

Bits, Bytes, and Integers

Professor Hugh C. Lauer
CS-2011, Machine Organization and Assembly Language

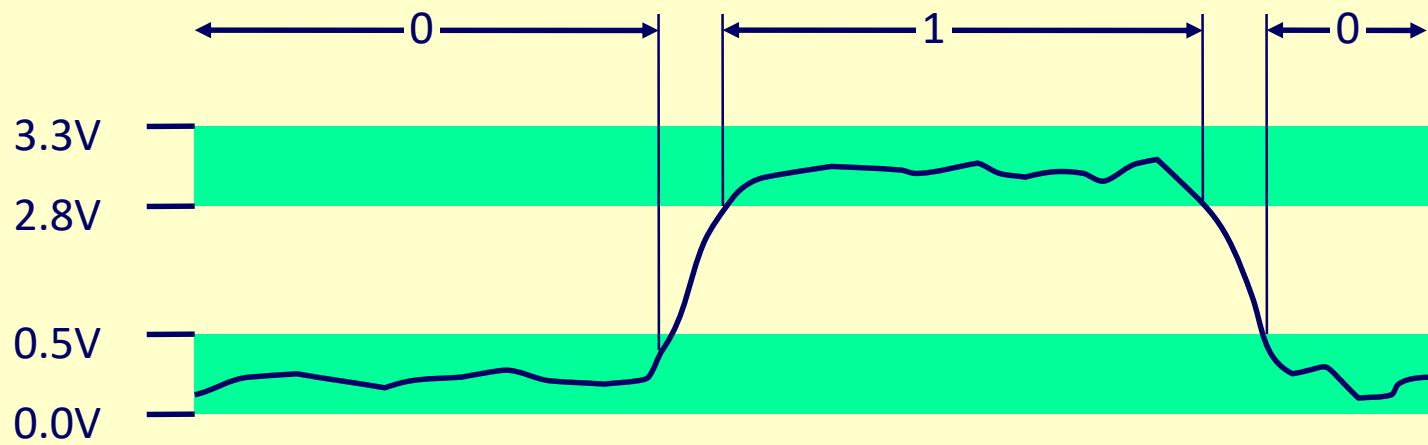
(Slides include copyright materials from *Computer Systems: A Programmer's Perspective*, by Bryant and O'Hallaron, and from *The C Programming Language*, by Kernighan and Ritchie)

Bits, Bytes, and Integers

Reading Assignment: §2 thru §2.3

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

Binary Representations



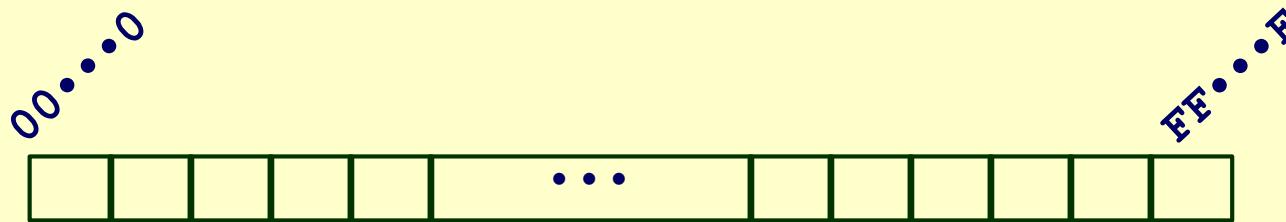
Encoding Byte Values

■ Byte = 8 bits

- Binary 0000000₂ to 1111111₂
- Decimal: 0₁₀ to 255₁₀
- Hexadecimal 00₁₆ to FF₁₆
 - Base 16 number representation
 - Use characters '0' to '9' and 'A' to 'F'
 - Write FA1D37B₁₆ 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	

Byte-oriented memory organization



- **Programs refer to virtual addresses**
 - Conceptually very large array of bytes
 - Actually implemented with hierarchy of different memory types
 - System provides address space private to particular “process”
 - Program being executed
 - Program can clobber its own data, but not that of others
- **Compiler + run-time system control allocation**
 - Where different program objects should be stored
 - All allocation within single virtual address space

Machine words

■ Machine has “word size”

- Nominal size of integer-valued data
 - Including addresses
- A lot of machines *still* use 32 bits (4 bytes) words
 - Limits addresses to 4GB
 - Becoming too small for memory-intensive applications
- High-end systems use 64 bits (8 bytes) words
 - Potential address space $\approx 1.8 \times 10^{19}$ bytes
 - x86-64 machines support 48-bit addresses: 256 Terabytes

Some machines *still* use
16 bits — 64 KB addresses

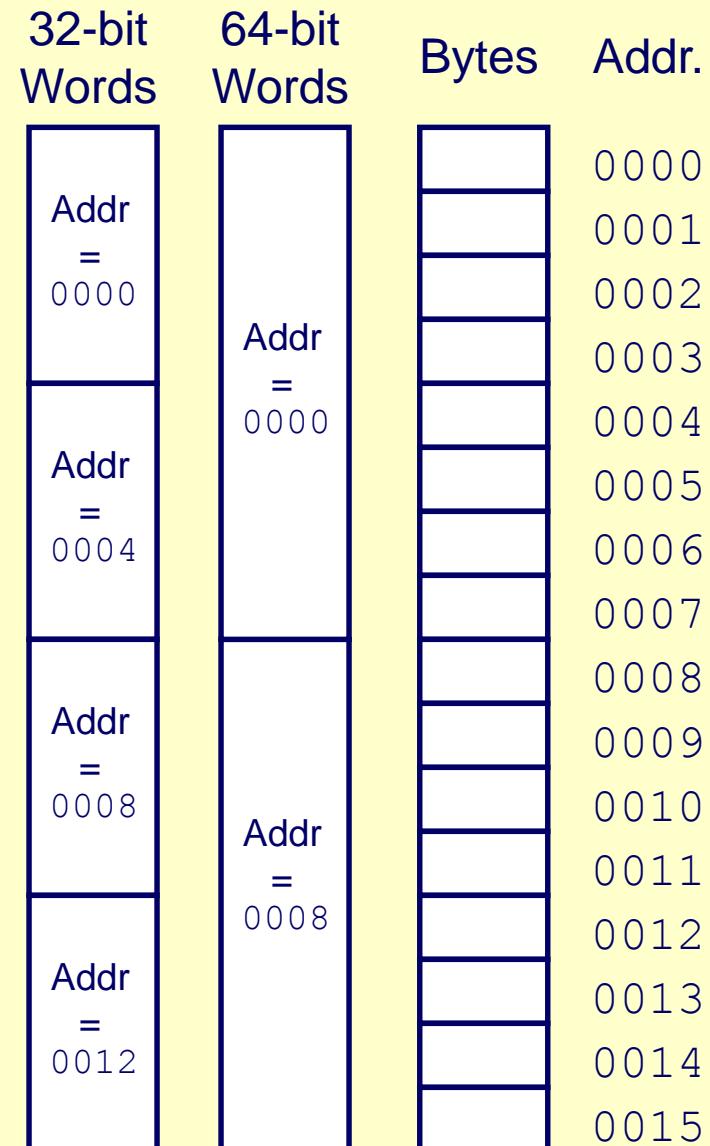
■ Machines support multiple data formats

- 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 2 (16-bit), 4 (32-bit), or 8 (64-bit)



Data Representations

C Data Type	Typical 32-bit	Intel IA32	x86-64
char	1	1	1
short	2	2	2
int	4	4	4
long	4	4	8
long long	8	8	8
float	4	4	4
double	8	8	8
long double	8	10/12	10/16
pointer	4	4	8

Byte Ordering

- How should bytes within a multi-byte word be ordered in memory?
- Conventions
 - Big Endian: Sun, PPC Mac, Internet
 - Least significant byte has highest address
 - Little Endian: x86
 - Least significant byte has lowest address
- Trivia question:– When and how did the terms “big endian” and “little endian” enter the English language?

Byte Ordering Example

■ BigEndian

- Least significant byte has highest address

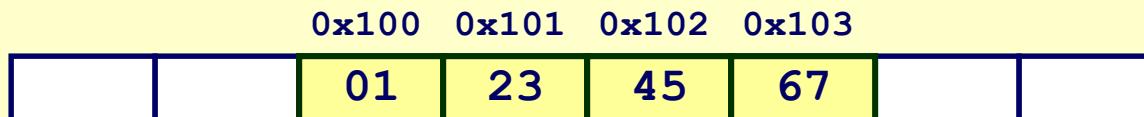
■ LittleEndian

- Least significant byte has lowest address

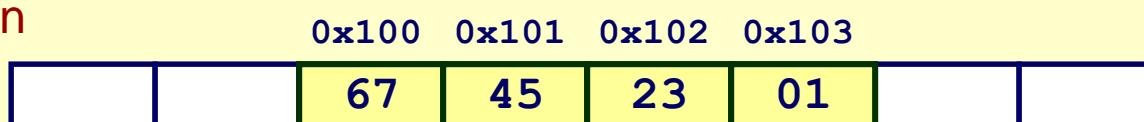
■ Example

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

BigEndian



LittleEndian



Reading Byte-Reversed Listings

■ Disassembly

- Text representation of binary machine code
- Generated by program that reads the machine code

■ Example Fragment

Address	Instruction Code	Assembly Rendition
8048365:	5b	pop %ebx
8048366:	81 c3 ab 12 00 00	add \$0x12ab,%ebx
804836c:	83 bb 28 00 00 00 00	cmpl \$0x0,0x28(%ebx)

■ Deciphering Numbers

- Value:
 - Pad to 32 bits:
 - Split into bytes:
 - Reverse:
- 0x12ab
0x000012ab
00 00 12 ab
ab 12 00 00

Examining Data Representations

■ Code to Print Byte Representation of Data

- Casting pointer to unsigned char * creates byte array

```
typedef unsigned char *pointer;

void show_bytes(pointer start, int len) {
    int 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

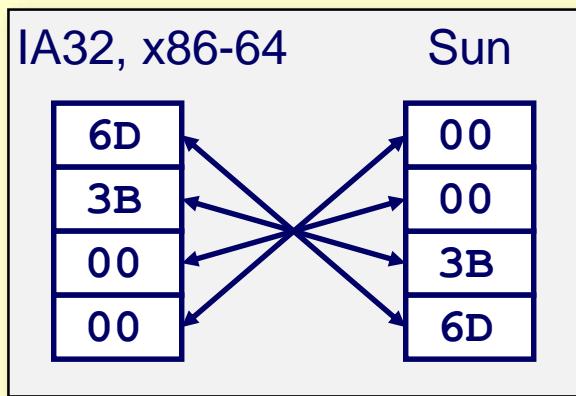
```
int a = 15213;  
printf("int a = 15213;\n");  
show_bytes((pointer) &a, sizeof(int));
```

Result (Linux):

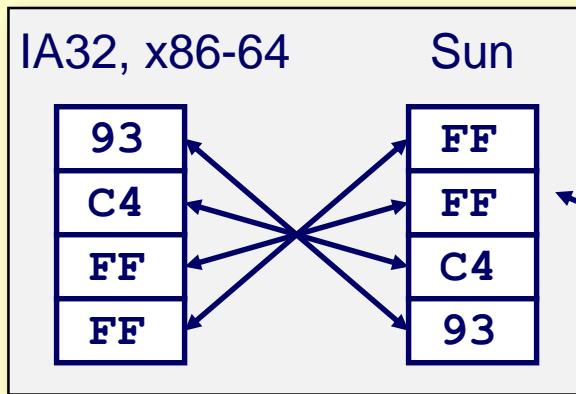
```
int a = 15213;  
0x11ffffcb8 0x6d  
0x11ffffcb9 0x3b  
0x11ffffcba 0x00  
0x11ffffcbb 0x00
```

Representing Integers

```
int A = 15213;
```

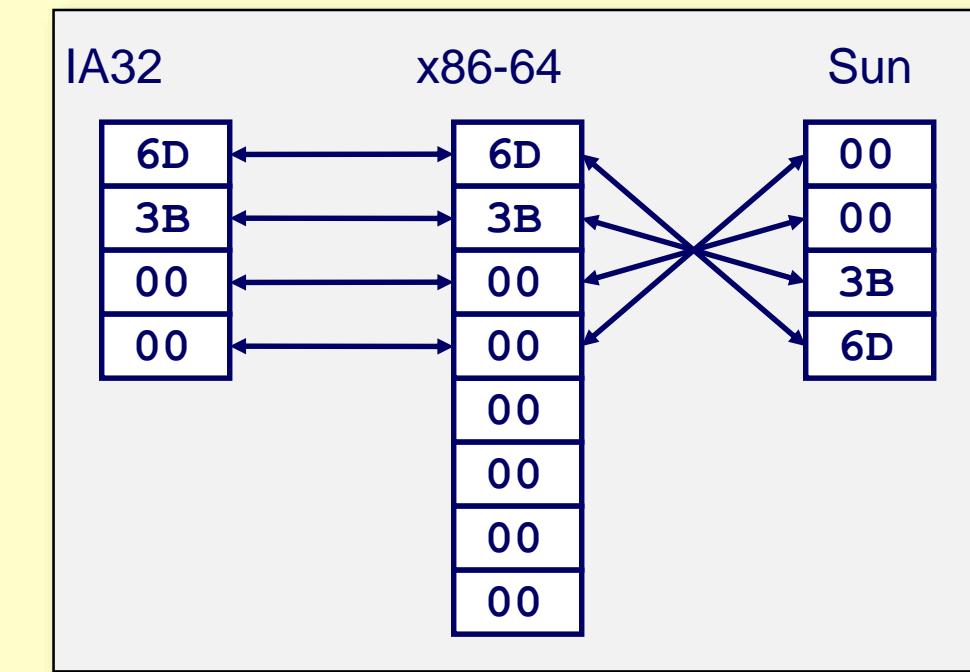


```
int B = -15213;
```



Decimal: 15213
 Binary: 0011 1011 0110 1101
 Hex: 3 B 6 D

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

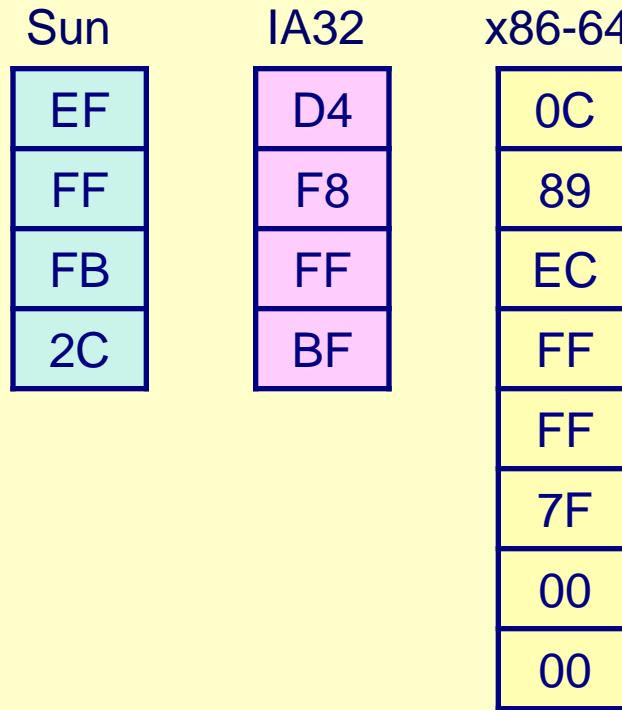


Smallest memory addr at top ↑

Two's complement representation
 (covered later)

Representing Pointers

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



Different compilers & machines assign different locations to objects

Representing Strings

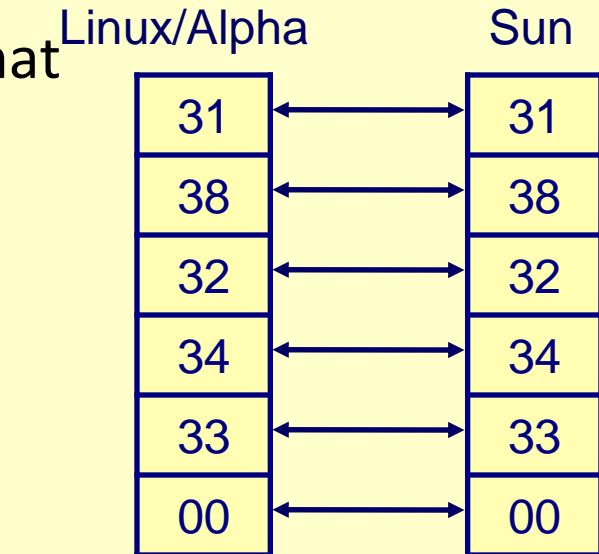
```
char S[6] = "18243";
```

■ 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



Questions?

Today: Bits, Bytes, and Integers

■ Representing information as bits

■ Bit-level manipulations

Reading Assignment: §2.1.7–§2.1.10

■ Integers

- Representation: unsigned and signed
- Conversion, casting
- Expanding, truncating
- Addition, negation, multiplication, shifting

■ Summary

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)

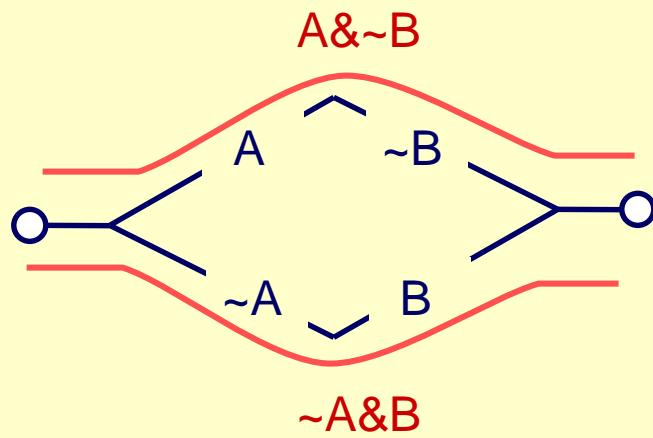
- $A ^ B = 1$ when either $A=1$ or $B=1$, but not both

$^$	0	1
0	0	1
1	1	0

Application of Boolean Algebra

■ Applied to Digital Systems by Claude Shannon

- 1937 MIT Master's Thesis
- Reason about networks of relay switches
 - Encode closed switch as 1, open switch as 0



Connection when

$$A \& \sim B \mid \sim A \& B$$

$$= A^{\wedge}B$$

General Boolean Algebras

■ Operate on Bit Vectors

- Operations applied bitwise

$$\begin{array}{rcl} 01101001 & 01101001 & 01101001 \\ \& 01010101 & | \ 01010101 & ^ \ 01010101 \\ \hline 01000001 & 01111101 & 00111100 & \sim 01010101 \\ & & & \hline & & & 10101010 \end{array}$$

■ All of the Properties of Boolean Algebra Apply

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)

- $\sim 0x41 \rightarrow 0xBE$
 - $\sim 01000001_2 \rightarrow 1011110_2$
- $\sim 0x00 \rightarrow 0xFF$
 - $\sim 00000000_2 \rightarrow 11111111_2$
- $0x69 \& 0x55 \rightarrow 0x41$
 - $01101001_2 \& 01010101_2 \rightarrow 01000001_2$
- $0x69 | 0x55 \rightarrow 0x7D$
 - $01101001_2 | 01010101_2 \rightarrow 01111101_2$

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`
`p && p -> f`
(avoids null pointer access)

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 right

■ Undefined Behavior

- Shift amount < 0 or \geq word size
- Different machines behave differently

Argument x	01100010
$\ll 3$	00010000
Log. $\gg 2$	00011000
Arith. $\gg 2$	00011000

Argument x	10100010
$\ll 3$	00010000
Log. $\gg 2$	00101000
Arith. $\gg 2$	11101000

Questions?

Today: 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

Reading Assignment: §2.2

Encoding integers

Unsigned

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

§ 2.2.2 in Bryant and O'Hallaron

Two's Complement

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

Sign Bit

§ 2.2.3 in Bryant and O'Hallaron



Encoding integers

Unsigned

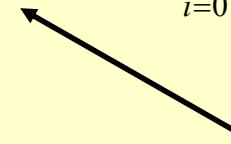
$$B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

Two's Complement

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

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

Sign
Bit



■ C short 2 bytes long

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

■ Sign bit

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

Encoding example (continued)

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
Bits, Bytes, and Integers				

Numeric ranges

■ Unsigned Values

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

Values for $W = 16$

■ Two's Complement Values

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

■ Other Values

- Minus 1
111...1

	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

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

■ 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

Today: 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

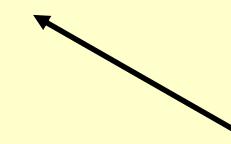
Reading Assignment: §2.2 (continued)

(Review) 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

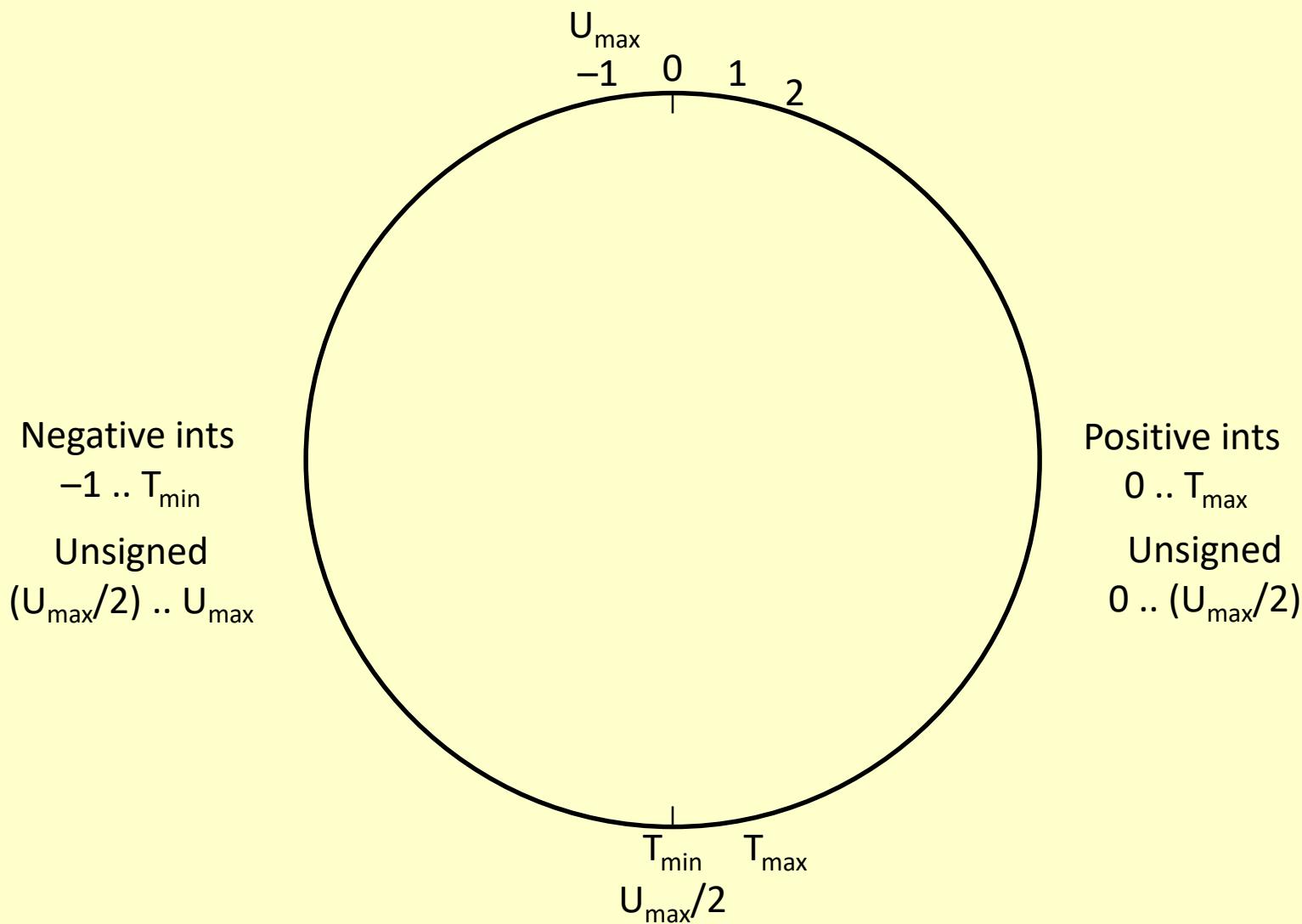
■ Unsigned Values

- $U_{Min} = 0$
000...0
- $U_{Max} = 2^w - 1$
111...1

■ Two's Complement Values

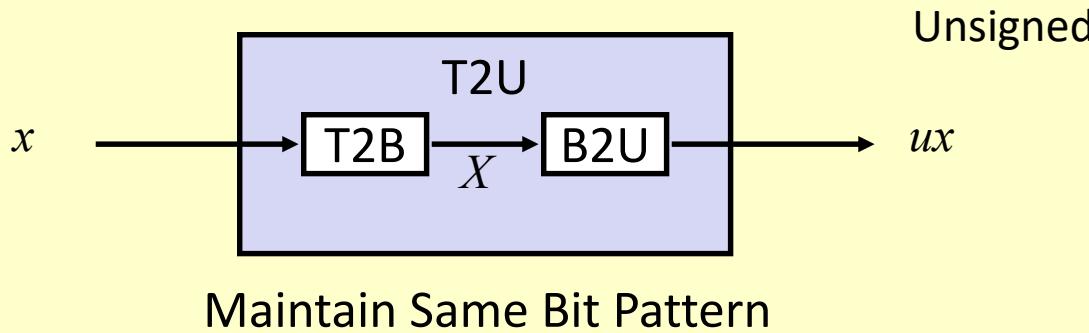
- $T_{Min} = -2^{w-1}$
100...0
- $T_{Max} = 2^{w-1} - 1$
011...1

(Review) Signed vs. Unsigned

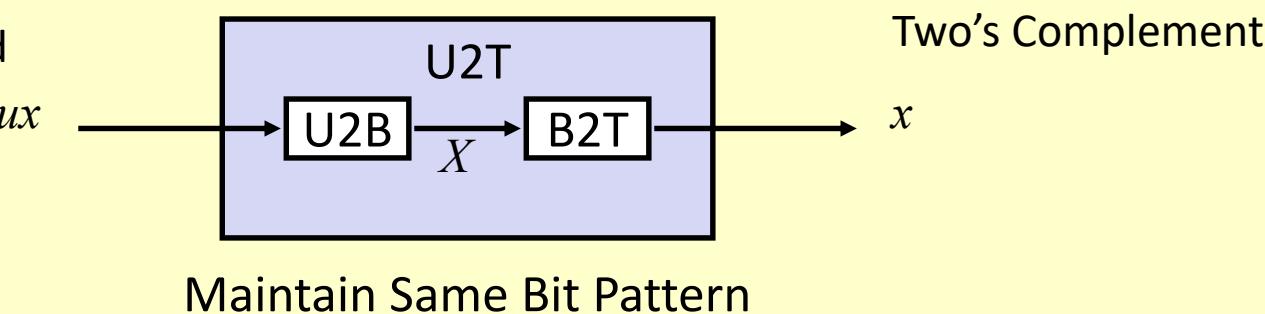


Mapping Between Signed & Unsigned

Two's Complement

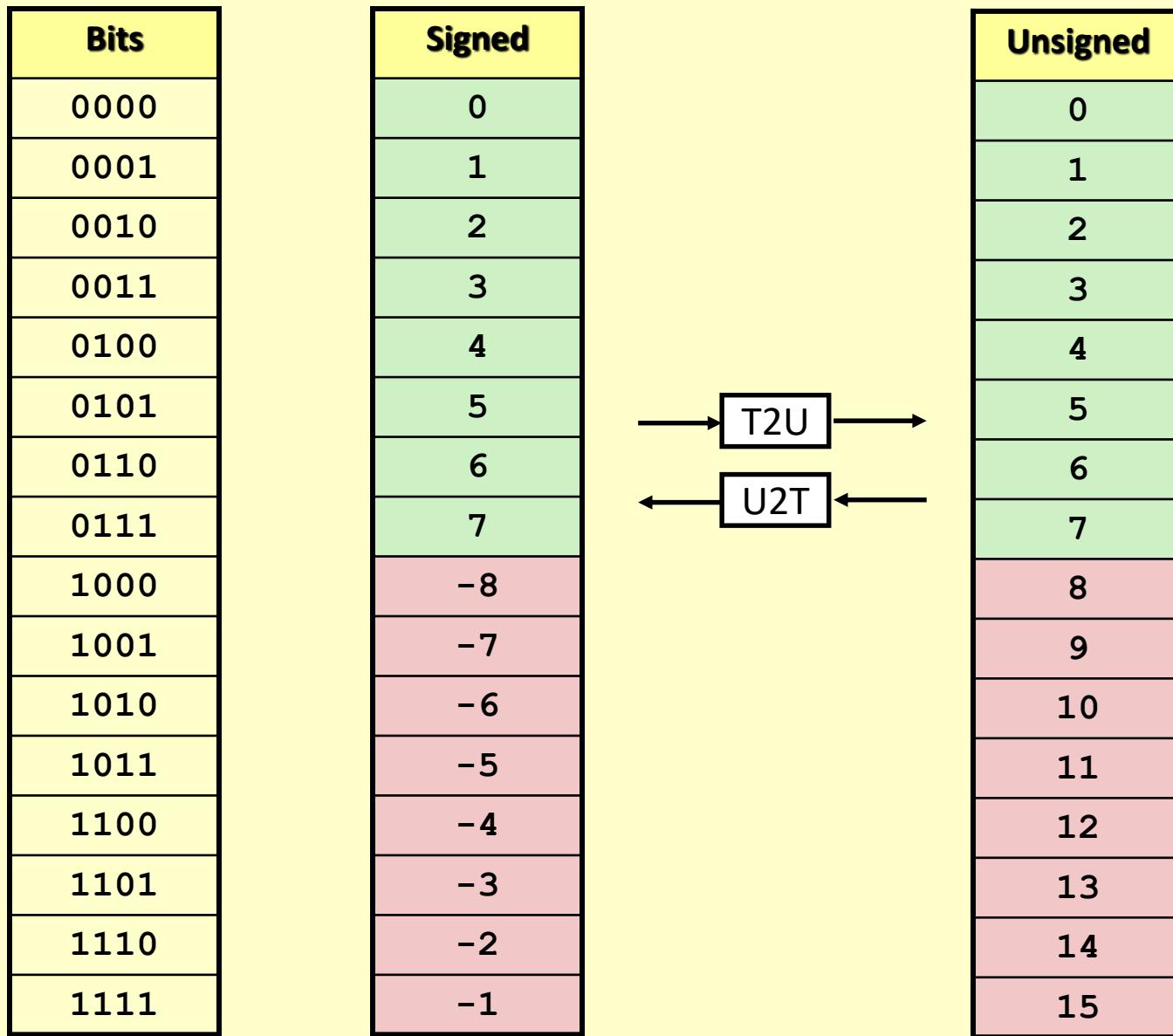


Unsigned

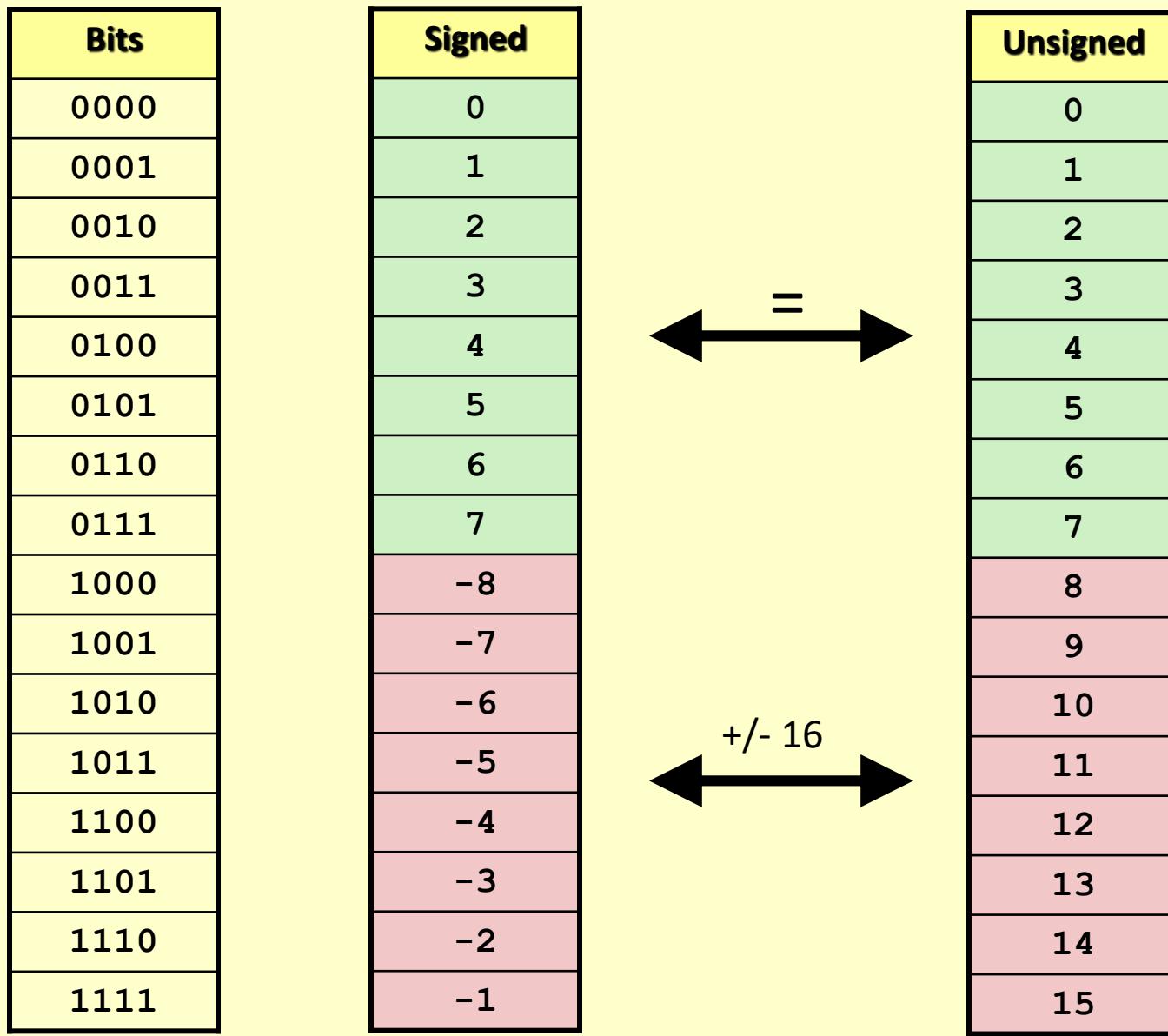


- **Mappings between unsigned and two's complement numbers:-**
keep same bit representations and reinterpret

Mapping Signed \leftrightarrow Unsigned

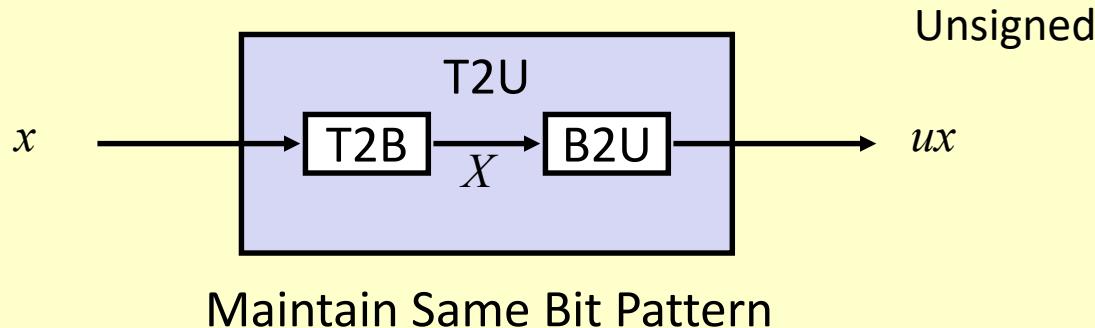


Mapping Signed \leftrightarrow Unsigned



Relation between Signed & Unsigned

Two's Complement



$w-1 \quad 0$

$ux \quad [+, +, +, \dots, +, +, +]$

$x \quad [-, +, +, \dots, +, +, +]$



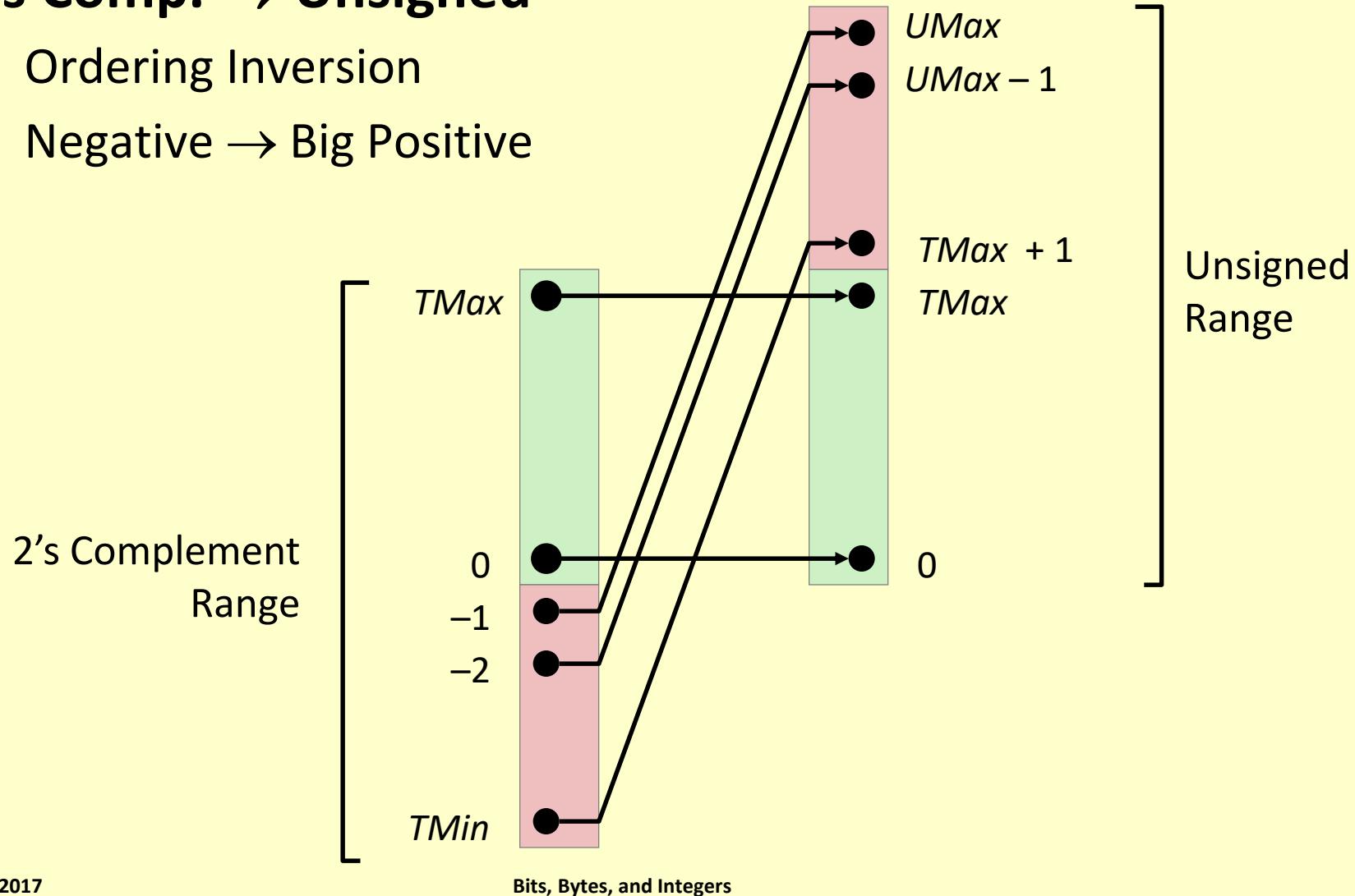
Large negative weight
becomes
Large positive weight

$$ux = \begin{cases} x & x \geq 0 \\ x + 2^w & x < 0 \end{cases}$$

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 function 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 ₁	Constant ₂	Relation	Evaluation
0	0U	<code>==</code>	unsigned
-1	0	<code><</code>	signed
-1	0U	<code>></code>	unsigned
2147483647	-2147483647-1	<code>></code>	signed
2147483647U	-2147483647-1	<code><</code>	unsigned
-1	-2	<code>></code>	signed
(unsigned)-1	-2	<code>></code>	unsigned
2147483647	2147483648U	<code><</code>	unsigned
2147483647	(int) 2147483648U	<code>></code>	signed

Code Security Example

```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

- Similar to code found in FreeBSD's implementation of `getpeername`
- There are legions of smart people trying to find vulnerabilities in programs

Typical Usage

```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

```
#define MSIZE 528

void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, MSIZE);
    printf("%s\n", mybuf);
}
```

Malicious Usage

```
/* Declaration of library function memcpy */
void *memcpy(void *dest, void *src, size_t n);
```

```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

```
#define MSIZE 528

void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, -MSIZE);
    . . .
}
```

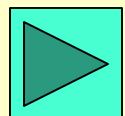
size_t is defined as
an *unsigned* integer
(K&R §A7.4.8)
<stddef.h>

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`!!



Questions?

Today: 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

Reading Assignment: §2.2 (continued)

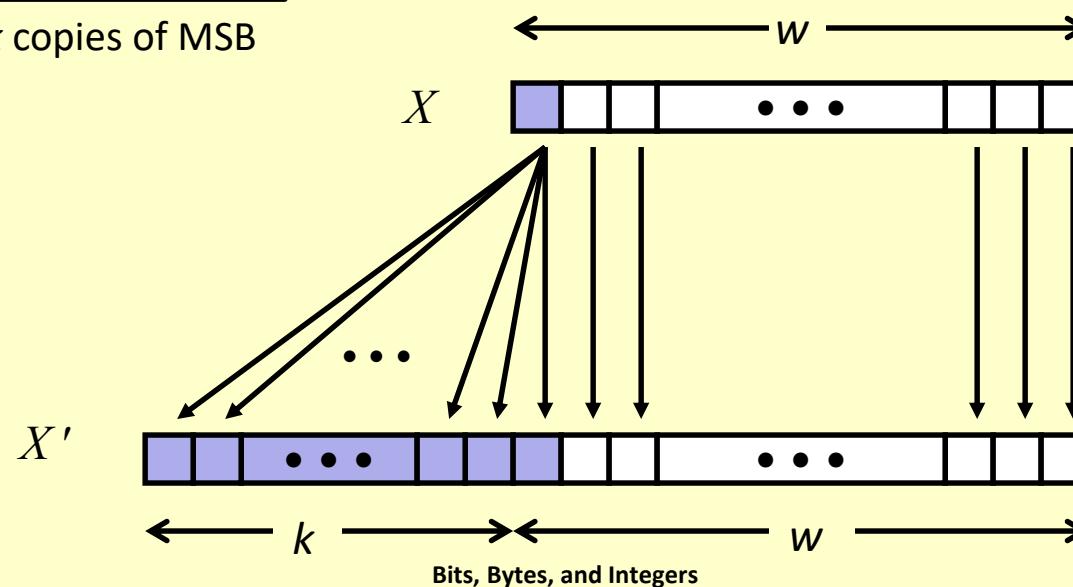
Sign Extension

■ Task:

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

■ Rule:

- Make k copies of sign bit:
- $X' = \underbrace{X_{w-1}, \dots, X_{w-1}}_{k \text{ copies of MSB}}, X_{w-1}, X_{w-2}, \dots, X_0$



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

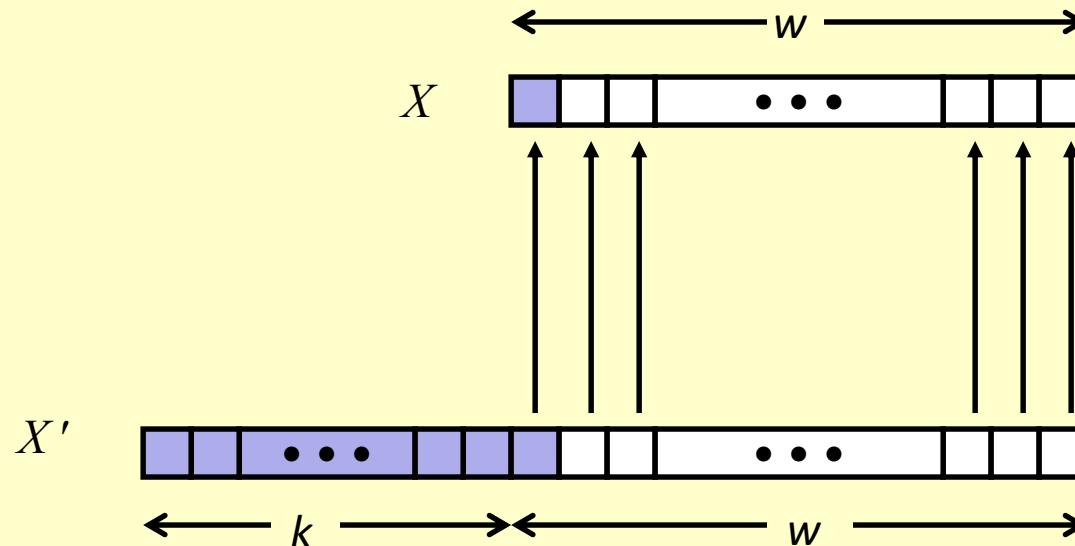
■ Expanding (e.g., short int to int)

- 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

Truncating – Illustration



■ For unsigned numbers:-

- Equivalent to dividing by 2^k and keeping the remainder
 - i.e., $\text{truncate}(x, k) = x \bmod 2^k$

■ For signed numbers:-

- Same bit result ...
- ... but truncated number may have different sign!

Today: 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

Reading Assignment: §2.3

Negation: Complement & Increment

■ Claim: Following Holds for 2's Complement

$$\sim x + 1 == -x$$

■ Complement

- Observation: $\sim x + x == 1111\dots111 == -1$

x	<table border="1" style="display: inline-table;"><tr><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td></tr></table>	1	0	0	1	1	1	0	1
1	0	0	1	1	1	0	1		
+	<table border="1" style="display: inline-table;"><tr><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr></table>	0	1	1	0	0	0	1	0
0	1	1	0	0	0	1	0		
<hr/>									
-1	<table border="1" style="display: inline-table;"><tr><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr></table>	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1		

■ Complete Proof?



Complement & Increment Examples

x = 15213

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
$\sim x$	-15214	C4 92	11000100 10010010
$\sim x + 1$	-15213	C4 93	11000100 10010011
y	-15213	C4 93	11000100 10010011

x = 0

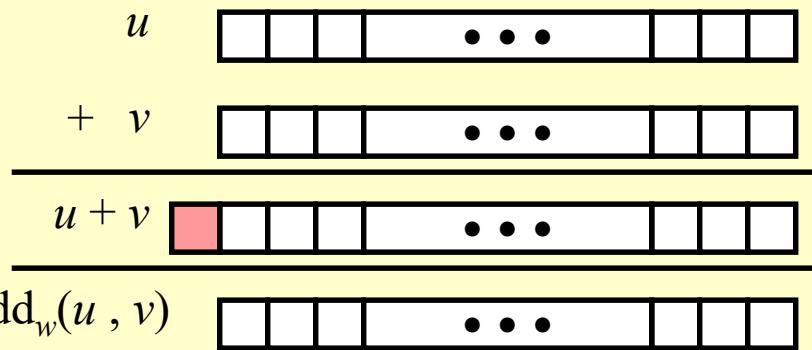
	Decimal	Hex	Binary
0	0	00 00	00000000 00000000
~ 0	-1	FF FF	11111111 11111111
$\sim 0 + 1$	0	00 00	00000000 00000000

Unsigned Addition

Operands: w bits

True Sum: $w+1$ bits

Discard Carry: w bits



■ Standard Addition Function

- Ignores carry output

■ Implements Modular Arithmetic

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

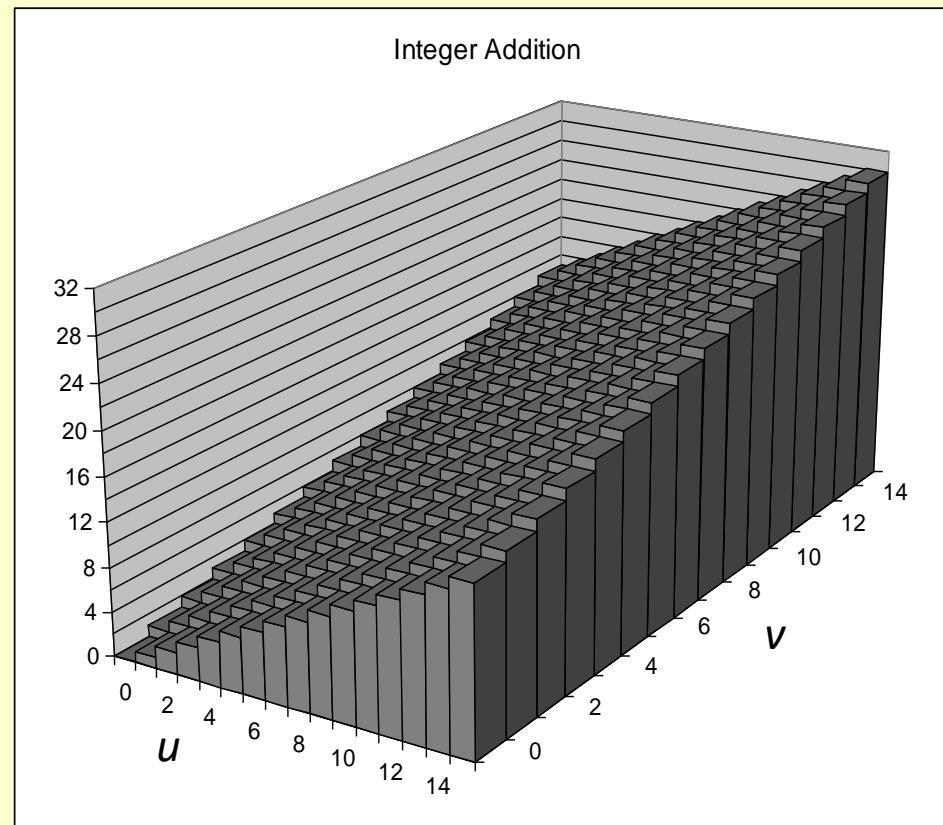
$$\text{UAdd}_w(u, v) = \begin{cases} u + v & u + v < 2^w \\ u + v - 2^w & u + v \geq 2^w \end{cases}$$

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

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

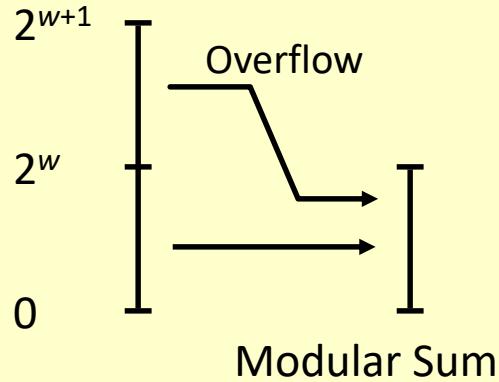


Visualizing Unsigned Addition

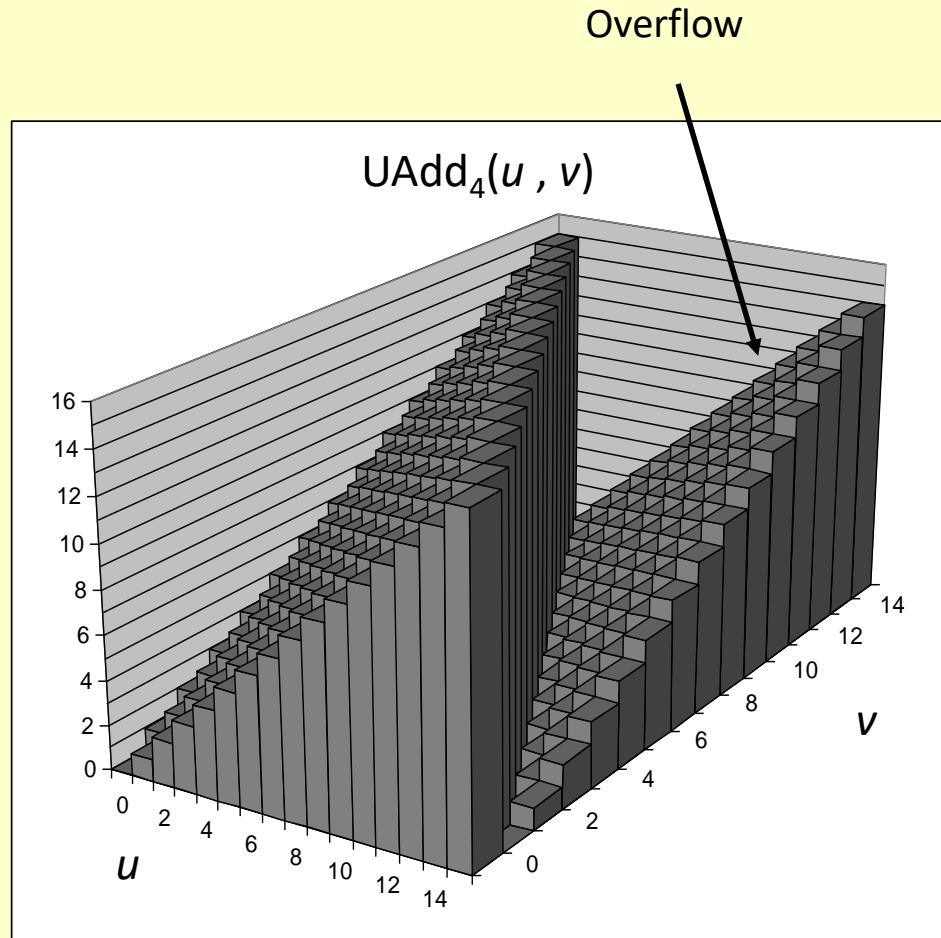
Wraps Around

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

True Sum



Overflow



Mathematical Properties

■ Modular Addition Forms an *Abelian Group*

- **Closed** under addition

$$0 \leq \text{UAdd}_w(u, v) \leq 2^w - 1$$

- **Commutative**

$$\text{UAdd}_w(u, v) = \text{UAdd}_w(v, u)$$

- **Associative**

$$\text{UAdd}_w(t, \text{UAdd}_w(u, v)) = \text{UAdd}_w(\text{UAdd}_w(t, u), v)$$

- **0** is additive identity

$$\text{UAdd}_w(u, 0) = u$$

- Every element has additive **inverse**

- Let $\text{UComp}_w(u) = 2^w - u$

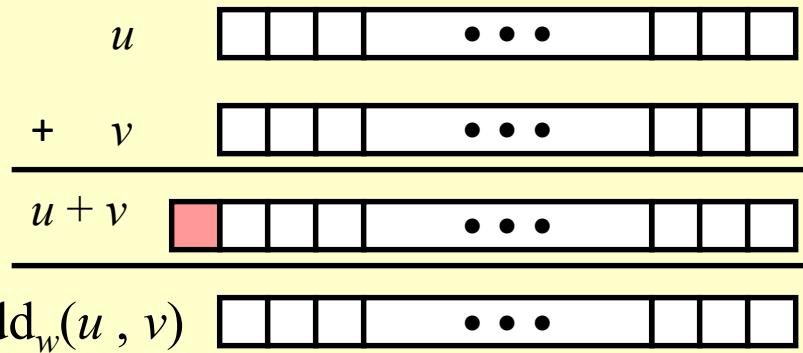
$$\text{UAdd}_w(u, \text{UComp}_w(u)) = 0$$

Two's Complement Addition

Operands: w bits

True Sum: $w+1$ bits

Discard Carry: w bits



■ TAdd and UAdd have Identical Bit-Level Behavior

- Signed vs. unsigned addition in C:

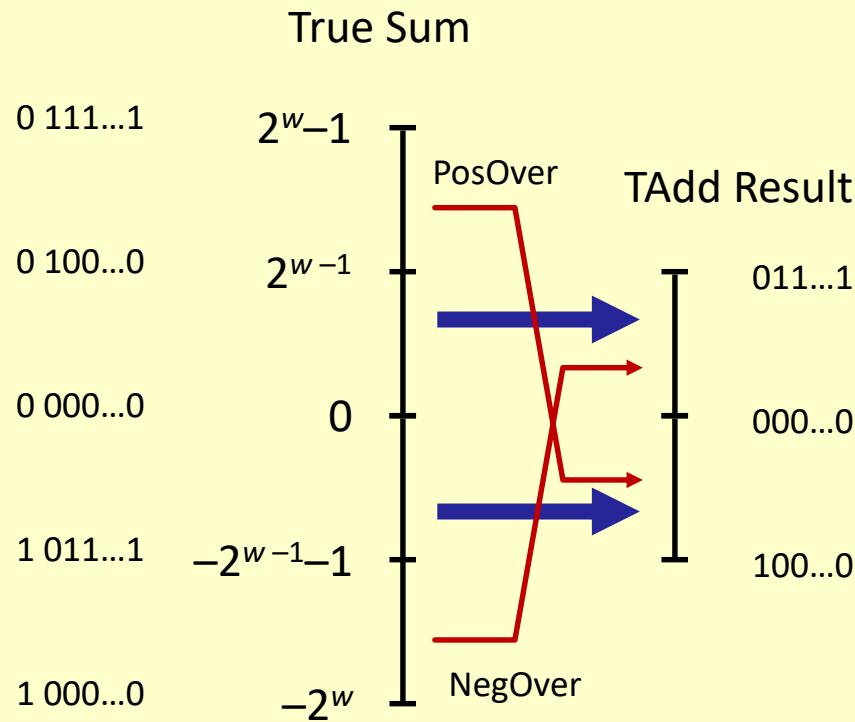
```
int s, t, u, v;
s = (int) ((unsigned) u + (unsigned) v);
t = u + v
```

- 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

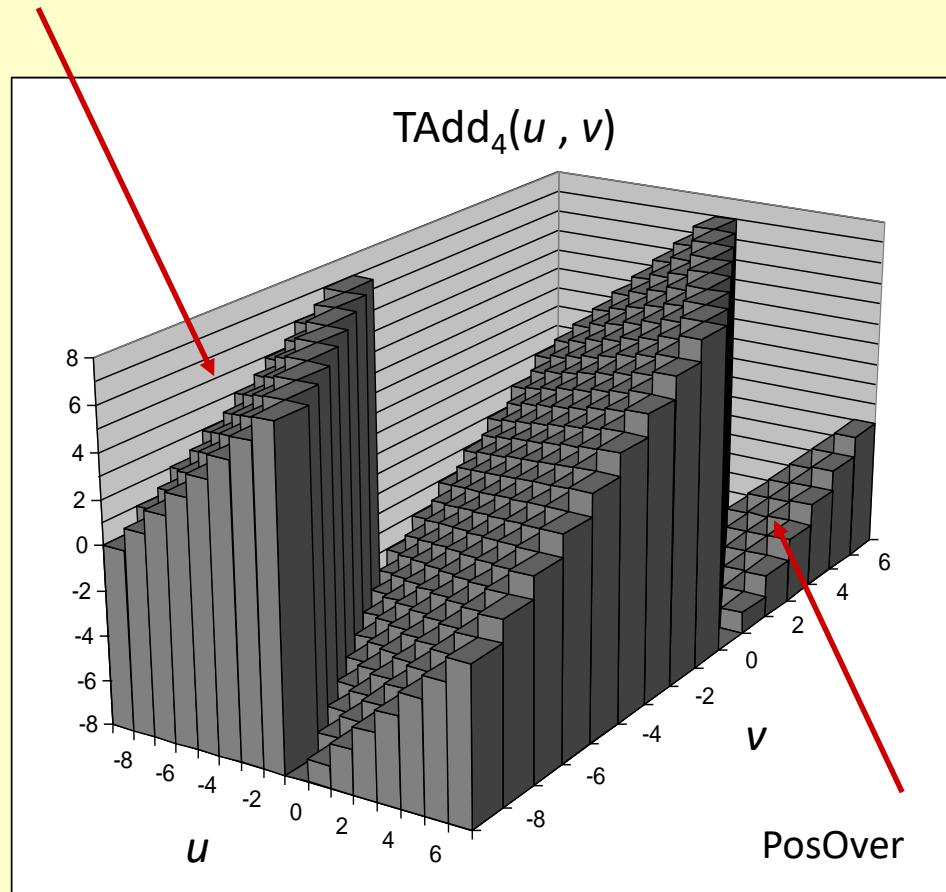
■ Values

- 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

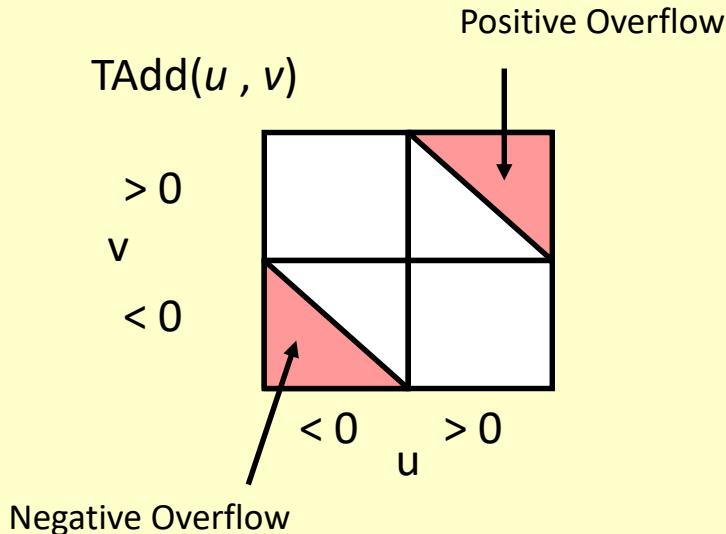
NegOver



Characterizing TAdd

■ Functionality

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



$$TAdd_w(u, v) = \begin{cases} u + v + 2^w & u + v < TMin_w \text{ (NegOver)} \\ u + v & TMin_w \leq u + v \leq TMax_w \\ u + v - 2^w & TMax_w < u + v \text{ (PosOver)} \end{cases}$$

Mathematical Properties of TAdd

■ Isomorphic Group to unsigned with UAdd

- $TAdd_w(u, v) = U2T(UAdd_w(T2U(u), T2U(v)))$
 - Since both have identical bit patterns

■ Two's Complement Under TAdd Forms a Group

- Closed, Commutative, Associative, 0 is additive identity
- Every element has additive inverse

$$TComp_w(u) = \begin{cases} -u & u \neq TMin_w \\ TMin_w & u = TMin_w \end{cases}$$

Questions?

Multiplication — shifting and adding

$$\begin{array}{r} 15213 \\ \times \quad \underline{2011} \\ 15213 \\ 152130 \\ \underline{30426000} \\ 30593343 \end{array}$$

Multiplication in Binary

Decimal	Hex	Binary
15213	3B 6D	00000000 00000000 00111011 01101101
2011	07 DB	00000111 11011011
		00111011 01101101
		0 01110110 1101101
		001 11011011 01101
		0011 10110110 1101
		001110 11011011 01
		0011101 10110110 1
		00111011 01101101
		0 01110110 1101101
		00 11101101 101101
30593343	1 D2 D1 3F	0001 11010010 11010001 00111111

Multiplication in Binary – mod 2^w

Decimal	Hex	Binary	
15213	3B 6D	00000000 00000000	00111011 01101101
2011	07 DB		00000111 11011011
			00111011 01101101
		0	01110110 1101101
		001	11011011 01101
		0011	10110110 1101
		001110	11011011 01
		0011101	10110110 1
		00111011	01101101
		0 01110110	1101101
		00 11101101	101101
30593343	1 D2 D1 3F	0001 11010010	11010001 00111111

Overflow bits truncated



Multiplication in Binary

- Any difference between unsigned and twos complement?
- A:- No!
 - Same bit pattern
 - Different interpretation
 - Different overflows

Multiplication performance

- **10 or more machine cycles**
 - In modern processors
- **Multiply-and-Add instruction**
- **Easily pipelined**
 - E.g., dot product

Compiler optimizations

- Small, constant multipliers
 - E.g., array indexes
- Shift instructions followed by adds
- Specialized instructions

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

k

u

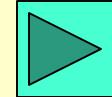
$*$ 2^k

$u \cdot 2^k$

$$\begin{aligned} \text{UMult}_w(u, 2^k) \\ \text{TMult}_w(u, 2^k) \end{aligned}$$

■ Examples

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



Compiled Multiplication Code

C Function

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

Compiled Arithmetic Operations

```
leal (%eax,%eax,2), %eax
sall $2, %eax
```

Explanation

```
t <- x+x*2
return t << 2;
```

- C compiler automatically generates shift/add code when multiplying by constant

Questions?

1

Division — subtracting and shifting

$$\begin{array}{r} 1170 \\ \hline 13) 15213 \\ \underline{13} \quad 213 \\ 22 \quad 00 \\ \underline{13} \quad 00 \\ 91 \quad 0 \\ \underline{91} \quad 0 \\ \qquad \qquad 3 \end{array}$$

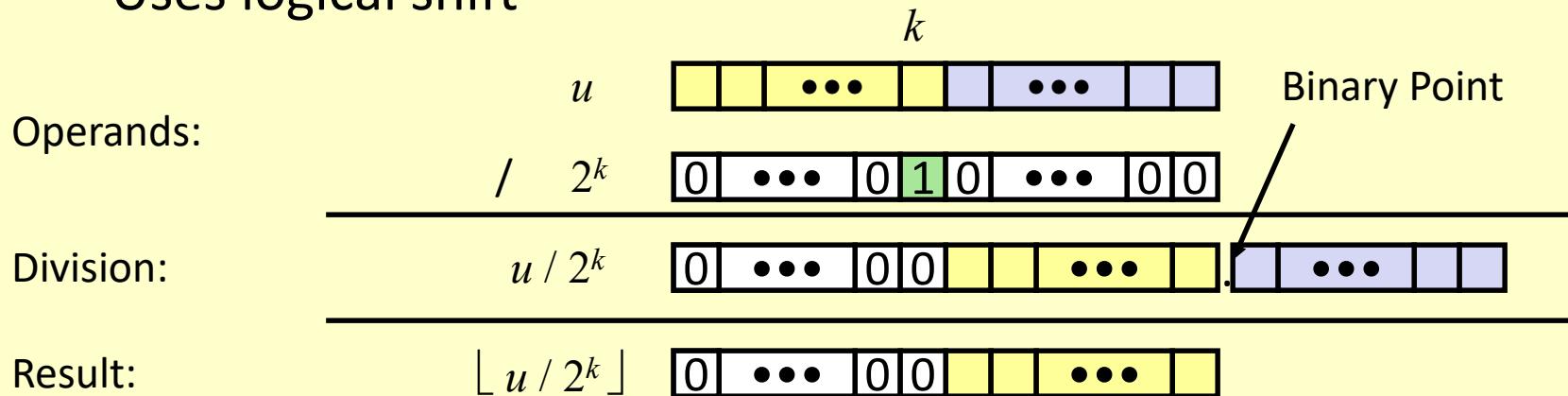
Binary Division

- **Similar principle:-**
 - Subtract divisor from high-order bits of dividend
 - Shift, bring down next bit
 - Repeat
- **Very costly in number of cycles**
 - 30 or more for integer divide
- **Not very amenable to pipelining**
- **Highly specialized designs**
 - Mostly implemented in Floating Point arithmetic units

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



Compiled Unsigned Division Code

C Function

```
unsigned udiv8(unsigned x)
{
    return x/8;
}
```

Compiled Arithmetic Operations

```
shrl $3, %eax
```

Explanation

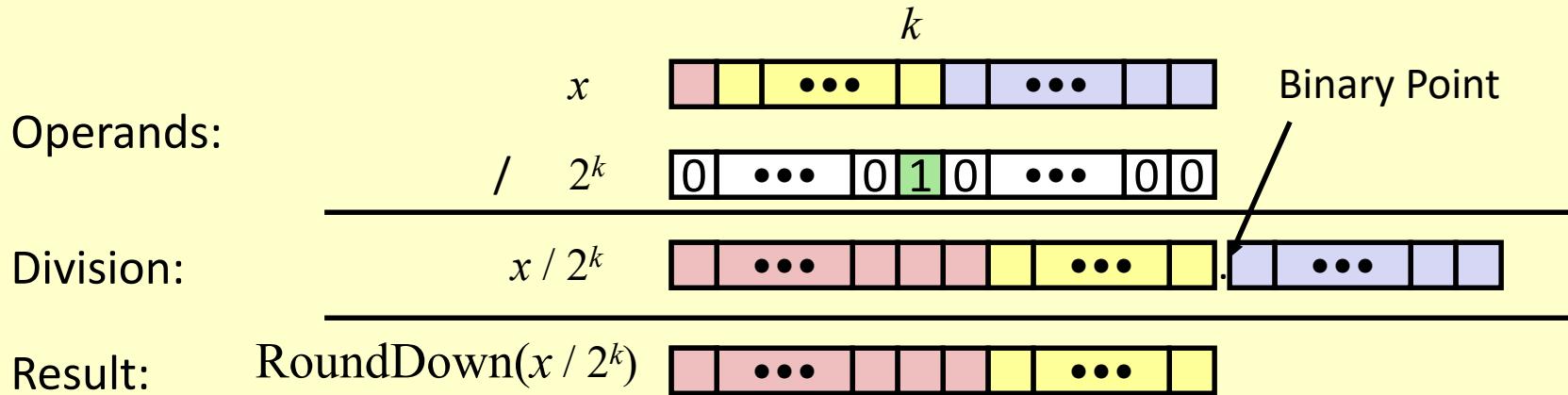
```
# Logical shift
return x >> 3;
```

- Uses logical shift for unsigned
- For Java Users
 - Logical shift written as >>>

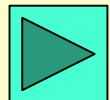
Signed Power-of-2 Divide with Shift

■ Quotient of Signed by Power of 2

- $x \gg k$ gives $\lfloor x / 2^k \rfloor$
- Uses arithmetic shift
- Rounds wrong direction when $u < 0$



	Division	Computed	Hex	Binary
y	-15213	-15213	C4 93	11000100 10010011
y >> 1	-7606.5	-7607	E2 49	11100010 01001001
y >> 4	-950.8125	-951	FC 49	11111100 01001001
y >> 8	-59.4257813	-60	FF C4	11111111 11000100



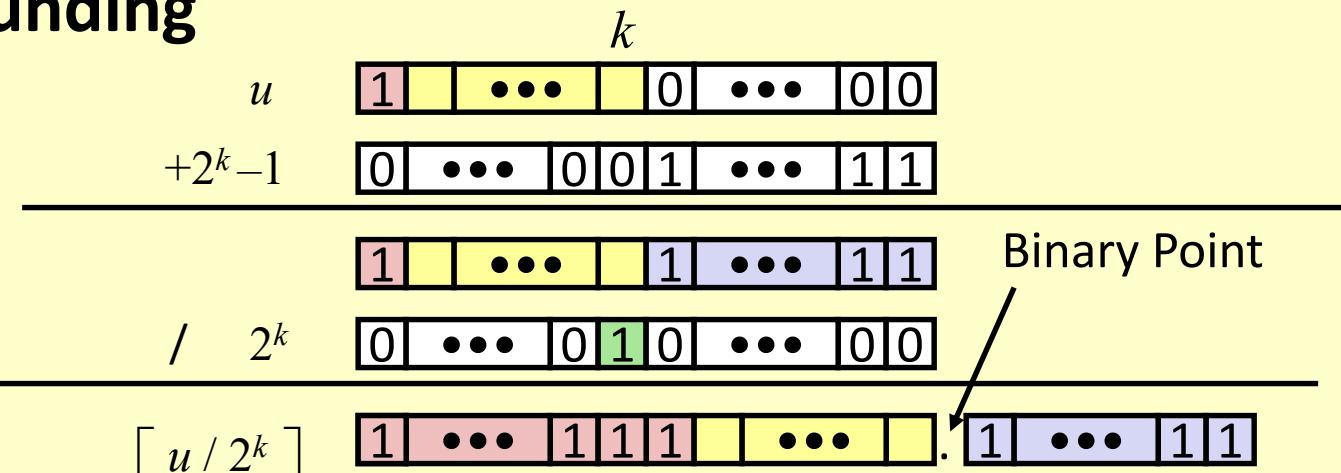
Correct Power-of-2 Divide

■ Quotient of Negative Number by Power of 2

- Want $\lceil x / 2^k \rceil$ (Round Toward 0)
- Compute as $\lfloor (x+2^k-1) / 2^k \rfloor$
 - In C: $(x + (1<<k) - 1) \gg k$
 - Biases dividend toward 0

Case 1: No rounding

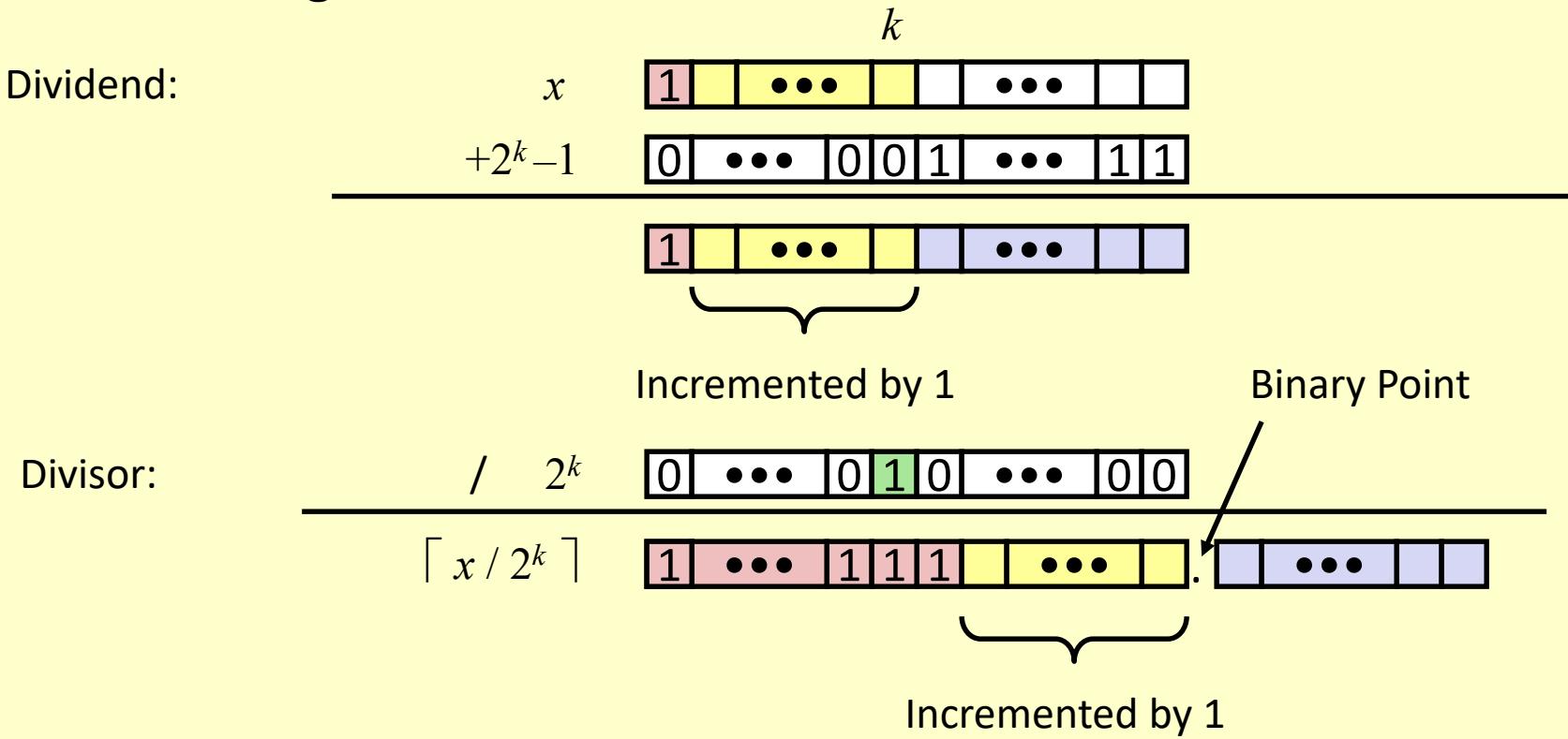
Dividend:



Biasing has no effect

Correct Power-of-2 Divide (Cont.)

Case 2: Rounding



Biasing adds 1 to final result

Compiled Signed Division Code

C Function

```
int idiv8(int x)
{
    return x/8;
}
```

Compiled Arithmetic Operations

```
testl %eax, %eax
js    L4
L3:
sarl $3, %eax
ret
L4:
addl $7, %eax
jmp  L3
```

Explanation

```
if x < 0
    x += 7;
# Arithmetic shift
return x >> 3;
```

- **Uses arithmetic shift for int**
- **For Java Users**
 - Arith. shift written as `>>`

Questions?

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)

Arithmetic: Basic Rules

- Unsigned ints, 2's complement ints are isomorphic rings: isomorphism = casting
- Left shift
 - Unsigned/signed: multiplication by 2^k
 - Always logical shift
- Right shift
 - Unsigned: logical shift, div (division + round to zero) by 2^k
 - Signed: arithmetic shift
 - Positive numbers: div (division + round to zero) by 2^k
 - Negative numbers: div (division + round away from zero) by 2^k

Questions?

Today: Integers

- Representation: unsigned and signed
- Conversion, casting
- Expanding, truncating
- Addition, negation, multiplication, shifting
- Summary

Properties of Unsigned Arithmetic

■ Unsigned Multiplication with Addition Forms Commutative Ring

- Addition is commutative group

- Closed under multiplication

$$0 \leq \text{UMult}_w(u, v) \leq 2^w - 1$$

- Multiplication Commutative

$$\text{UMult}_w(u, v) = \text{UMult}_w(v, u)$$

- Multiplication is Associative

$$\text{UMult}_w(t, \text{UMult}_w(u, v)) = \text{UMult}_w(\text{UMult}_w(t, u), v)$$

- 1 is multiplicative identity

$$\text{UMult}_w(u, 1) = u$$

- Multiplication distributes over addition

$$\text{UMult}_w(t, \text{UAdd}_w(u, v)) = \text{UAdd}_w(\text{UMult}_w(t, u), \text{UMult}_w(t, v))$$

Properties of Two's Comp. Arithmetic

■ Isomorphic Algebras

- Unsigned multiplication and addition
 - Truncating to w bits
- Two's complement multiplication and addition
 - Truncating to w bits

■ Both Form Rings

- Isomorphic to ring of integers mod 2^w

■ Comparison to (Mathematical) Integer Arithmetic

- Both are rings
- Integers obey ordering properties, e.g.,

$$u > 0 \quad \Rightarrow \quad u + v > v$$

$$u > 0, v > 0 \quad \Rightarrow \quad u \cdot v > 0$$

- These properties are not obeyed by two's comp. arithmetic

$$TMax + 1 == TMin$$

$$15213 * 30426 == -10030 \text{ (16-bit words)}$$

Why Should I Use Unsigned?

■ *Don't Use Just Because Number Nonnegative*

- 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)  
    . . .
```

■ *Do Use When Performing Modular Arithmetic*

- Multiprecision arithmetic

■ *Do Use When Using Bits to Represent Sets*

- Logical right shift, no sign extension



Course Theme: Abstraction *vs.* Reality

Professor Hugh C. Lauer
CS-2011, Machine Organization and Assembly Language

(Slides include copyright materials from *Computer Systems: A Programmer's Perspective*, by Bryant and O'Hallaron, and from *The C Programming Language*, by Kernighan and Ritchie)

Course Theme: Abstraction is good but don't forget reality

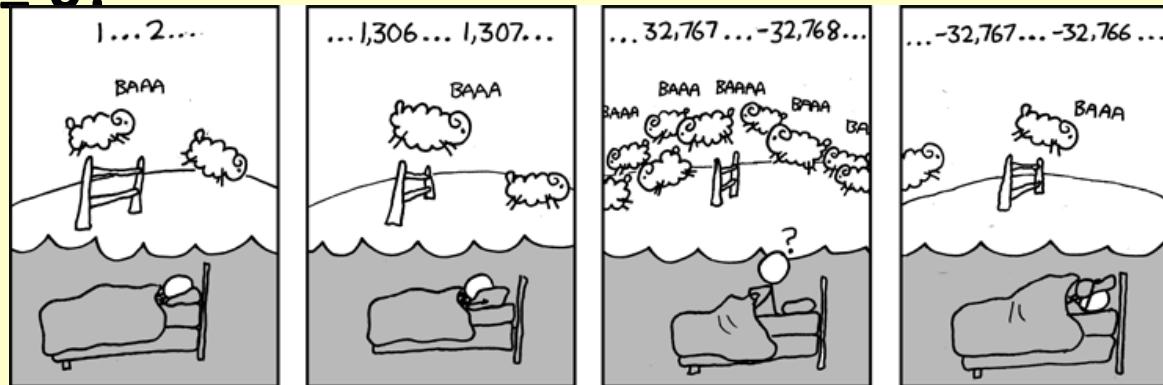
- Most CS courses emphasize abstraction
 - Abstract data types
 - Asymptotic analysis
- Abstractions have limits
 - Especially in the presence of bugs
 - Need to understand details of underlying implementations
- Useful outcomes
 - Become more effective programmers
 - Able to find and eliminate bugs efficiently
 - Able to understand and tune for program performance
 - Prepare for later “systems” classes in CS & ECE
 - Compilers, Operating Systems, Networks, Computer Architecture, Embedded Systems

Great Reality #1:

Ints are not integers, floats are not reals

■ Example 1: Is $x^2 \geq 0$?

- Float's: Yes!



- Int's:

- $40000 * 40000 \rightarrow 1,600,000,000$
- $50000 * 50000 \rightarrow ??$

-352,516,352

■ Example 2: Is $(x + y) + z = x + (y + z)$?

- Unsigned & Signed Int's: Yes!

- Float's:

- $(1e20 + -1e20) + 3.14 \rightarrow 3.14$
- $1e20 + (-1e20 + 3.14) \rightarrow ??$

Code security example

```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

- Similar to code found in FreeBSD's implementation of `getpeername()`
- There are legions of smart people trying to find vulnerabilities in programs

Typical usage

```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

```
#define MSIZE 528

void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, MSIZE);
    printf("%s\n", mybuf);
}
```

Malicious usage

```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

```
#define MSIZE 528

void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, -MSIZE);
    . . .
}
```

This line turns out to
be *toxic!*

Computer arithmetic

- **Does not generate random values**
 - Arithmetic operations have important mathematical properties
- **Cannot assume all “usual” mathematical properties**
 - Due to finiteness of representations
 - Integer operations satisfy “ring” properties
 - Commutative, associative, distributive
 - Floating point operations satisfy “ordering” properties
 - Monotonicity, values of signs
- **Observation**
 - Need to understand which abstractions apply in which contexts
 - Important issues for compiler writers and serious application programmers

Great Reality #2:

You've got to know assembly language

- **Chances are, you'll never write programs in assembly**
 - Compilers are much better & more patient than you are
- **But: Understanding assembly is key to machine-level execution model**
 - Behavior of programs in presence of bugs
 - High-level language models break down
 - Tuning program performance
 - Understand optimizations done / not done by the compiler
 - Understanding sources of program inefficiency
 - Implementing system software
 - Compiler has machine code as target
 - Operating systems must manage process state
 - Creating / fighting malware
 - x86 assembly is the language of choice!

Assembly code example

■ Time Stamp Counter

- Special 64-bit register in Intel-compatible machines
- Incremented every clock cycle
- Read with `rdtsc` instruction

■ Application

- Measure time (in clock cycles) required by procedure

```
double t;
start_counter();
P();
t = get_counter();
printf("P required %f clock cycles\n", t);
```

Code to read counter

- Write small amount of assembly code using GCC's `asm` facility
- Inserts assembly code into machine code generated by compiler

```
static unsigned cyc_hi = 0;
static unsigned cyc_lo = 0;

/* Set *hi and *lo to the high and low order bits
   of the cycle counter.

*/
void access_counter(unsigned *hi, unsigned *lo)
{
    asm("rdtsc; movl %%edx,%0; movl %%eax,%1"
        : "=r" (*hi), "=r" (*lo)
        :
        : "%edx", "%eax");
}
```

Great Reality #3: Memory matters

Random access memory is an unphysical abstraction

- **Memory is not unbounded**

- It must be allocated and managed
 - Many applications are memory dominated

- **Memory referencing bugs especially pernicious**

- Effects are distant in both time and space

- **Memory performance is not uniform**

- Cache and virtual memory effects can greatly affect program performance
 - Adapting program to characteristics of memory system can lead to major speed improvements

Memory referencing bug example

```
double fun(int i)
{
    volatile double d[1] = {3.14};
    volatile long int a[2];
    a[i] = 1073741824; /* Possibly out of bounds */
    return d[0];
}
```

fun(0) →	3.14
fun(1) →	3.14
fun(2) →	3.1399998664856
fun(3) →	2.00000061035156
fun(4) →	3.14, then segmentation fault

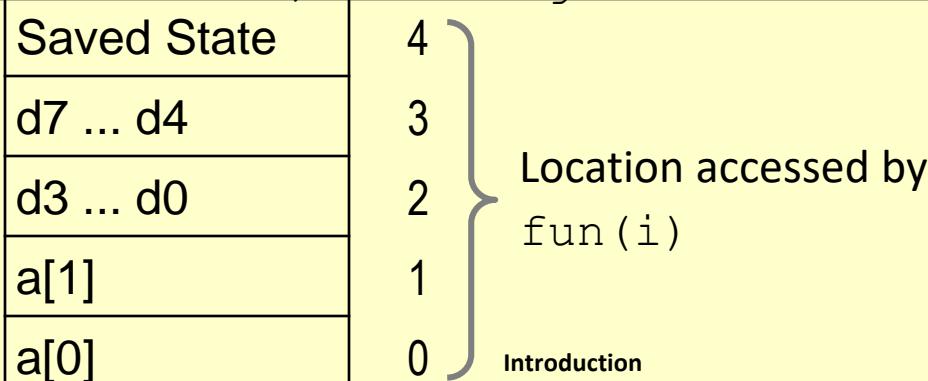
■ Result is architecture specific

Memory referencing bug example

```
double fun(int i)
{
    volatile double d[1] = {3.14};
    volatile long int a[2];
    a[i] = 1073741824; /* Possibly out of bounds */
    return d[0];
}
```

fun(0)	→	3.14
fun(1)	→	3.14
fun(2)	→	3.1399998664856
fun(3)	→	2.00000061035156
fun(4)	→	3.14, then segmentation fault

Explanation:



Memory referencing errors

■ C and C++ do not provide any memory protection

- Out of bounds array references
- Invalid pointer values
- Abuses of `malloc()`/`free()`

■ Can lead to nasty bugs

- Whether or not bug has any effect depends on system and compiler
- Action at a distance
 - Corrupted object logically unrelated to one being accessed
 - Effect of bug may be first observed long after it is generated

■ How can I deal with this?

- Program in Java, Ruby. or ML
- Understand what possible interactions may occur
- Use or develop tools to detect referencing errors (e.g. **Valgrind**)

Great Reality #4:

More to performance than asymptotic complexity

- Constant factors matter too!
- And even exact op count does not predict performance
 - Easily see 10:1 performance range depending on how code written
 - Must optimize at multiple levels: algorithm, data representations, procedures, and loops
- Must understand system to optimize performance
 - How programs compiled and executed
 - How to measure program performance and identify bottlenecks
 - How to improve performance without destroying code modularity and generality

Memory system performance example

```
void copyij(int src[2048][2048],  
           int dst[2048][2048])  
{  
    int i,j;  
    for (i = 0; i < 2048; i++)  
        for (j = 0; j < 2048; j++)  
            dst[i][j] = src[i][j];  
}
```

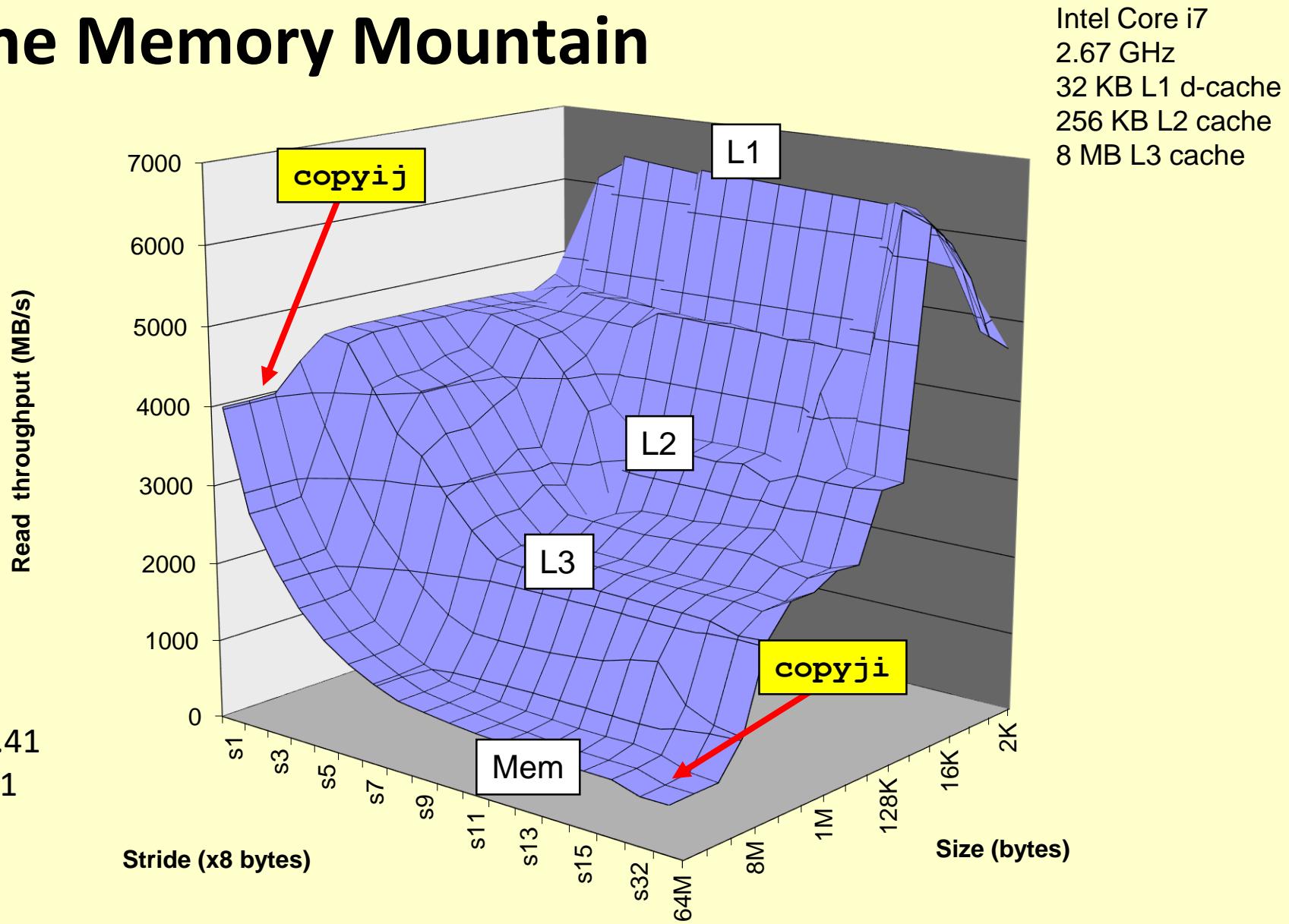


```
void copyji(int src[2048][2048],  
           int dst[2048][2048])  
{  
    int i,j;  
    for (j = 0; j < 2048; j++)  
        for (i = 0; i < 2048; i++)  
            dst[i][j] = src[i][j];  
}
```

21 times slower
(Pentium 4)

- Hierarchical memory organization
- Performance depends on access patterns
 - Including how step through multi-dimensional array

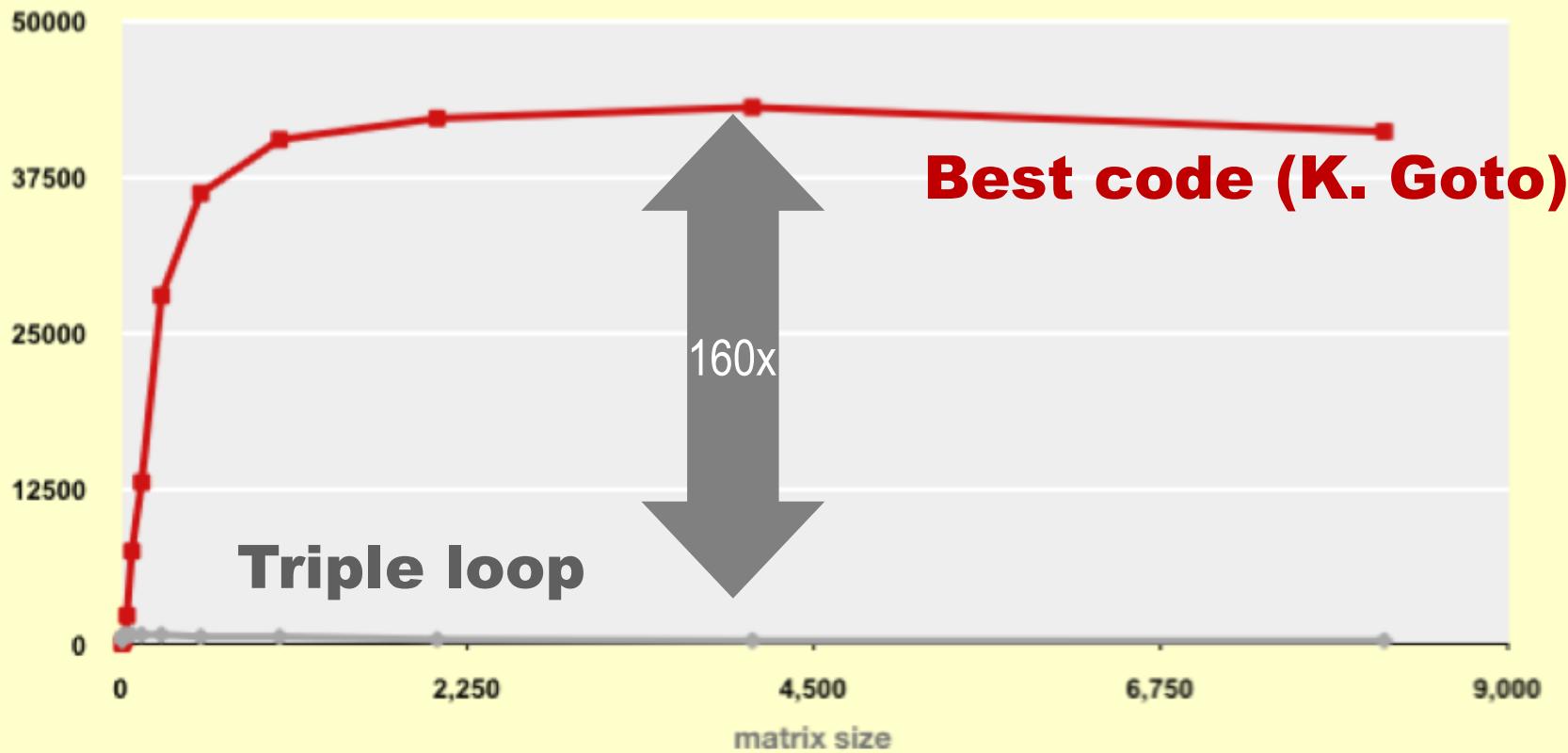
The Memory Mountain



Example Matrix Multiplication

Matrix-Matrix Multiplication (MMM) on 2 x Core 2 Duo 3 GHz (double precision)

Gflop/s

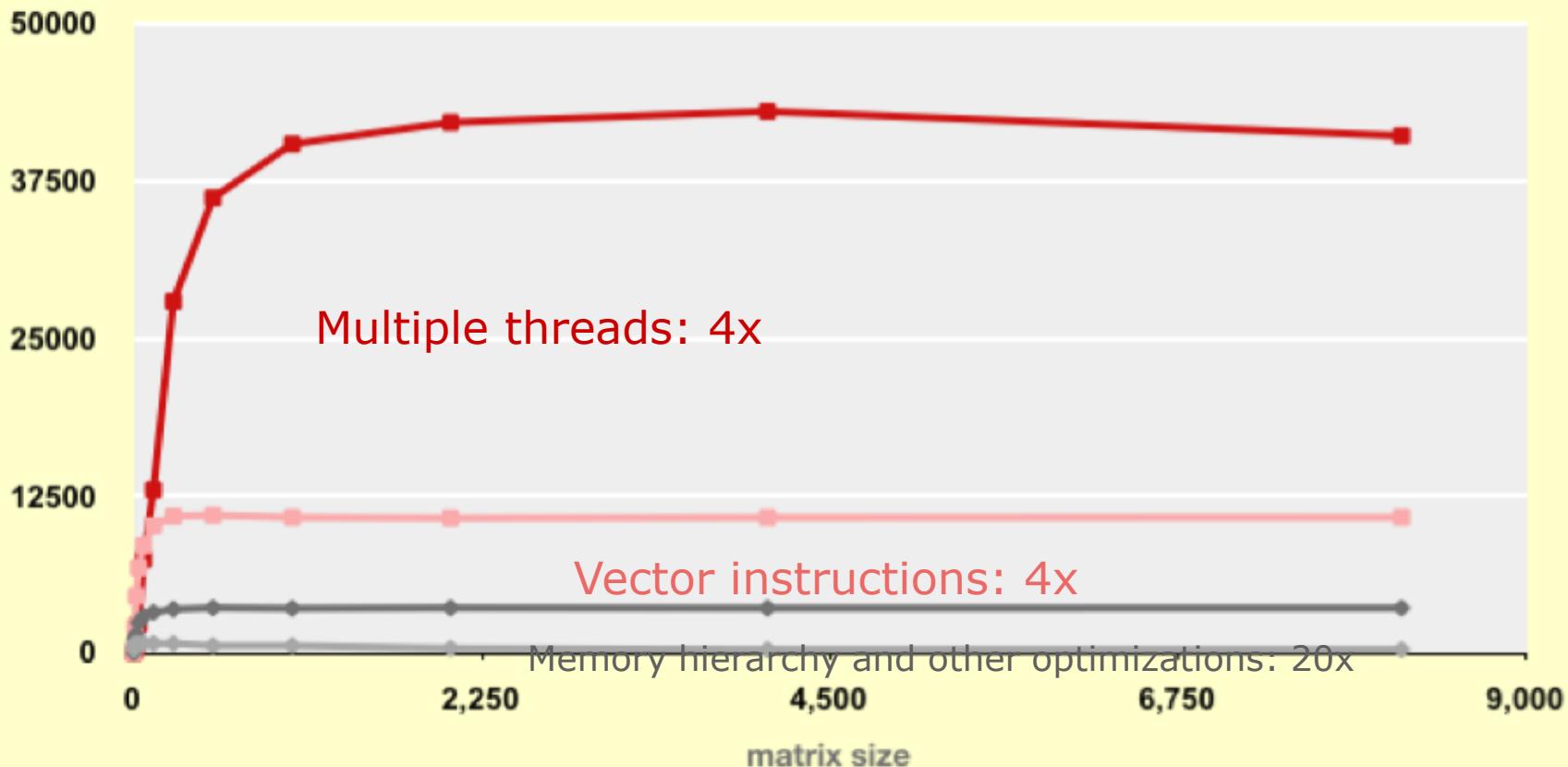


- Standard desktop computer, vendor compiler, using optimization flags
- Both implementations have **exactly the same operations count ($2n^3$)**
- **What is going on?**

MMM Plot: Analysis

Matrix-Matrix Multiplication (MMM) on 2 x Core 2 Duo 3 GHz

Gflop/s



- Reason for 20x: Blocking or tiling, loop unrolling, array scalarization, instruction scheduling, search to find best choice
- ***Effect: fewer register spills, L1/L2 cache misses, and TLB misses***

Great Reality #5:

Computers do more than execute programs

- **They need to get data in and out**
 - I/O system critical to program reliability and performance
- **They communicate with each other over networks**
 - Many system-level issues arise in presence of network
 - Concurrent operations by autonomous processes
 - Coping with unreliable media
 - Cross platform compatibility
 - Complex performance issues

CS-2011 — Machine Organization and Assembly Language

Professor Hugh C. Lauer
CS-2011, Machine Organization and Assembly Language

(Slides include copyright materials from *Computer Systems: A Programmer's Perspective*, by Bryant and O'Hallaron, and from *The C Programming Language*, by Kernighan and Ritchie)

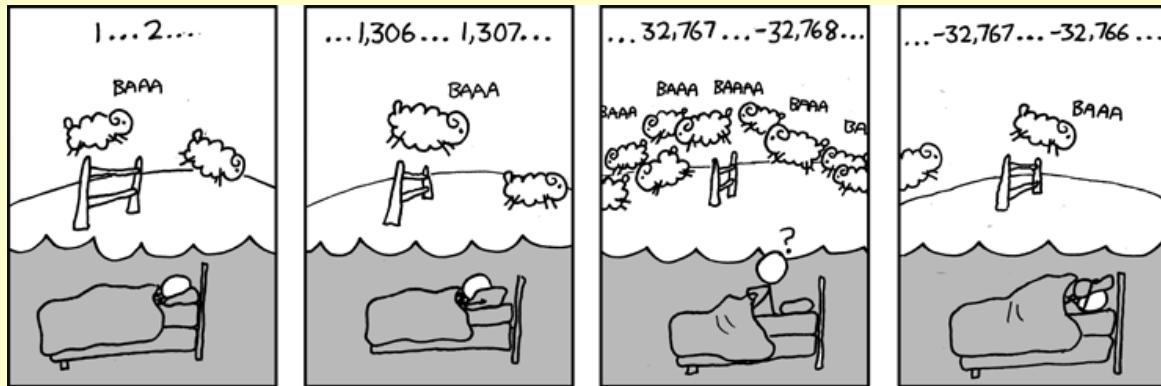
Overview

- A tantalizing problem (or two)
- Course introduction and logistics
- Course Survey and Photos
- Datalab assignment
- Course overview and theme

Two Tantalizing Problems

■ Is $x^2 \geq 0$?

- Float's: Yes!



- Int's:

- $40000 * 40000 \rightarrow 1,600,000,000$
- $50000 * 50000 \rightarrow ??$

-352,516,352

■ Example 2: Is $(x + y) + z = x + (y + z)$?

- Unsigned & Signed Int's: Yes!

- Float's:

- $(1e20 + -1e20) + 3.14 \rightarrow 3.14$
- $1e20 + (-1e20 + 3.14) \rightarrow ??$

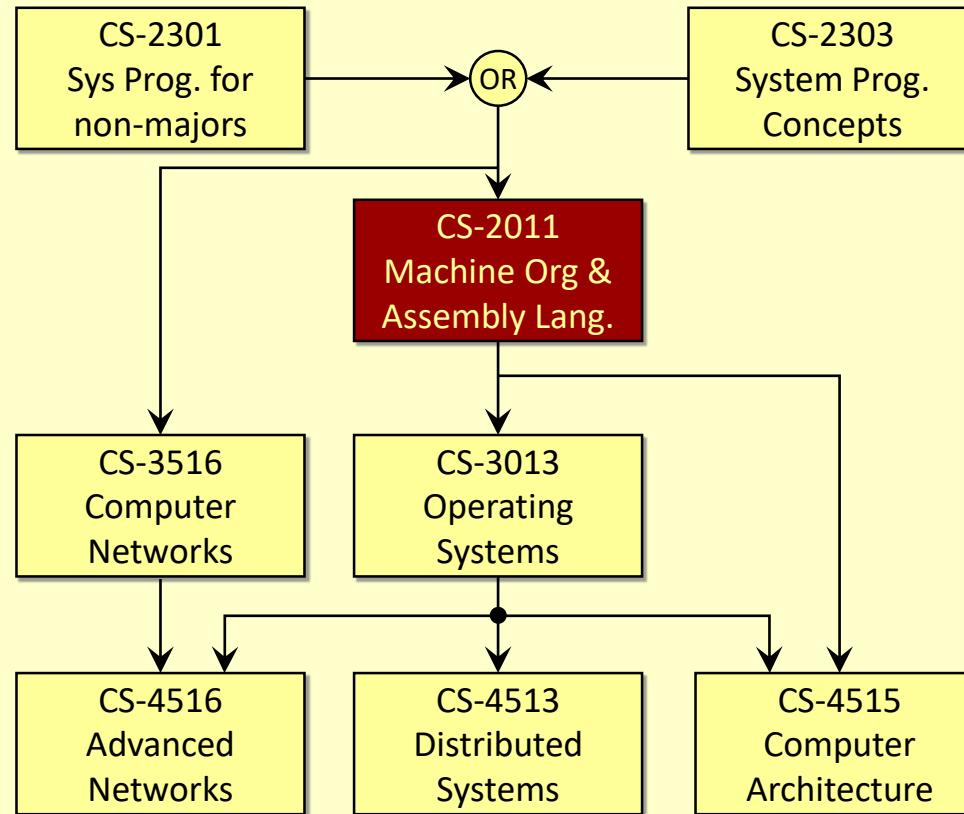
Computer Arithmetic

- **Does not generate random values**
 - Arithmetic operations have important mathematical properties
- **Cannot assume all “usual” mathematical properties**
 - Due to finiteness of representations
 - Integer operations satisfy “ring” properties
 - Commutativity, associativity, distributivity
 - Floating point operations satisfy “ordering” properties
 - Monotonicity, values of signs
- **Observation**
 - Need to understand which abstractions apply in which contexts
 - Important issues for compiler writers and serious application programmers

This course

- **Understand how numbers are represented in modern computers**
 - And the inner workings of arithmetic!
- **Read and work with Intel x86 and x86_64 assembly language**
- **Understand how C compiles to machine code**
- **Understand how modern computers work**
 - At 2000-level course

Role within CS Curriculum



Course perspective

■ Most Systems Courses are Builder-Centric

- Computer Architecture
 - Design pipelined processor in Verilog
- Operating Systems
 - Implement large portions of operating system
- Compilers
 - Write compiler for simple language
- Networking
 - Implement and simulate network protocols

Course perspective (continued)

■ This course is programmer-centric

- Purpose is to show how by knowing more about the underlying system, one can be more effective as a programmer
- Enable you to
 - Write programs that are more reliable and efficient
 - Incorporate features that require hooks into OS
 - E.g., concurrency, signal handlers
- Not just a course for dedicated hackers
 - We bring out the hidden hacker in everyone
- We cover material in this course that you won't see elsewhere

Course components

■ Lectures

- Higher level concepts, applied concepts, clarifications of texts, etc.

■ Labs (i.e., programming) projects — 4

- The heart of the course
- 1-2 weeks each
- Provide in-depth understanding of an aspect of systems
- Programming and measurement

■ Recitation sessions

- Get started on Lab projects
- Work through problems and details interactively

■ Weekly quizzes (5 small + 1 medium)

- Test your understanding of concepts & mathematical principles
- Work out problems from textbook

Lab rationale

All projects/labs
programmed in C

- Each lab has a well-defined goal such as solving a puzzle or winning a contest
- Doing the lab should result in new skills and concepts
- We try to use competition in a fun and healthy way
 - Set a reasonable threshold for full credit
 - Post intermediate results (anonymized) on Web page for glory!

More About This Course

- **This is *not* a traditional course in which**
 - We feed you a bunch of facts and methods
 - We assign you homework and quizzes to demonstrate that you know those facts and methods ...
 - ... and how to apply them!
- **Instead, we assign you problems that you don't know how to solve**
 - It is your responsibility to study them early, and ...
 - ... to figure out what it is that you don't know but need to know, and ...
 - ... to ask questions

More About This Course (continued)

- ... Ask us
 - ... Ask the textbook
 - ... Ask each other
- Share your knowledge
- Raise questions and have discussions at the start of any class
-

More About This Course (continued)

- According to feedback in Course Evaluations:-
- This is the most difficult course that many students have encountered so far at WPI
- This course requires the most work and time of any encountered so far at WPI
- The course is the most fun of any course encountered so far at WPI
- At the end of the course, I hope tell you that ...
 - ... you ain't seen nothin' yet!

Logistics

■ Lectures:-

- Monday, Tuesday, Thursday, Friday, 9:00-10:00 AM
- Salisbury 115
- No classes during Thanksgiving break, November 22-24,
- Also, no class on Tuesday, December 12 (“Reading Day” or makeup day for snowstorms)
- Quizzes:- Fridays November 3 – December 15
- At *start* of class time. Approx 20-25 minutes (except last quiz)
 - You may begin as soon as you arrive in classroom
 - Stay in seat when you are finished

Anyone needing to take quiz at EPC should start early, be back in class at 9:20 AM

■ No make-up quizzes!

- Best four out of six
- See Professor in advance if you must be away — *e.g., interviews, projects, etc.*
- Let me know ASAP if you are sick or injured

Last quiz is mandatory in order to pass course!

Logistics (continued)

- **Recitation sections (called “labs” by Registrar)**
 - 9:00 AM, 10:00 AM (Zoo Lab)
 - 11:00 AM, 12:00 noon (Salisbury 123)
- **Attendance Counts!**
- **Must attend your own, registered session**
 - Space not guaranteed in later sections if you oversleep!
- **Waitlist:-**
 - Please bring an ADD-DROP slip ASAP!

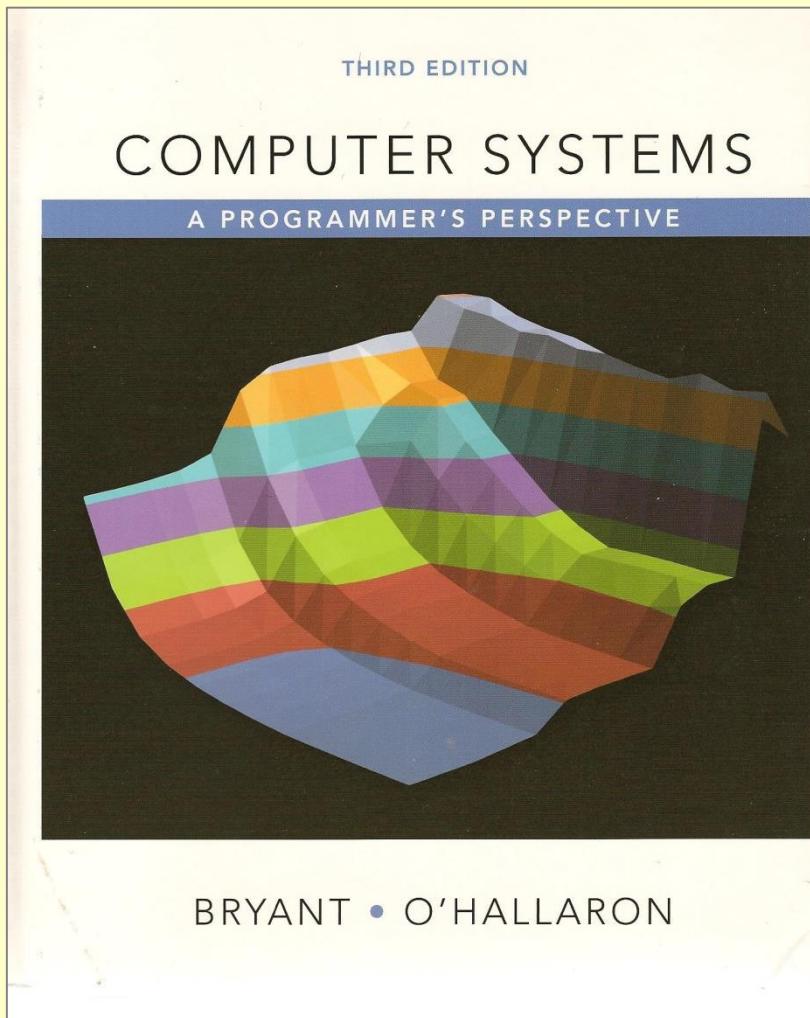
Questions?

Textbooks

- **Randal E. Bryant and David R. O'Hallaron,**
 - “Computer Systems: A Programmer’s Perspective, Third Edition” (CS:APP3e), Prentice Hall, 2016
 - <http://csapp.cs.cmu.edu>
 - This book really matters for the course!
 - How to solve labs
 - Practice problems typical of quiz problems

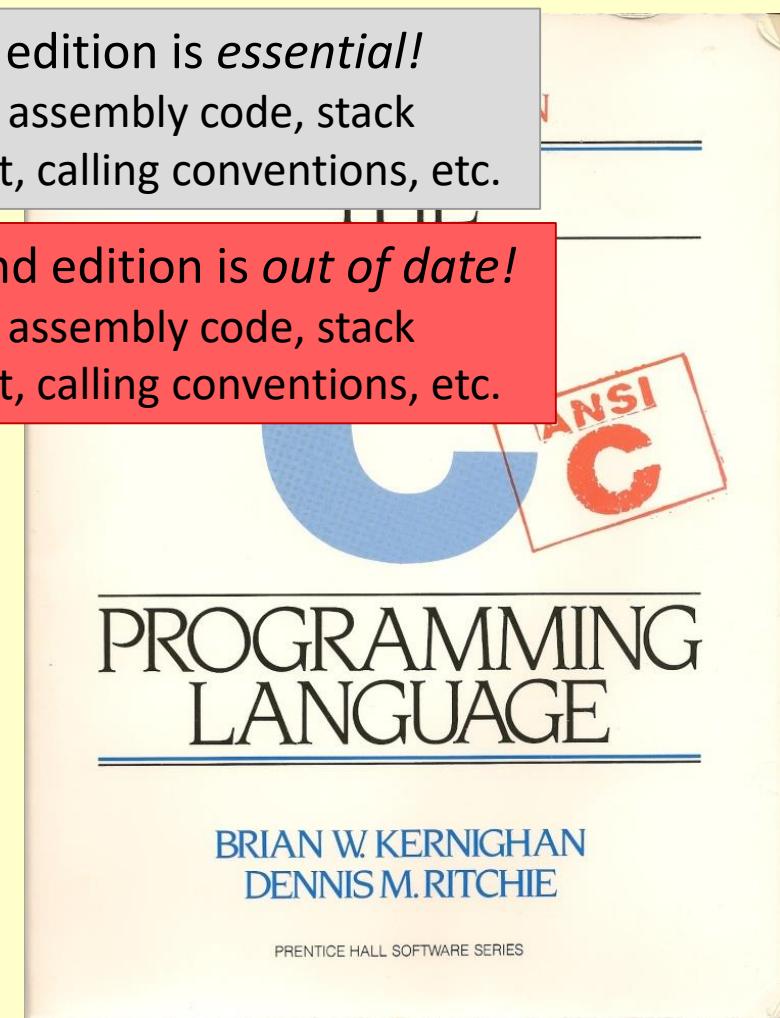
- **Brian Kernighan and Dennis Ritchie,**
 - “The C Programming Language, Second Edition,” Prentice Hall, 1988
 - You should keep a copy of this on your desk for the rest of your (professional) life!

Textbooks (continued)

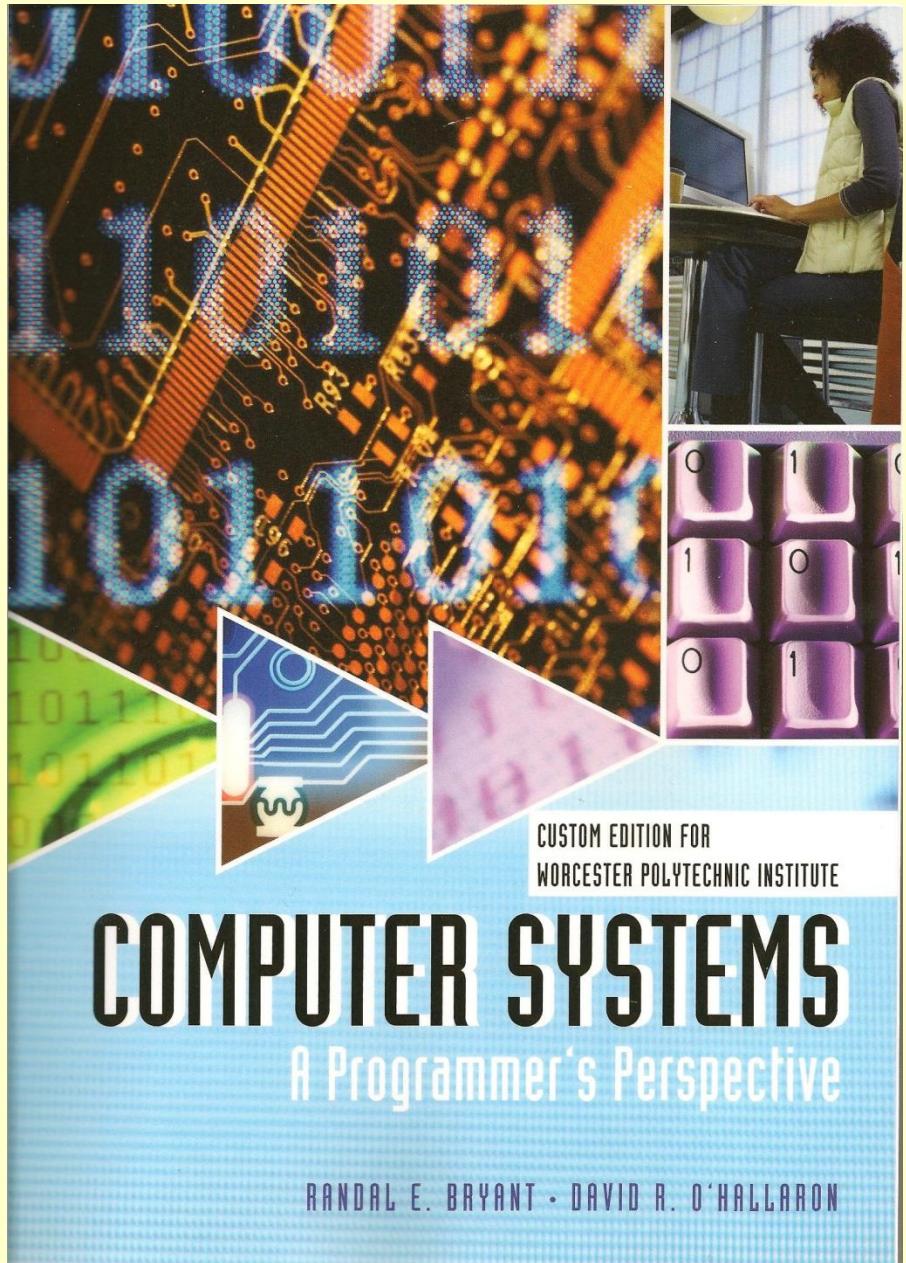


Third edition is *essential!*
64-bit assembly code, stack
format, calling conventions, etc.

Second edition is *out of date!*
32-bit assembly code, stack
format, calling conventions, etc.

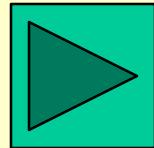


Custom 3rd edition (chapters 1, 2, 3 and 5, 6, 7)



Tomorrow's Recitation Session

Introduction to Datalab

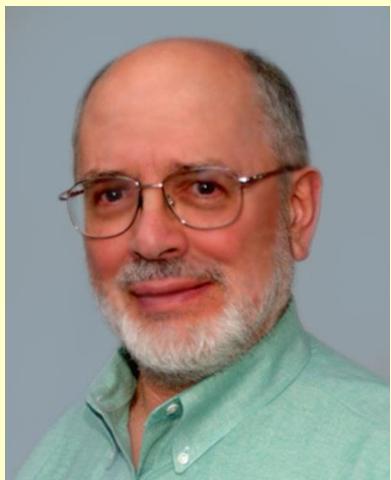


TAs and SAs



Rachel Plante

Teaching staff



Hugh C. Lauer
Adjunct Professor

- **Ph. D. Carnegie-Mellon, 1972-73**
 - Dissertation “Correctness in Operating Systems”
- **Faculty at University of Newcastle upon Tyne, UK**
- **30+ years in industry in USA**
- **WPI since 2006**
- **21 US patents issued**
- **2 seminal contributions to Computer Science**

Reading Assignment — Chapter 1 of Bryant and O'Hallaron

- We will assume full knowledge of this chapter throughout the course
- There may be questions on the contents of this chapter on *any* quiz

Grading

- **40-45% quizzes**
 - Final quiz is worth 2 times any other quiz
- **40-45% Lab projects**
 - Roughly equal in weight
- **10-20% Class and Recitation participation**
 - Helping each other
 - Contributing to discussion groups
 - Asking questions & asking for help
 - It is in your interest that the Professor and TAs know you by name
 - Please don't be embarrassed/annoyed if we ask you often

Getting Help

■ Class Web Page: <http://www.cs.wpi.edu/~cs2011/b17>

- Complete schedule of lectures, exams, and assignments
- Copies of lectures, assignments, exams, solutions
- Clarifications to assignments

■ Canvas

- Repository for copyright materials (lecture notes)
- Linked directly from course web page
- Submissions (backup for course servers)

■ Lecture Capture

- Lectures will be recorded using WPI's Echo Lecture Capturing system.
- Link on Canvas home page for this course

Getting Help

- Course mailing list: cs2011-all@cs.wpi.edu
 - Use as discussion list
- Staff mailing list: cs2011-staff@cs.wpi.edu
 - Use this for ALL communication with the teaching staff
 - Send email to individual instructors or TAs only to schedule appointments
- Office hours (Prof. Lauer):
 - Monday, TBD
 - Tuesday, TBD
 - Thursday, TBD
 - Friday, TBD
- 1:1 Appointments
 - You can schedule 1:1 appointments with Professor or any of the TAs

Ground Rule #1

- **There are no “stupid” questions.**
- **It is a waste of your time and the class’s time to proceed when you don’t understand the basic terms.**
- **If you don’t understand it, someone else probably doesn’t it, either.**

Ground Rule #2

- Help each other!
- Even when a project or assignment is specified as *individual*, ask your friends or classmates about stuff you don't understand.
- It is a waste of your time try to figure out some obscure detail on your own when there are lots of resources around.
- When you have the answer, *write it in your own words* (or own coding style).

Names and Faces

- **It is *in your own interest* that I know who you are.**
 - Class picture — will circulate next time
- **Students who speak up in class usually get more favorable grades than those who don't**
- **When speaking in class, please identify yourselves**

Policies: Assignments (Labs) And Exams

■ No work groups or project teams

- You must work alone on all assignments (unless otherwise specified)

■ Canvas

- Assignments due at 6:00 pm on the due date
- Electronic turn-ins using *Canvas* (unless otherwise specified)

■ Conflict exams, other irreducible conflicts

- Make PRIOR arrangements with Prof. Lauer
- Notifying us well ahead of time shows maturity and makes us like you more (and thus to work harder to help you out of *your* problem)

■ Appealing grades

- Within 7 days of completion of grading
- Labs: Email to the staff mailing list
- Quizzes: Talk to Prof. Lauer

Facilities

- Lab projects to be completed on 64-bit Ubuntu virtual machines
- gcc for compilation
gdb for debugging
Eclipse or Code::Blocks for GUI
 - DDD no longer supported!
- You may use any other platform
 - On your own!
 - Projects graded on Ubuntu virtual machines

Some students have used
nemiver as a gui debugger

Timeliness

- Lateness penalties
 - 5% per hour (first two hours)
 - 10% per hour after that
- Catastrophic events
 - Major illness, death in family, ...
 - Formulate a plan (with your professor *and* academic advisor) to get back on track
- Advice
 - Once you start running late, it's really hard to catch up

WPI Honesty Policy

■ What is cheating?

- Sharing code: by copying, pasting, retyping, looking at, or supplying a *file*
- Coaching: helping your friend to write a program, *line by line*
- Copying code from previous course or from elsewhere on WWW
 - Only allowed to use code we supply, or from CS:APP website

■ What is NOT cheating?

- Explaining how to use systems or tools
- Helping others with abstract design issues
- Working out a problem together on a whiteboard, etc.
 - Diagrams of data structures — *not code*
 - Discussing outlines of algorithms *in English, Chinese, etc!*

■ ...

WPI Honesty Policy (continued)

- ...
- **It is a *violation* of the WPI Academic Honesty Policy to submit someone else's work as your own.**
- **It is *not a violation* of WPI's Academic Honesty Policy to ask for help!**
 - Classmates, TAs, friends, mentors, ...
 - Explanations of things you don't understand

Unauthorized hacking

- Any hacking into server will be treated as equivalent to *Academic Dishonesty*
 - Reported accordingly to Dean of Students Office.
- If you suspect a vulnerability in *any system*, come and get permission to investigate *BEFORE* actually trying to probe.

Note on Quizzes

- **Open book, open notes**
- **Many students have electronic textbooks**
 - Kindle
 - Tablet
 - Laptop
- **Okay to consult electronic textbook, electronic copies of notes, lecture slides, etc.**
- **May *NOT* connect to network, web, etc.**

Topics for this course

- Programs and Data
- The Memory Hierarchy
- Performance
- ~~Exceptions Control Flow~~
- ~~Virtual Memory~~
- ~~Networking and Concurrency~~

Questions?

Floating Point

Professor Hugh C. Lauer
CS-2011, Machine Organization and Assembly Language

(Slides include copyright materials from *Computer Systems: A Programmer's Perspective*, by Bryant and O'Hallaron, and from *The C Programming Language*, by Kernighan and Ritchie)

Today: Floating Point

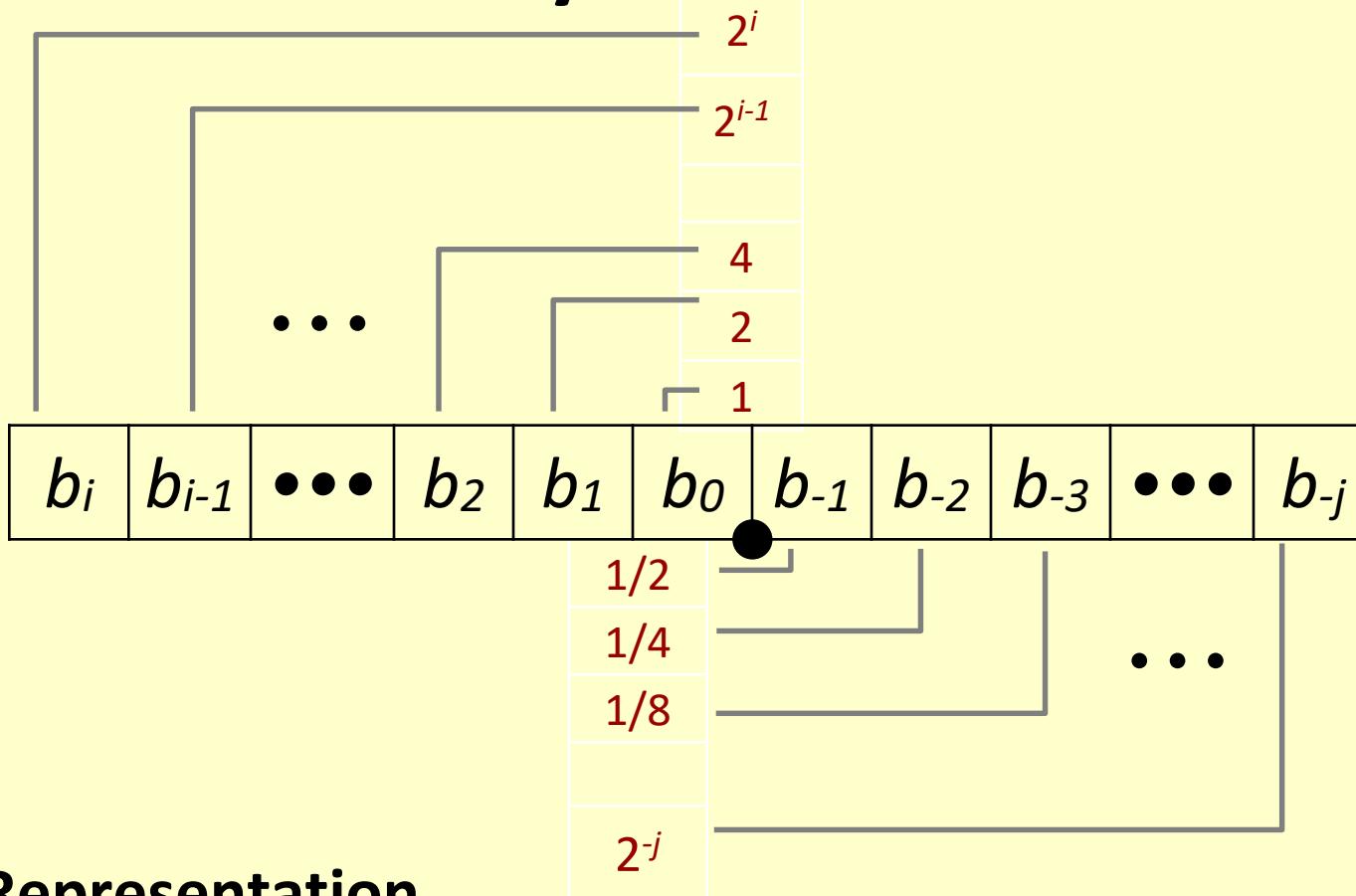
Reading Assignment: §2.4

- **Background: Fractional binary numbers**
- **IEEE floating point standard: Definition**
- **Example and properties**
- **Rounding, addition, multiplication**
- **Floating point in C**
- **Summary**

Fractional binary numbers

- What is 1011.101_2 ?

Fractional binary numbers



■ Representation

- Bits to right of “binary point” represent fractional powers of 2
- Represents rational number: $\sum_{k=-j}^i b_k \times 2^k$

Fractional binary numbers: examples

■ Value	Representation
5 3/4	101.11 ₂
2 7/8	010.111 ₂
1 7/16	001.0111 ₂

■ Observations

- Divide by 2 by shifting right
- Multiply by 2 by shifting left
- Numbers of form 0.111111...₂ are just below 1.0
 - $1/2 + 1/4 + 1/8 + \dots + 1/2^i + \dots \rightarrow 1.0$
 - Use notation $1.0 - \varepsilon$

Representable numbers

■ Limitation #1

- Can only exactly represent numbers of the form $x/2^k$
- Other rational numbers have repeating bit representations

Value	Representation
▪ $1/3$	$0.0101010101[01]..._2$
▪ $1/5$	$0.001100110011[0011]..._2$
▪ $1/10$	$0.0001100110011[0011]..._2$

■ Limitation #2

- Many, many bits needed for very large or small numbers with fixed binary point
 - Planck's constant:– 6.626068×10^{-34} erg sec
 - Avogadro's number:– 6.022×10^{23} particles per mole

Today: Floating Point

- **Background: Fractional binary numbers**
- **IEEE floating point standard: Definition**
- **Example and properties**
- **Rounding, addition, multiplication**
- **Floating point in C**
- **Summary**

Floating point

- A way to *approximate* real numbers in computers
 - And also most rational numbers
- Examples:-
 - 3.14159265358979323846 — π
 - 2.99792458×10^8 m/s — c , the velocity of light
 - $6.62606885 \times 10^{-27}$ erg sec — h , Planck's constant
- In C (and most other programming languages):-
 - 3.14159265358979323846
 - 2.99792458e8
 - 6.62606885e-27

IEEE floating point

■ IEEE Standard 754

- Established in 1985 as uniform standard for floating point arithmetic
 - Before that, many idiosyncratic formats
 - Same equations gave different answers on different machines!
- Now supported by all major processors

■ Driven by numerical concerns

- Nice standards for rounding, overflow, underflow
- Difficult to make fast in hardware
 - Numerical analysts predominated over hardware designers in defining standard

Floating point representation

■ Numerical Form:

$$(-1)^s \times M \times 2^E$$

- **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

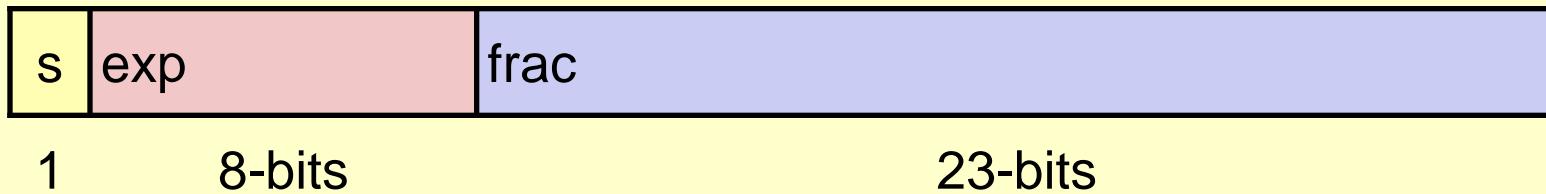
■ Encoding

- MSB *s* is sign bit *s*
- *exp* field encodes *E* (but is not equal to E)
- *frac* field encodes *M* (but is not equal to M)

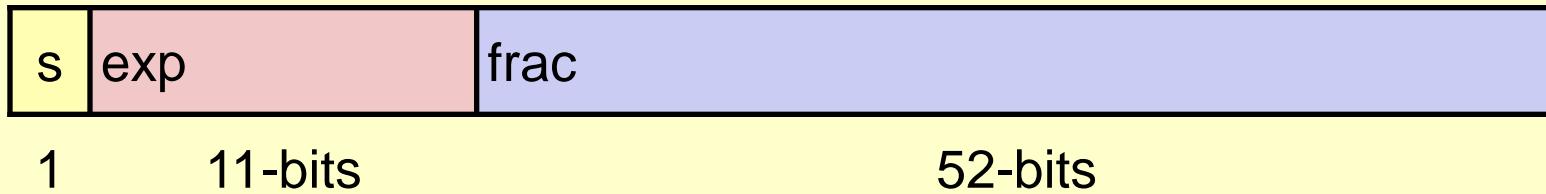


Precisions

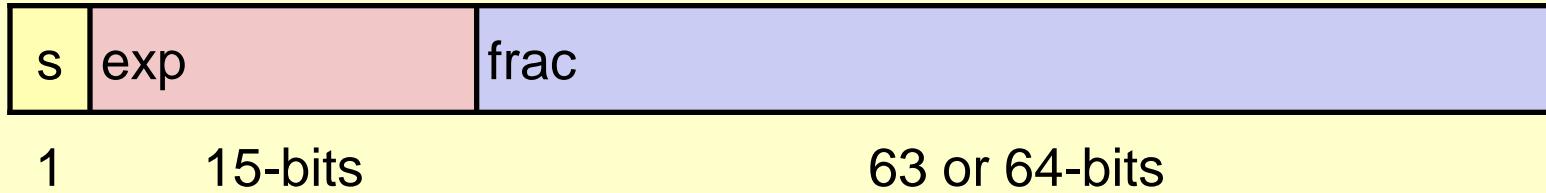
■ Single precision: 32 bits



■ Double precision: 64 bits



■ Extended precision: 80 bits (Intel only)



Normalized values

$$v = (-1)^s M 2^E$$

- Condition: $\text{exp} \neq 000\dots0$ and $\text{exp} \neq 111\dots1$
- Exponent coded as *biased* value: $E = \text{Exp} - \text{Bias}$
 - Exp : unsigned value exp
 - $\text{Bias} = 2^{k-1} - 1$, where k is number of exponent bits
 - Single precision: 127 ($\text{Exp}: 1\dots254$, $E: -126\dots127$)
 - Double precision: 1023 ($\text{Exp}: 1\dots2046$, $E: -1022\dots1023$)
- Significand coded with implied leading 1: $M = 1.\text{xxx}\dots\text{x}_2$
 - $\text{xxx}\dots\text{x}$: bits of frac
 - Minimum when $000\dots0$ ($M = 1.0$)
 - Maximum when $111\dots1$ ($M = 2.0 - \epsilon$)
 - Get extra leading bit for “free”

Normalized encoding example

$$v = (-1)^s M 2^E$$

$$E = Exp - Bias$$

- Value: `Float F = 15213.0;`
 - $15213_{10} = 11101101101101_2$
 $= 1.1101101101101_2 \times 2^{13}$

- Significand

$$M = 1.\underline{1101101101101}_2$$

$$\text{frac} = \underline{1101101101101}0000000000_2$$

- Exponent

$$E = 13$$

$$Bias = 127$$

$$Exp = 140 = 10001100_2$$

- Result:

0	10001100	1101101101101000000000000
s	exp	frac

Denormalized values

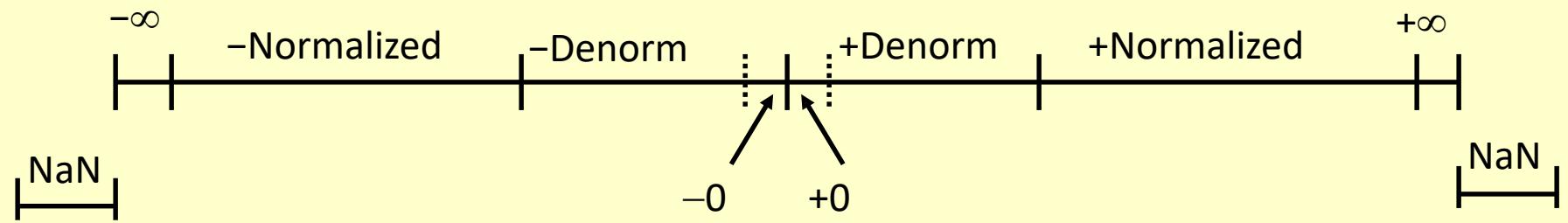
$$v = (-1)^s M 2^E$$
$$E = 1 - \text{Bias}$$

- Condition: $\text{exp} = 000\dots0$
- Exponent value: $E = -\text{Bias} + 1$ (instead of $E = 0 - \text{Bias}$)
- Significand coded with implied leading 0: $M = 0.\text{xxx}\dots\text{x}_2$
 - $\text{xxx}\dots\text{x}$: bits of **frac**
- Cases
 - $\text{exp} = 000\dots0$, **frac** = 000...0
 - Represents zero value
 - Note distinct values: +0 and -0 (why?)
 - $\text{exp} = 000\dots0$, **frac** \neq 000...0
 - Numbers very close to 0.0
 - Lose precision as get smaller
 - Equispaced

Special values

- Condition: **exp = 111...1**
- Case: **exp = 111...1, frac = 000...0**
 - Represents value ∞ (infinity)
 - Operation that overflows
 - Both positive and negative
 - E.g., $1.0/0.0 = -1.0/-0.0 = +\infty$, $1.0/-0.0 = -\infty$
- Case: **exp = 111...1, frac \neq 000...0**
 - Not-a-Number (NaN)
 - Represents case when no numeric value can be determined
 - E.g., $\sqrt{-1}$, $\infty - \infty$, $\infty \times 0$

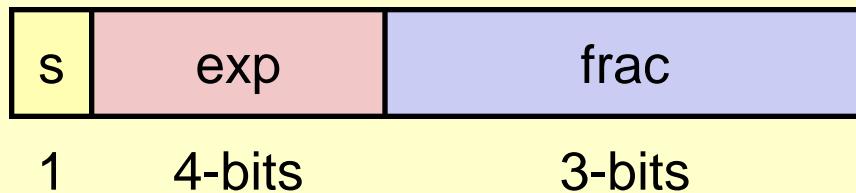
Visualization: floating point encodings



Today: Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- Rounding, addition, multiplication
- Floating point in C
- Summary

Tiny floating point example



■ 8-bit Floating Point Representation

- the sign bit is in the most significant bit
- the next four bits are the exponent, with a bias of 7
- the last three bits are the **frac**

■ Same general form as IEEE Format

- normalized, denormalized
- representation of 0, NaN, infinity

Dynamic range (positive only)

$$v = (-1)^s M 2^E$$

$$n: E = \text{Exp} - \text{Bias}$$

$$d: E = 1 - \text{Bias}$$

	s	exp	frac	E	Value	
Denormalized numbers	0	0000	000	-6	0	
	0	0000	001	-6	$1/8 * 1/64 = 1/512$	closest to zero
	0	0000	010	-6	$2/8 * 1/64 = 2/512$	
	...					
	0	0000	110	-6	$6/8 * 1/64 = 6/512$	
	0	0000	111	-6	$7/8 * 1/64 = 7/512$	largest denorm
	0	0001	000	-6	$8/8 * 1/64 = 8/512$	smallest norm
Normalized numbers	0	0001	001	-6	$9/8 * 1/64 = 9/512$	
	...					
	0	0110	110	-1	$14/8 * 1/2 = 14/16$	
	0	0110	111	-1	$15/8 * 1/2 = 15/16$	closest to 1 below
	0	0111	000	0	$8/8 * 1 = 1$	
	0	0111	001	0	$9/8 * 1 = 9/8$	closest to 1 above
	0	0111	010	0	$10/8 * 1 = 10/8$	
	...					
	0	1110	110	7	$14/8 * 128 = 224$	
	0	1110	111	7	$15/8 * 128 = 240$	largest norm
	0	1111	000	n/a	inf	

Dynamic range (positive only)

$$v = (-1)^s M 2^E$$

$$n: E = Exp - Bias$$

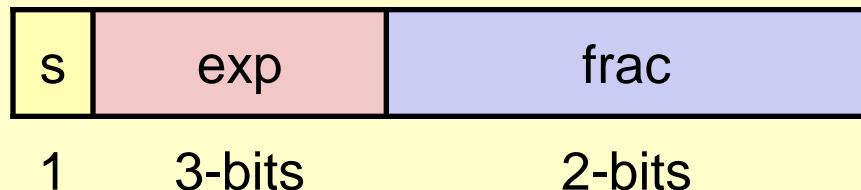
$$d: E = 1 - Bias$$

	s	exp	frac	E	Value	
Denormalized numbers	0	0000	000	-6	0	
	0	0000	001	-6	$1/8 * 1/64 = 1/512$	closest to zero
	0	0000	010	-6	$2/8 * 1/64 = 2/512$	
	...					
	0	0000	110	-6	$6/8 * 1/64 = 6/512$	
	0	0000	111	-6	$7/8 * 1/64 = 7/512$	largest denorm
	0	0001	000	-6	$8/8 * 1/64 = 8/512$	smallest norm
Normalized numbers	0	0001	001	-6	$9/8 * 1/64 = 9/512$	
	...					
	0	0110	110	-1	$14/8 * 1/2 = 14/16$	
	0	0110	111	-1	$15/8 * 1/2 = 15/16$	closest to 1 below
	0	0111	000	0	$8/8 * 1 = 1$	
	0	0111	001	0	$9/8 * 1 = 9/8$	closest to 1 above
	0	0111	010	0	$10/8 * 1 = 10/8$	
Note: the value 1 has exponent = bias and significand = all zeros					$4/8 * 128 = 224$ $5/8 * 128 = 240$	largest norm
	0	1111	000	n/a	inf	

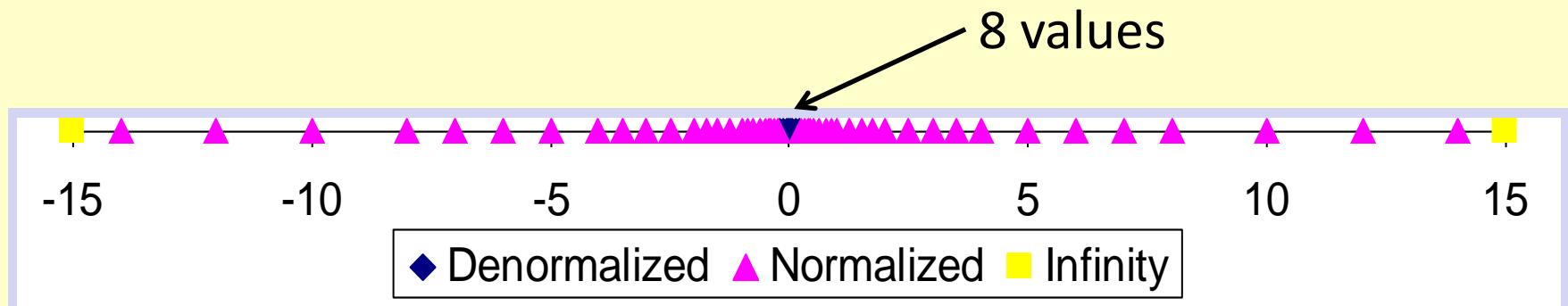
Distribution of values

■ 6-bit IEEE-like format

- $e = 3$ exponent bits
- $f = 2$ fraction bits
- Bias is $2^{(3-1)} - 1 = 3$



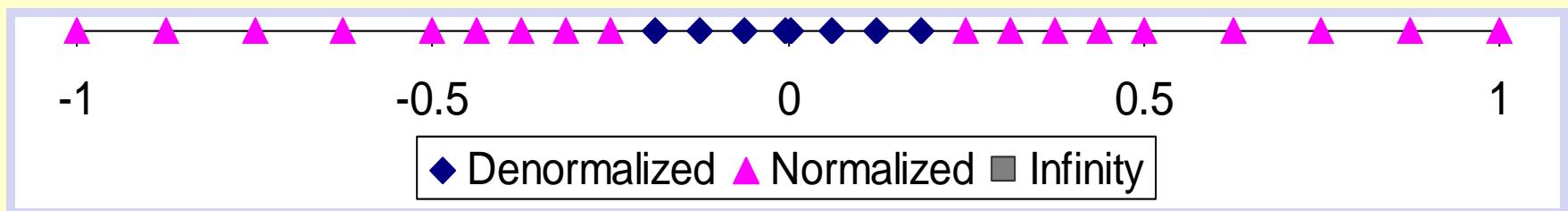
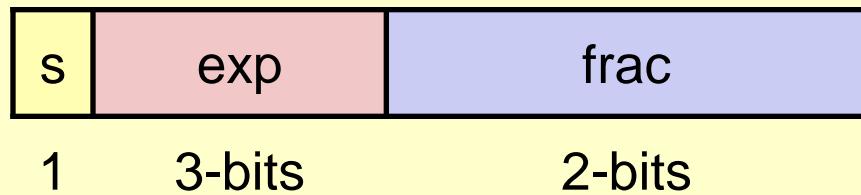
■ Notice how the distribution gets denser toward zero.



Distribution of values (close-up view)

■ 6-bit IEEE-like format

- $e = 3$ exponent bits
- $f = 2$ fraction bits
- Bias is 3



Interesting numbers

{single, double}

<i>Description</i>	<i>exp</i>	<i>frac</i>	<i>Numeric Value</i>
■ Zero	00...00	00...00	0.0
■ Smallest Pos. Denorm.	00...00	00...01	$2^{-\{23,52\}} \times 2^{-\{126,1022\}}$
■ Single $\approx 1.4 \times 10^{-45}$			
■ Double $\approx 4.9 \times 10^{-324}$			
■ Largest Denormalized	00...00	11...11	$(1.0 - \varepsilon) \times 2^{-\{126,1022\}}$
■ Single $\approx 1.18 \times 10^{-38}$			
■ Double $\approx 2.2 \times 10^{-308}$			
■ Smallest Pos. Normalized	00...01	00...00	$1.0 \times 2^{-\{126,1022\}}$
■ Just larger than largest denormalized			
■ One	01...11	00...00	1.0
■ Largest Normalized	11...10	11...11	$(2.0 - \varepsilon) \times 2^{\{127,1023\}}$
■ Single $\approx 3.4 \times 10^{38}$			
■ Double $\approx 1.8 \times 10^{308}$			

Special properties of encoding

- FP zero same as integer zero
 - All bits = 0
- Can (almost) use unsigned integer comparison
 - Must first compare sign bits
 - Must consider $-0 = 0$
 - NaNs problematic
 - Will be greater than any other values
 - What should comparison yield?
 - Otherwise OK
 - Denorm vs. normalized
 - Normalized vs. infinity

Today: Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- Rounding, addition, multiplication
- Floating point in C
- Summary

Floating point operations: Basic idea

■ Basic idea

- First **compute exact result**
- Make it fit into desired precision
 - Possibly overflow if exponent too large
 - Possibly **round to fit into `frac`**

■ $x +_f y = \text{Round}(x + y)$

■ $x \times_f y = \text{Round}(x \times y)$

Rounding

■ Rounding Modes (illustrate with \$ rounding)

	\$1.40	\$1.60	\$1.50	\$2.50	-\$1.50
■ Towards zero	\$1	\$1	\$1	\$2	-\$1
■ Round down ($-\infty$)	\$1	\$1	\$1	\$2	-\$2
■ Round up ($+\infty$)	\$2	\$2	\$2	\$3	-\$1
■ Nearest Even (default)	\$1	\$2	\$2	\$2	-\$2

■ What are the advantages of the modes?

Closer look at round-to-even

■ Default Rounding Mode

- Hard to get any other kind without dropping into assembly
- All others are statistically biased
 - Sum of set of positive numbers will consistently be over- or under-estimated

■ Applying to Other Decimal Places / Bit Positions

- When exactly halfway between two possible values
 - Round so that least significant digit is even
- E.g., round to nearest hundredth

1.2349999 1.23 (Less than half way)

1.2350001 1.24 (Greater than half way)

1.2350000 1.24 (Half way—round up)

1.2450000 1.24 (Half way—round down)

Rounding binary numbers

■ Binary Fractional Numbers

- “Even” when least significant bit is 0
- “Half way” when bits to right of rounding position = 100...₂

■ Examples

- Round to nearest 1/4 (2 bits right of binary point)

Value	Binary	Rounded	Action	Rounded Value
2 3/32	10.000 11 ₂	10.00 ₂	(<1/2—down)	2
2 3/16	10.00 110 ₂	10.01 ₂	(>1/2—up)	2 1/4
2 7/8	10.11 100 ₂	11.00 ₂	(1/2—up)	3
2 5/8	10.10 100 ₂	10.10 ₂	(1/2—down)	2 1/2

FP multiplication

- $(-1)^{s_1} M_1 2^{E_1} \times (-1)^{s_2} M_2 2^{E_2}$

- Exact Result: $(-1)^s M 2^E$

- Sign s : $s_1 \wedge s_2$
- Significand M : $M_1 \times M_2$
- Exponent E : $E_1 + E_2$

- Fixing

- If $M \geq 2$, shift M right, increment E
- If E out of range, overflow
- Round M to fit frac precision

- Implementation

- Biggest chore is multiplying significands

Floating point addition

- $(-1)^{s_1} M_1 2^{E_1} + (-1)^{s_2} M_2 2^{E_2}$

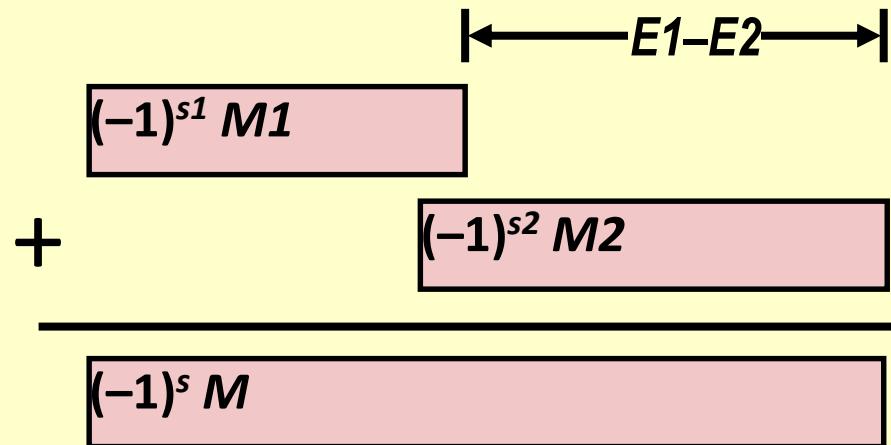
- Assume $E_1 > E_2$

- Exact Result: $(-1)^s M 2^E$

- Sign s , significand M :
 - Result of signed align & add
- Exponent E : E_1

- Fixing

- If $M \geq 2$, shift M right, increment E
- if $M < 1$, shift M left k positions, decrement E by k
- Overflow if E out of range
- Round M to fit **frac** precision



Mathematical properties of FP add

■ Compare to those of Abelian Group

- Closed under addition?
 - But may generate infinity or NaN
- Commutative?
Yes
- Associative?
No
 - Overflow and inexactness of rounding
 - $(3.14+1e10)-1e10 = 0, 3.14+(1e10-1e10) = 3.14$
- 0 is additive identity?
Yes
- Every element has additive inverse
 - Except for infinities & NaNs

■ Monotonicity

- $a \geq b \Rightarrow a+c \geq b+c?$
 - Except for infinities & NaNs

Mathematical Properties of FP Mult

■ Compare to Commutative Ring

- Closed under multiplication?
 - But may generate infinity or NaN
- Multiplication Commutative?
- Multiplication is Associative?
 - Possibility of overflow, inexactness of rounding
 - Ex: $(1e20 * 1e20) * 1e-20 = \inf$, $1e20 * (1e20 * 1e-20) = 1e20$
- 1 is multiplicative identity?
- Multiplication distributes over addition?
 - Possibility of overflow, inexactness of rounding

Yes

Yes

No

Yes

No

■ Monotonicity

- $a \geq b \ \& \ c \geq 0 \Rightarrow a * c \geq b * c?$
 - Except for infinities & NaNs

Almost

Today: Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- Rounding, addition, multiplication
- **Floating point in C**
- Summary

Floating point in C

■ C guarantees two levels

- **float** single precision
- **double** double precision

■ Conversions/casting

- Casting between **int**, **float**, and **double** changes bit representations
- **double/float → int**
 - Truncate fractional part — i.e., rounding toward zero
 - Not defined when out-of-range, **NaN**, etc.; — generally set to *TMin*
- **int → double**
 - Exact conversion for numbers that fit into ≤ 53 bits
- **int → float**
 - Round according to rounding mode

Floating Point Puzzles

■ For each of the following C expressions, either:

- Argue that it is true for all argument values
- Explain why not true

```
int x = ...;
float f = ...;
double d = ...;
```

Assume neither
d nor **f** is NaN

- $x == (\text{int})(\text{float}) \ x$
- $x == (\text{int})(\text{double}) \ x$
- $f == (\text{float})(\text{double}) \ f$
- $d == (\text{float}) \ d$
- $f == -(-f);$
- $2/3 == 2/3.0$
- $d < 0.0 \Rightarrow ((d^2) < 0.0)$
- $d > f \Rightarrow -f > -d$
- $d * d >= 0.0$
- $(d+f)-d == f$

Today: Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- Rounding, addition, multiplication
- Floating point in C
- Summary

Summary

- IEEE Floating Point has clear mathematical properties
- Represents numbers of form $M \times 2^E$
- One can reason about operations independent of implementation
 - As if computed with perfect precision and then rounded
- Not the same as real arithmetic
 - Violates associativity/distributivity
 - Makes life difficult for compilers & serious numerical applications programmers

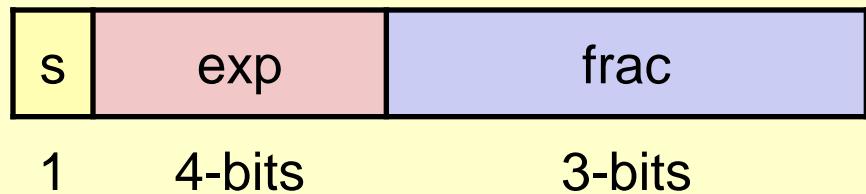
Questions?

More Slides

Creating Floating Point Number

■ Steps

- Normalize to have leading 1
- Round to fit within fraction
- Postnormalize to deal with effects of rounding



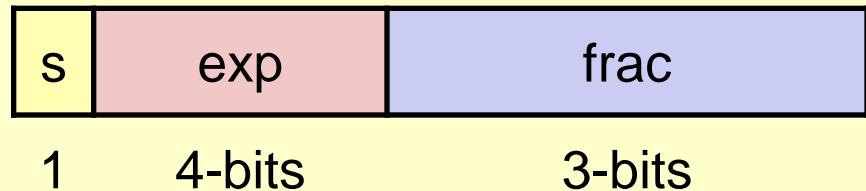
■ Case Study

- Convert 8-bit unsigned numbers to tiny floating point format

Example Numbers

128	10000000
15	00001101
33	00010001
35	00010011
138	10001010
63	00111111

Normalize



■ Requirement

- Set binary point so that numbers of form 1.xxxxxx
- Adjust all to have leading one
 - Decrement exponent as shift left

<i>Value</i>	<i>Binary</i>	<i>Fraction</i>	<i>Exponent</i>
128	10000000	1.0000000	7
15	00001101	1.1010000	3
17	00010001	1.0001000	4
19	00010011	1.0011000	4
138	10001010	1.0001010	7
63	00111111	1.1111100	5

Rounding

1.BBGRXXX

Guard bit: LSB of result

Round bit: 1st bit removed

Sticky bit: OR of remaining bits

■ Round up conditions

- Round = 1, Sticky = 1 → > 0.5
- Guard = 1, Round = 1, Sticky = 0 → Round to even

Value	Fraction	GRS	Incr?	Rounded
128	1.0000000	000	N	1.000
15	1.1010000	100	N	1.101
17	1.0001000	010	N	1.000
19	1.0011000	110	Y	1.010
138	1.0001010	011	Y	1.001
63	1.1111100	111	Y	10.000

Postnormalize

■ Issue

- Rounding may have caused overflow
- Handle by shifting right once & incrementing exponent

<i>Value</i>	<i>Rounded</i>	<i>Exp</i>	<i>Adjusted</i>	<i>Result</i>
128	1.000	7		128
15	1.101	3		15
17	1.000	4		16
19	1.010	4		20
138	1.001	7		134
63	10.000	5	1.000/6	64

Questions?

Machine-Level Programming I: Basics

Professor Hugh C. Lauer
CS-2011, Machine Organization and Assembly Language

(Slides include copyright materials from *Computer Systems: A Programmer's Perspective*, by Bryant and O'Hallaron, and from *The C Programming Language*, by Kernighan and Ritchie)

Today: Machine Programming I: Basics

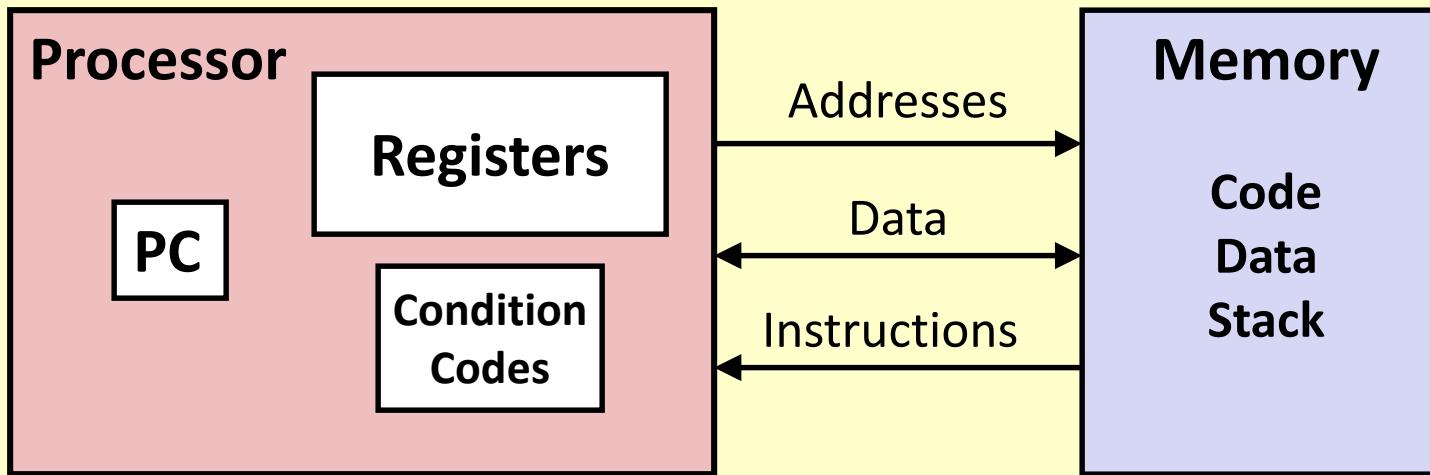
- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Arithmetic & logical operations

Deferred to end
of this topic

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

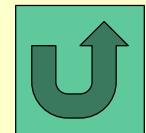
Assembly/Machine Code View



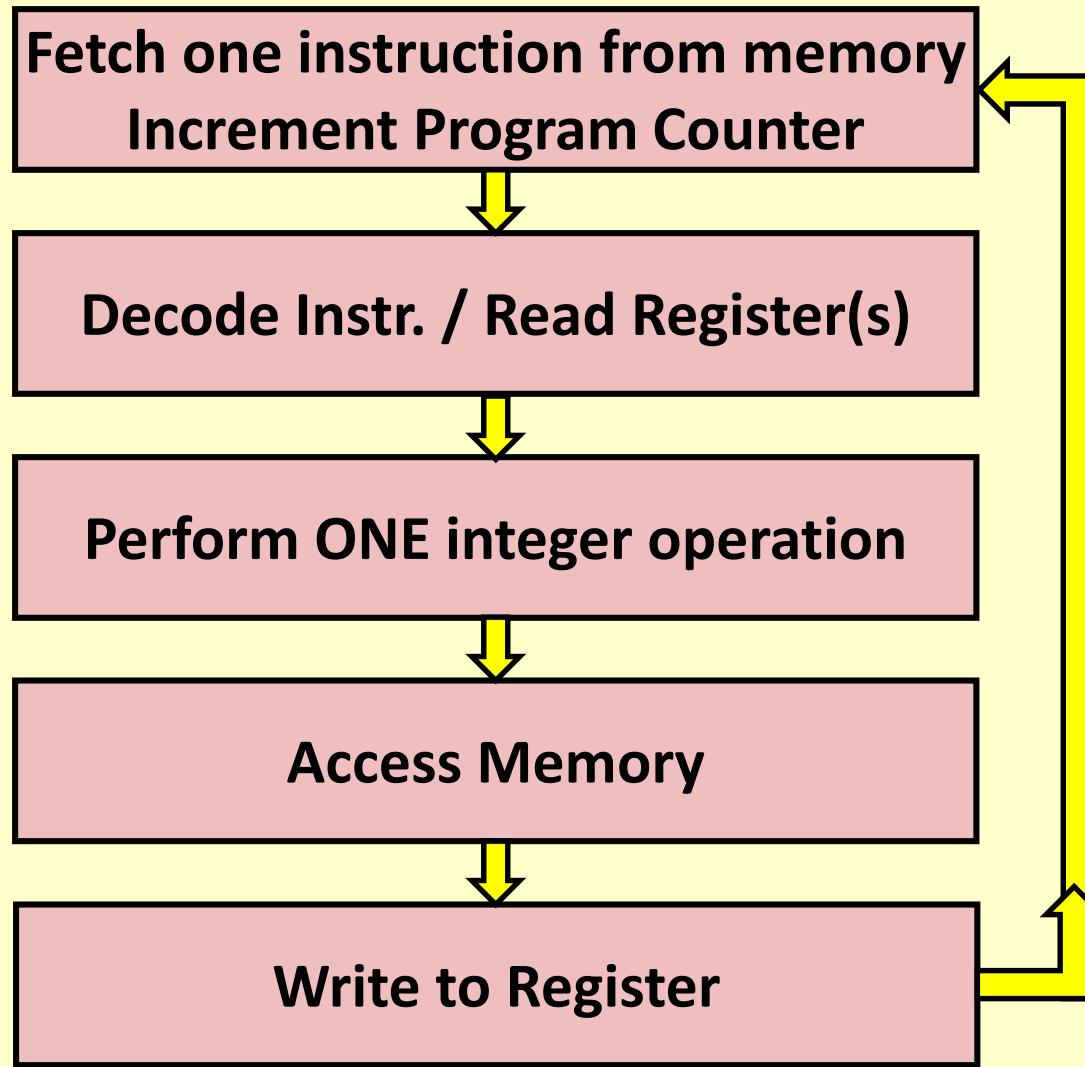
Programmer-Visible State

- **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 (aka *functions*)

CPU:– Archaic term for “Processor”
— “Central Processing Unit”



Execution Model for Modern Processors



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    // save on stack
    movq    %rdx, %rbx
    call    plus
    movq    %rax, (%rbx)
    popq   %rbx    // restore from
    ret     // stack
```

Obtain (on course virtual machine) with command

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

Produces file sum.s

Warning: May get very different results on other systems (CCC Linux, Mac OS-X, ...) due to different versions of *gcc* and different compiler settings.



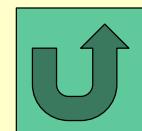
Assembly Characteristics: Data Types

- “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

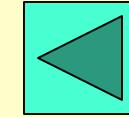
- Transfer data between memory and register
 - Load data from memory into register
 - Store register data into memory
- Perform arithmetic function on register or memory data
- Transfer control
 - Unconditional jumps to/from procedures
 - Conditional branches

Another archaic term:–
“procedure” in the
textbook (and these slides)
means “function” in
modern terminology!



Object Code

Code for `sumstore`



0x0400595:

0x53

0x48

0x89

0xd3

0xe8

0xf2

0xff

0xff

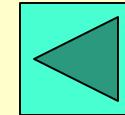
- Total of 14 bytes
- Each instruction 1, 3, or 5 bytes
- Starts at address 0x0400595

■ Assembler

- Translates .s into .o
- Binary encoding of each instruction
- Nearly-complete image of executable code
- Missing linkages between code in different files

■ Linker

- Resolves references between files
- Combines with static run-time libraries
 - E.g., code for `malloc`, `printf`
- Some libraries are *dynamically linked*
 - Linking occurs when program begins execution



Machine Instruction Example

```
*dest = t;
```

■ C Code

- Store value **t** to memory designated by **dest**

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

■ Assembly

- Move 8-byte value to memory
 - Quad words in x86-64 parlance
- Operands:

t: Register **%rax**

dest: Register **%rbx**

***dest:** Memory **M[%rbx]**

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

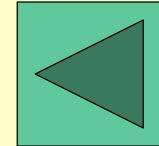
■ Object Code

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

Disassembling Object Code

Disassembled

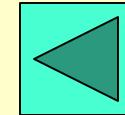
```
0000000000400595 <sumstore>:  
 400595: 53          push    %rbx  
 400596: 48 89 d3    mov      %rdx,%rbx  
 400599: e8 f2 ff ff ff  callq   400590 <plus>  
 40059e: 48 89 03    mov      %rax,(%rbx)  
 4005a1: 5b          pop     %rbx  
 4005a2: c3          retq
```



■ Disassembler

`objdump -d sum`

- Useful tool for examining object code
- Analyzes bit pattern of series of instructions
- Produces approximate rendition of assembly code
- Can dump either `a.out` file (complete executable) or `.o` file (single module)



Alternate Disassembly

Object

0x0400595:

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

0xff
0xff
0x48
0x89
0x03
0x5b
0xc3

Disassembled

Dump of assembler code for function **sumstore**:

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

■ Within gdb Debugger

gdb sum

disassemble sumstore

■ Disassemble procedure

x/14xb sumstore

■ Examine the 14 bytes starting at **sumstore**

What Can be Disassembled?

```
% objdump -d WINWORD.EXE

WINWORD.EXE:      file format pei-i386

No symbols in "WINWORD.EXE".
Disassembly of section .text:

30001000 <.text>:
30001000:
30001001:
30001003:
30001005:
3000100a:
```

Reverse engineering forbidden by
Microsoft End User License Agreement

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

Today: Machine Programming I: Basics

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

x86-64 Integer Registers

Reading Assignment: §3.4

%rax	%eax
------	------

%r8	%r8d
-----	------

%rbx	%ebx
------	------

%r9	%r9d
-----	------

%rcx	%ecx
------	------

%r10	%r10d
------	-------

%rdx	%edx
------	------

%r11	%r11d
------	-------

%rsi	%esi
------	------

%r12	%r12d
------	-------

%rdi	%edi
------	------

%r13	%r13d
------	-------

%rsp	%esp
------	------

%r14	%r14d
------	-------

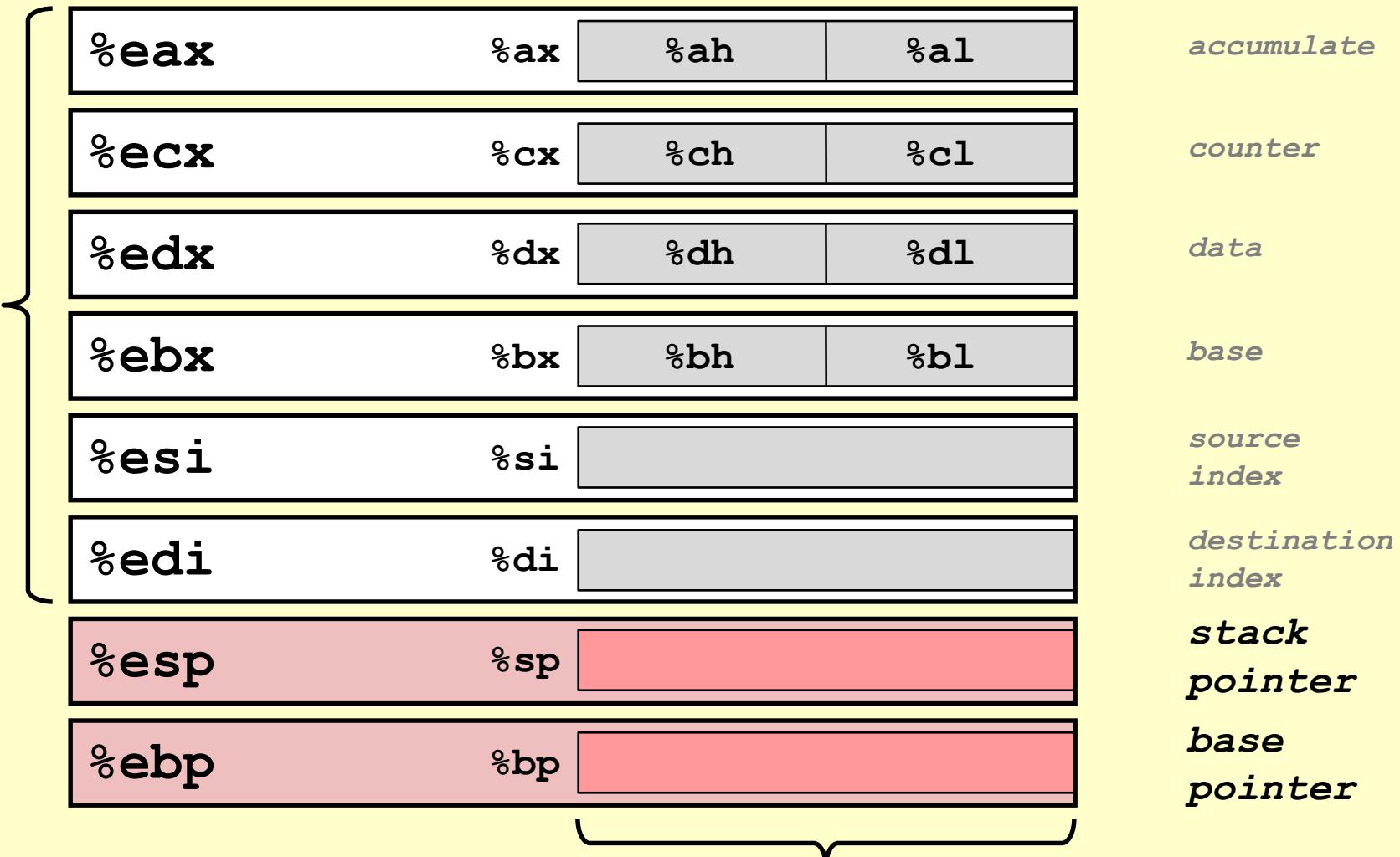
%rbp	%ebp
------	------

%r15	%r15d
------	-------

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

Some History: IA32 Registers

general purpose



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`

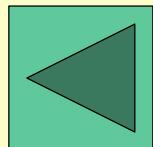
`%rsp`

`%rbp`

`%rN`

movq Operand Combinations

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



Cannot do memory-memory transfer with a single instruction

Simple Memory Addressing Modes

■ Normal (R) Mem[Reg[R]]

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

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

■ Displacement D(R) Mem[Reg[R]+D]

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

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

Example of Simple Addressing Modes

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

swap:

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

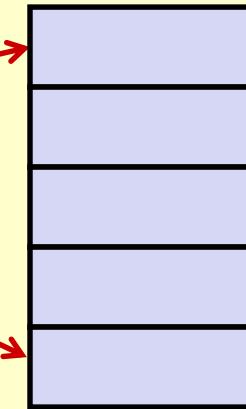
Understanding Swap()

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

Memory

Registers

%rdi	
%rsi	
%rax	
%rdx	



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	

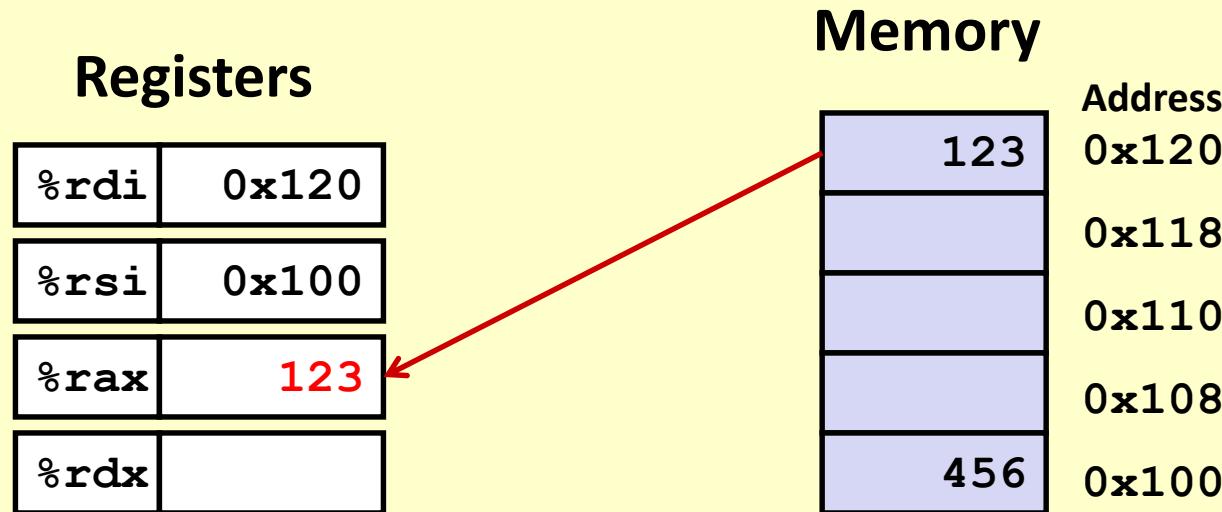
Memory

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

swap:

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

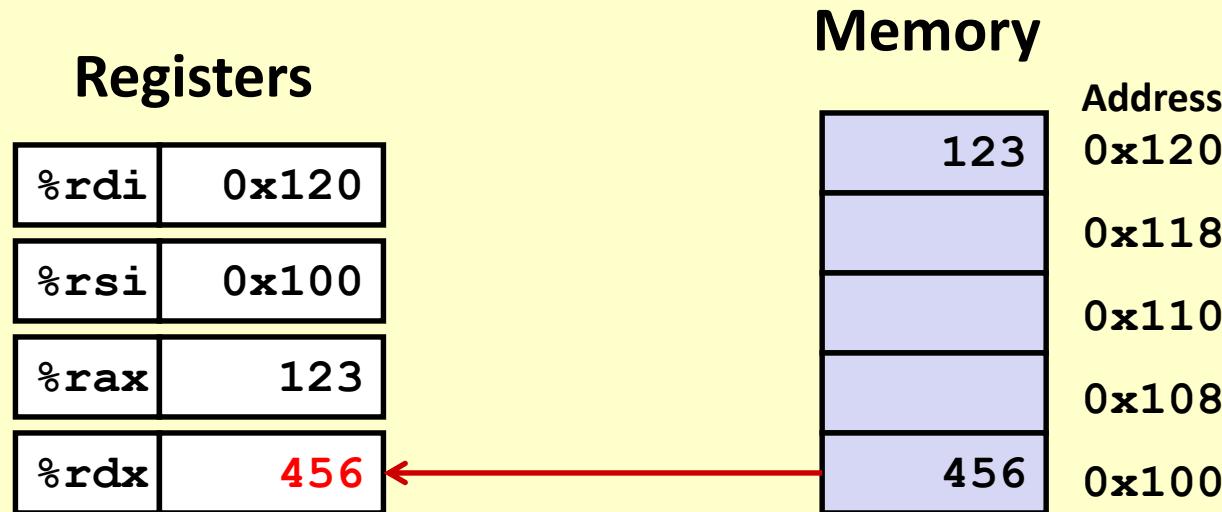
Understanding Swap()



swap:

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

Understanding Swap()

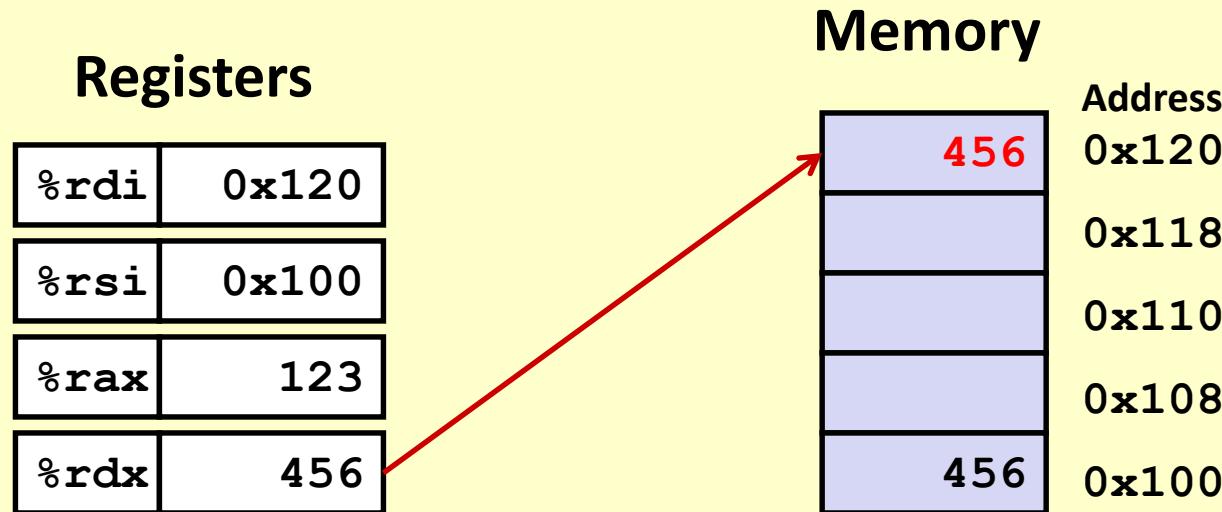


swap:

```

    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()

Registers

%rdi	0x120
%rsi	0x100
%rax	123
%rdx	456

Memory

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



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

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

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

Address Computation Examples

%rdx	0xf000
%rcx	0x0100

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

Address Computation Examples

%rdx	0xf000
%rcx	0x0100

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

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	
0x80(,%rdx,2)	2*0xf000 + 0x80	

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	

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

Today: Machine Programming I: Basics

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

Address Computation Instruction

Reading Assignment: §3.5

■ **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 $x + k^*y$
 - $k = 1, 2, 4, \text{ or } 8$

```
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:

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

■ 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;
}
```

Where is the constant “48” in machine code?

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

Today: Machine Programming I: Basics

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

Intel x86 Processors

Reading Assignment: §3.1

- Dominate laptop/desktop/server market
- Evolutionary design
 - Backwards compatible up until 8086, introduced in 1978
 - Added more features as time goes on
- Complex instruction set computer (CISC)
 - Many different instructions with many different formats
 - But, only small subset encountered with Linux programs
 - Hard to match performance of Reduced Instruction Set Computers (RISC)
 - But, Intel has done just that!
 - In terms of speed. Less so for low power.

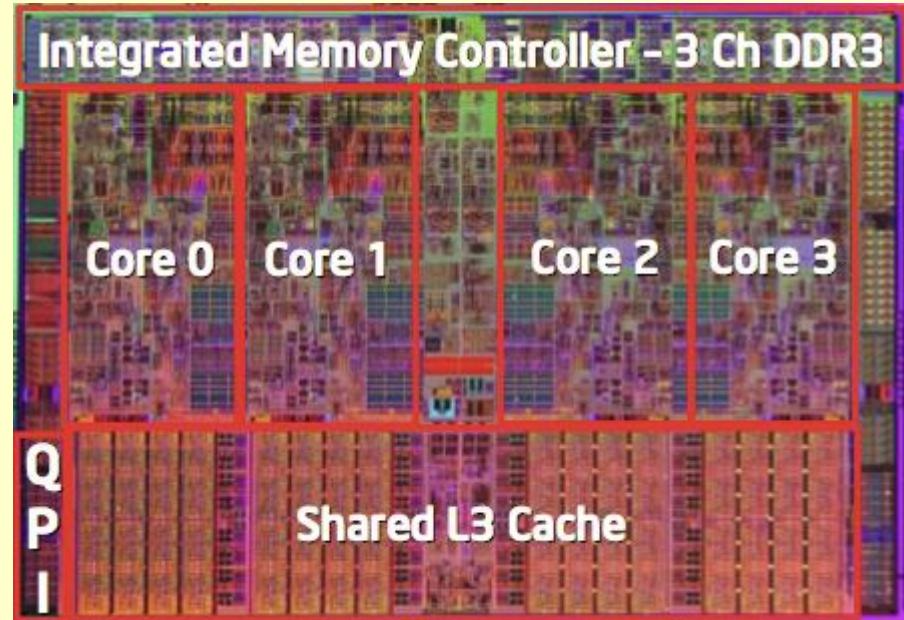
Intel x86 Evolution: Milestones

<i>Name</i>	<i>Date</i>	<i>Transistors</i>	<i>MHz</i>
■ 8086	1978	29K	5-10
		■ First 16-bit Intel processor. Basis for IBM PC & DOS	
		■ 1MB address space	
■ 386	1985	275K	16-33
		■ First 32 bit Intel processor , referred to as IA32	
		■ Added “flat addressing”, capable of running Unix	
■ Pentium 4E	2004	125M	2800-3800
		■ First 64-bit Intel x86 processor, referred to as x86-64	
■ Core 2	2006	291M	1060-3500
		■ First multi-core Intel processor	
■ Core i7	2008	731M	1700-3900
		■ Four cores (most modern desktop PCs)	

Intel x86 Processors, cont.

■ Machine Evolution

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



■ Added Features

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

2015 State of the Art

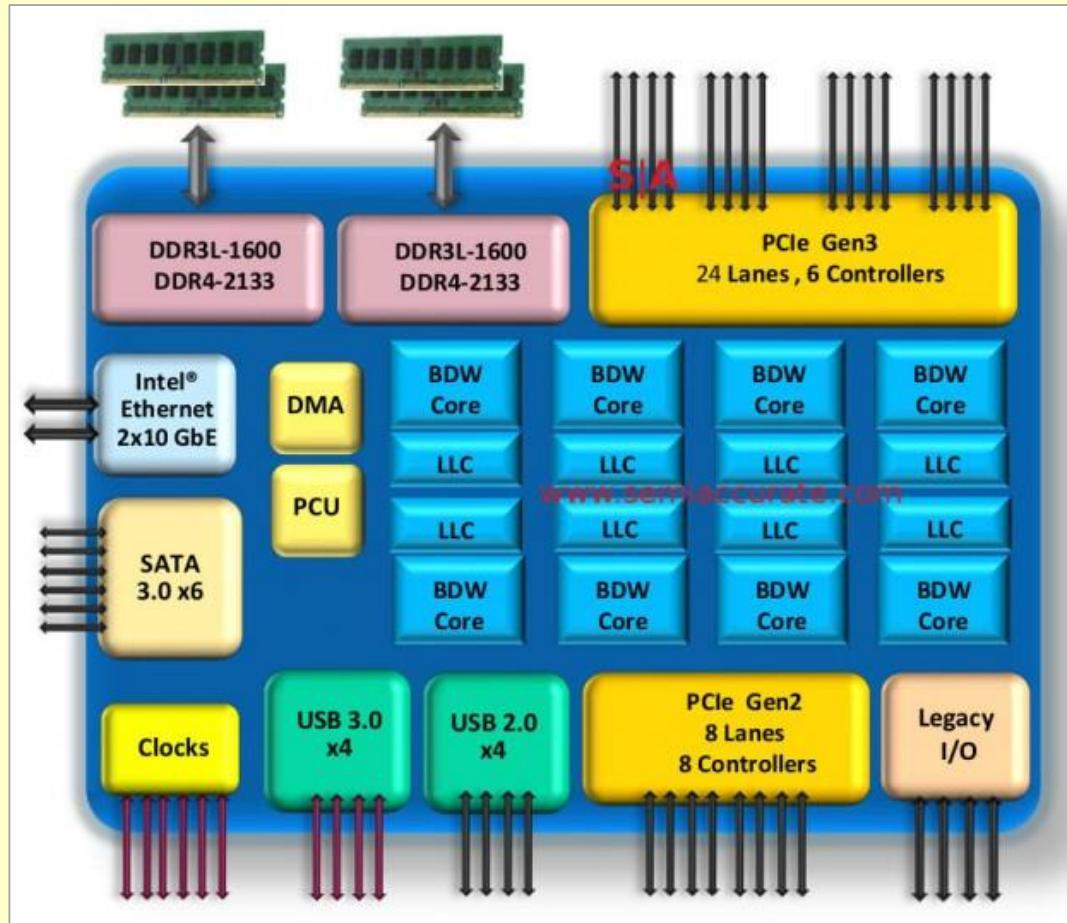
- Core i7 Broadwell 2015

■ Desktop Model

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

■ Server Model

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



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

x86 Clones: Advanced Micro Devices (AMD)

■ Historically

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

■ Then

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

■ Recent Years

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

Intel's 64-Bit History

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

Our Coverage

■ IA32

- The traditional x86
- For CS-2011: RIP, D-Term 2016

■ x86-64

- The standard
- CS-2011> `gcc hello.c`
- CS-2011> `gcc -m64 hello.c //same code!`

■ Presentation

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

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

Questions?

Machine-Level Programming II: Control

Professor Hugh C. Lauer
CS-2011, Machine Organization and Assembly Language

(Slides include copyright materials from *Computer Systems: A Programmer's Perspective*, 3rd edition, by Bryant and O'Hallaron, and from *The C Programming Language*, by Kernighan and Ritchie)

Today

Reading Assignment: §3.6

- 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>%r8</code>
<code>%rbx</code>	<code>%r9</code>
<code>%rcx</code>	<code>%r10</code>
<code>%rdx</code>	<code>%r11</code>
<code>%rsi</code>	<code>%r12</code>
<code>%rdi</code>	<code>%r13</code>
<code>%rsp</code>	<code>%r14</code>
<code>%rbp</code>	<code>%r15</code>

`%rip`

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)

■ An “implicit” side effect of (most) 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)
 - **SF set** if $(a-b) < 0$ (as signed)
 - **ZF set** if $a == b$
 - **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
sete	ZF	Equal / Zero
setne	$\sim ZF$	Not Equal / Not Zero
sets	SF	Negative
setns	$\sim SF$	Nonnegative
setg	$\sim (SF^OF) \ \& \ \sim ZF$	Greater (Signed)
setge	$\sim (SF^OF)$	Greater or Equal (Signed)
setl	(SF^OF)	Less (Signed)
setle	$(SF^OF) \ \ ZF$	Less or Equal (Signed)
seta	$\sim CF \ \& \ \sim ZF$	Above (unsigned)
setb	CF	Below (unsigned)

x86-64 Integer Registers

%rax	%al	%r8	%r8b
%rbx	%bl	%r9	%r9b
%rcx	%cl	%r10	%r10b
%rdx	%dl	%r11	%r11b
%rsi	%sil	%r12	%r12b
%rdi	%dil	%r13	%r13b
%rsp	%spl	%r14	%r14b
%rbp	%bp1	%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 0

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

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

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

Today

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

Jumping

■ jX Instructions

- Jump to different part of code depending on condition codes

jX	Condition	Description
jmp	1	Unconditional
je	ZF	Equal / Zero
jne	$\sim ZF$	Not Equal / Not Zero
js	SF	Negative
jns	$\sim SF$	Nonnegative
jg	$\sim (SF \wedge OF) \ \& \ \sim ZF$	Greater (Signed)
jge	$\sim (SF \wedge OF)$	Greater or Equal (Signed)
jl	$(SF \wedge OF)$	Less (Signed)
jle	$(SF \wedge OF) \mid ZF$	Less or Equal (Signed)
ja	$\sim CF \ \& \ \sim ZF$	Above (unsigned)
jb	CF	Below (unsigned)

Conditional Branch Example (Old Style)

■ Generation

Linux> 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

Today

- 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 x
%rax	result

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

General “Do-While” Translation

C Code

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

Goto Version

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

■ Body:{

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

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

“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

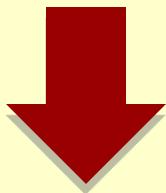
```
{  
    unsigned bit =  
        (x >> i) & 0x1;  
    result += bit;  
}
```

“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

Goto Version

```
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;
}
```

- Initial test can be optimized away

```
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;
}
```

Today

- 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

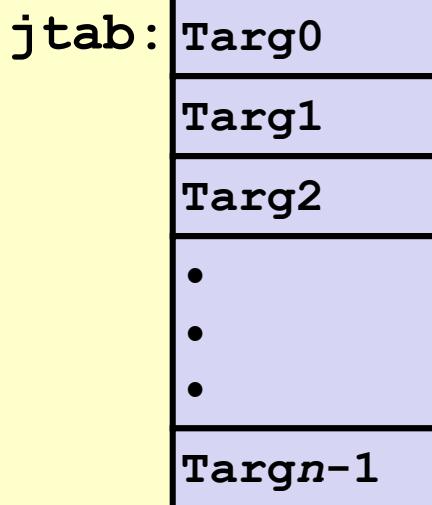
- Multiple case labels
 - Here: 5 & 6
- Fall through cases
 - Here: 2
- Missing cases
 - Here: 4

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

•
•
•

Targn-1:

Code Block
n-1

Translation (Extended C)

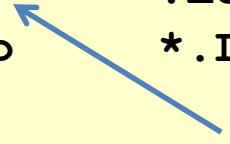
```
goto *JTab[x];
```

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

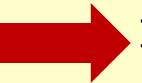
Switch Statement Example

```
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
    jmp    * .L4(,%rdi,8) # goto *JTab[x]
```

*Indirect
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 x
%rsi	Argument y
%rdx	Argument z
%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
    movl    $1, %eax   # w = 1
.L6:          # merge:
    addq    %rcx, %rax # w += z
    ret

```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%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
    movl $1, %eax      # w = 1
    subq %rdx, %rax   # w -= z
    ret
.L8:                      # Default:
    movl $2, %eax      # 2
    ret
```

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

■ Today

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

■ Next Time

- Stack
- Call / return
- Procedure (i.e., function) call discipline

Machine-Level Programming IV: Data (Arrays, Structures, Alignment)

Professor Hugh C. Lauer
CS-2011, Machine Organization and Assembly Language

(Slides include copyright materials from *Computer Systems: A Programmer's Perspective*, by Bryant and O'Hallaron, and from *The C Programming Language*, by Kernighan and Ritchie)

Today

■ Arrays

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

■ Structures

- Allocation
- Access
- Alignment

■ Floating Point

Definition — Array

From Systems course

- A collection of objects of the *same type* stored contiguously in memory under one name
 - May be type of any kind of variable
 - May be objects of the same class (C++)
 - May even be collection of arrays!
- For ease of access to any member of array
- For passing to functions as a collection

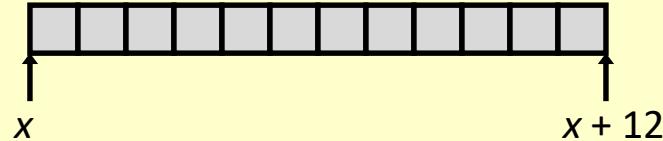
Array Allocation

■ Basic Principle

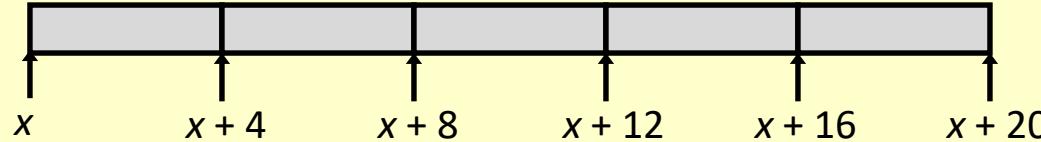
$T \mathbf{A}[L];$

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

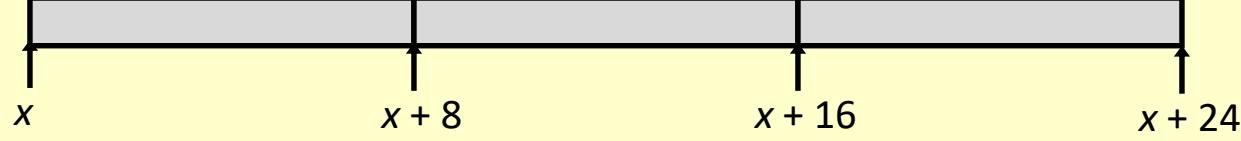
`char string[12];`



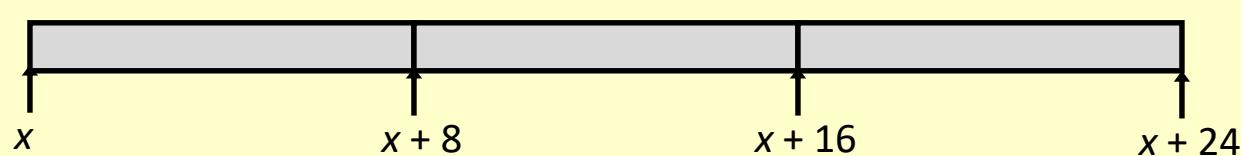
`int val[5];`



`double a[3];`



`char *p[3];`

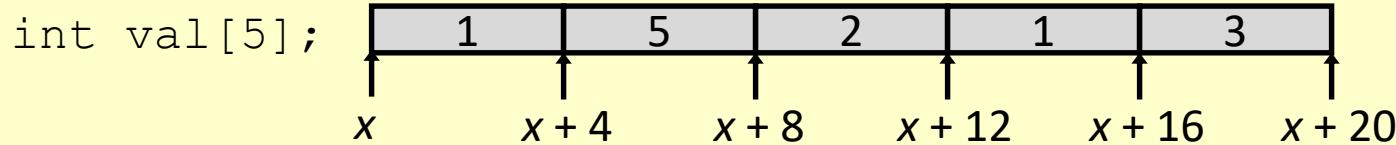


Array Access

■ Basic Principle

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

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



■ Reference Type Value

`val[4]`

`val`

`val+1`

`&val[2]`

`val[5]`

`* (val+1)`

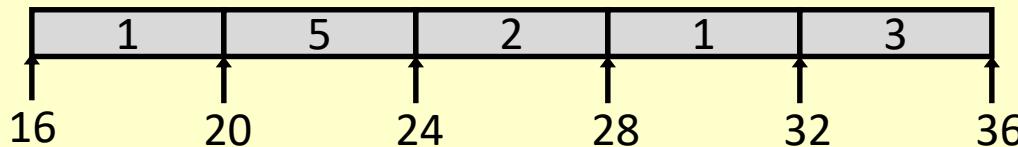
`val + i`

Array Example

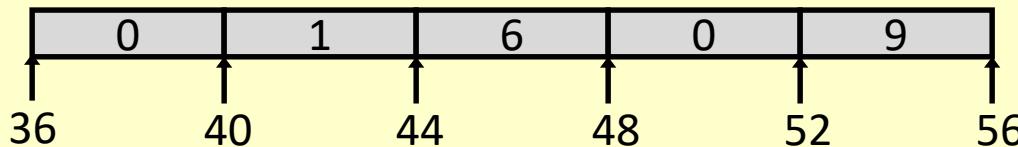
```
#define ZLEN 5
typedef int zip_dig[ZLEN];

zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig wpi = { 0, 1, 6, 0, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```

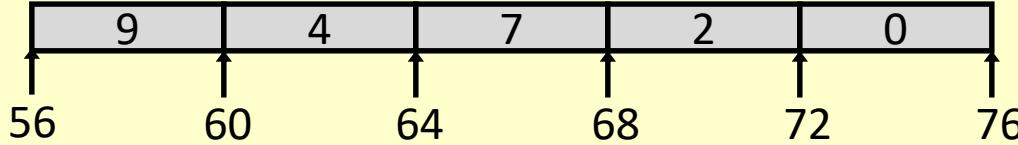
zip_dig cmu;



zip_dig wpi;



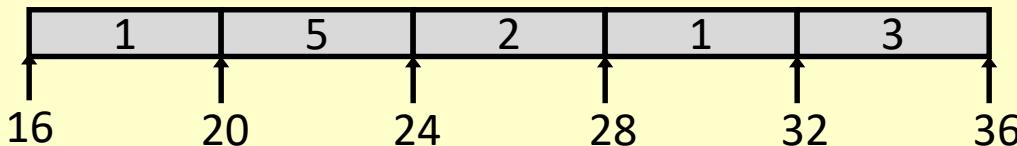
zip_dig ucb;



- 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];
}
```

X86-64

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

Questions about one-dimension arrays?

Multidimensional (Nested) Arrays

■ Declaration

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

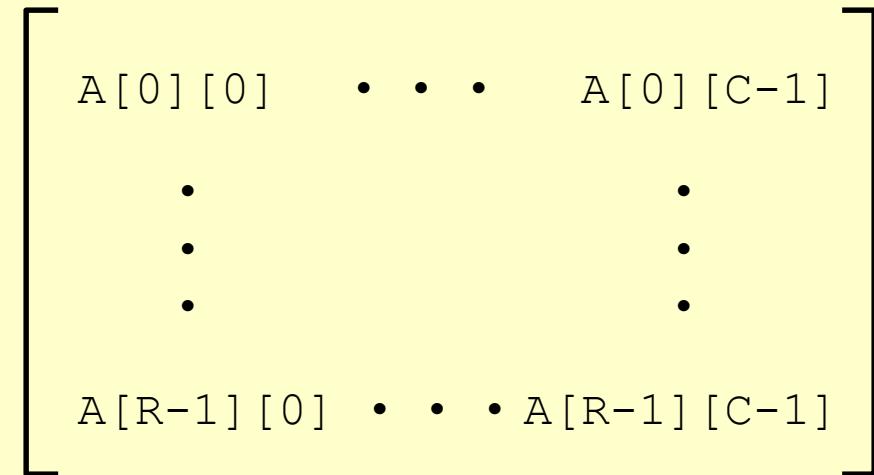
- 2D array of data type T
- R rows, C columns
- Type T element requires K bytes

■ Array Size

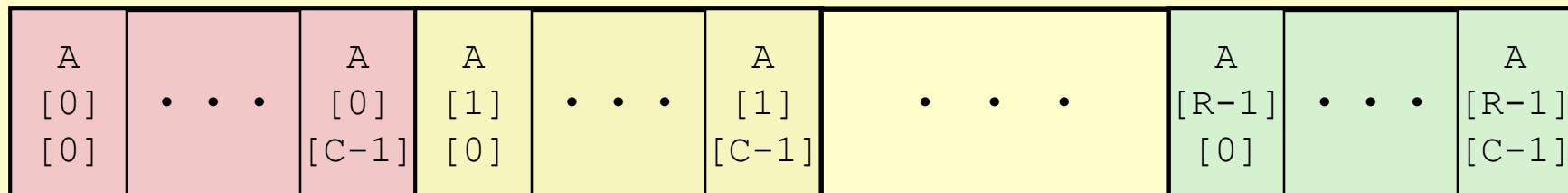
- $R * C * K$ bytes

■ Arrangement

- Row-Major Ordering



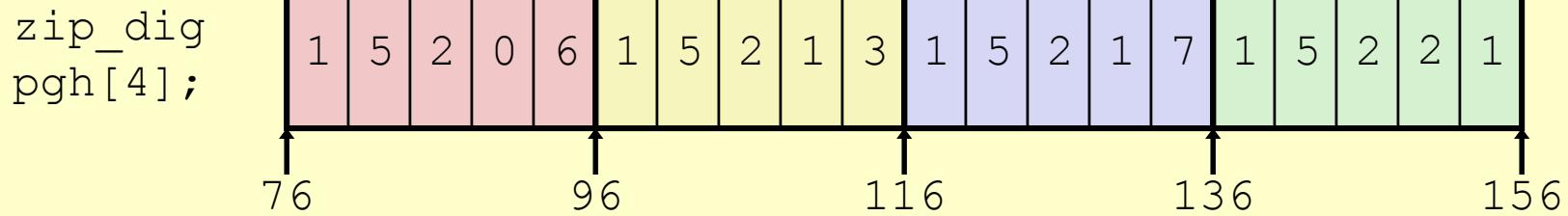
`int A[R][C];`



$4 * R * C$ Bytes

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 }};
```



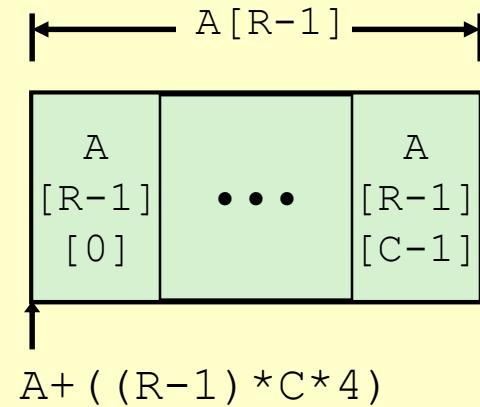
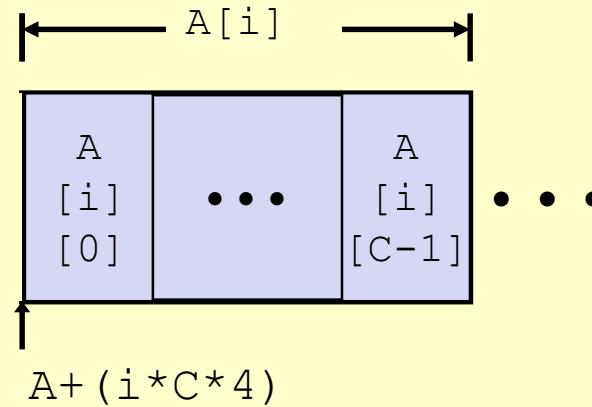
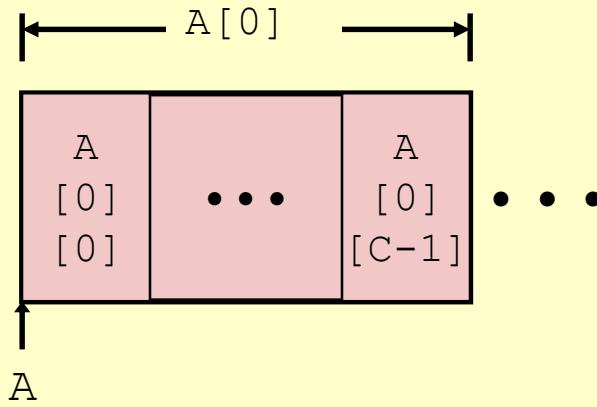
- “`zip_dig pgh[4]`” equivalent to “`int pgh[4][5]`”
 - Variable `pgh`: array of 4 elements, allocated contiguously
 - Each element is an array of 5 `int`’s, allocated contiguously
- “Row-Major” ordering of all elements in memory

Nested Array Row Access

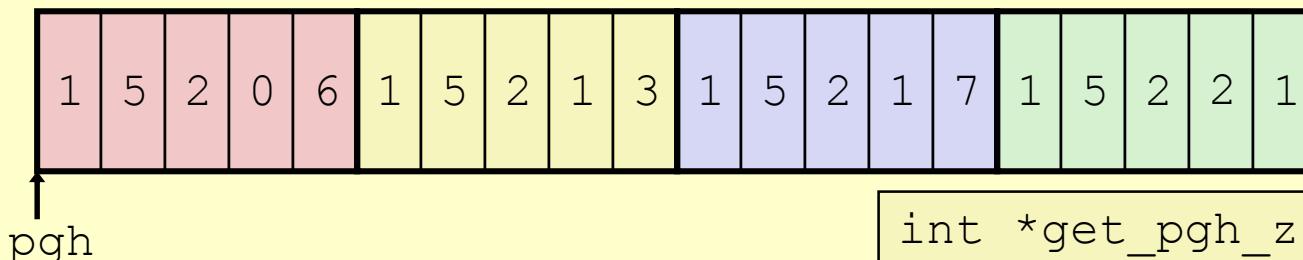
■ Row Vectors

- $\mathbf{A}[i]$ 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

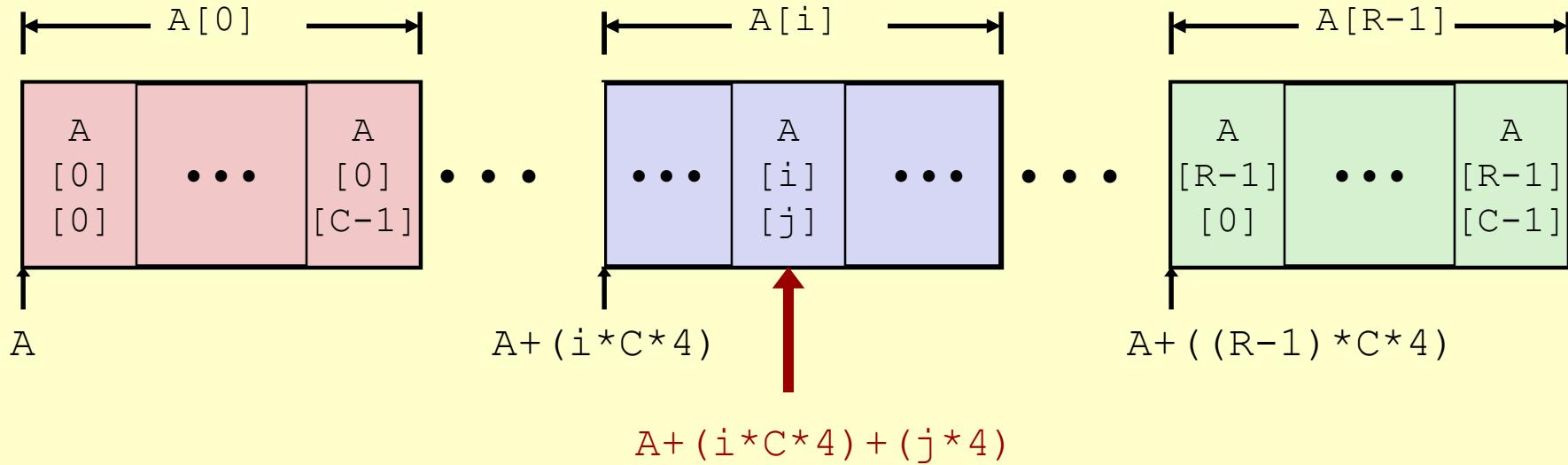
- Computes and returns address
- Compute as **pgh + 4*(index+4*index)**

Nested Array Element Access

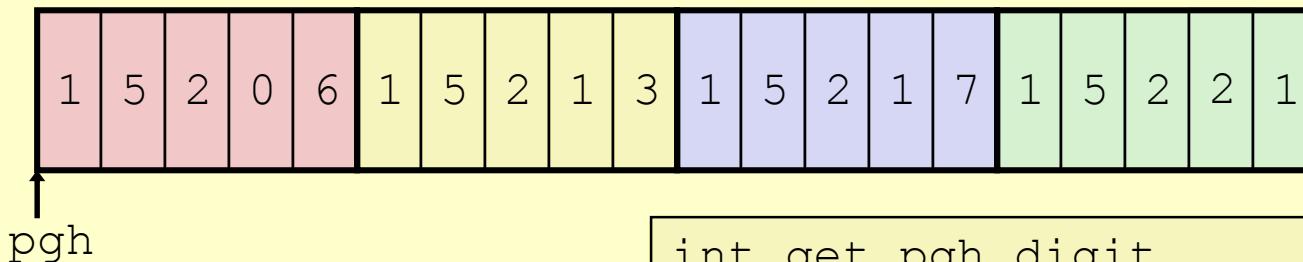
■ Array Elements

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

```
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)`

Correct vs. Traditional approach

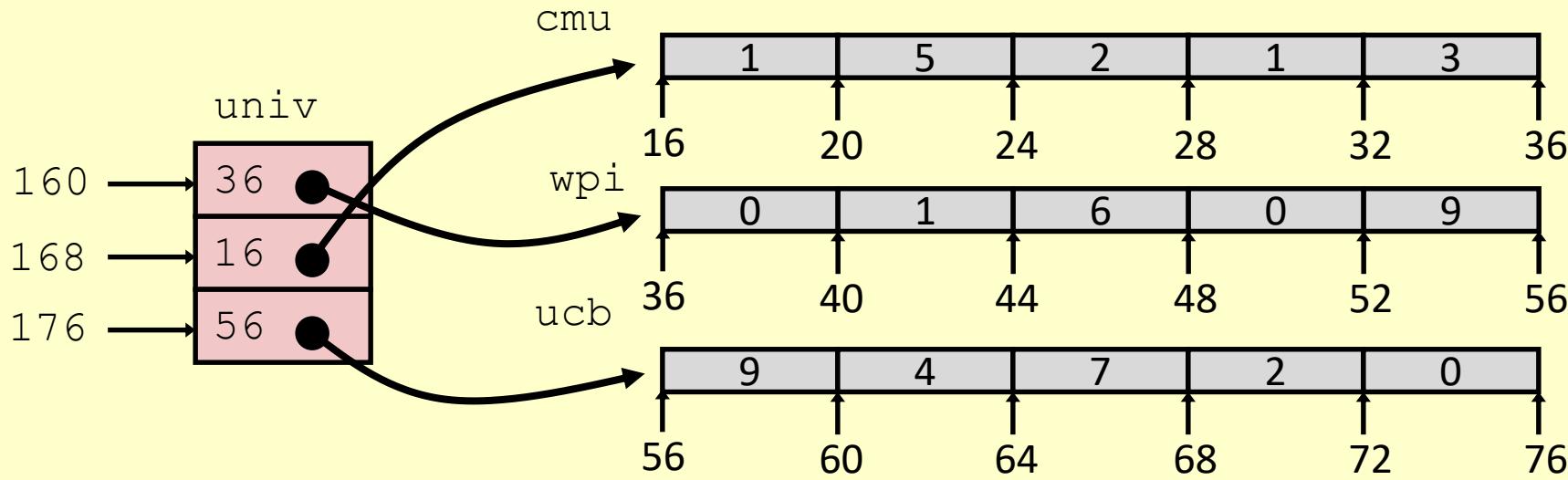
- The previous approach is the mathematically “correct” way to access multi-dimensional arrays
 - Requires multiplication for intermediate dimension
- Traditionally, old, small systems did not handle multiplication very well (if at all!)
 - Solution:– allocate intermediate arrays of pointers to access individual rows.

Multi-Level Array Example

```
zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig wpi = { 0, 1, 6, 0, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```

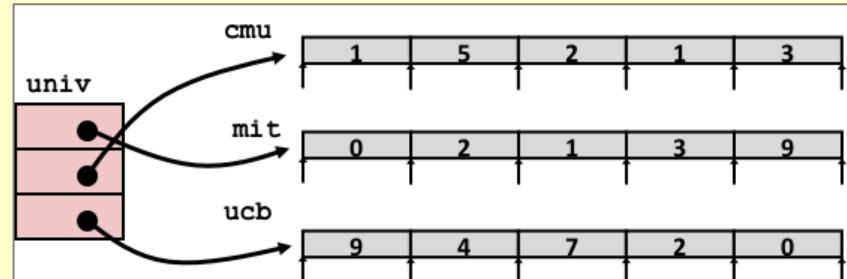
```
#define UCOUNT 3
int *univ[UCOUNT] = {wpi, cmu, ucb};
```

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



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
```

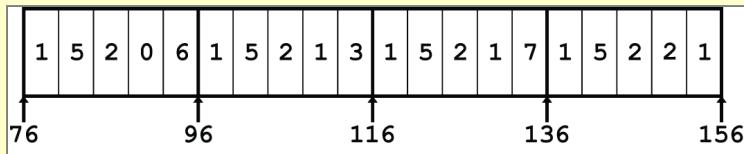
■ Computation

- 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

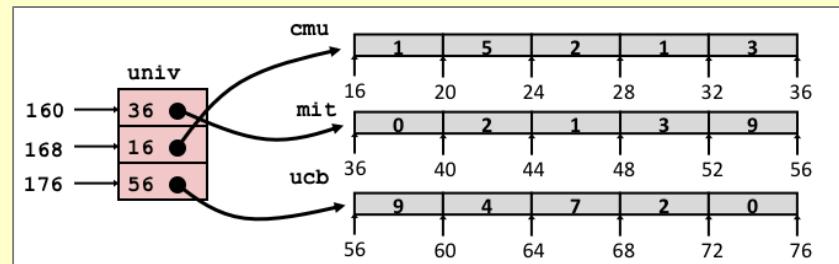
Nested array

```
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]`

N X N 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

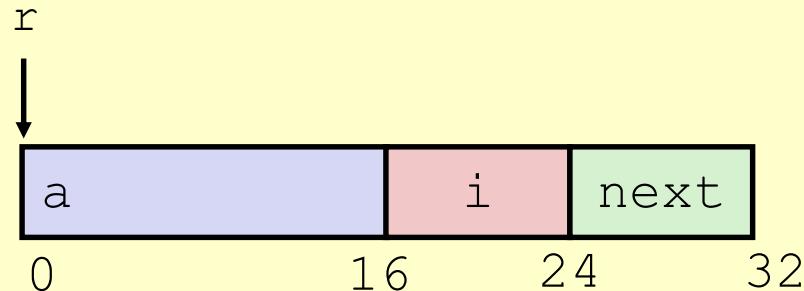
■ Structures

- Allocation
- Access
- Alignment

■ Floating Point

Structure Representation

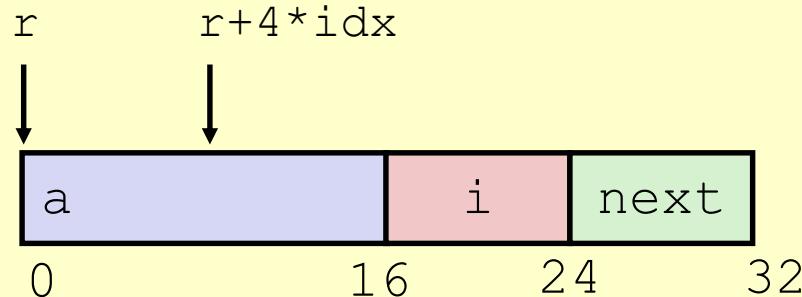
```
struct rec {  
    int a[4];  
    size_t i;  
    struct rec *next;  
};
```



- 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];
}
```

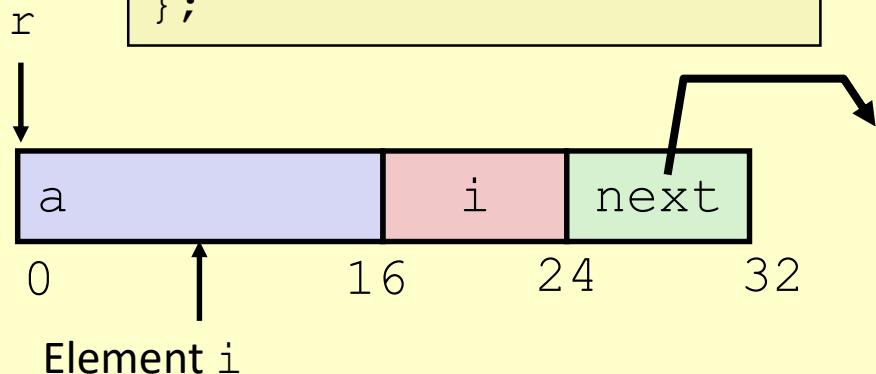
```
# r in %rdi, idx in %rsi
leaq (%rdi,%rsi,4), %rax
ret
```

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;
};
```

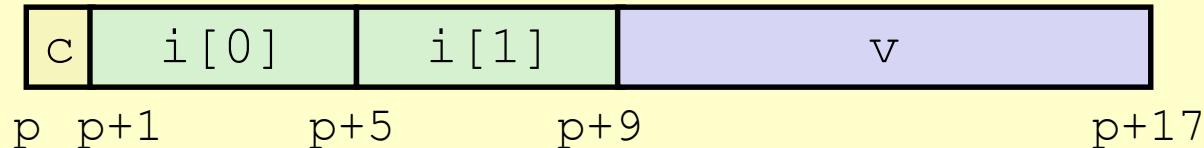


Register	Value
%rdi	r
%rsi	val

```
.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

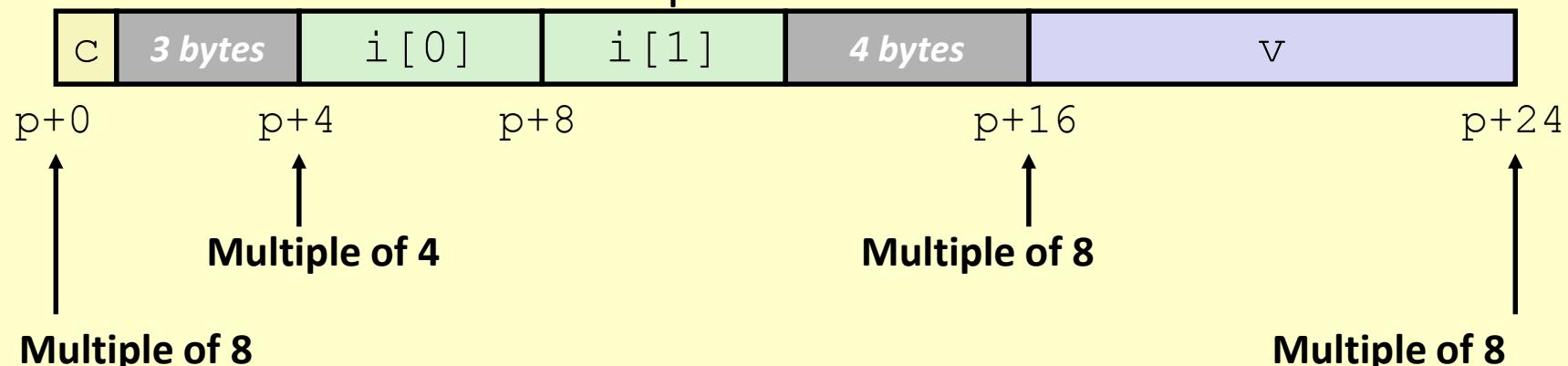
■ Unaligned Data



```
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

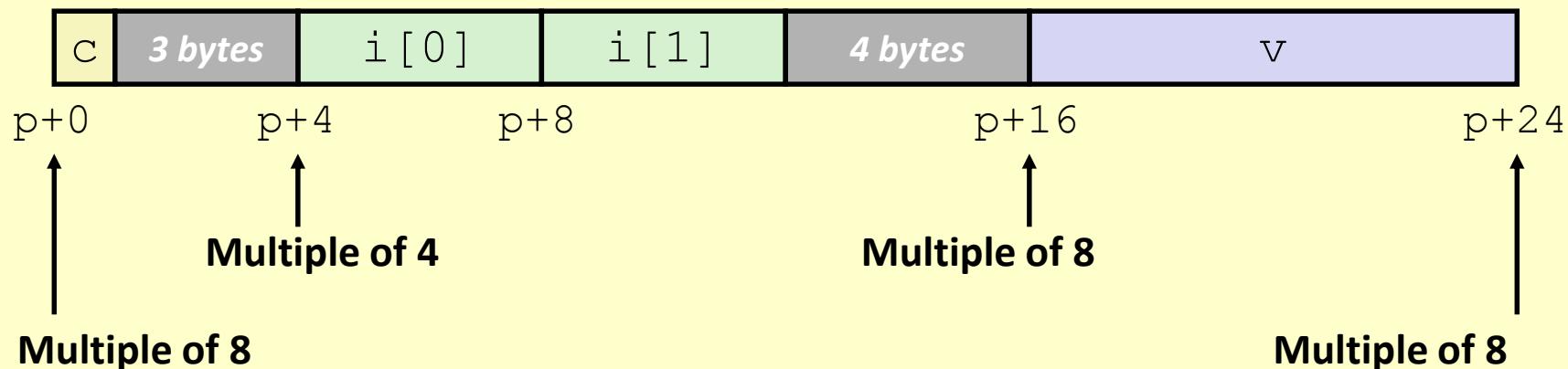
■ Overall structure placement

- Each structure has alignment requirement **K**
 - **K** = Largest alignment of any element
- Initial address & structure length must be multiples of **K**

■ Example:

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

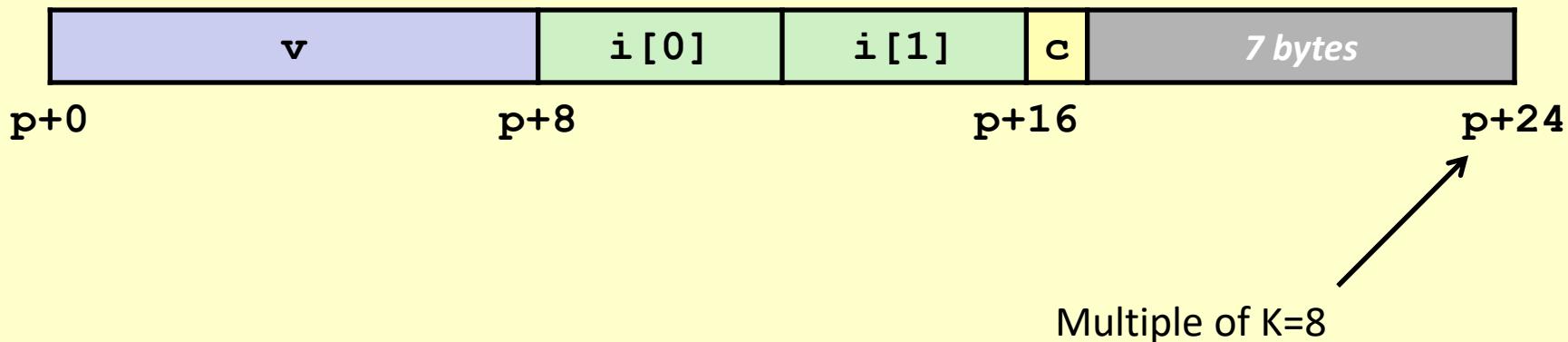
```
struct S1 {
    char c;
    int i[2];
    double v;
} *p;
```



Meeting Overall Alignment Requirement

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

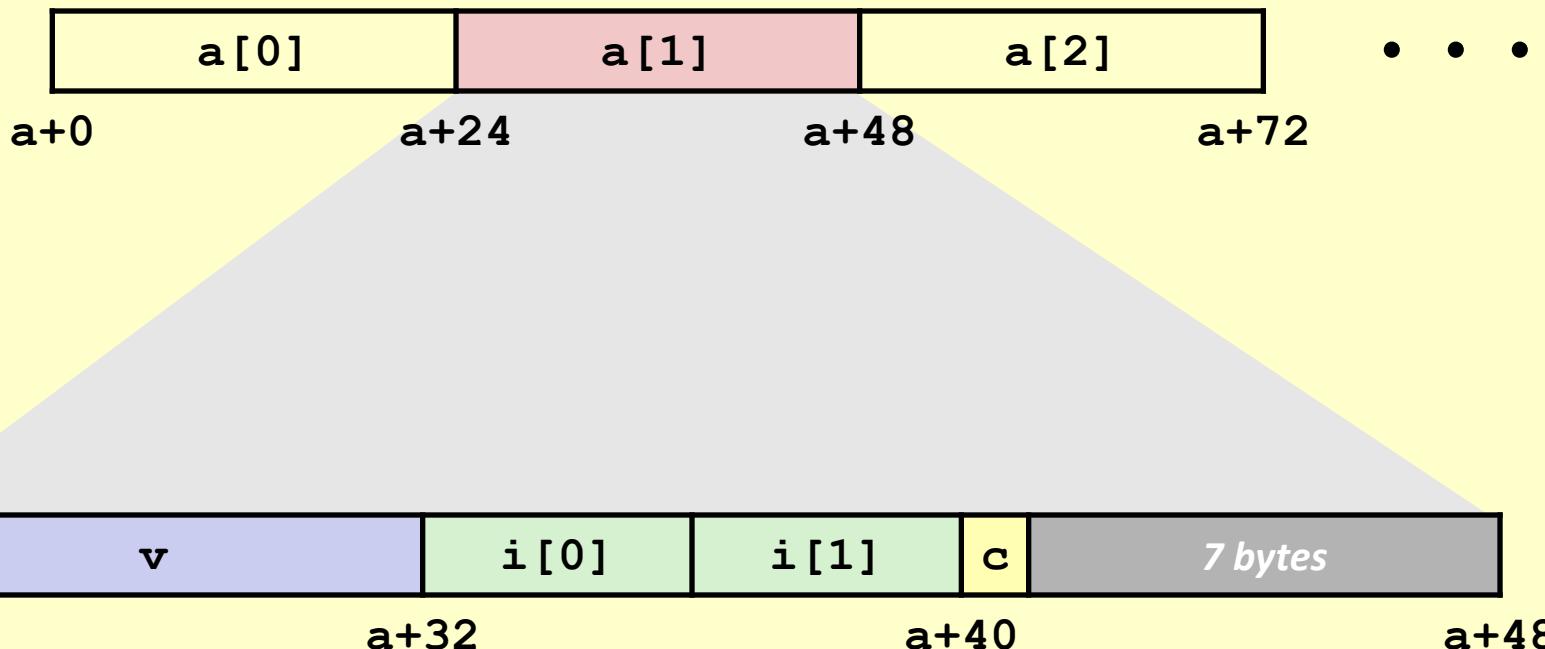
```
struct S2 {  
    double v;  
    int i[2];  
    char c;  
} *p;
```



Arrays of Structures

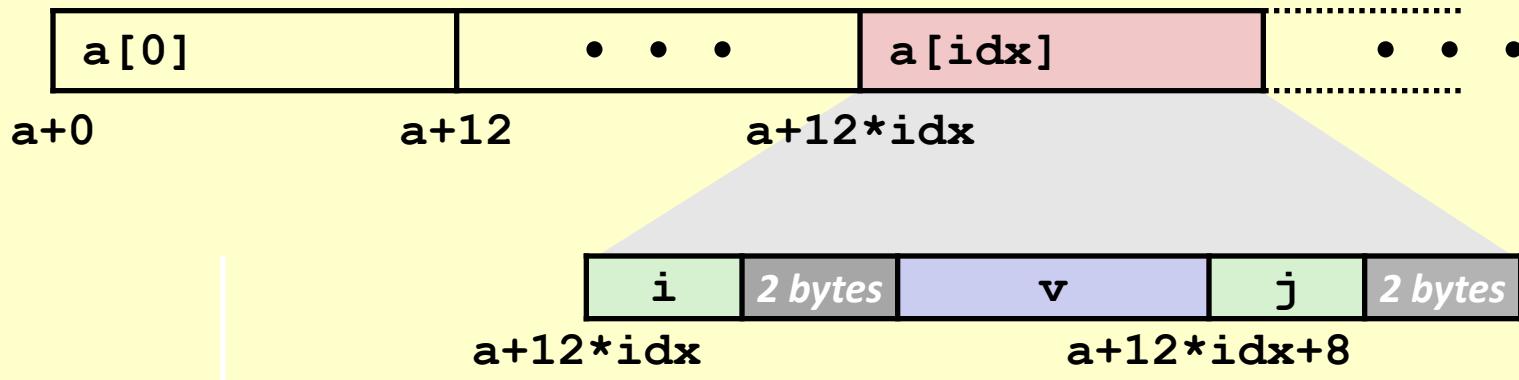
- Overall structure length multiple of K
- Satisfy alignment requirement for every element

```
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



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

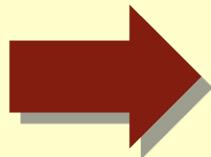
```
# %rdi = idx
leaq (%rdi,%rdi,2),%rax # 3*idx
movzwl a+8(%rax,4),%eax
```

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

Saving Space

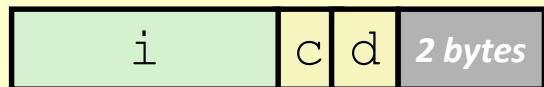
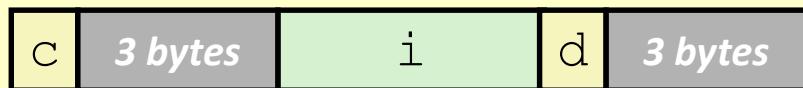
■ Put large data types first

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



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

■ Effect (K=4)



Today

■ Arrays

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

■ Structures

- Allocation
- Access
- Alignment

■ Floating Point

Background

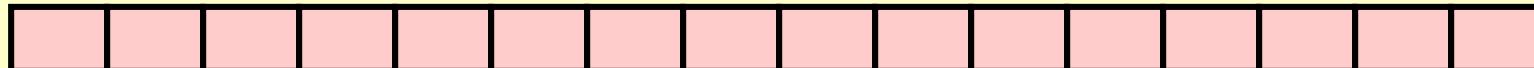
■ History

- x87 FP
 - Legacy, very ugly
- SSE FP
 - Supported by Shark machines
 - Special case use of vector instructions
- AVX FP
 - Newest version
 - Similar to SSE
 - Documented in book

Programming with SSE3

XMM Registers

- 16 total, each 16 bytes
- 16 single-byte integers



- 8 16-bit integers



- 4 32-bit integers



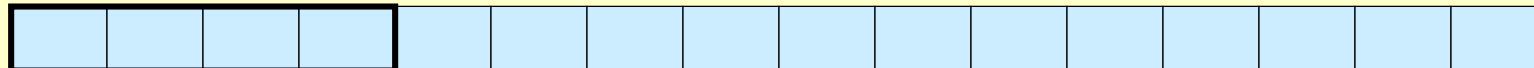
- 4 single-precision floats



- 2 double-precision floats



- 1 single-precision float

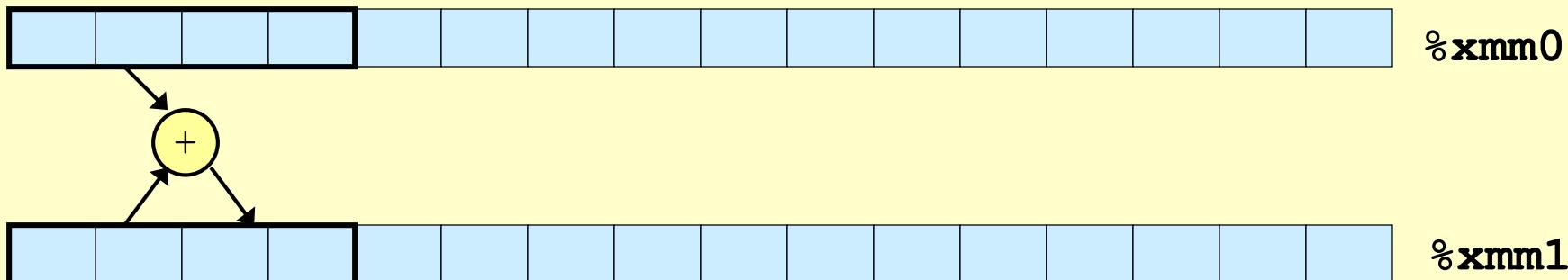


- 1 double-precision float

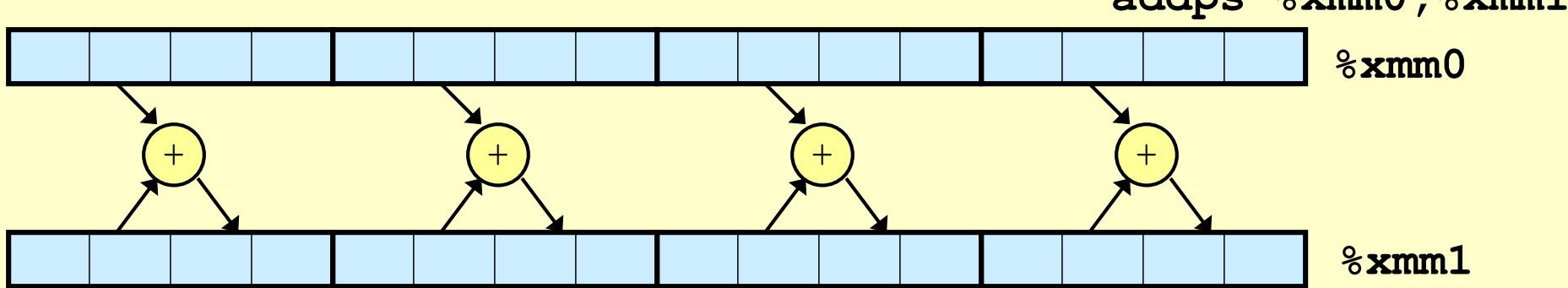


Scalar & SIMD Operations

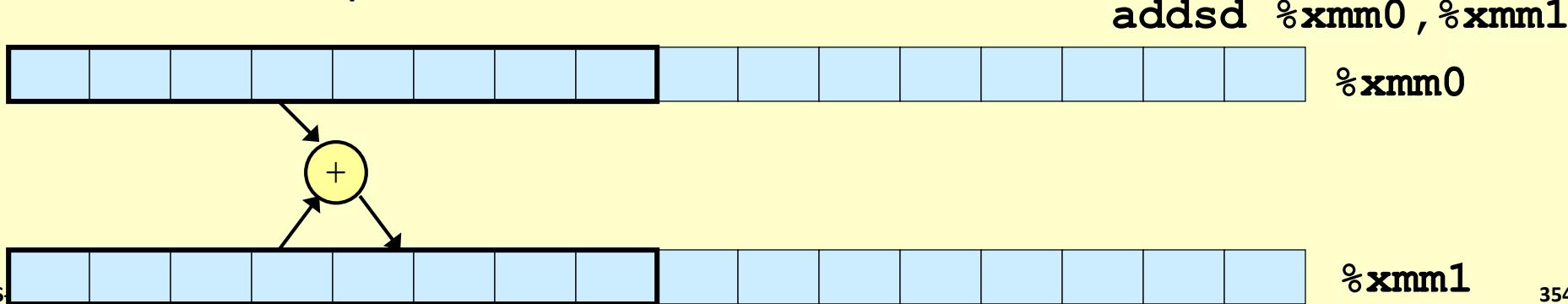
■ Scalar Operations: Single Precision



■ SIMD Operations: Single Precision



■ Scalar Operations: Double Precision



FP Basics

- Arguments passed in `%xmm0, %xmm1, ...`
- Result returned in `%xmm0`
- All XMM registers caller-saved

```
float fadd(float x, float y)
{
    return x + y;
}
```

```
double dadd(double x, double y)
{
    return x + y;
}
```

```
# x in %xmm0, y in %xmm1
addss    %xmm1, %xmm0
ret
```

```
# x in %xmm0, y in %xmm1
addsd    %xmm1, %xmm0
ret
```

FP Memory Referencing

- Integer (and pointer) arguments passed in regular registers
- FP values passed in XMM registers
- Different mov instructions to move between XMM registers, and between memory and XMM registers

```
double dincr(double *p, double v)
{
    double x = *p;
    *p = x + v;
    return x;
}
```

```
# p in %rdi, v in %xmm0
movapd  %xmm0, %xmm1    # Copy v
movsd   (%rdi), %xmm0  # x = *p
addsd   %xmm0, %xmm1    # t = x + v
movsd   %xmm1, (%rdi)  # *p = t
ret
```

Other Aspects of FP Code

- **Lots of instructions**
 - Different operations, different formats, ...
- **Floating-point comparisons**
 - Instructions **ucomiss** and **ucomisd**
 - Set condition codes CF, ZF, and PF
- **Using constant values**
 - Set XMM0 register to 0 with instruction **xorpd %xmm0, %xmm0**
 - Others loaded from memory

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

■ Floating Point

- Data held and operated on in XMM registers

Understanding Pointers & Arrays #1

Decl	An			*An		
	Cmp	Bad	Size	Cmp	Bad	Size
int A1[3]						
int *A2						

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

Machine-Level Programming III: Procedures (*aka Functions*)

Professor Hugh C. Lauer
CS-2011, Machine Organization and Assembly Language

(Slides include copyright materials from *Computer Systems: A Programmer's Perspective*, by Bryant and O'Hallaron, and from *The C Programming Language*, by Kernighan and Ritchie)

Reading Assignment: §3.7

Mechanisms in Functions

■ Passing control

- To beginning of function code
- Back to return point

■ Passing data

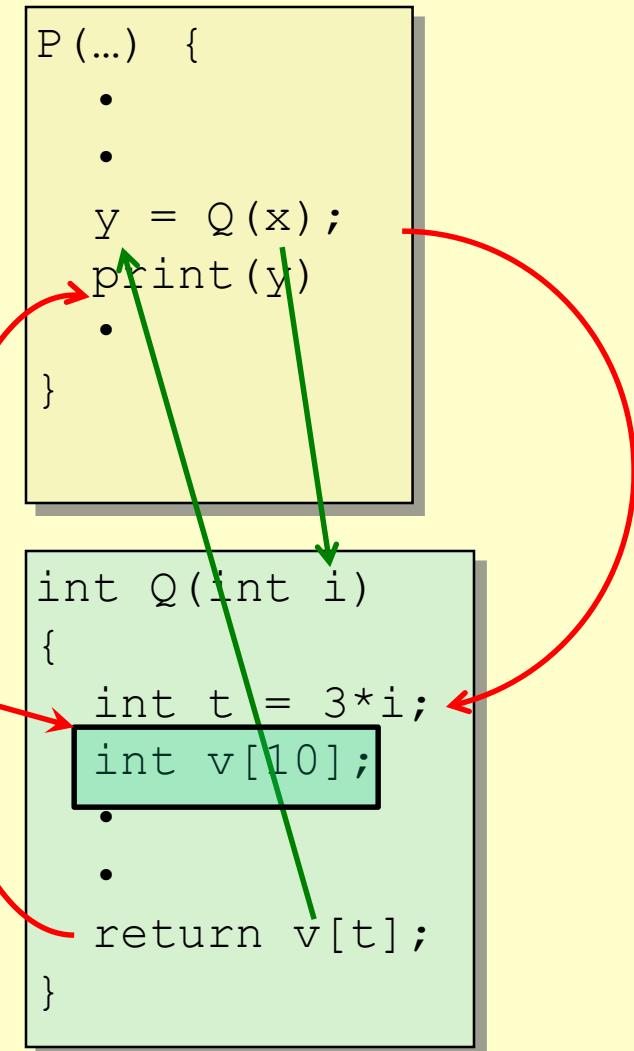
- function arguments
- Return value

■ Memory management

- Allocate during function execution
- Deallocate upon return

■ Mechanisms all implemented with machine instructions

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



Today

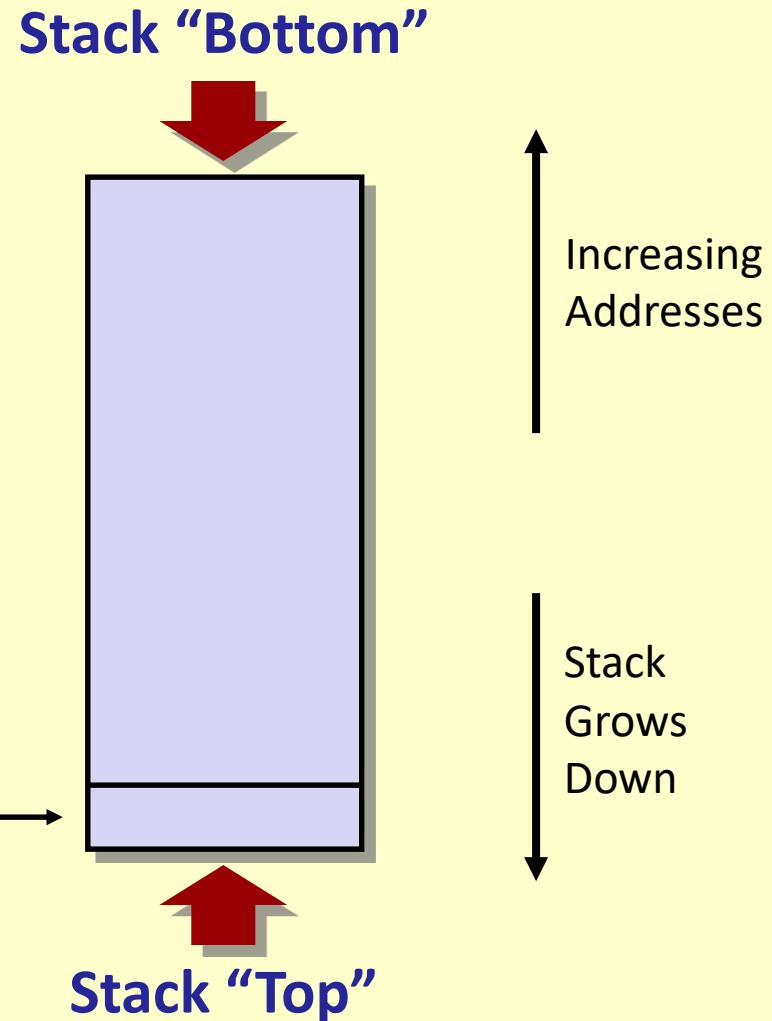
■ Procedures Functions

- 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

Stack Pointer: `%rsp` →



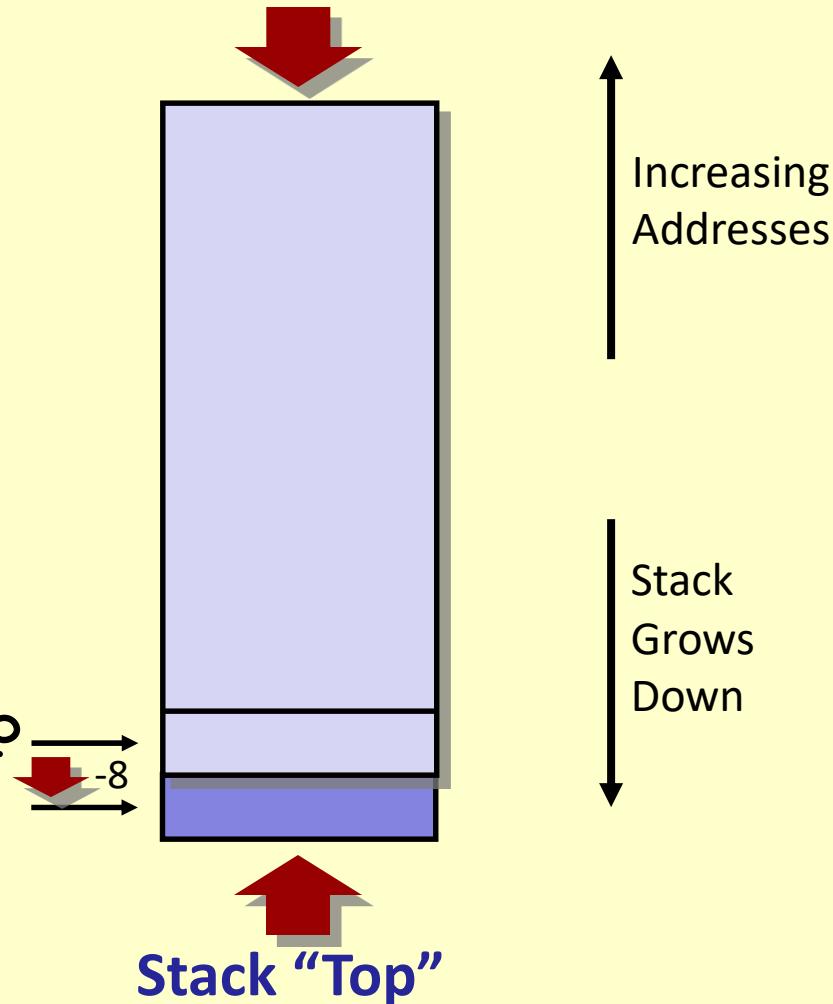
x86-64 Stack: Push

■ **pushq Src**

- Fetch operand at *Src*
- Decrement $\%rsp$ by 8
- Write operand at address given by $\%rsp$

Stack Pointer: $\%rsp$

Stack “Bottom”

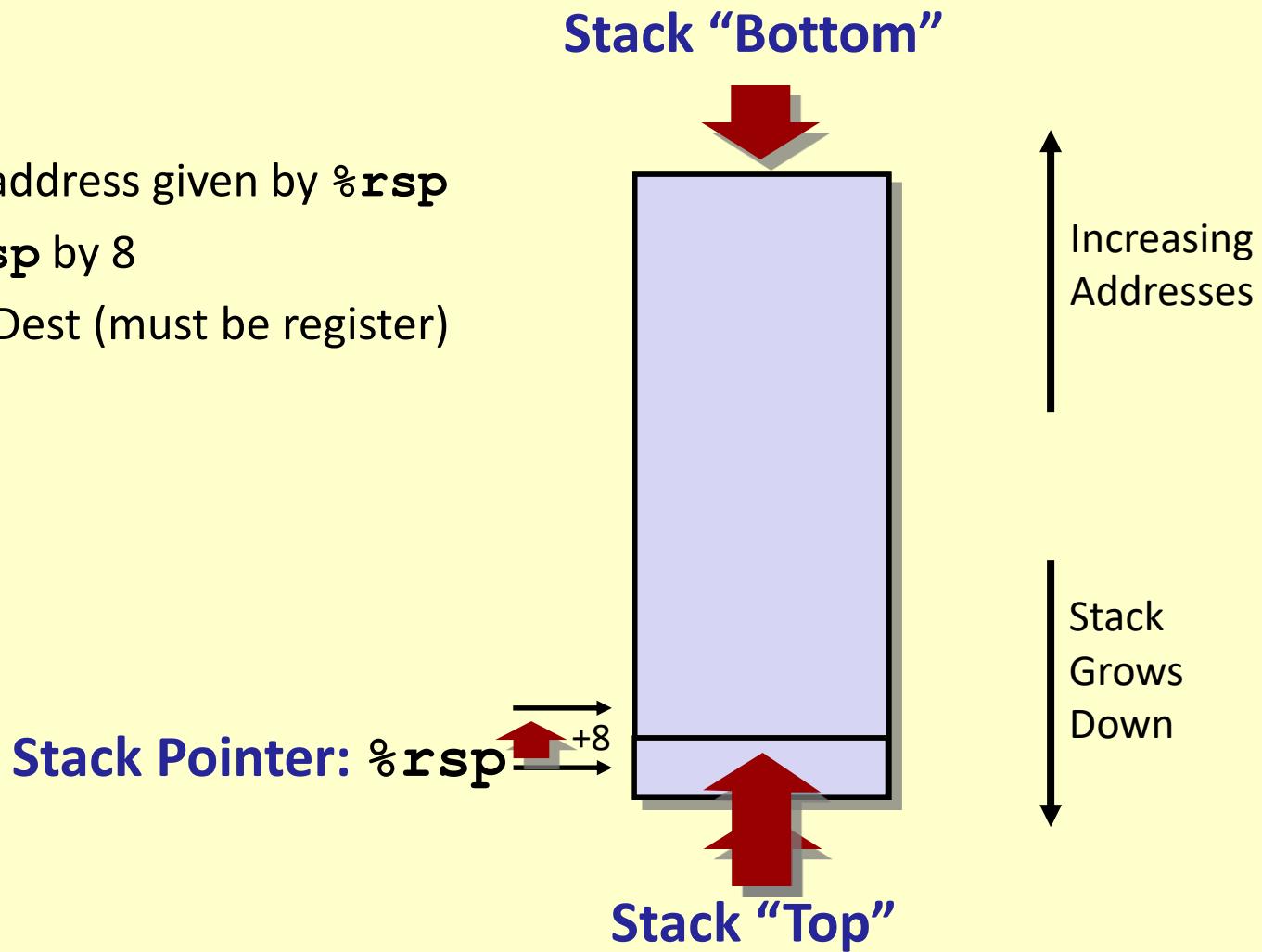


Stack “Top”

x86-64 Stack: Pop

■ **popq Dest**

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



Today

■ Procedures Functions

- 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;
}
```

```
000000000400540 <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	

Caller save
register
(p. 251,
§3.7.5)

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

```
000000000400550 <mult2>:
```

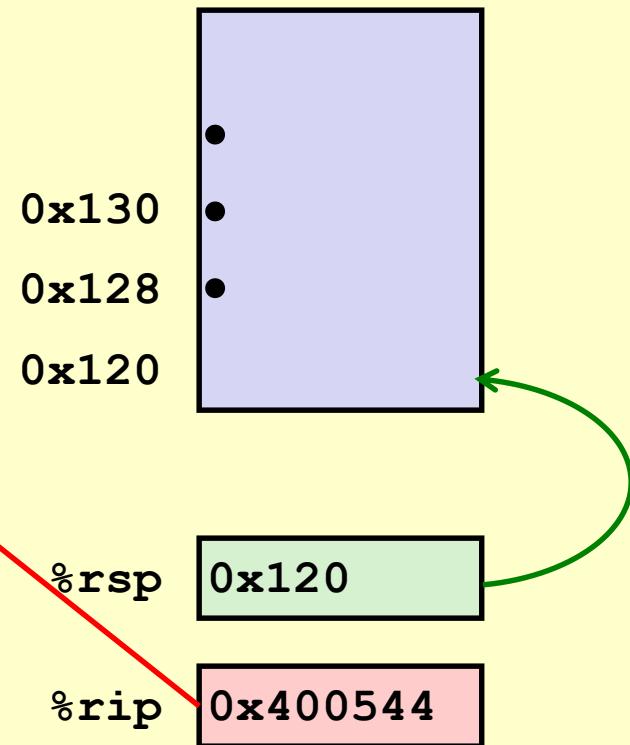
400550: mov %rdi,%rax	# a
400553: imul %rsi,%rax	# a * b
400557: retq	# Return

Function Control Flow

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

Control Flow Example #1

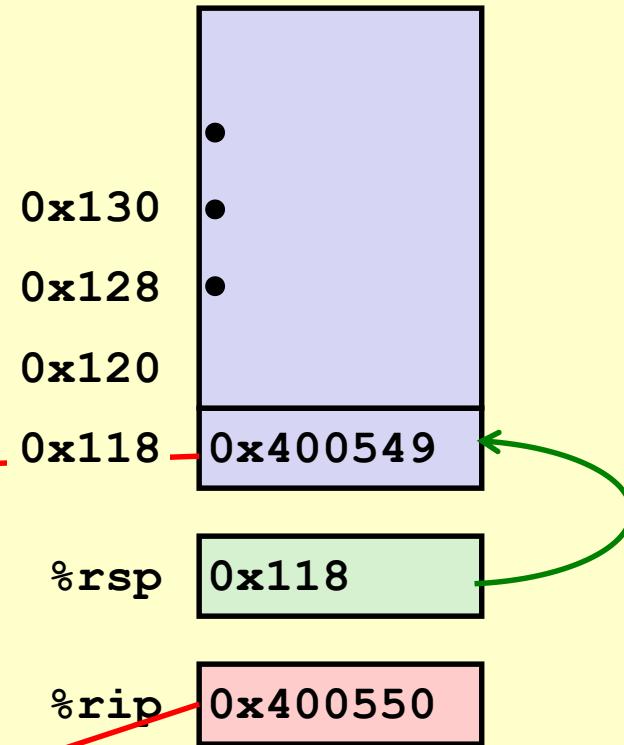
```
0000000000400540 <multstore>:  
.  
.  
400544: callq 400550 <mult2>  
400549: mov    %rax, (%rbx)  
.  
.
```



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

Control Flow Example #2

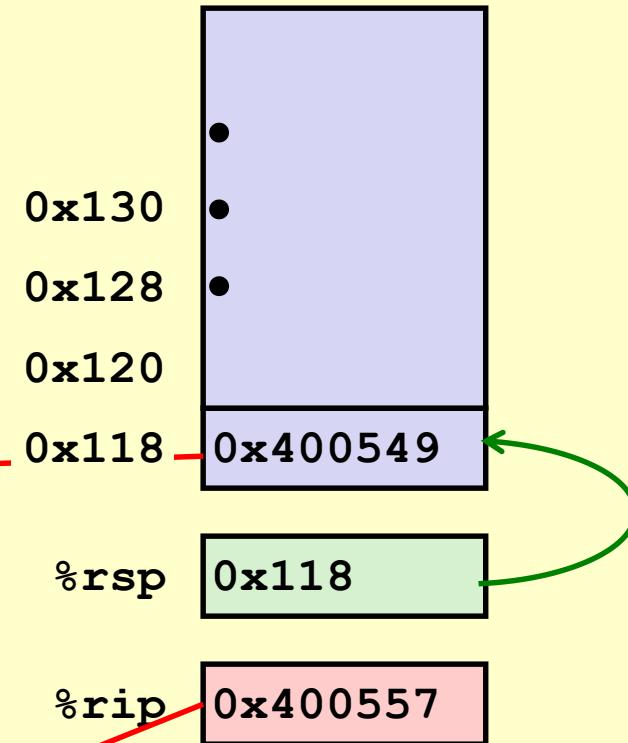
```
0000000000400540 <multstore>:  
.  
.  
400544: callq 400550 <mult2>  
400549: mov    %rax, (%rbx) ←
```



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

Control Flow Example #3

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

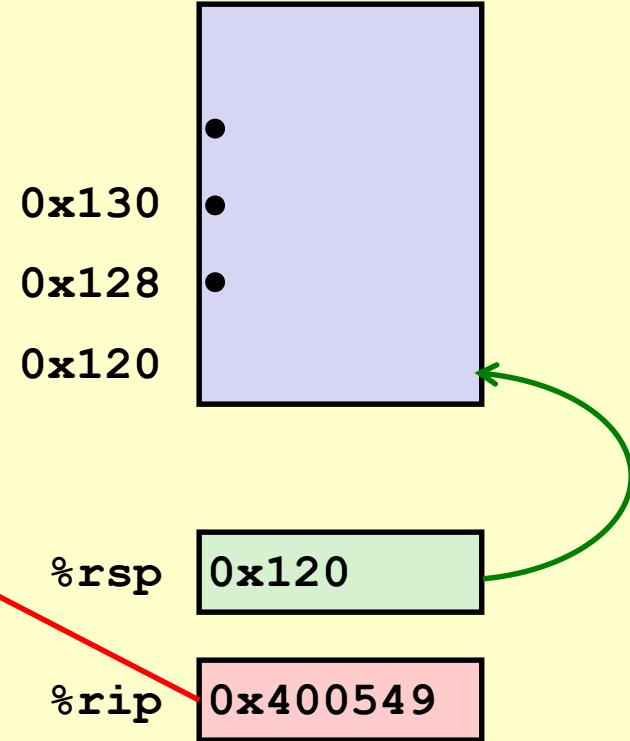


```
0000000000400550 <mult2>:  
400550: mov    %rdi,%rax  
. .  
400557: retq ←
```

Control Flow Example #4

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

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



Note: *Callee* must return stack
to condition it found it
(subject to side effects)

Today

■ Procedures Functions

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

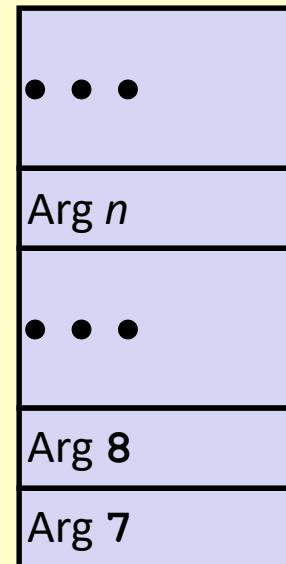
Procedure Data Flow

Registers

- First 6 arguments



Stack



- Return value



- Only allocate stack space when needed

Data Flow Examples

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

000000000400540 <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;
}
```

000000000400550 <mult2>:

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

Today

■ Procedures Functions

- 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 function
- Need some place to store state of each instantiation
 - Arguments
 - Local variables
 - Return pointer

■ Stack discipline

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

■ Stack allocated in *Frames*

- state for single function instantiation

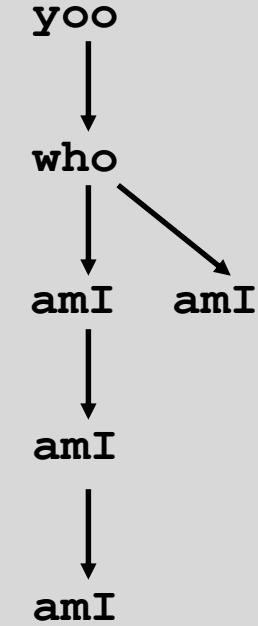
Call Chain Example

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

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

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

Example Call Chain

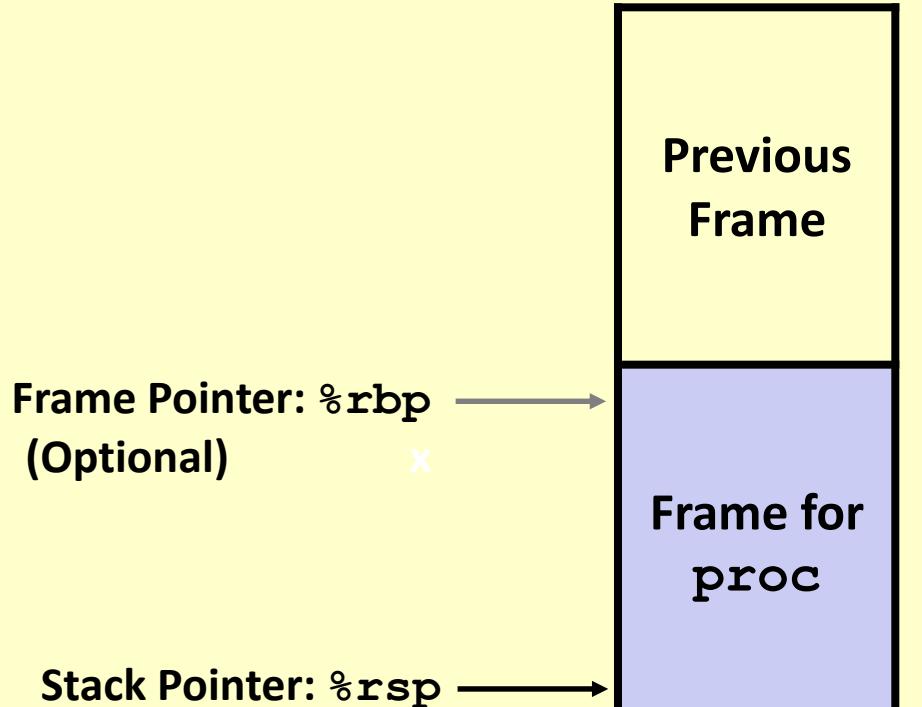


function amI () is recursive

Stack Frames

■ Contents

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

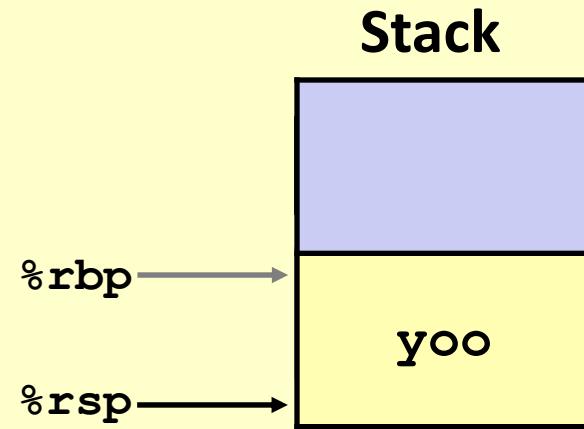
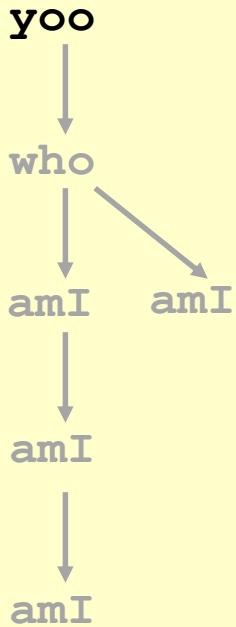


■ Management

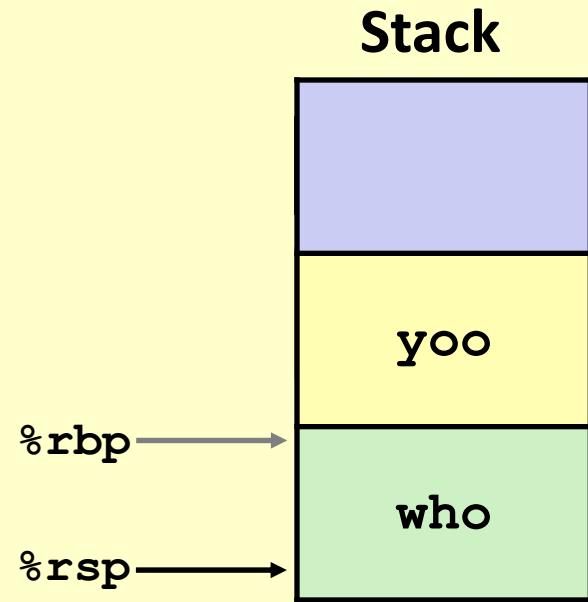
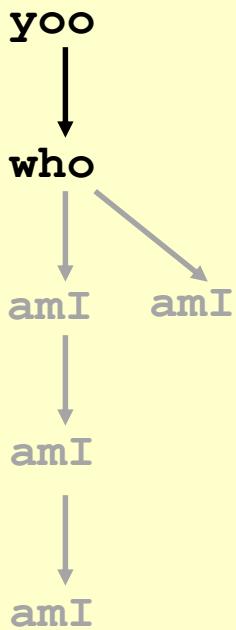
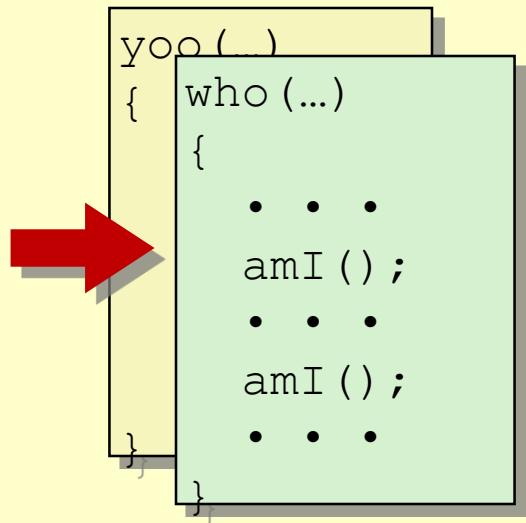
- Space allocated when enter function
 - “Set-up” code
 - Includes push by **call** instruction
- Deallocated when return
 - “Finish” code
 - Includes pop by **ret** instruction

Example

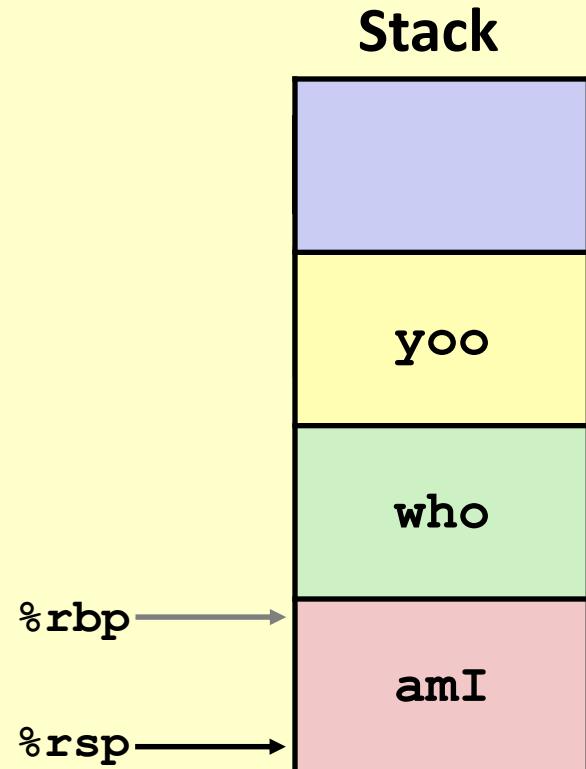
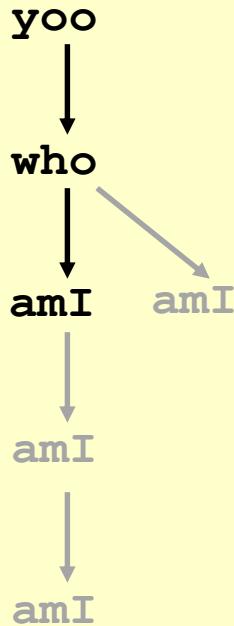
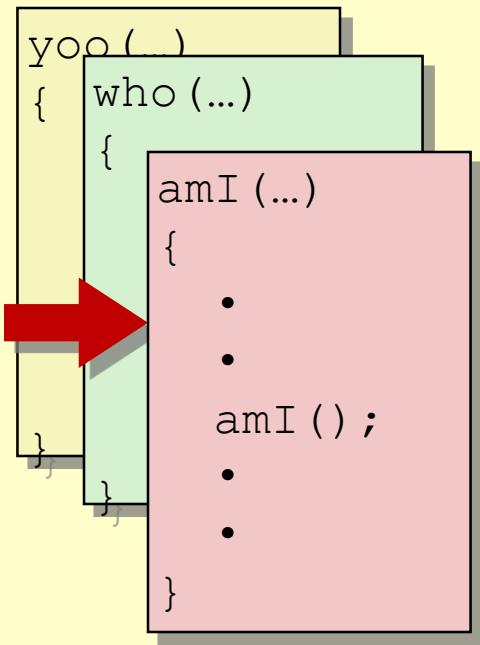
```
yoo (...)  
{  
    •  
    •  
    who () ;  
    •  
    •  
}
```



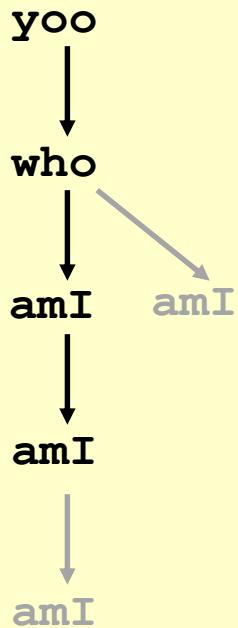
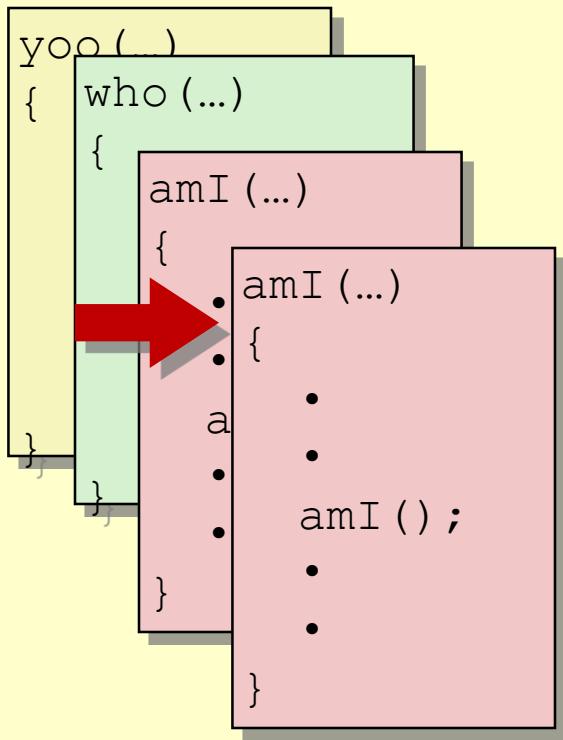
Example



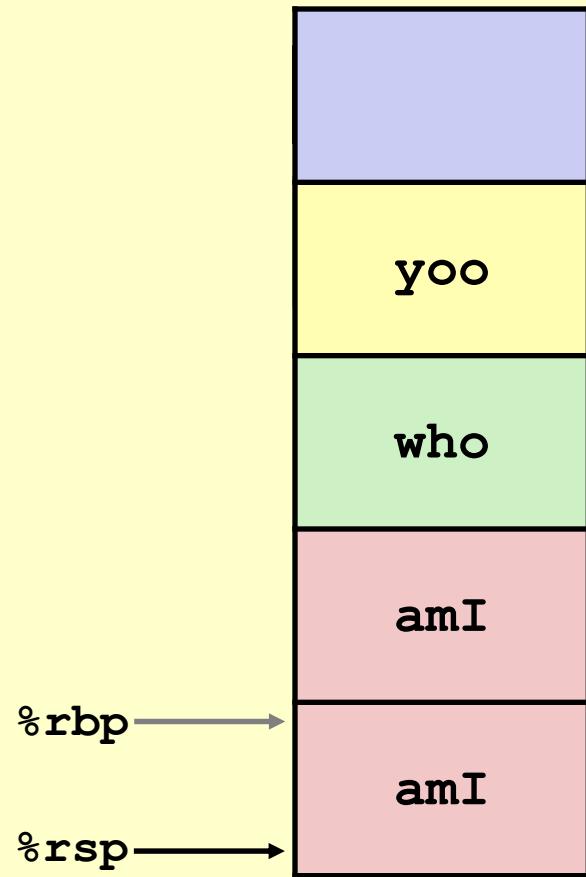
Example



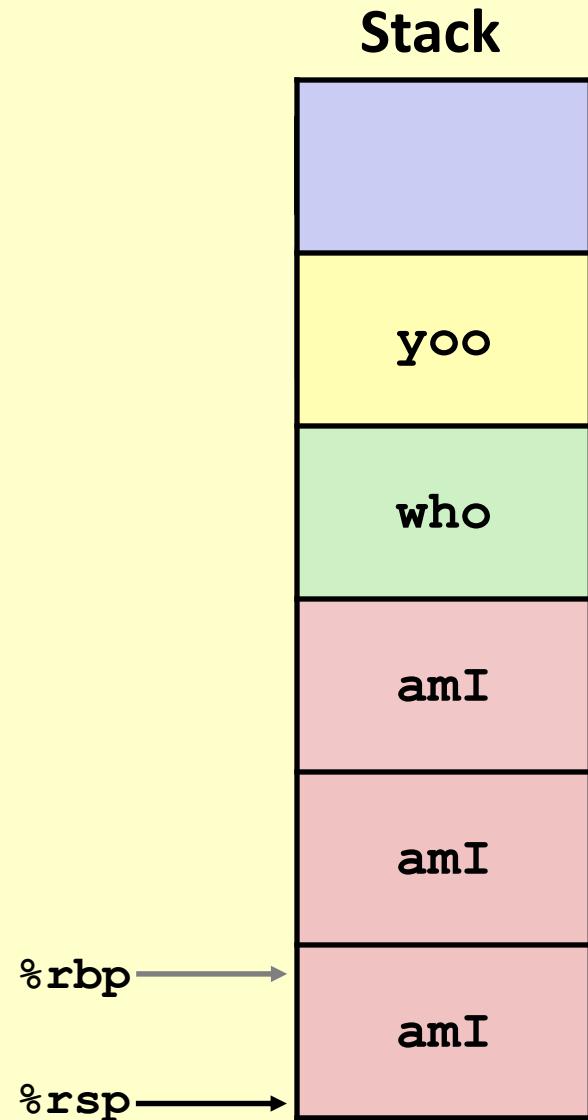
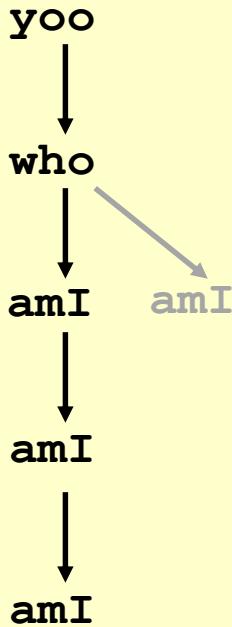
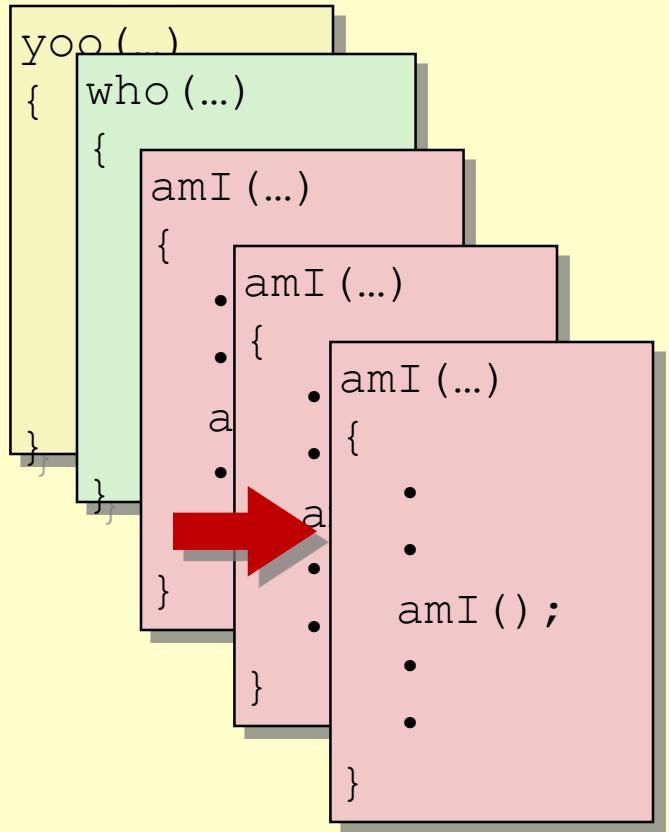
Example



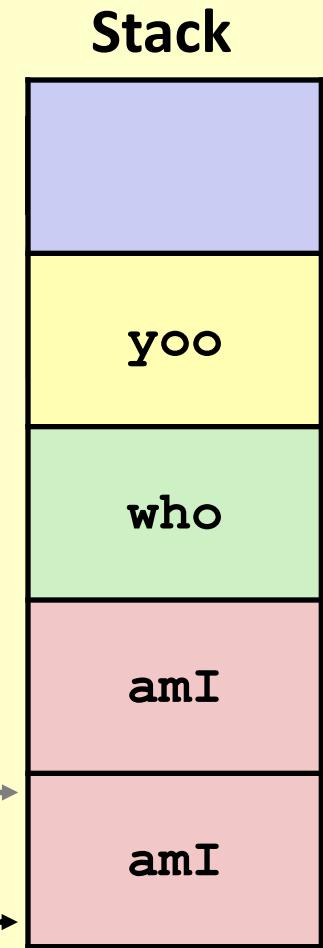
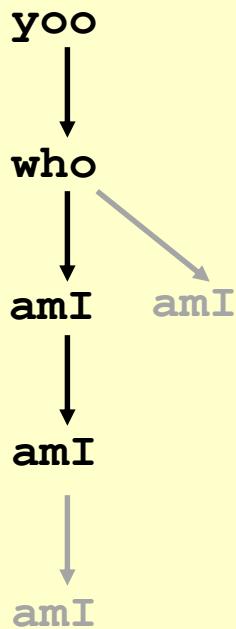
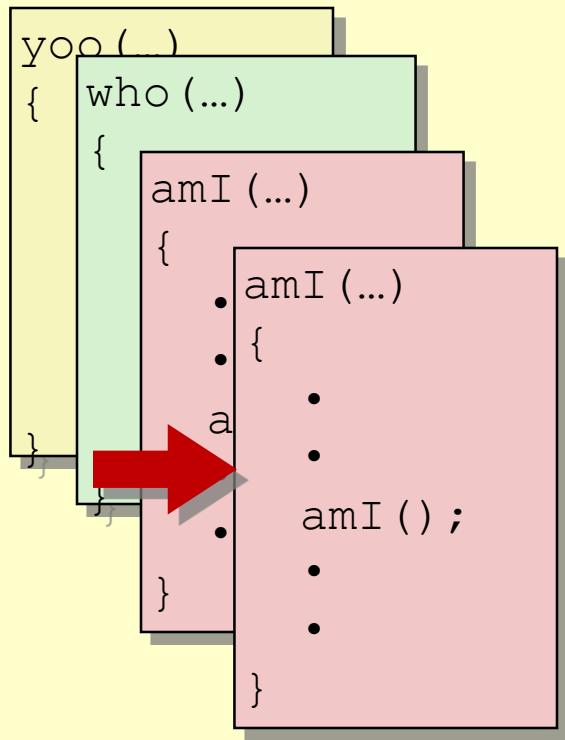
Stack



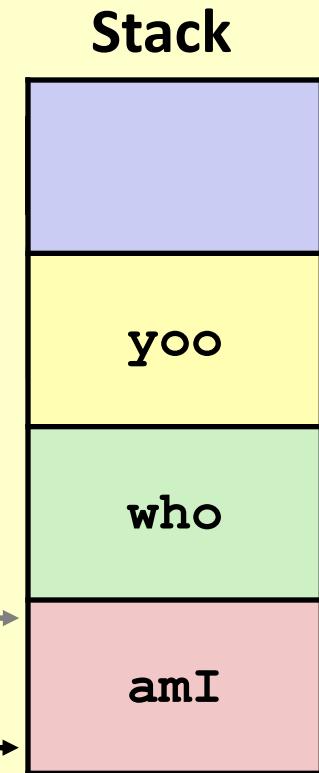
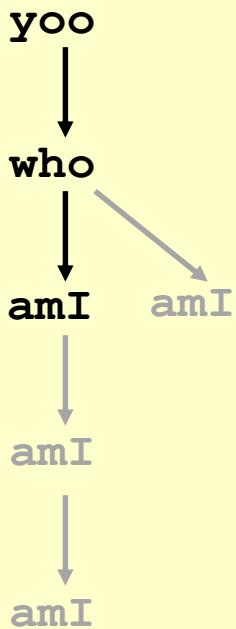
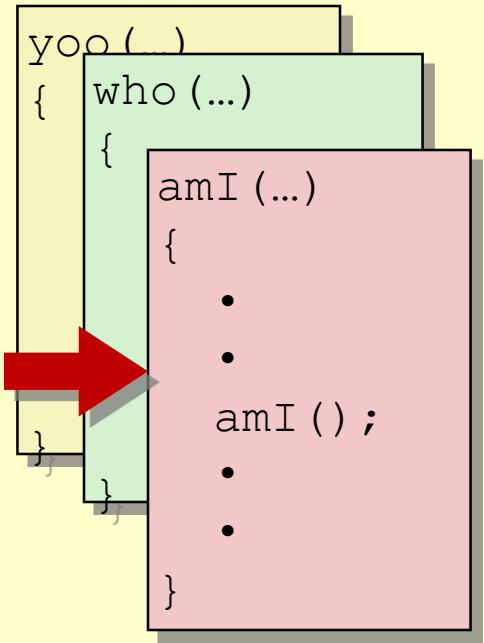
Example



Example

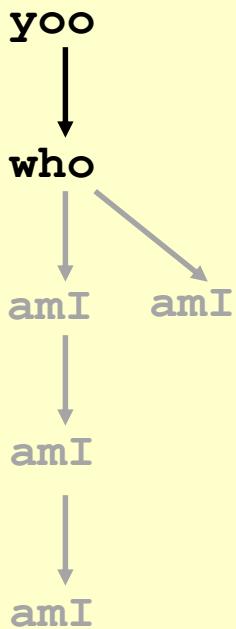


Example

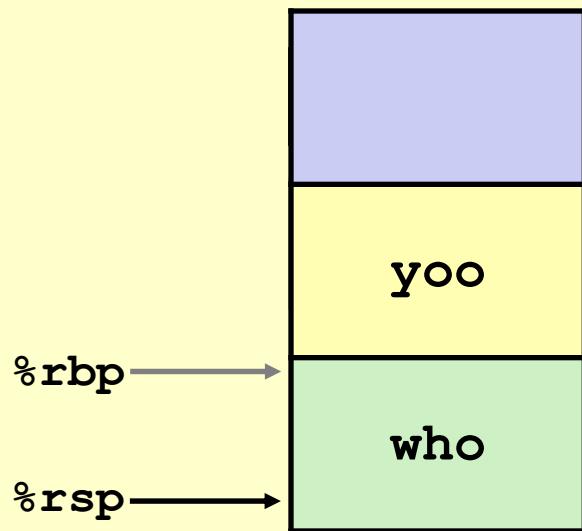


Example

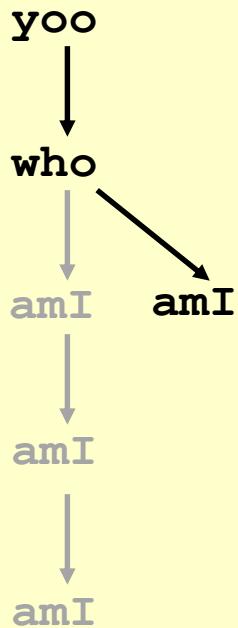
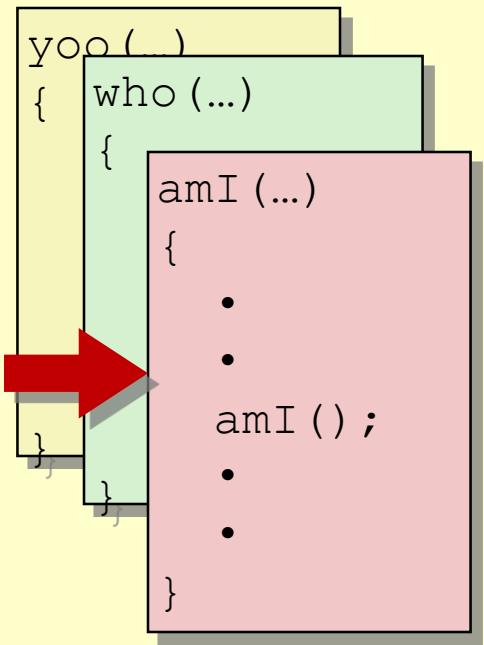
```
yoo(...)  
{    who(...)  
    {  
        . . .  
        amI();  
        . . .  
        amI();  
        . . .  
    }  
}
```



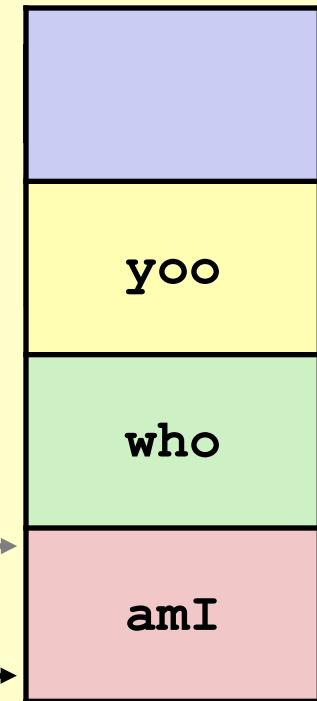
Stack



Example

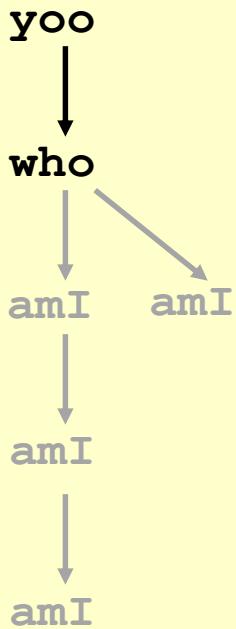
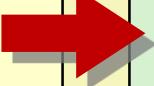


Stack

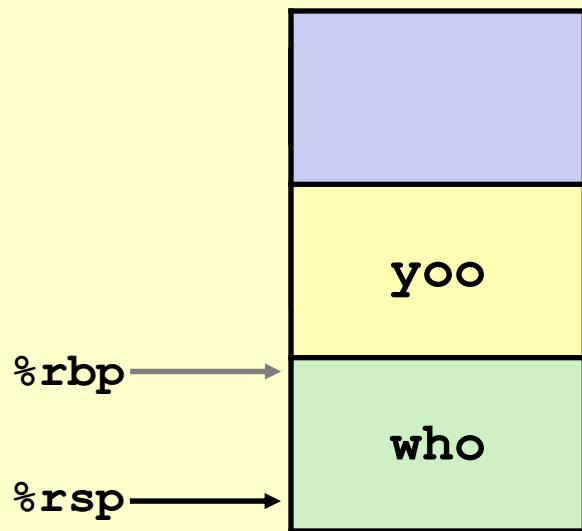


Example

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

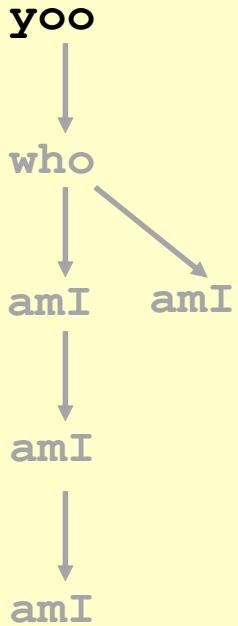
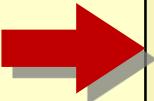


Stack

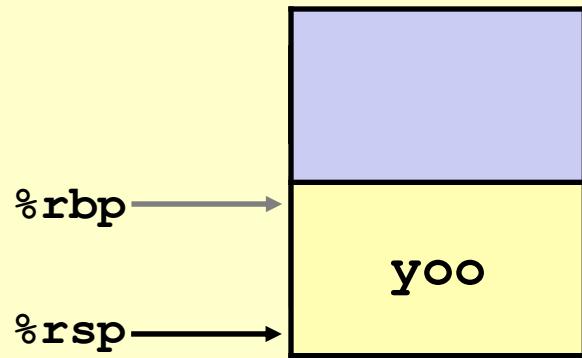


Example

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



Stack

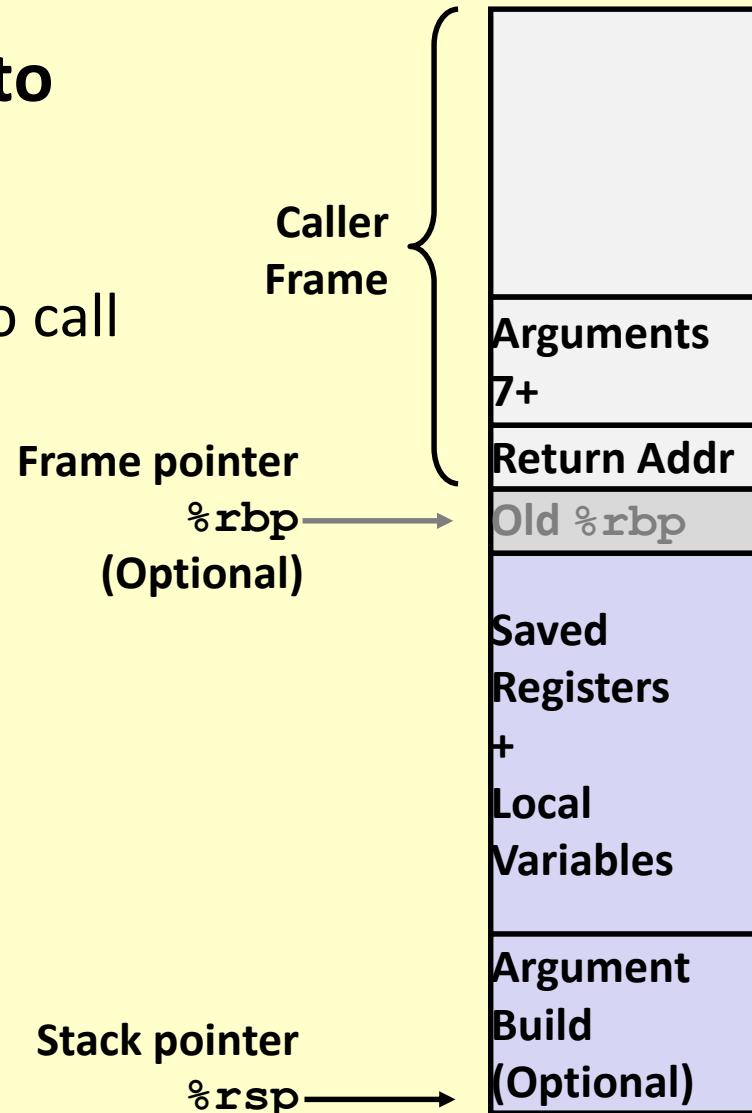


See also: Fig 3.25

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

See also: Fig 3.25

x86-64/Linux Stack Frame

■ Current Stack Frame (“Top” to Bottom)

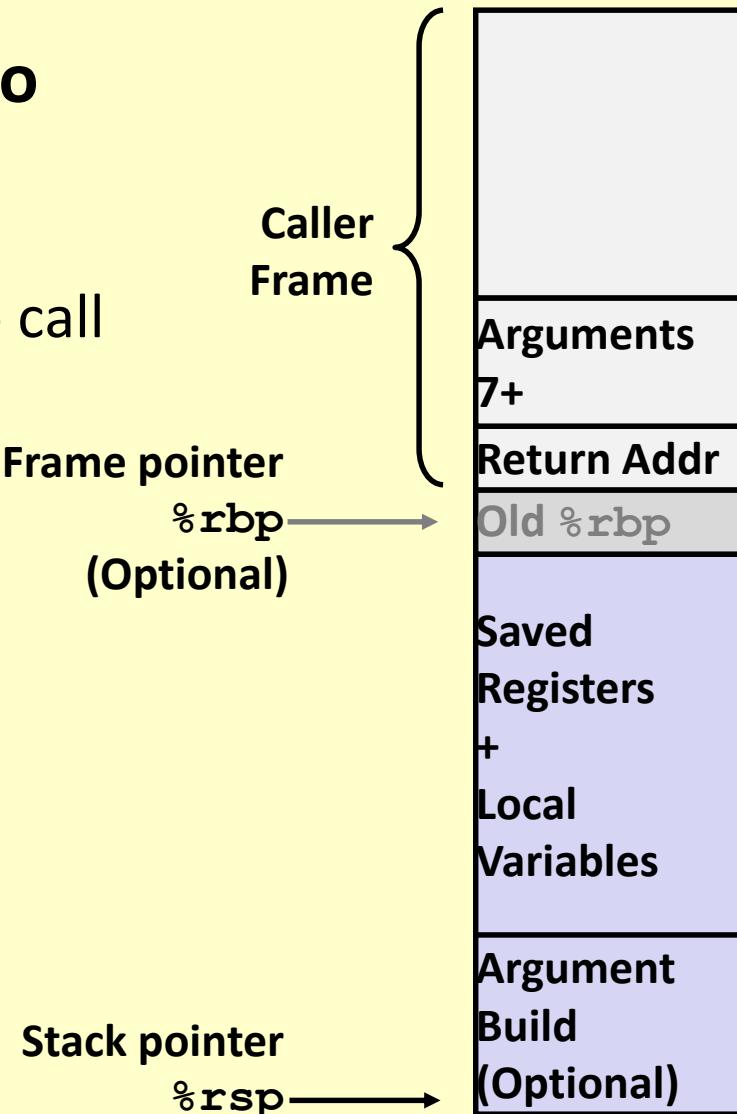
Definition:- Area on the stack bounded by %rsp (at top) and caller's stack frame (below)

out 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
```

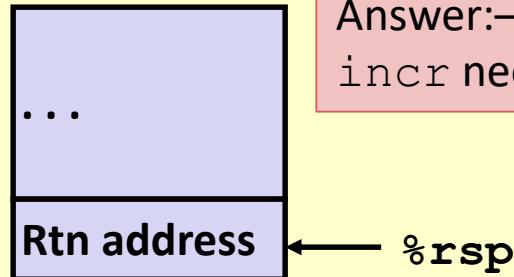
Register	Use(s)
%rdi	Argument p
%rsi	Argument val , y
%rax	x , Return value

Example: Calling `incr` #1

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

Why do we need to save
15213 on the stack?

Initial Stack Structure

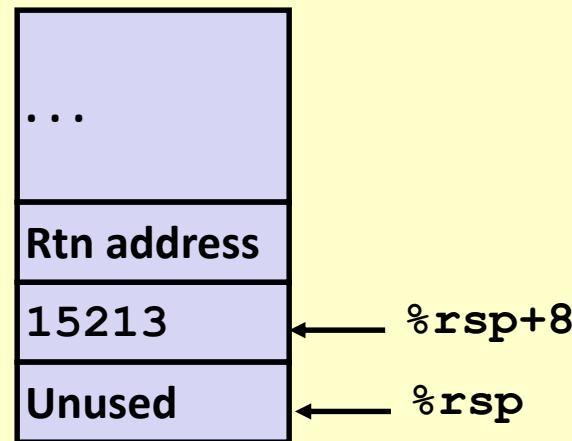


Answer:- because
`incr` needs its address

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

Can only make pointers
to things in memory!
E.g. on The Stack

Resulting Stack Structure



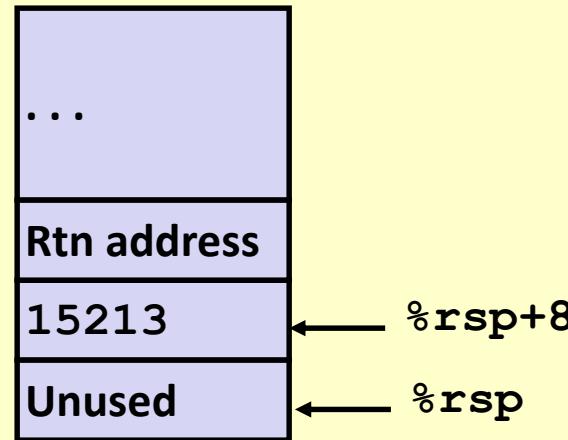
Reading Assignment: §3.7.2

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)
    movl    $3000, %esi
    leaq    8(%rsp), %rdi
    call    incr
    addq    8(%rsp), %rax
    addq    $16, %rsp
    ret
```

Stack Structure



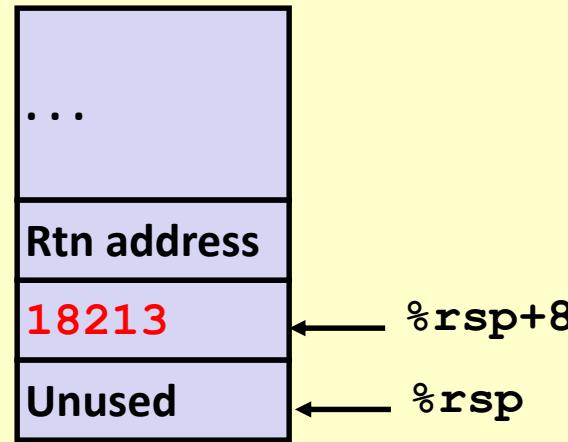
Register	Use(s)
<code>%rdi</code>	<code>&v1</code>
<code>%rsi</code>	<code>3000</code>

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)
    movl    $3000, %esi
    leaq    8(%rsp), %rdi
    call    incr
    addq    8(%rsp), %rax
    addq    $16, %rsp
    ret
```

Stack Structure

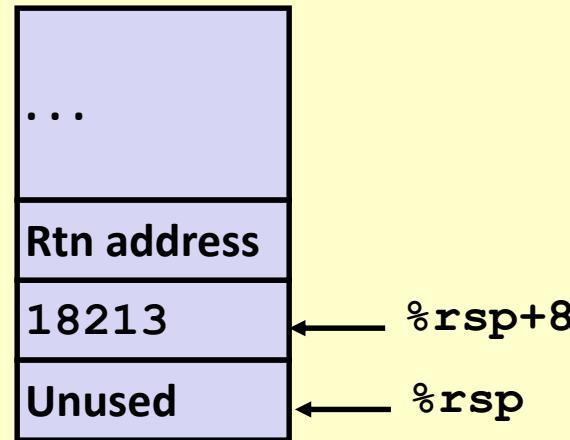


Register	Use(s)
%rdi	&v1
%rsi	3000

Example: Calling `incr` #4

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

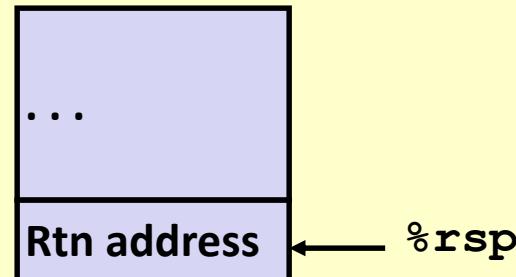
Stack Structure



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

Register	Use(s)
<code>%rax</code>	Return value

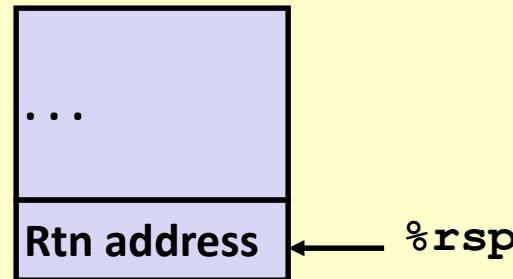
Updated Stack Structure



Example: Calling `incr` #5

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

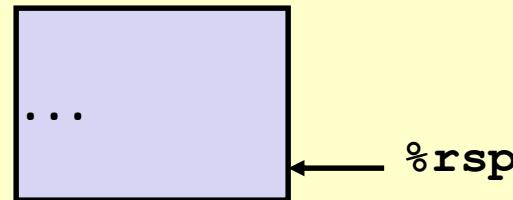
Updated Stack Structure



```
call_incr:
subq    $16, %rsp
movq    $15213, 8(%rsp)
movl    $3000, %esi
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 function `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 function `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 function
- **%rdi, ..., %r9**
 - Arguments
 - Also caller-saved
 - Can be modified by procedure
- **%r10, %r11**
 - Caller-saved
 - Can be modified by function

Return value

%rax

%rdi

%rsi

%rdx

%rcx

%r8

%r9

%r10

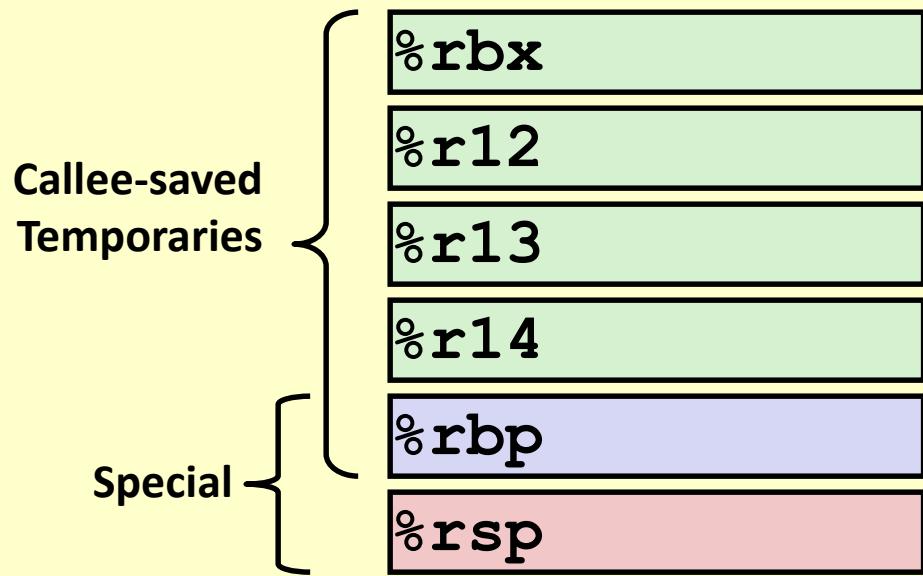
%r11

Arguments

Caller-saved
temporaries

x86-64 Linux Register Usage #2

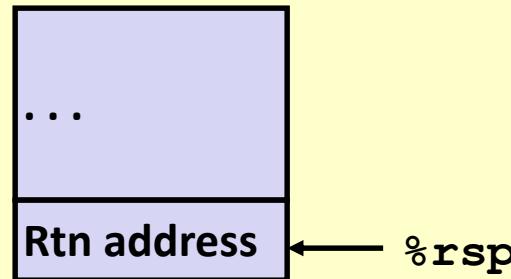
- **%rbx, %r12, %r13, %r14**
 - Callee-saved
 - Callee must save & restore
- **%rbp**
 - Callee-saved
 - Callee must save & restore
 - May be used as frame pointer
 - Can mix & match
- **%rsp**
 - Special form of callee save
 - Restored to original value upon exit from function



Callee-Saved Example #1

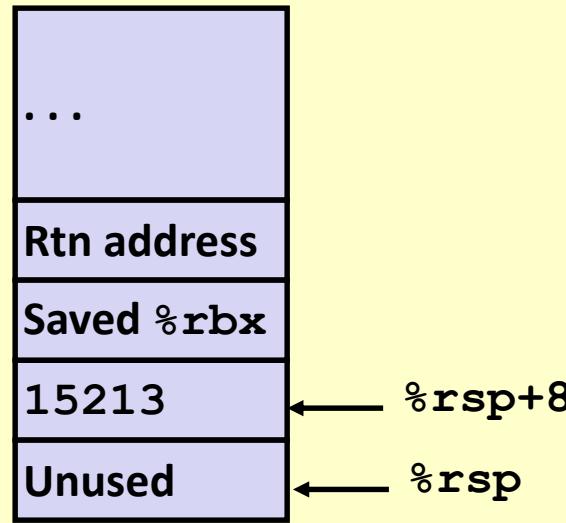
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

Initial Stack Structure



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

Resulting Stack Structure

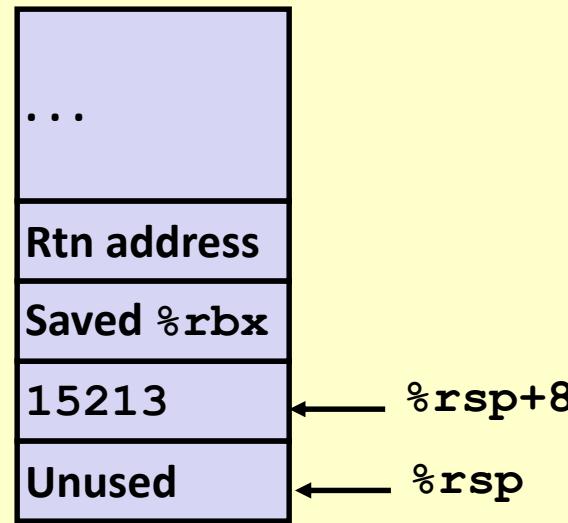


Callee-Saved Example #2

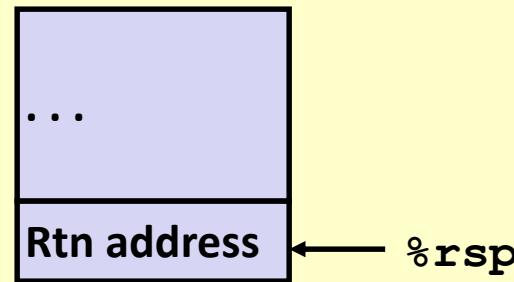
Resulting Stack Structure

```
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)
    movl $3000, %esi
    leaq 8(%rsp), %rdi
    call incr
    addq %rbx, %rax
    addq $16, %rsp
    popq %rbx
    ret
```



Pre-return Stack Structure



Today

■ Procedures Functions

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

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:

```
    movl    $0, %eax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq    %rdi, %rbx
    andl    $1, %ebx
    shrq    %rdi
    call    pcount_r
    addq    %rbx, %rax
    popq   %rbx
```

.L6:

```
    rep; ret
```

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:`

<code>movl</code>	<code>\$0, %eax</code>
<code>testq</code>	<code>%rdi, %rdi</code>
<code>je</code>	<code>.L6</code>
<code>pushq</code>	<code>%rbx</code>
<code>movq</code>	<code>%rdi, %rbx</code>
<code>andl</code>	<code>\$1, %ebx</code>
<code>shrq</code>	<code>%rdi</code>
<code>call</code>	<code>pcount_r</code>
<code>addq</code>	<code>%rbx, %rax</code>
<code>popq</code>	<code>%rbx</code>

`.L6:`

`rep; ret`

Register	Use(s)	Type
<code>%rdi</code>	<code>x</code>	Argument
<code>%rax</code>	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);
}
```

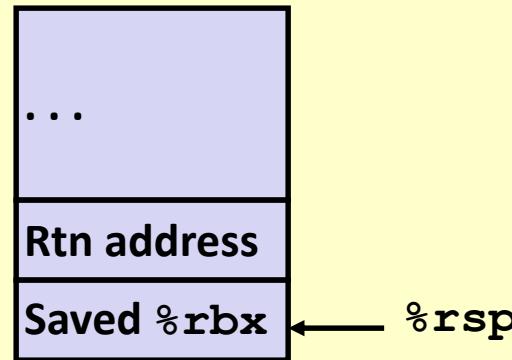
`pcount_r:`

<code>movl</code>	<code>\$0, %eax</code>
<code>testq</code>	<code>%rdi, %rdi</code>
<code>je</code>	<code>.L6</code>
<code>pushq</code>	<code>%rbx</code>
<code>movq</code>	<code>%rdi, %rbx</code>
<code>andl</code>	<code>\$1, %ebx</code>
<code>shrq</code>	<code>%rdi</code>
<code>call</code>	<code>pcount_r</code>
<code>addq</code>	<code>%rbx, %rax</code>
<code>popq</code>	<code>%rbx</code>

`.L6:`

`rep; ret`

Register	Use(s)	Type
<code>%rdi</code>	<code>x</code>	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:

movl	\$0, %eax
testq	%rdi, %rdi
je	.L6
pushq	%rbx
movq	%rdi, %rbx
andl	\$1, %ebx
shrq	%rdi
call	pcount_r
addq	%rbx, %rax
popq	%rbx

.L6:

rep; 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:

movl	\$0, %eax
testq	%rdi, %rdi
je	.L6
pushq	%rbx
movq	%rdi, %rbx
andl	\$1, %ebx
shrq	%rdi
call	pcount_r
addq	%rbx, %rax
popq	%rbx

.L6:

rep; 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:

movl	\$0, %eax
testq	%rdi, %rdi
je	.L6
pushq	%rbx
movq	%rdi, %rbx
andl	\$1, %ebx
shrq	%rdi
call	pcount_r
addq	%rbx, %rax
popq	%rbx

.L6:

rep; 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);
}
```

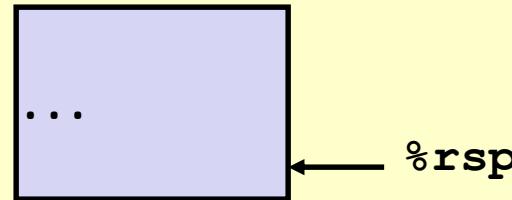
`pcount_r:`

```
    movl    $0, %eax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq   %rdi, %rbx
    andl   $1, %ebx
    shrq   %rdi
    call   pcount_r
    addq   %rbx, %rax
    popq   %rbx
```

`.L6:`

`rep; ret`

Register	Use(s)	Type
<code>%rax</code>	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

Multiple Threads

■ From OS course:-

- Process — a running program with its own address space, stack, etc.
- Thread — an independently executing function in the same address space as other threads.
 - Requires own stack
 - Shares all other variables
 - Pointers valid across threads

Threads and Stacks

- Stack discipline makes it possible for multiple threads to execute same function *independently*
 - Concurrently
- Each thread has own stack pointer
 - Separately executing threads use different stack frames on own stacks!

x86-64 Procedure Function Summary

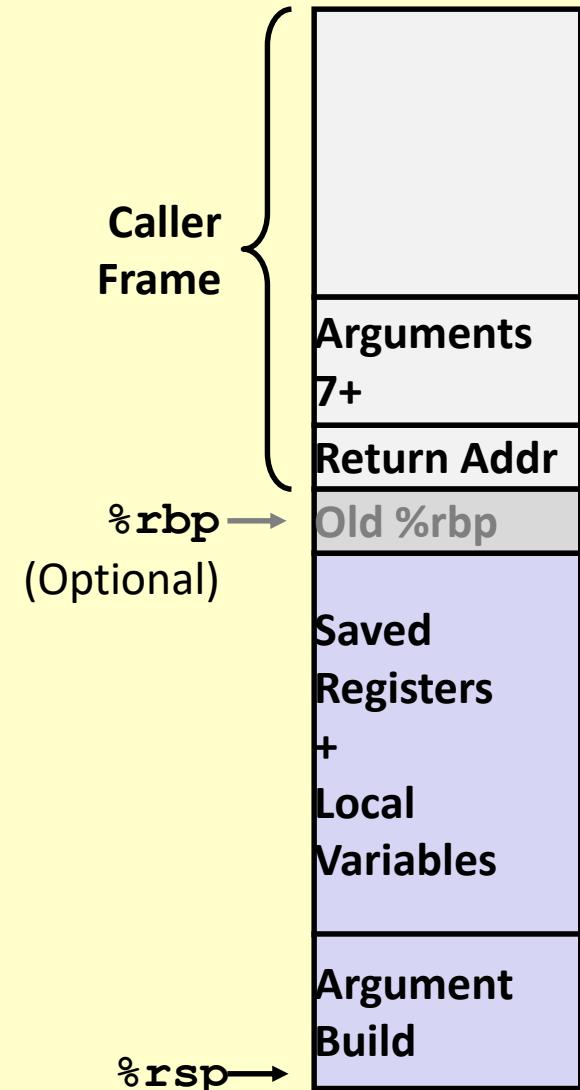
■ Important Points

- 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



Questions?

Machine-Level Programming V: Advanced Topics

Professor Hugh C. Lauer
CS-2011, Machine Organization and Assembly Language

(Slides include copyright materials from *Computer Systems: A Programmer's Perspective*, by Bryant and O'Hallaron, and from *The C Programming Language*, by Kernighan and Ritchie)

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

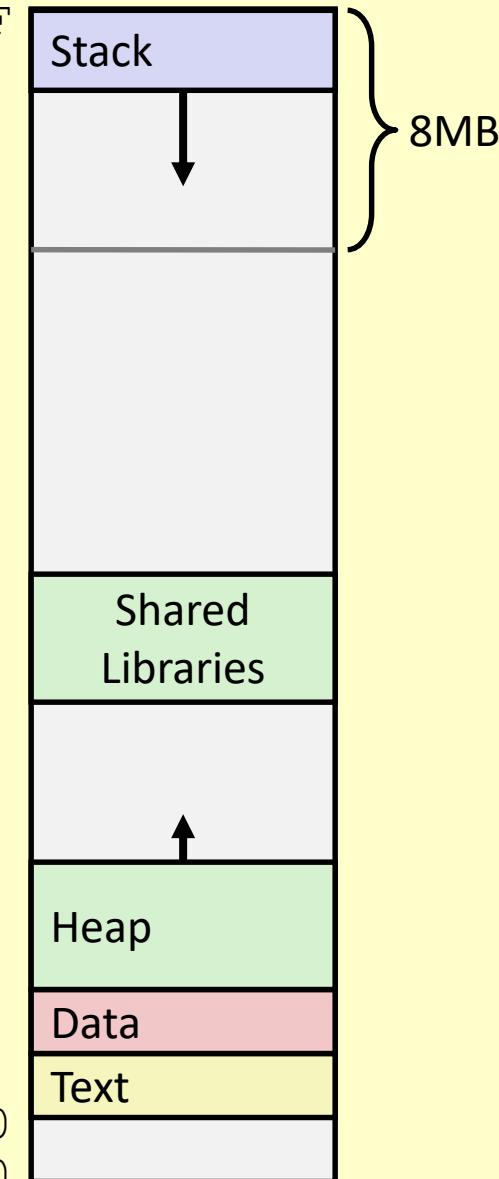
■ Text / Shared Libraries

- Executable machine instructions
- Read-only

00007FFFFFFFFF

Hex Address

400000
000000



not drawn to scale

Memory Allocation Example

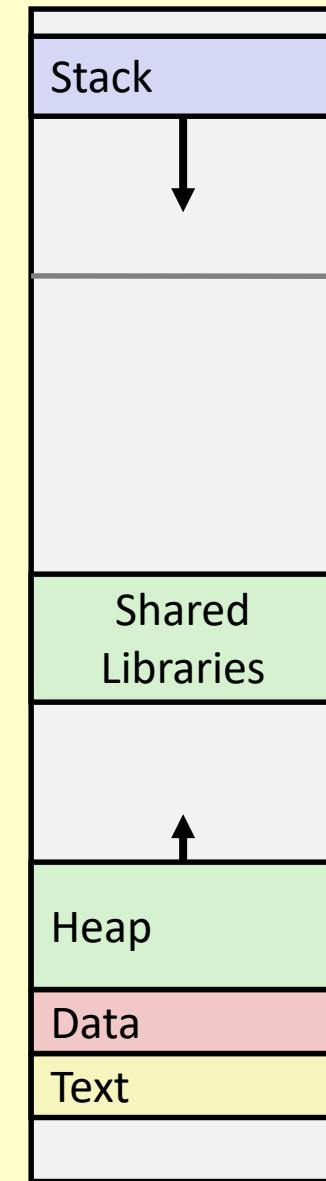
```
char big_array[1L<<24]; /* 16 MB */
char huge_array[1L<<31]; /* 2 GB */

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 ... */
}
```

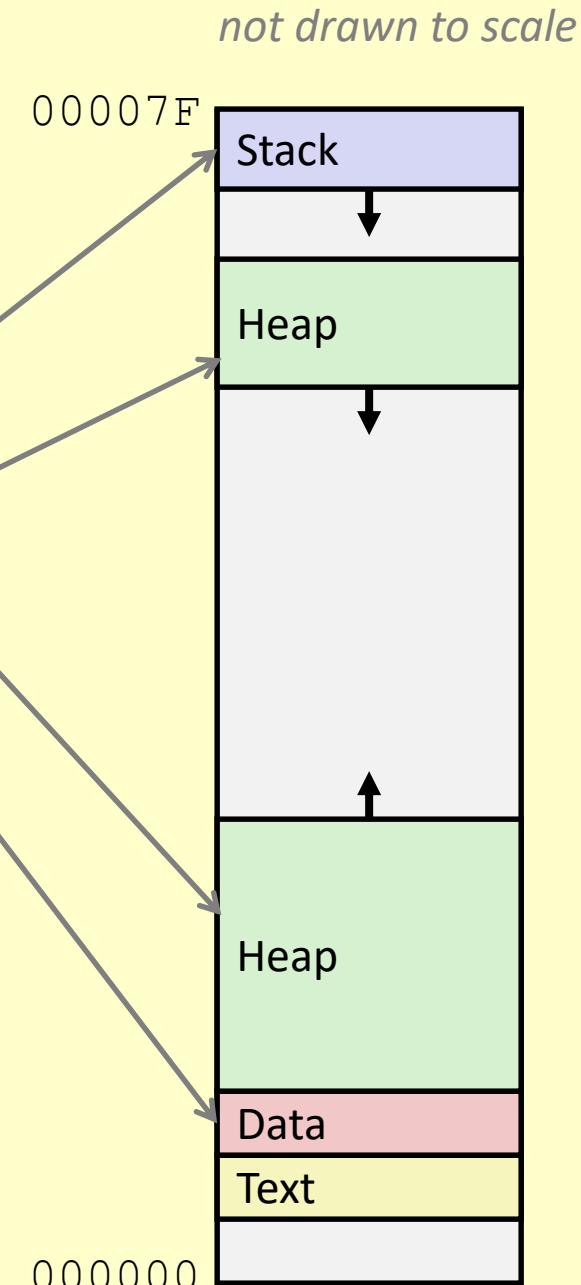
Where does everything go?



x86-64 Example Addresses

address range ~ 2^{47}

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



Reading Assignment: §3.10.3

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;
}
```

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

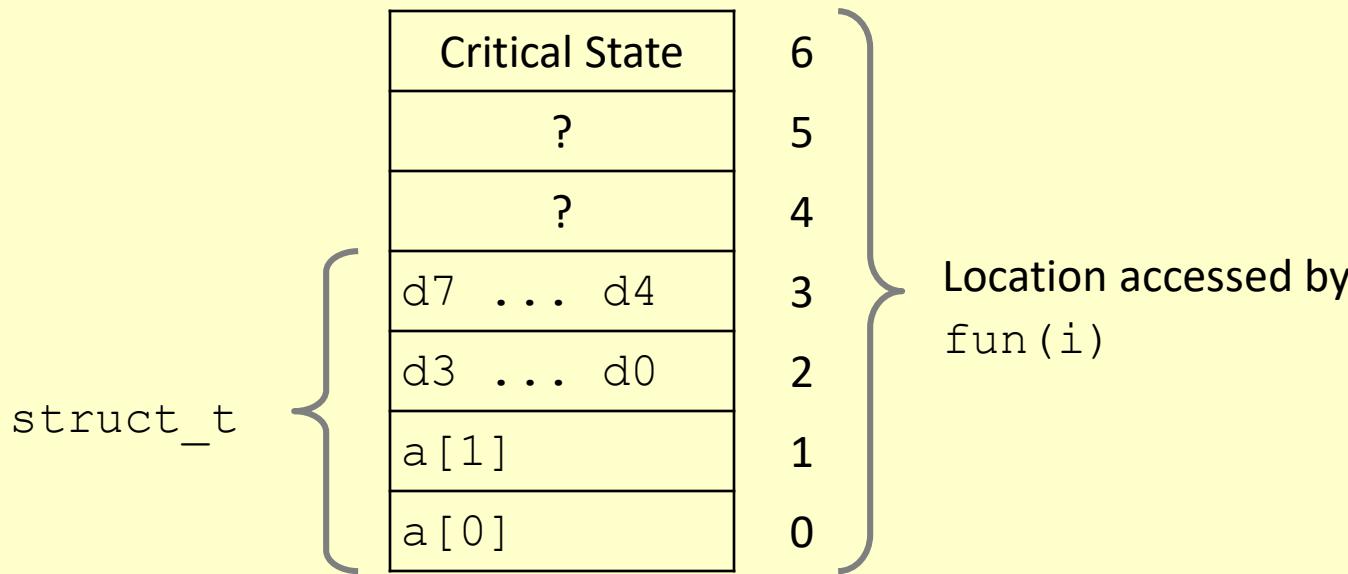
- Result is system specific

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:

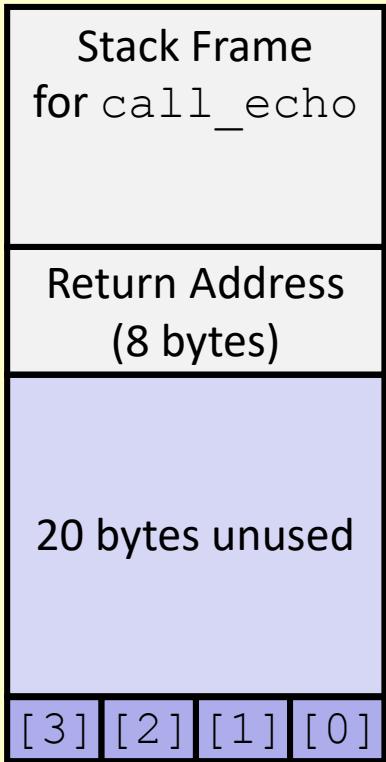
```
00000000004006cf <echo>:  
 4006cf: 48 83 ec 18          sub    $0x18,%rsp  
 4006d3: 48 89 e7          mov    %rsp,%rdi  
 4006d6: e8 a5 ff ff ff      callq   400680 <gets>  
 4006db: 48 89 e7          mov    %rsp,%rdi  
 4006de: e8 3d fe ff ff      callq   400520 <puts@plt>  
 4006e3: 48 83 c4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

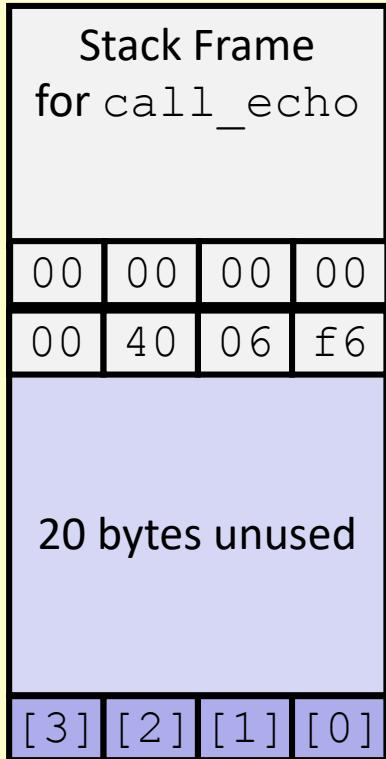


```
/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
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 call_echo			
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

<pre>void echo() { char buf[4]; gets(buf); ... }</pre>	<pre>echo: subq \$24, %rsp movq %rsp, %rdi call gets ... </pre>
--	---

call_echo:

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

buf ← %rsp

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

Overflowed buffer, but did not corrupt state

Buffer Overflow Stack Example #2

After call to gets

Stack Frame for call_echo			
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 ← %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 call_echo			
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

<pre>void echo() { char buf[4]; gets(buf); ... }</pre>	<pre>echo: subq \$24, %rsp movq %rsp, %rdi call gets ... </pre>
--	---

call_echo:

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

buf ← %rsp

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

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

Buffer Overflow Stack Example #3 Explained

After call to gets

Stack Frame for call_echo			
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

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

```

buf ← %rsp

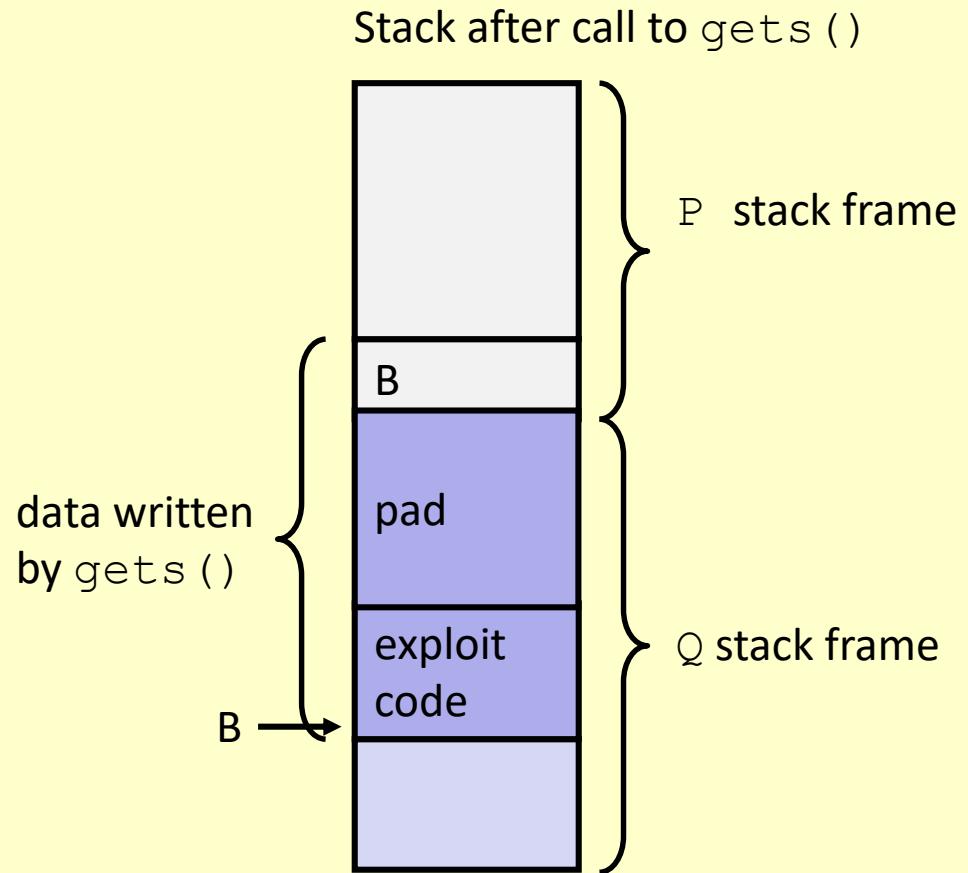
- “Returns” to unrelated code
- Lots of things happen, without modifying critical state
- Eventually executes `retq` back to main

Code Injection Attacks

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

return address
A

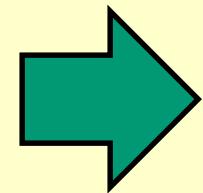
```
int Q() {
    char buf[64];
    gets(buf);
    ...
    return ...;
}
```



- 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 Attack Lab
 - Hopefully to convince you to never leave such holes in your programs!!



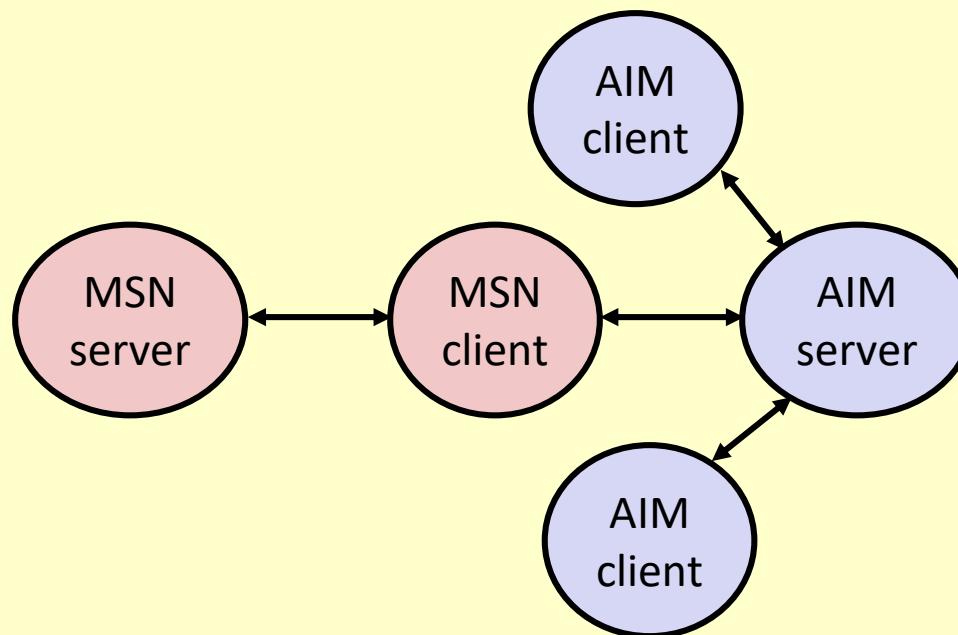
Example: the original Internet worm (1988)

- **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 nefariously compromising peoples' security.

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

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

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

Ken Thompson, 1984 Turing Award Lecture, "Reflections on Trusting Trust," CACM, August 1984, pp. 761-763

■ Both are (usually) designed to spread among computers and to wreak havoc

OK, what to do about buffer overflow attacks

- Avoid overflow vulnerabilities
- Employ system-level protections
- Have compiler use “stack canaries”
- Lets talk about each...

1. Avoid Overflow Vulnerabilities in Code (!)

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

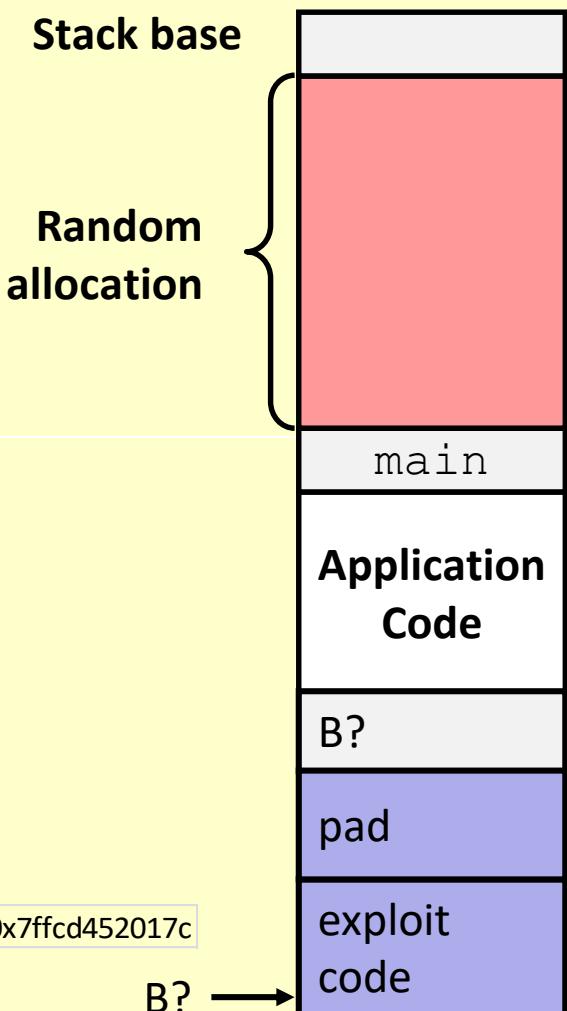
- For example, use library routines that limit string lengths
 - **fgets** instead of **gets**
 - **strncpy** instead of **strcpy**
 - Don't use **scanf** with **%s** conversion specification
 - Use **fgets** to read the string
 - Or use **%ns** where **n** is a suitable integer

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
 - Stack repositioned each time program executes

local	0x7ffe4d3be87c	0x7fff75a4f9fc	0x7ffeadb7c80c	0x7ffeaea2fdac	0x7ffcd452017c
-------	----------------	----------------	----------------	----------------	----------------

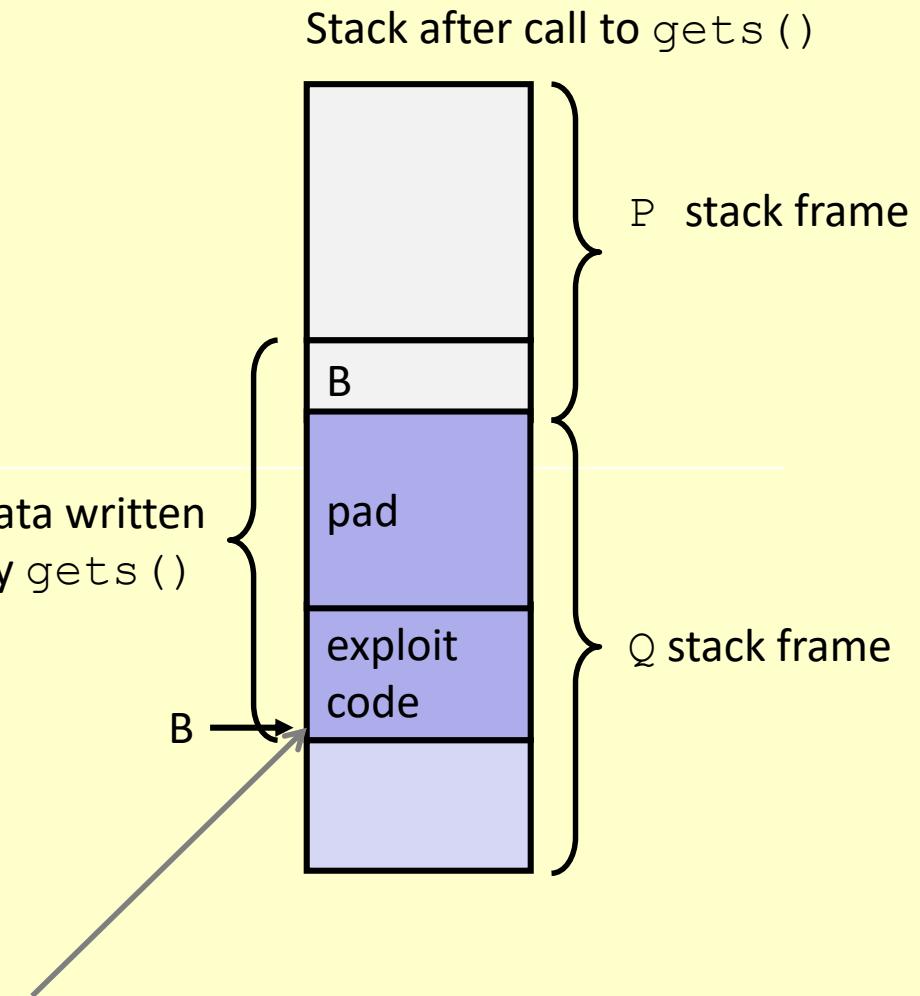


2. System-Level Protections can help

■ 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 in older versions of gcc)

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

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

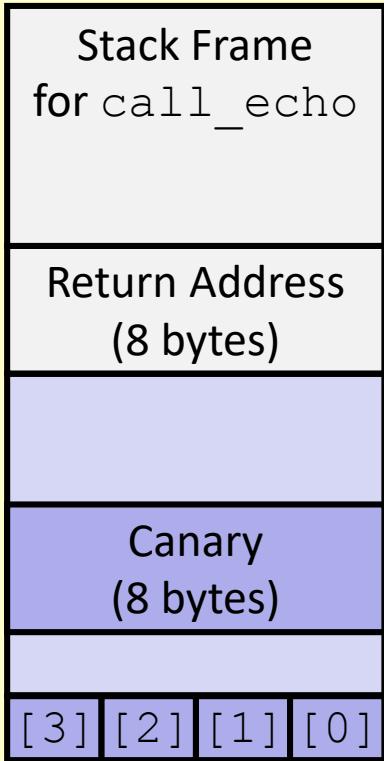
Protected Buffer Disassembly

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



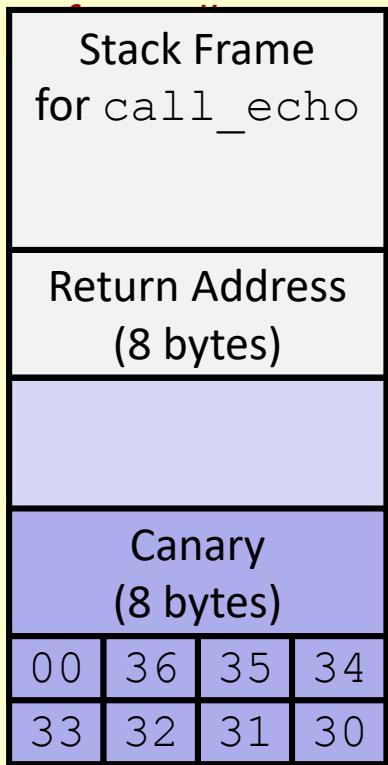
```
/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    gets(buf);
    puts(buf);
}
```

buf ← %rsp

```
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 non-executable 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;  
}
```

```
0000000004004d0 <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
```

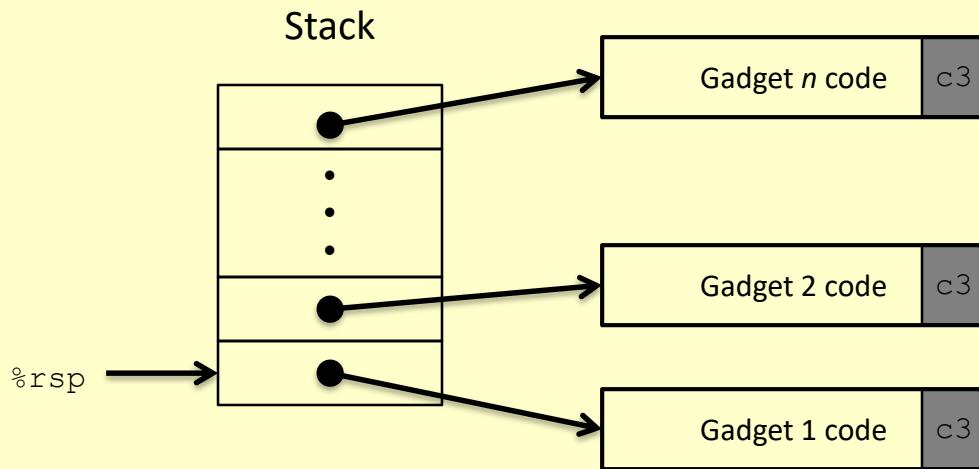
Encodes `movq %rax, %rdi`

`rdi ← rax`

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

Reading Assignment: § 3.9

Today

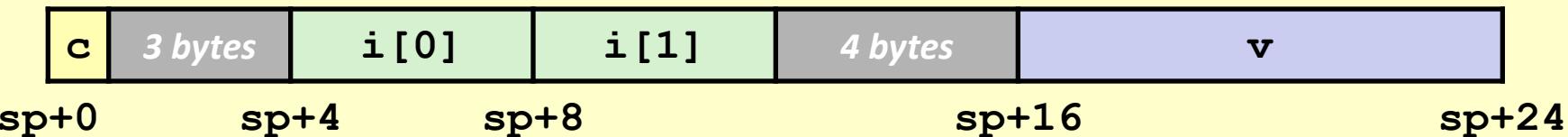
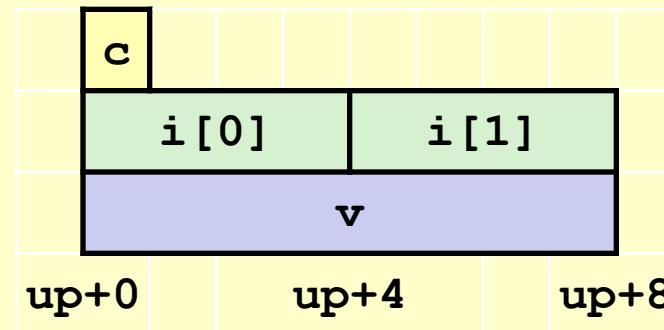
- Memory Layout
- Buffer Overflow
 - Vulnerability
 - Protection
- Unions (and structures again)

Union Allocation

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

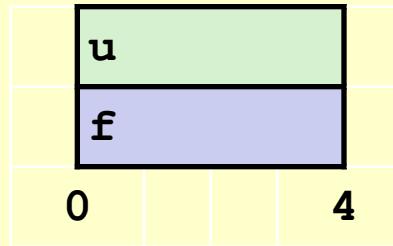
```
union U1 {
    char c;
    int i[2];
    double v;
} *up;
```

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



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;
}
```

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

Same as (`float`) `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]);

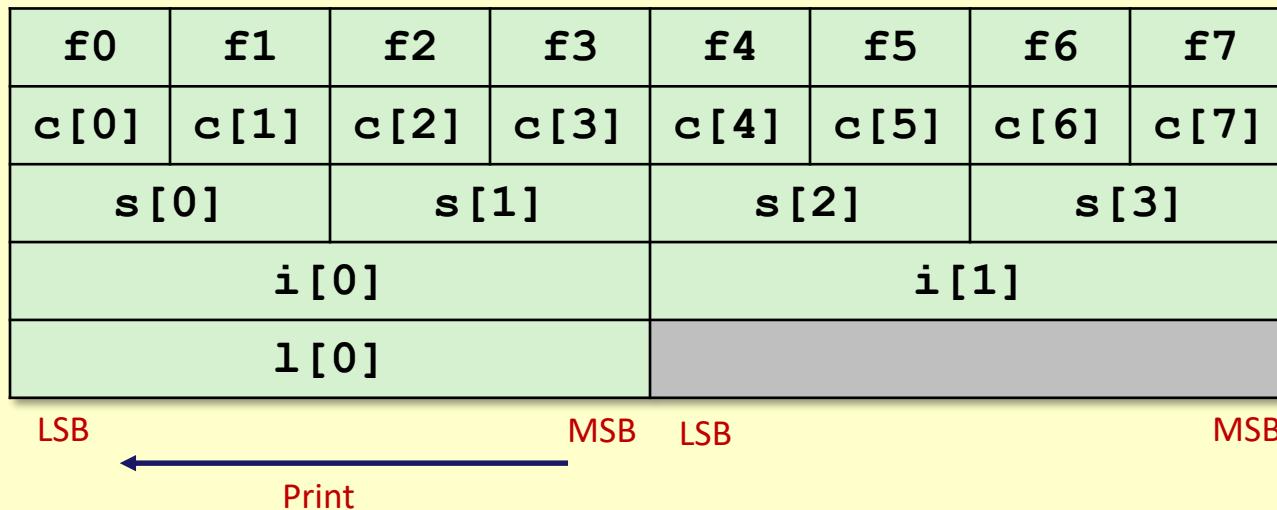
printf("Shorts 0-3 == [0x%x,0x%x,0x%x,0x%x]\n",
       dw.s[0], dw.s[1], dw.s[2], dw.s[3]);

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

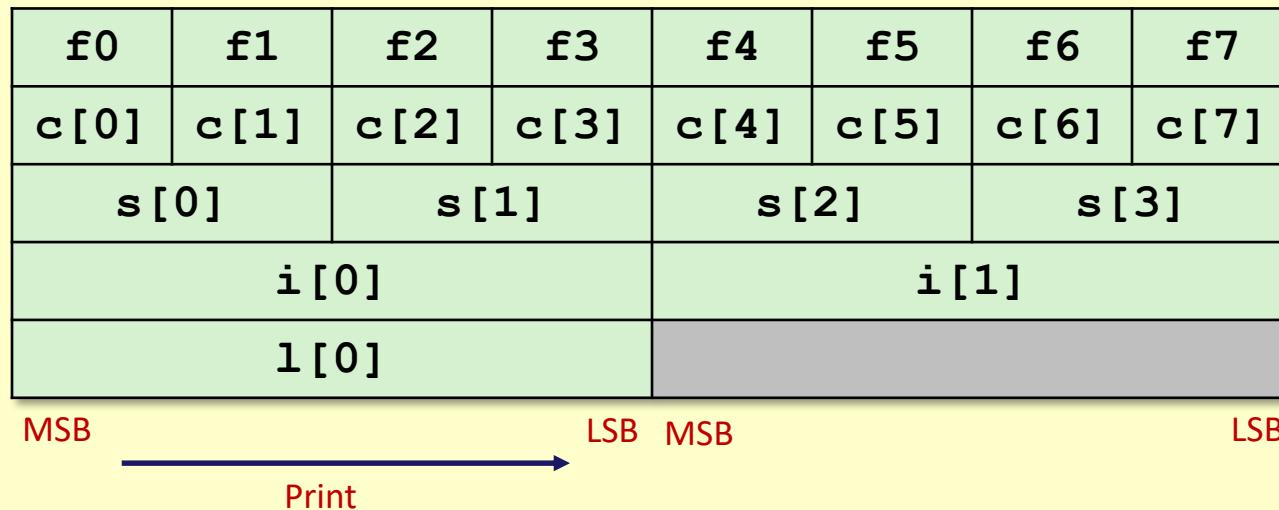


Output:

```
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

Big Endian

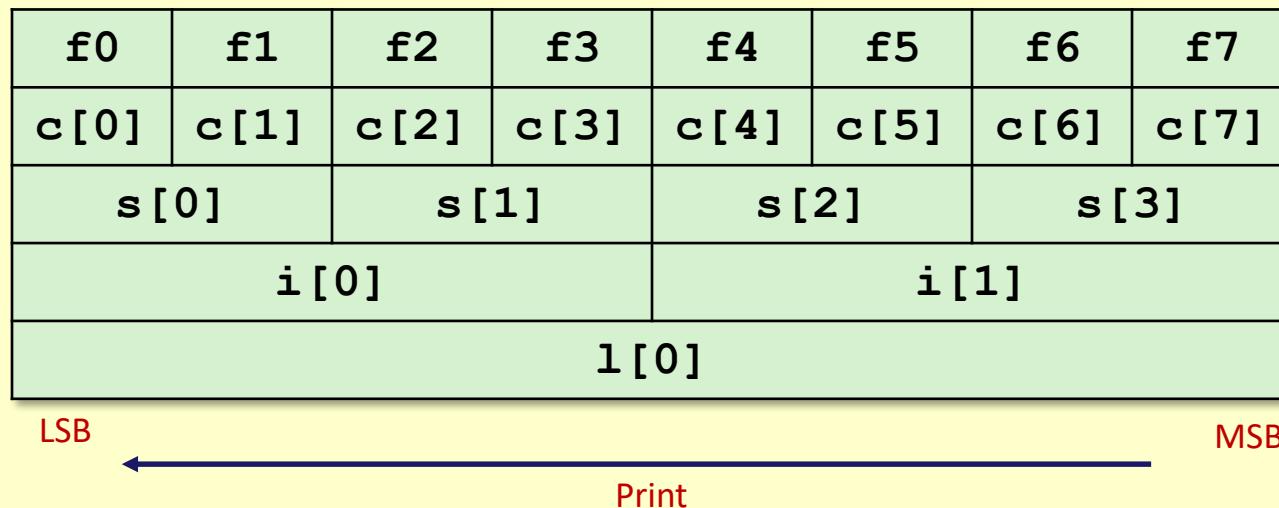


Output on Sun:

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

Little Endian



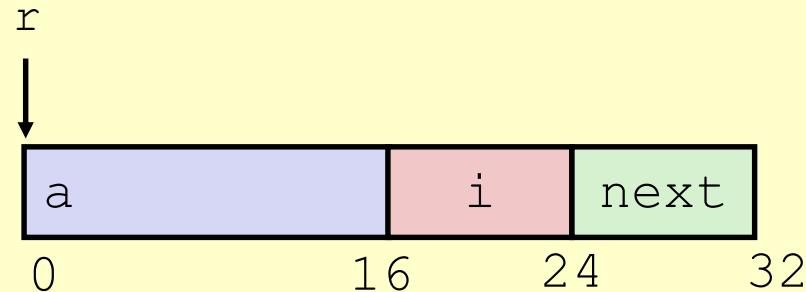
Output on x86-64:

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]

Questions?

Structure Representation (again)

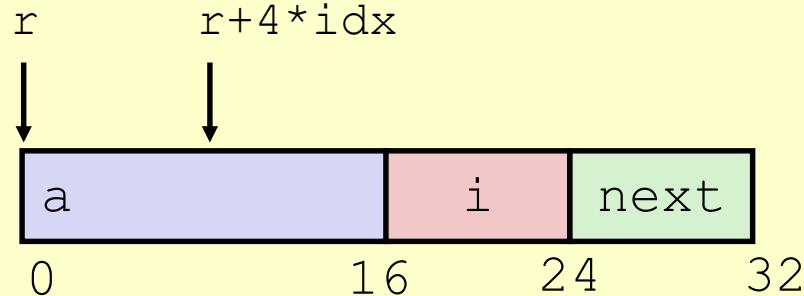
```
struct rec {  
    int a[4];  
    size_t i;  
    struct rec *next;  
};
```



- 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];
}
```

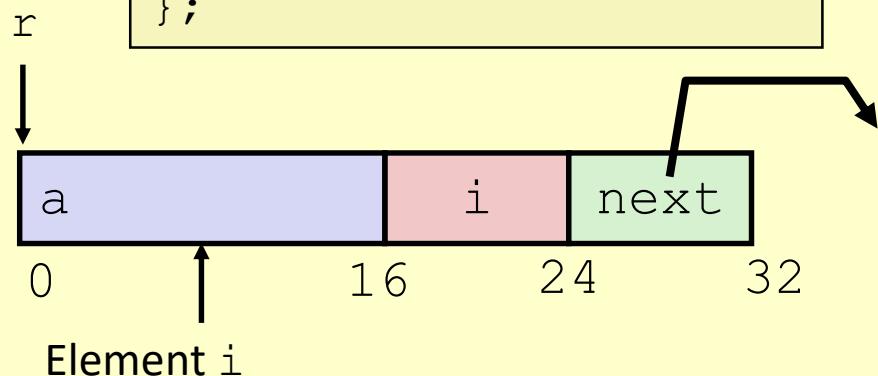
```
# r in %rdi, idx in %rsi
leaq (%rdi,%rsi,4), %rax
ret
```

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;
};
```

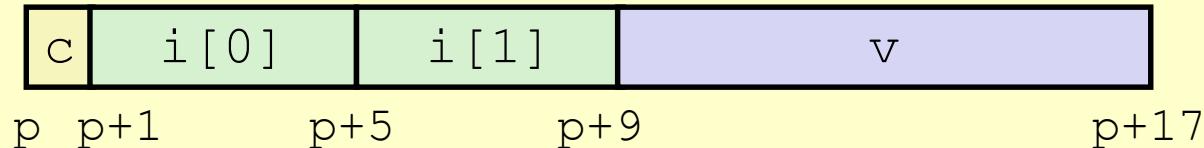


Register	Value
<code>%rdi</code>	<code>r</code>
<code>%rsi</code>	<code>val</code>

```
.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

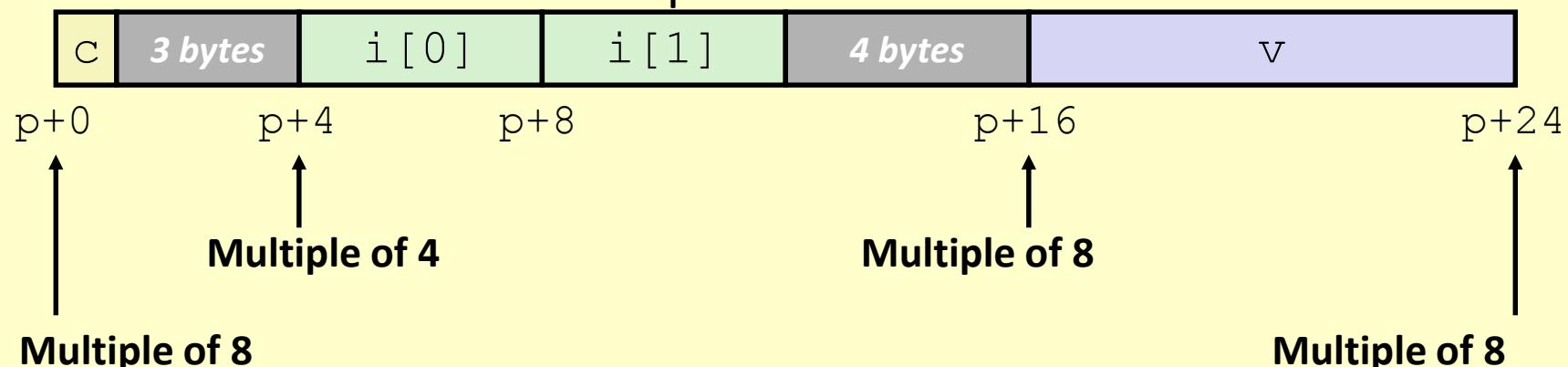
■ Unaligned Data



```
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

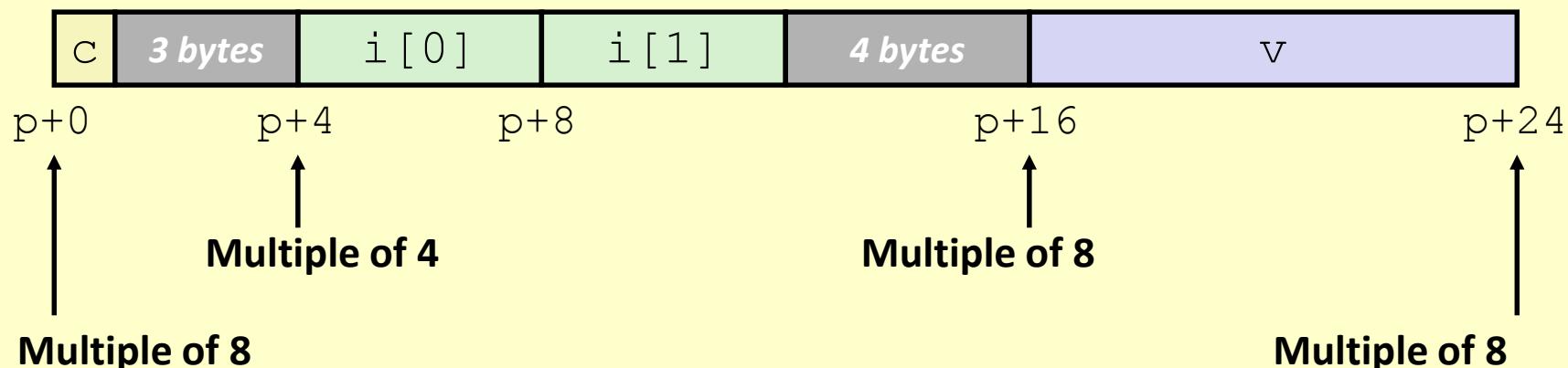
■ Overall structure placement

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

■ Example:

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

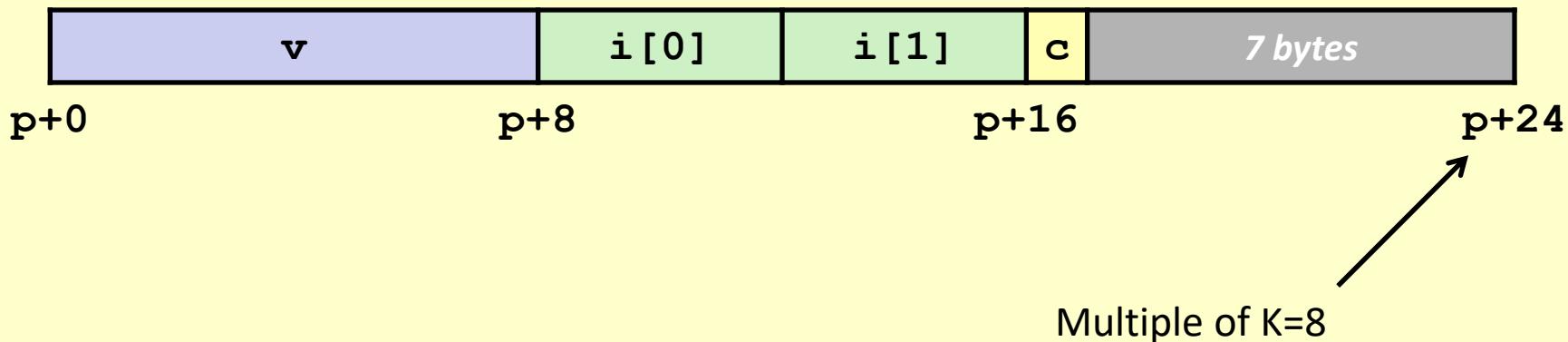
```
struct S1 {
    char c;
    int i[2];
    double v;
} *p;
```



Meeting Overall Alignment Requirement

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

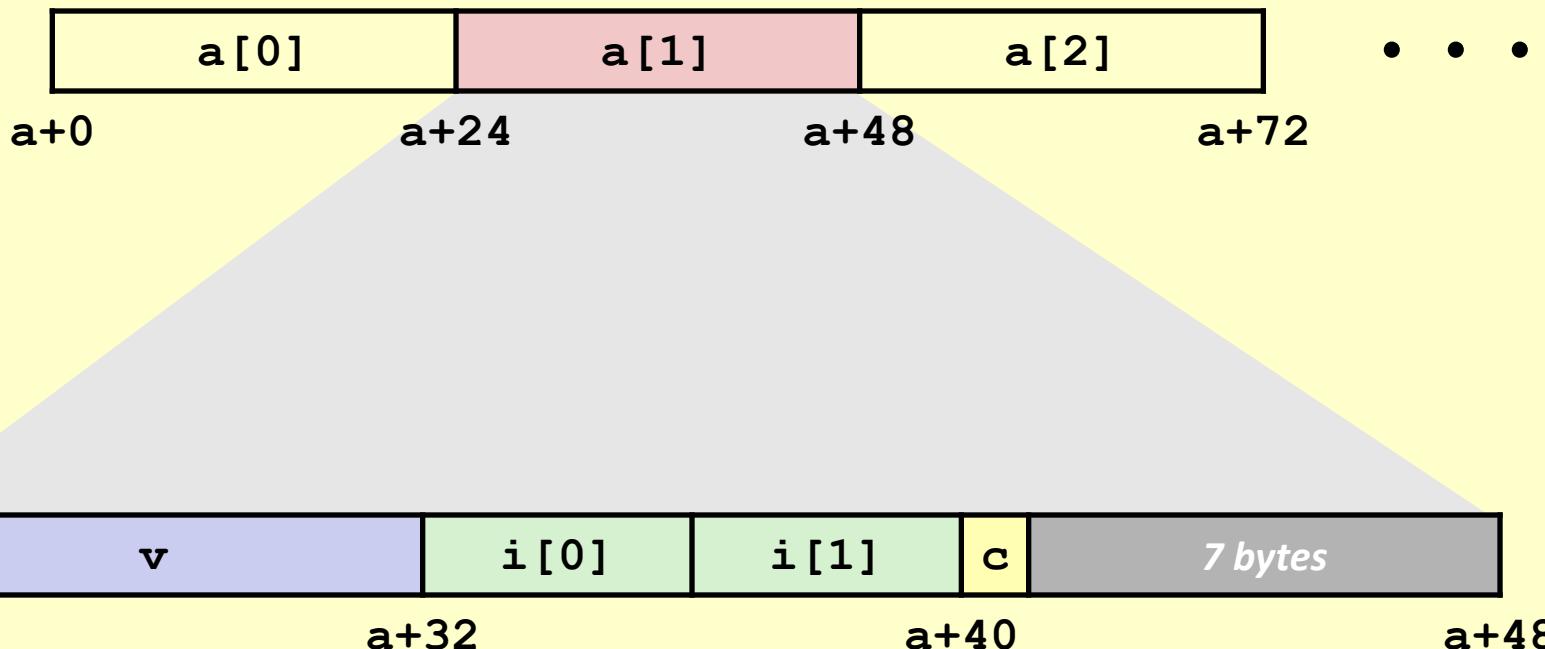
```
struct S2 {  
    double v;  
    int i[2];  
    char c;  
} *p;
```



Arrays of Structures

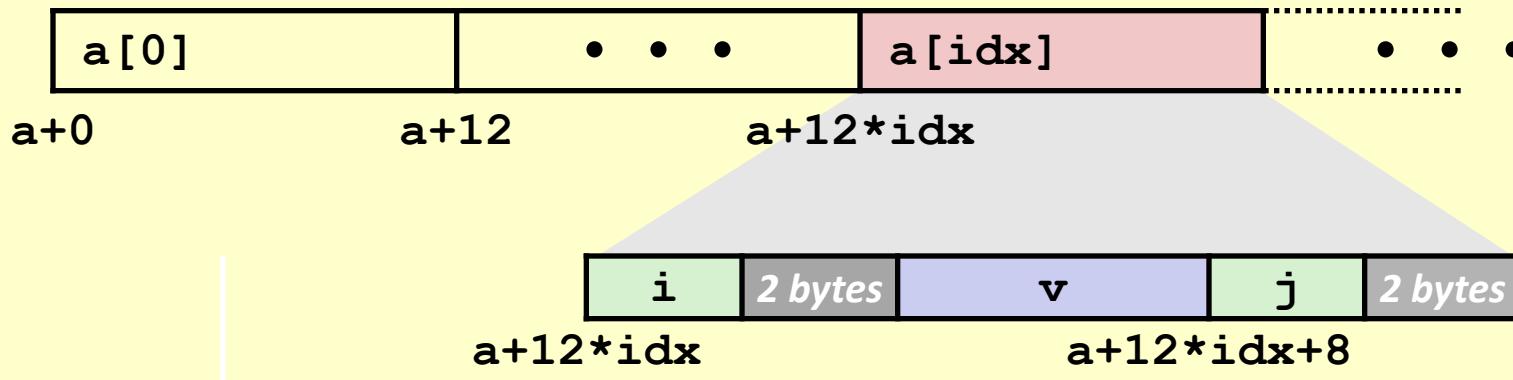
- Overall structure length multiple of K
- Satisfy alignment requirement for every element

```
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



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

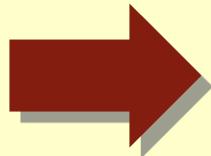
```
# %rdi = idx
leaq (%rdi,%rdi,2),%rax # 3*idx
movzwl a+8(%rax,4),%eax
```

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

Saving Space

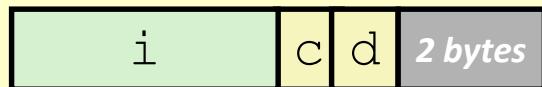
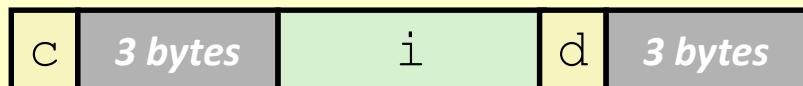
■ Put large data types first

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



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

■ Effect (K=4)



Summary 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

Questions?

Cache Memories

Professor Hugh C. Lauer
CS-2011, Machine Organization and Assembly Language

(Slides include copyright materials from *Computer Systems: A Programmer's Perspective*, by Bryant and O'Hallaron, and from *The C Programming Language*, by Kernighan and Ritchie)

Today

- Cache memory organization and operation

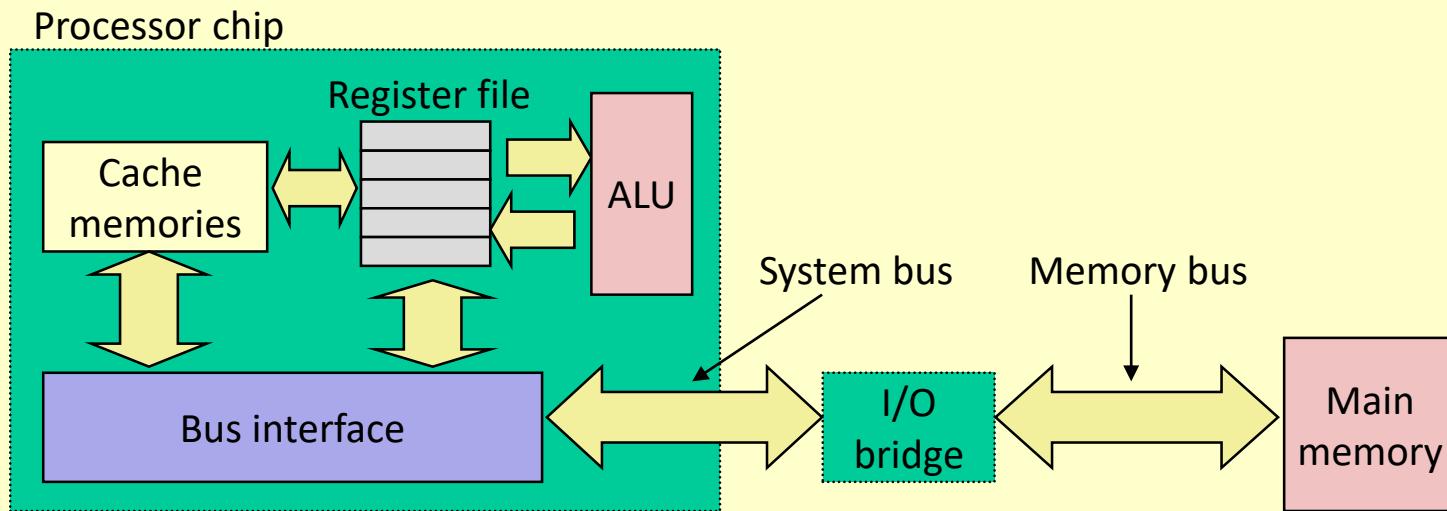
- Performance impact of caches

- The memory mountain
- Rearranging loops to improve spatial locality
- Using blocking to improve temporal locality

Reading Assignment: §6.1 – §6.5

Cache Memories in Processors

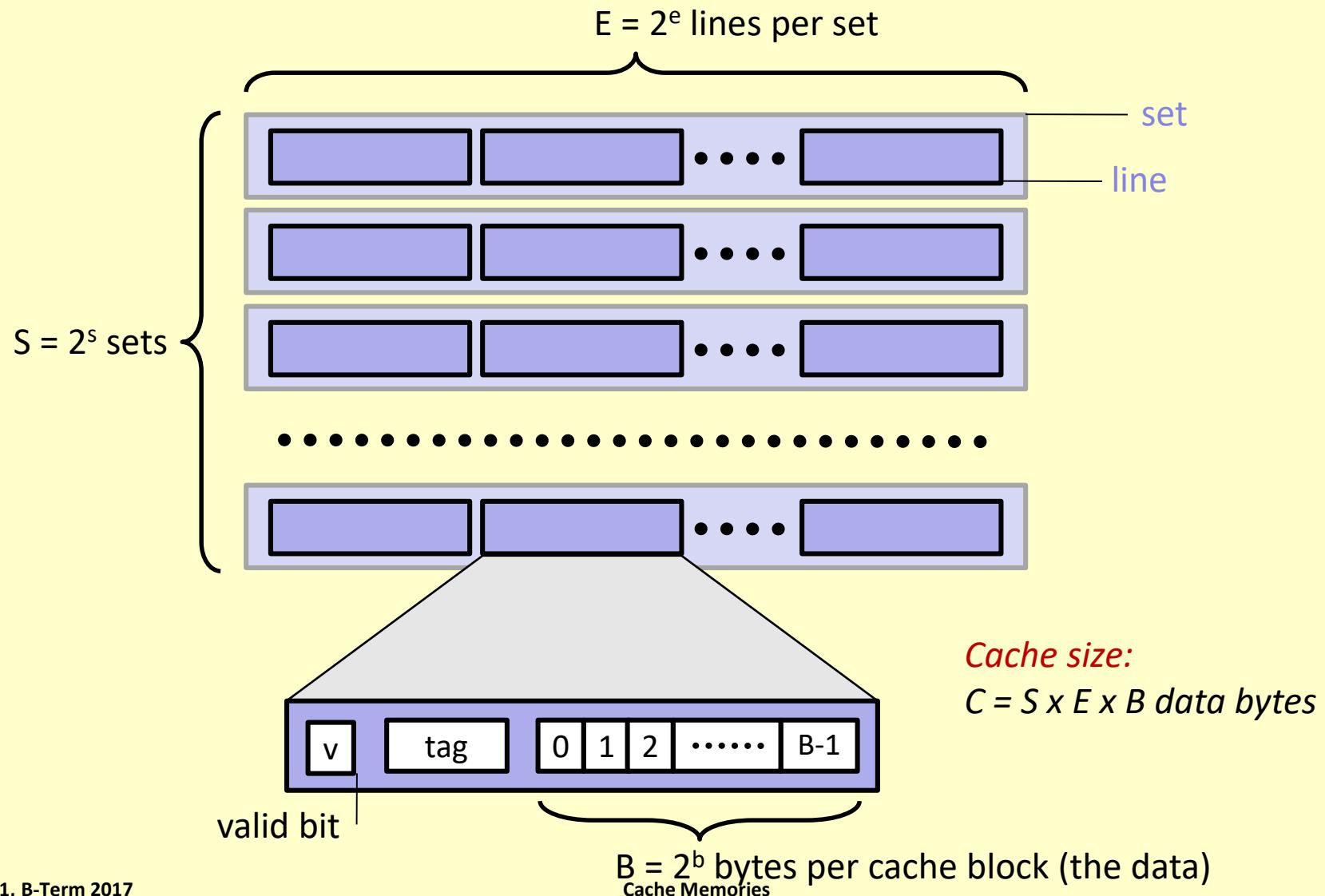
- Cache memories are small, fast SRAM-based memories managed automatically in hardware.
 - Hold frequently accessed blocks of main memory
- Processor looks first for data in caches (e.g., L1, L2, and L3), then in main memory.
- Typical system structure:



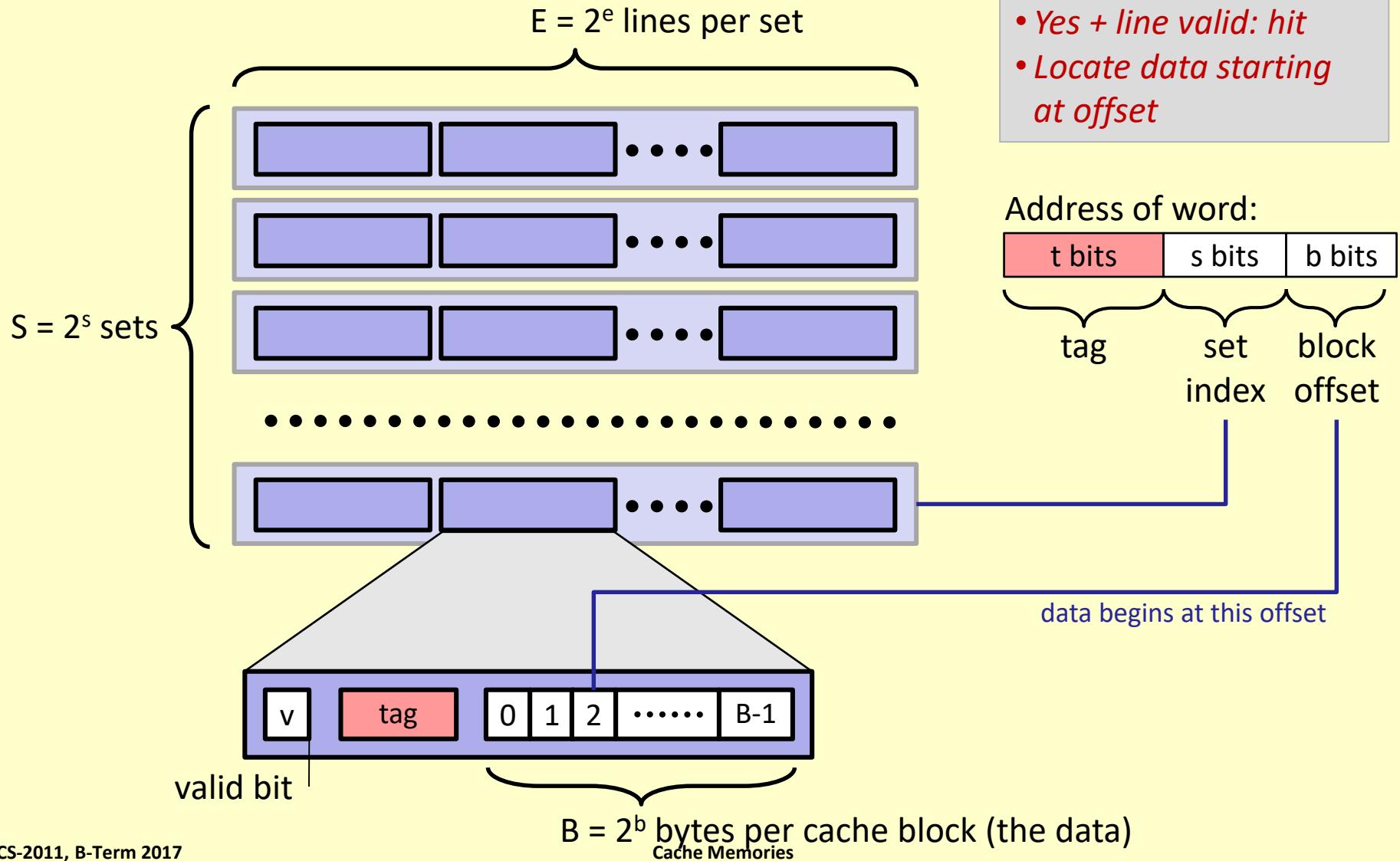
Caching issues

- **How do you know whether an item is in the cache or not?**
 - I.e., how do you find it quickly?
- **What do you do when the item you want is not in the cache**
 - And how do you make space for it?

Cache Organization (S, E, B)



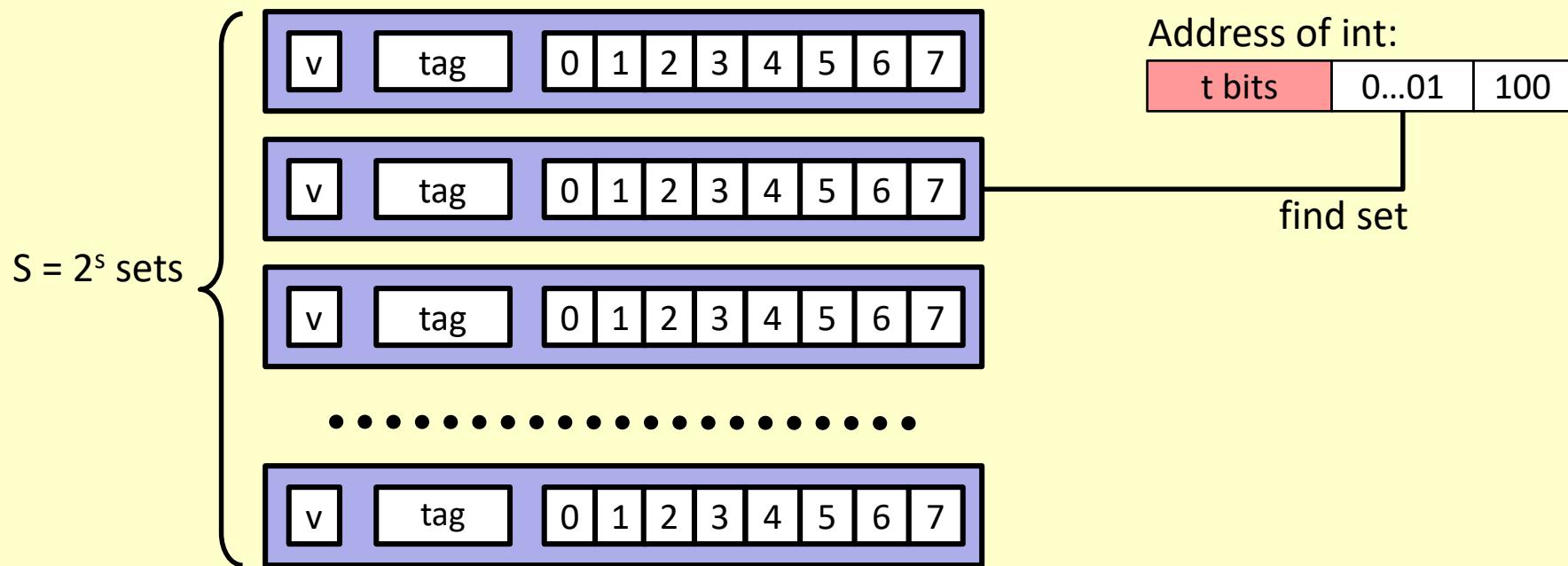
Cache Read



Example: Direct Mapped Cache ($E = 1$)

Direct mapped: One line per set

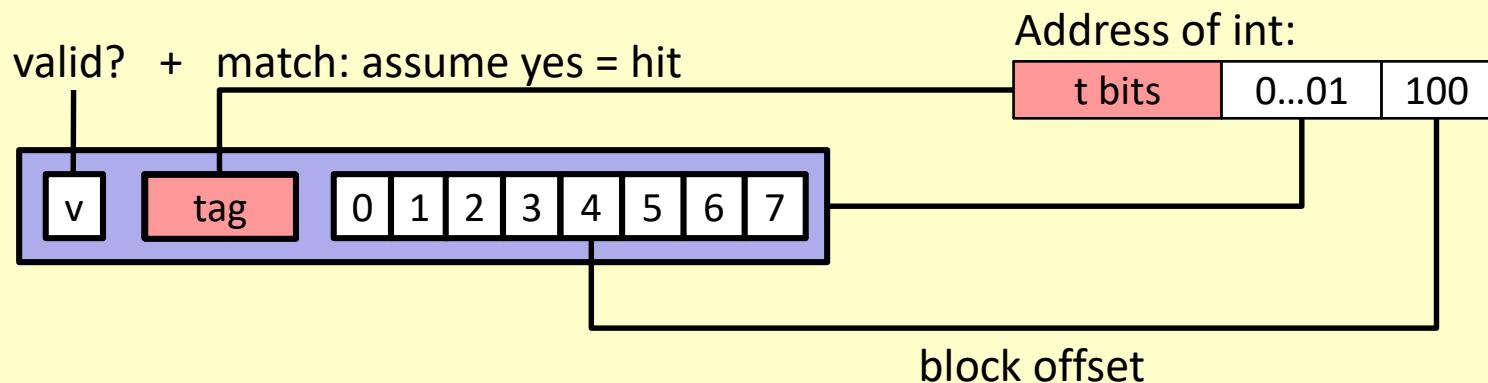
Assume: cache block size 8 bytes



Example: Direct Mapped Cache ($E = 1$)

Direct mapped: One line per set

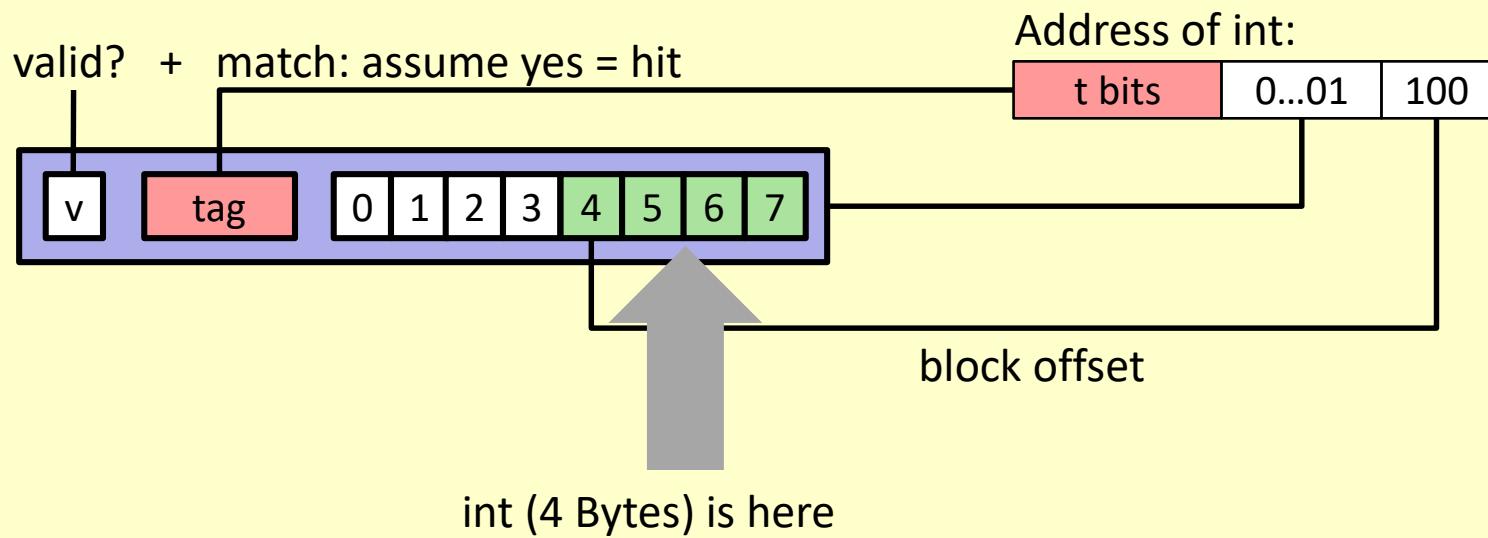
Assume: cache block size 8 bytes



Example: Direct Mapped Cache ($E = 1$)

Direct mapped: One line per set

Assume: cache block size 8 bytes



No match: old line is evicted and replaced

Direct-Mapped Cache Simulation

$t=1 \quad s=2 \quad b=1$

x	xx	x
---	----	---

$M=16$ byte addresses, $B=2$ bytes/block,
 $S=4$ sets, $E=1$ Blocks/set

Address trace (reads, one byte per read):

0	[0000] ₂ ,	miss
1	[0001] ₂ ,	hit
7	[0111] ₂ ,	miss
8	[1000] ₂ ,	miss
0	[0000] ₂	miss

	v	Tag	Block
Set 0	1	0	$M[0-1]$
Set 1			
Set 2			
Set 3	1	0	$M[6-7]$



A Higher Level Example

```
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

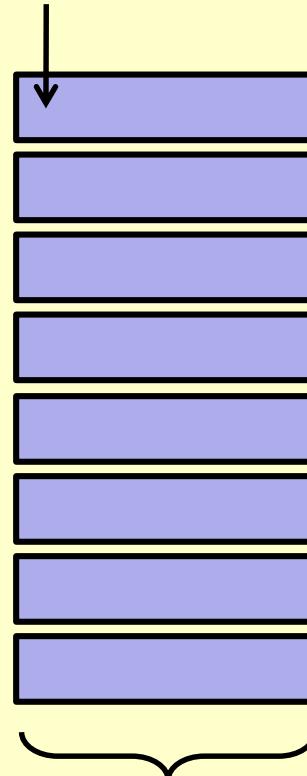
    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_cols(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (j = 0; j < 16; j++)
        for (i = 0; i < 16; i++)
            sum += a[i][j];
    return sum;
}
```

Ignore the variables sum, i, j

assume: cold (empty) cache,
a[0][0] goes here



32 bytes = 4 doubles

blackboard

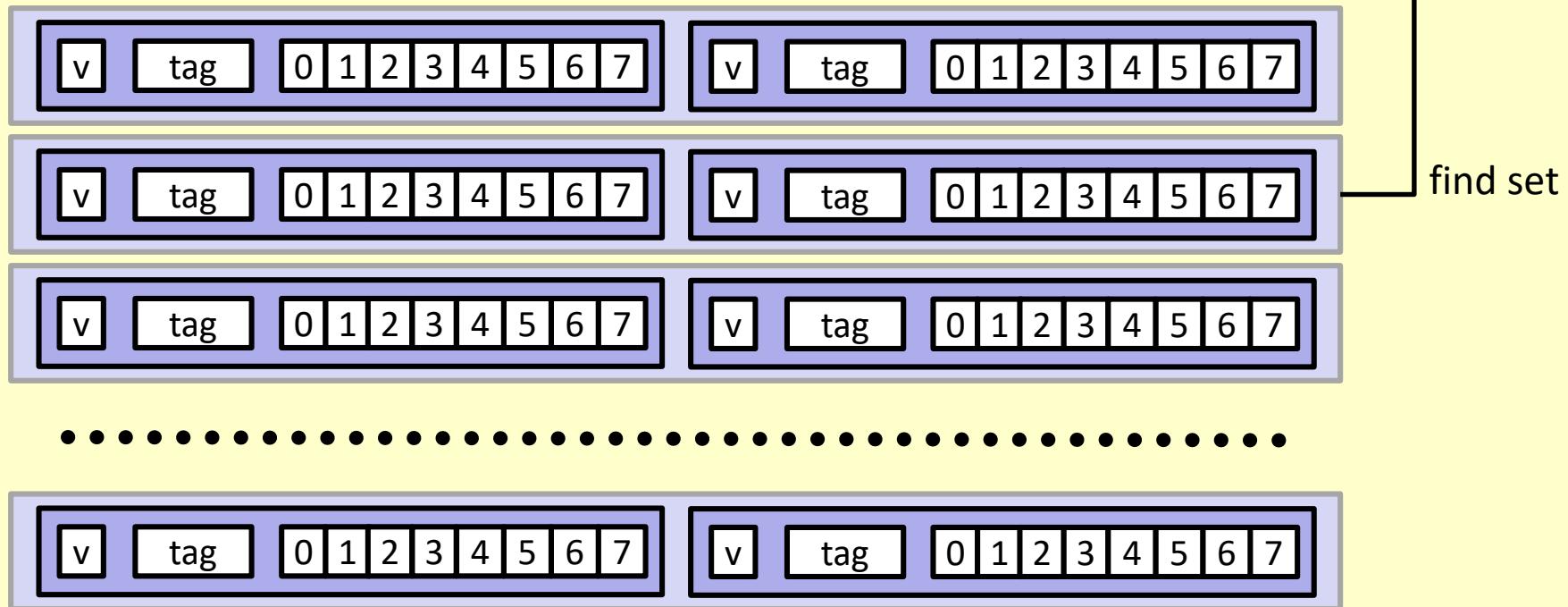
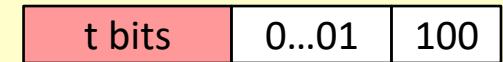
Questions?

E-way Set Associative Cache (Here: E = 2)

$E = 2$: Two lines per set

Assume: cache block size 8 bytes

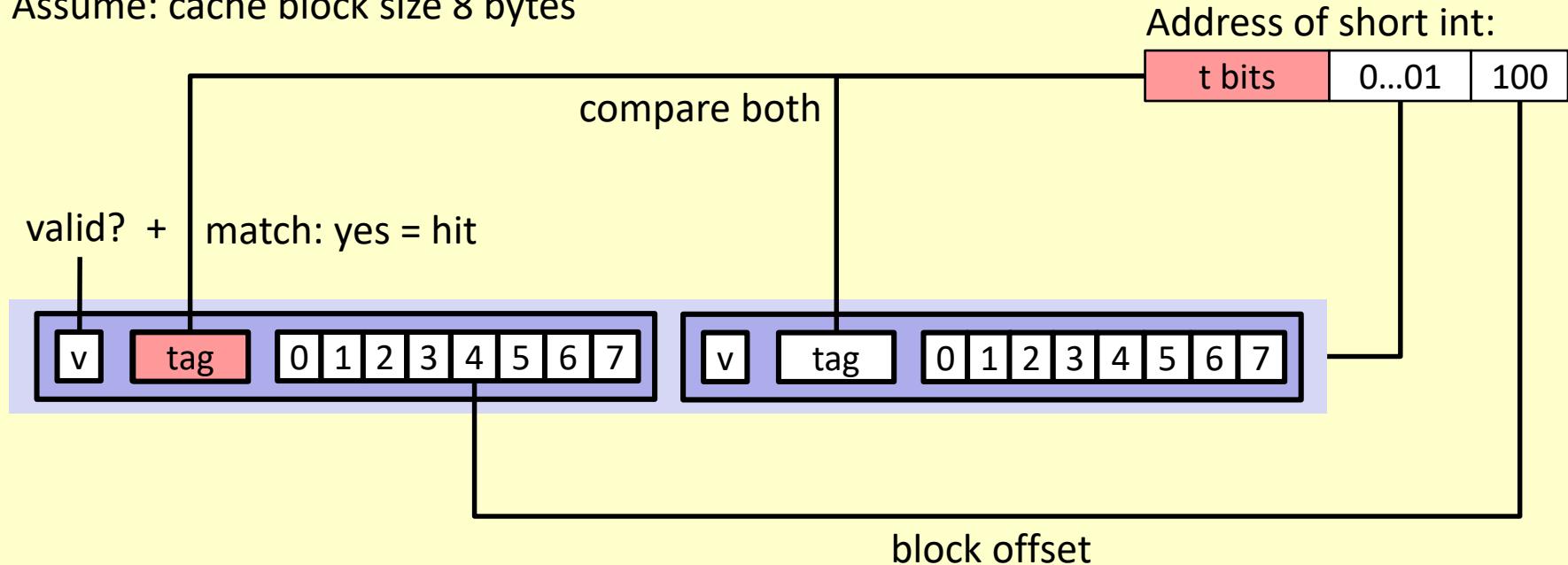
Address of short int:



E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set

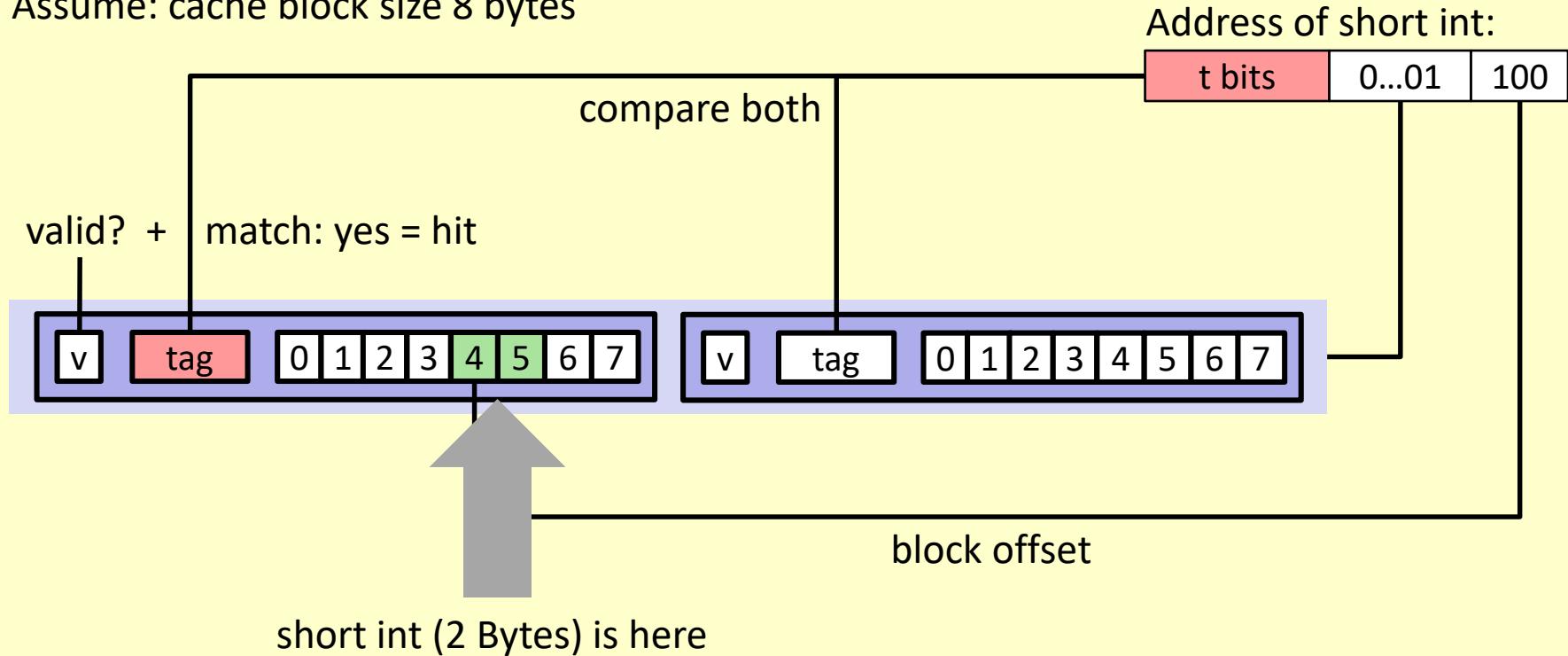
Assume: cache block size 8 bytes



E-way Set Associative Cache (Here: E = 2)

$E = 2$: Two lines per set

Assume: cache block size 8 bytes



No match:

- One line in set is selected for eviction and replacement
- Replacement policies: random, least recently used (LRU), ...

2-Way Set Associative Cache Simulation

$t=2 \quad s=1 \quad b=1$

xx	x	x
----	---	---

$M=16$ byte addresses, $B=2$ bytes/block,
 $S=2$ sets, $E=2$ blocks/set

Address trace (reads, one byte per read):

0	$[0000_2]$,	miss
1	$[0001_2]$,	hit
7	$[0111_2]$,	miss
8	$[1000_2]$,	miss
0	$[0000_2]$	hit

	v	Tag	Block
Set 0	1	00	$M[0-1]$
	1	10	$M[8-9]$
Set 1	1	01	$M[6-7]$
	0		
Cache Memories			



A Higher Level Example

```
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

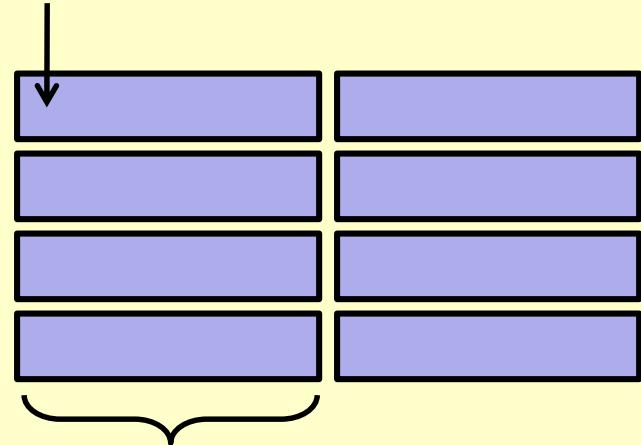
    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (j = 0; j < 16; j++)
        for (i = 0; i < 16; i++)
            sum += a[i][j];
    return sum;
}
```

Ignore the variables sum, i, j

assume: cold (empty) cache,
a[0][0] goes here



32 bytes = 4 doubles

blackboard

What about writes?

■ Multiple copies of data exist:

- L1, L2, Main Memory, Disk

■ What to do on a write-hit?

- **Write-through** (write immediately to memory)
- **Write-back** (defer write to memory until replacement of line)
 - Need a dirty bit (*meaning*: line different from memory or not)

■ What to do on a write-miss?

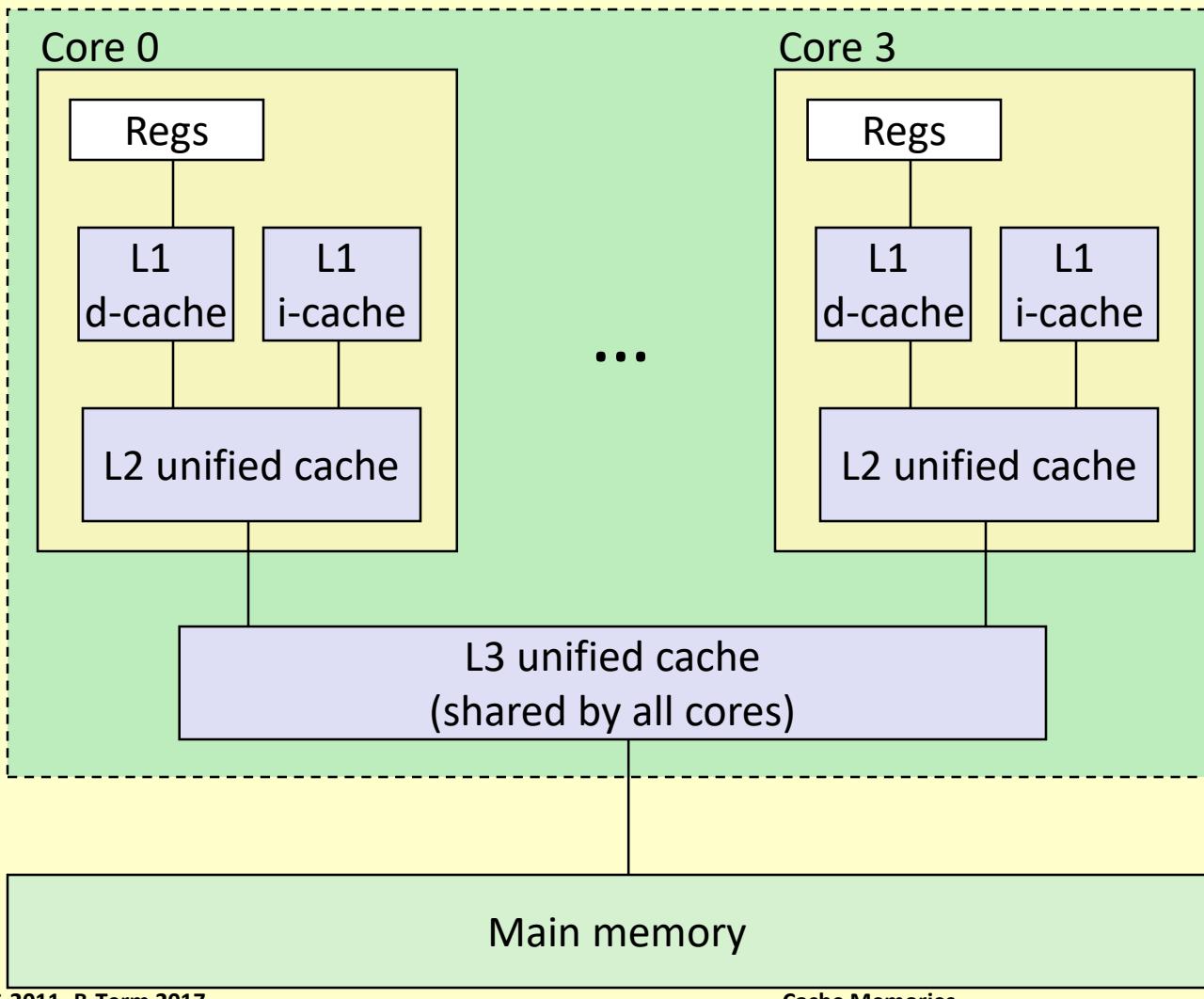
- **Write-allocate** (load into cache, update line in cache)
 - Good if more writes to the location follow
- **No-write-allocate** (writes immediately to memory)

■ Typical

- Write-through + No-write-allocate
- **Write-back + Write-allocate**

Intel Core i7 Cache Hierarchy

Processor package



L1 i-cache and d-cache:
32 KB each, 8-way,
Access: 4 cycles

L2 unified* cache:
256 KB, 8-way,
Access: 11 cycles

L3 unified cache:
8 MB, 16-way,
Access: 30-40 cycles

Block size: 64 bytes for
all caches.

* “Unified” = instructions +
data

Questions?

Today

- Cache organization and operation
- Performance impact of caches
 - The memory mountain
 - Rearranging loops to improve spatial locality
 - Using blocking to improve temporal locality

See: §6.6

The Memory Mountain

- **Read throughput (read bandwidth)**
 - Number of bytes read from memory per second (MB/s)
- **Memory mountain: Measured read throughput as a function of spatial and temporal locality.**
 - Compact way to characterize memory system performance.

Memory Mountain Test Function

```

long data[MAXELEMS]; /* Global array to traverse */

/* test - Iterate over first "elems" elements of
 *   array "data" with stride of "stride", using
 *   using 4x4 loop unrolling.
 */
int test(int elems, int stride) {
    long i, sx2=stride*2, sx3=stride*3, sx4=stride*4;
    long acc0 = 0, acc1 = 0, acc2 = 0, acc3 = 0;
    long length = elems, limit = length - sx4;

    /* Combine 4 elements at a time */
    for (i = 0; i < limit; i += sx4) {
        acc0 = acc0 + data[i];
        acc1 = acc1 + data[i+stride];
        acc2 = acc2 + data[i+sx2];
        acc3 = acc3 + data[i+sx3];
    }

    /* Finish any remaining elements */
    for (; i < length; i++) {
        acc0 = acc0 + data[i];
    }
    return ((acc0 + acc1) + (acc2 + acc3));
}

```

Call `test()` with many combinations of `elems` and `stride`.

For each `elems` and `stride`:

1. Call `test()` once to warm up the caches.
2. Call `test()` again and measure the read throughput (MB/s)

The Memory Mountain

Aggressive prefetching

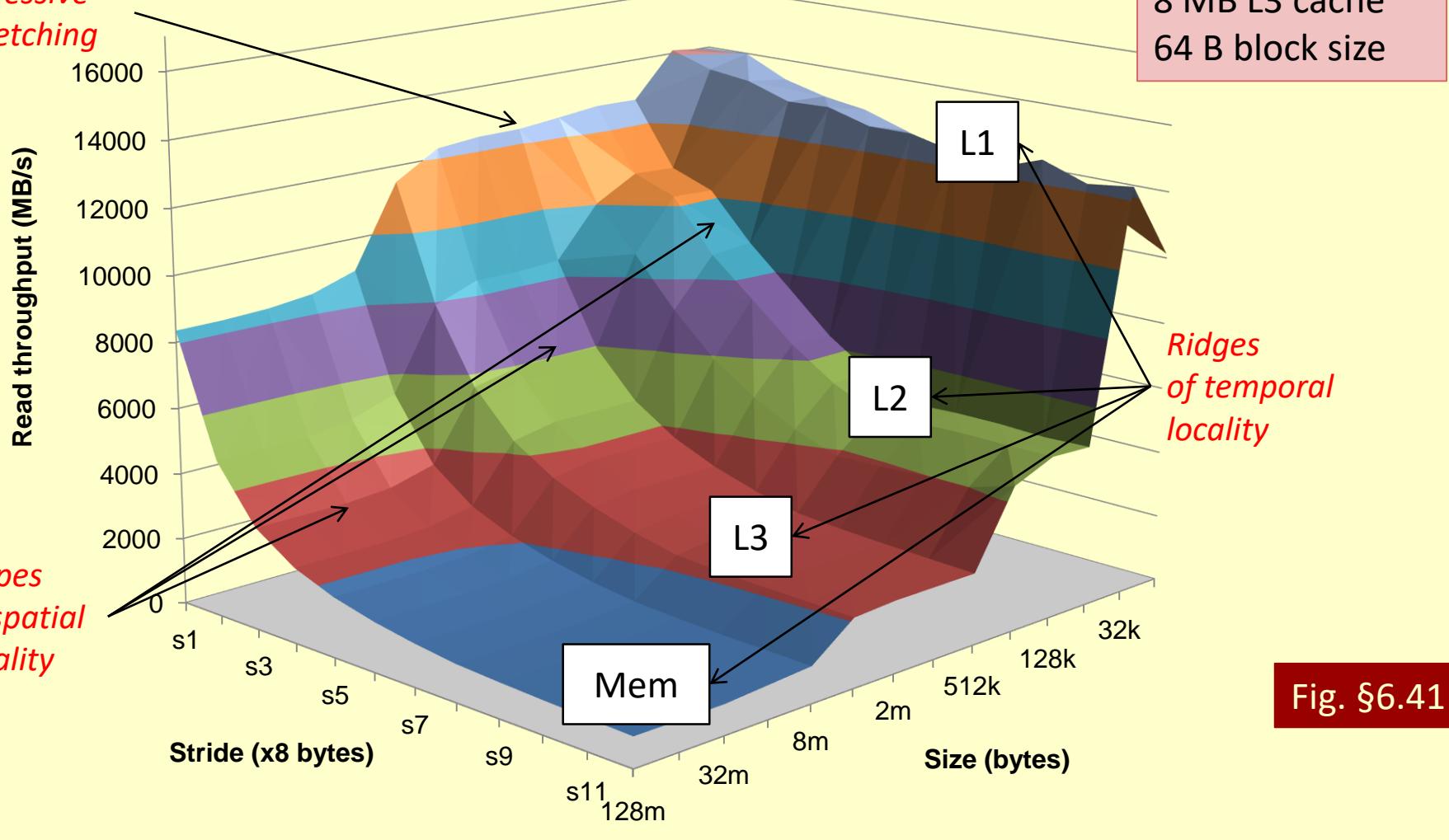


Fig. §6.41

Today

- Cache organization and operation
- Performance impact of caches
 - The memory mountain
 - Rearranging loops to improve spatial locality
 - Using blocking to improve temporal locality

Matrix Multiplication Example

■ Description:

- Multiply $N \times N$ matrices
- $O(N^3)$ total operations
- N reads per source element
- N values summed per destination
 - but may be able to hold in register

```
/* ijk */  
for (i=0; i<n; i++) {  
    for (j=0; j<n; j++) {  
        sum = 0.0; ←  
        for (k=0; k<n; k++)  
            sum += a[i] [k] * b[k] [j];  
        c[i] [j] = sum;  
    }  
}
```

Variable sum held in register

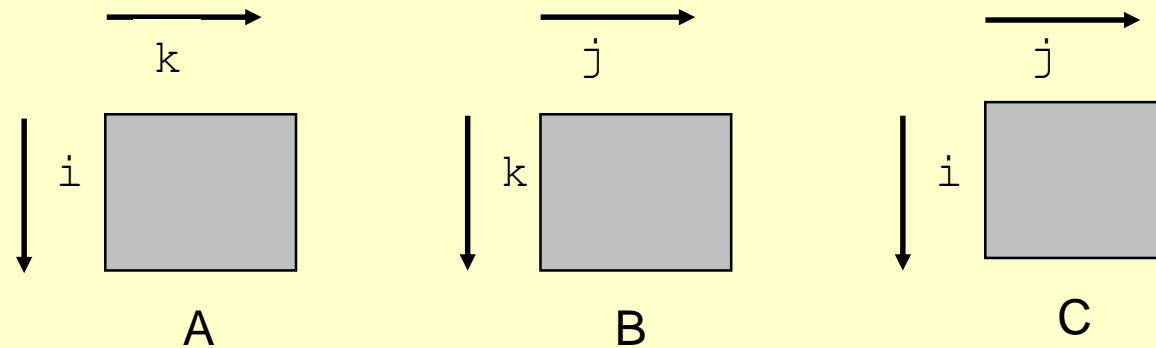
Miss Rate Analysis for Matrix Multiply

■ Assume:

- Cache line size = 32B (big enough for four 64-bit words)
- Matrix dimension (N) is very large
 - Approximate $1/N$ as 0.0
- Cache is not even big enough to hold multiple rows

■ Analysis Method:

- Look at access pattern of inner loop



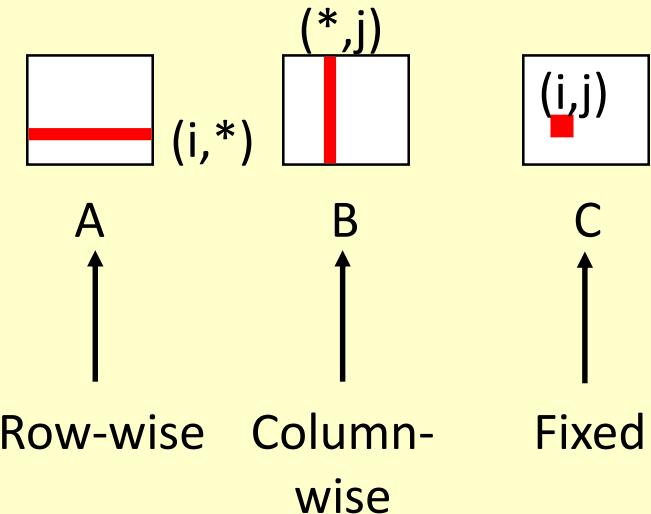
Layout of C Arrays in Memory (review)

- C arrays allocated in row-major order
 - each row in contiguous memory locations
- Stepping through columns in one row:
 - ```
for (i = 0; i < N; i++)
 sum += a[0][i];
```
  - accesses successive elements
  - if block size ( $B$ ) > 4 bytes, exploit spatial locality
    - compulsory miss rate =  $4 \text{ bytes} / B$
- Stepping through rows in one column:
  - ```
for (i = 0; i < n; i++)  
    sum += a[i][0];
```
 - accesses distant elements
 - no spatial locality!
 - compulsory miss rate = 1 (i.e. 100%)

Matrix Multiplication (ijk)

```
/* ijk */
for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
        sum = 0.0;
        for (k=0; k<n; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum;
    }
}
```

Inner loop:



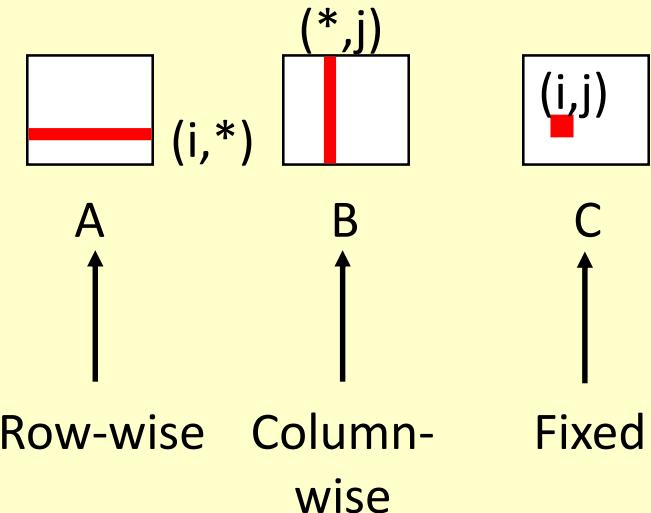
Misses per inner loop iteration:

<u>A</u>	<u>B</u>	<u>C</u>
0.25	1.0	0.0

Matrix Multiplication (jik)

```
/* jik */
for (j=0; j<n; j++) {
    for (i=0; i<n; i++) {
        sum = 0.0;
        for (k=0; k<n; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum
    }
}
```

Inner loop:



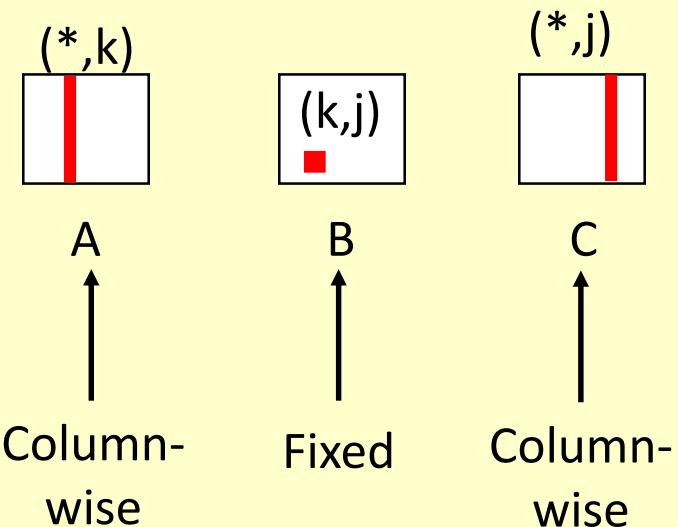
Misses per inner loop iteration:

<u>A</u>	<u>B</u>	<u>C</u>
0.25	1.0	0.0

Matrix Multiplication (jki)

```
/* jki */
for (j=0; j<n; j++) {
    for (k=0; k<n; k++) {
        r = b[k][j];
        for (i=0; i<n; i++)
            c[i][j] += a[i][k] * r;
    }
}
```

Inner loop:



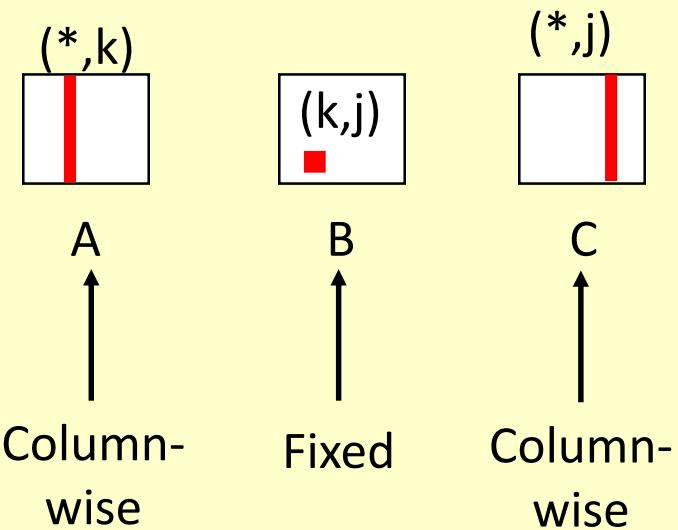
Misses per inner loop iteration:

<u>A</u>	<u>B</u>	<u>C</u>
1.0	0.0	1.0

Matrix Multiplication (kji)

```
/* kji */
for (k=0; k<n; k++) {
    for (j=0; j<n; j++) {
        r = b[k][j];
        for (i=0; i<n; i++)
            c[i][j] += a[i][k] * r;
    }
}
```

Inner loop:



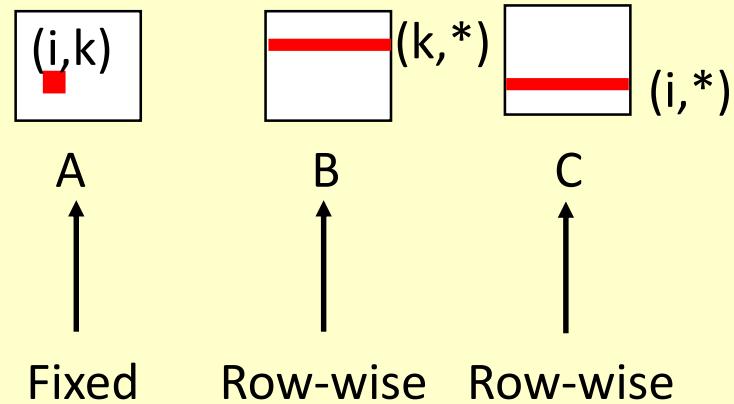
Misses per inner loop iteration:

<u>A</u>	<u>B</u>	<u>C</u>
1.0	0.0	1.0

Matrix Multiplication (kij)

```
/* kij */
for (k=0; k<n; k++) {
    for (i=0; i<n; i++) {
        r = a[i][k];
        for (j=0; j<n; j++)
            c[i][j] += r * b[k][j];
    }
}
```

Inner loop:



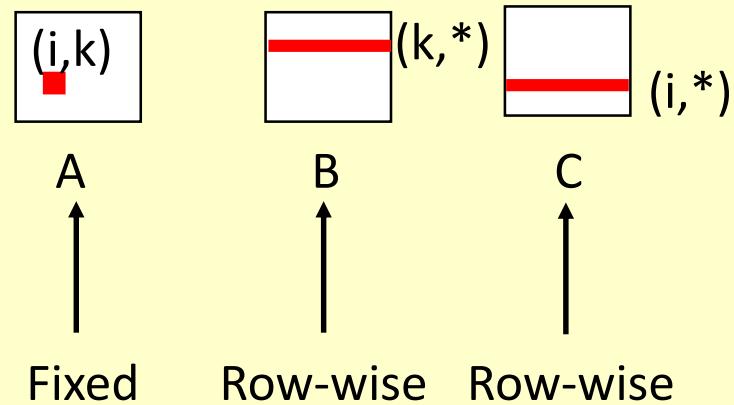
Misses per inner loop iteration:

<u>A</u>	<u>B</u>	<u>C</u>
0.0	0.25	0.25

Matrix Multiplication (ikj)

```
/* ikj */
for (i=0; i<n; i++) {
    for (k=0; k<n; k++) {
        r = a[i][k];
        for (j=0; j<n; j++)
            c[i][j] += r * b[k][j];
    }
}
```

Inner loop:



Misses per inner loop iteration:

<u>A</u>	<u>B</u>	<u>C</u>
0.0	0.25	0.25

Summary of Matrix Multiplication

```

for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
        sum = 0.0;
        for (k=0; k<n; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum;
    }
}

```

```

for (k=0; k<n; k++) {
    for (i=0; i<n; i++) {
        r = a[i][k];
        for (j=0; j<n; j++)
            c[i][j] += r * b[k][j];
    }
}

```

```

for (j=0; j<n; j++) {
    for (k=0; k<n; k++) {
        r = b[k][j];
        for (i=0; i<n; i++)
            c[i][j] += a[i][k] * r;
    }
}

```

ijk (& jik):

- 2 loads, 0 stores
- misses/iter = 1.25

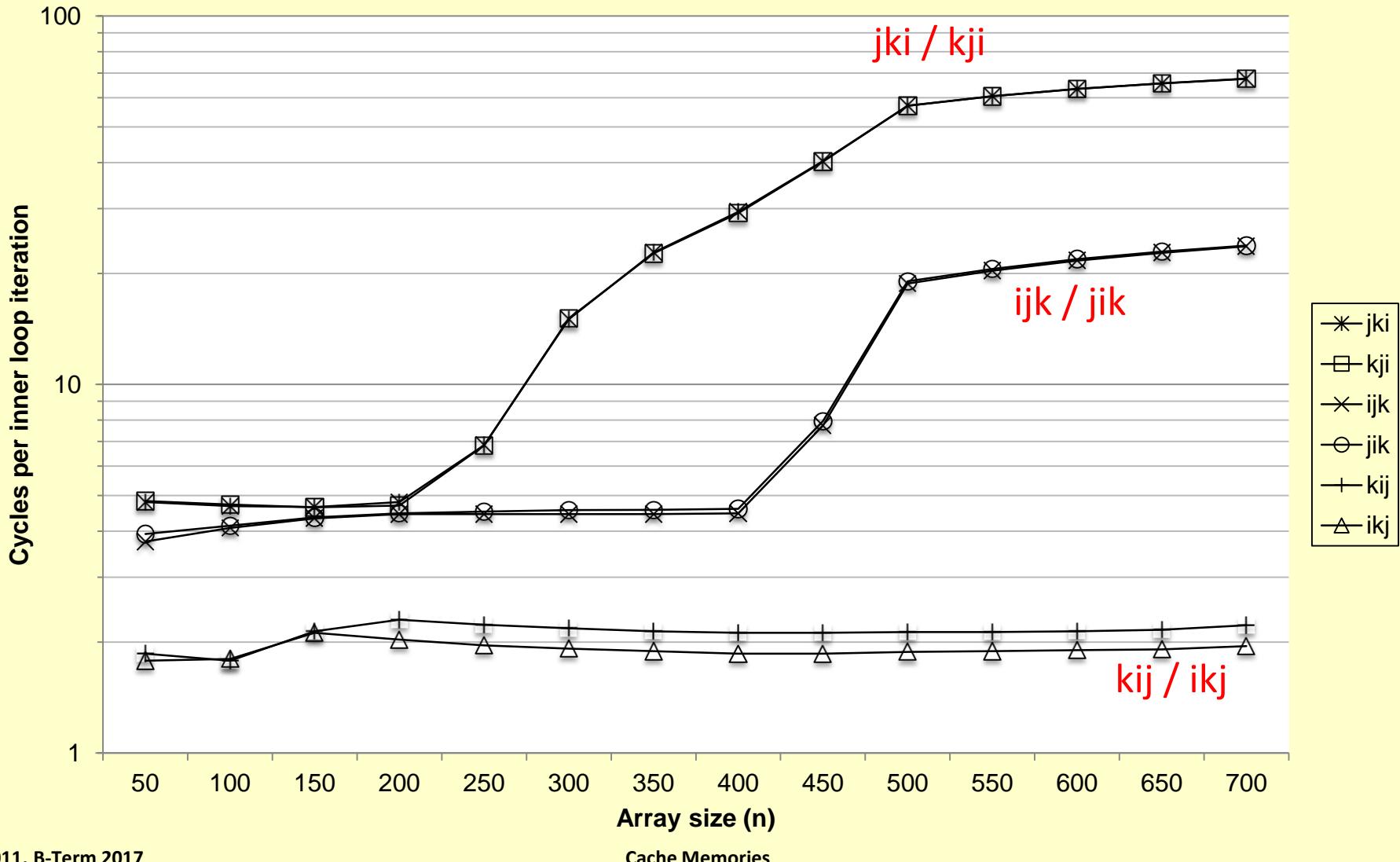
kij (& ikj):

- 2 loads, 1 store
- misses/iter = 0.5

jki (& kji):

- 2 loads, 1 store
- misses/iter = 2.0

Core i7 Matrix Multiply Performance

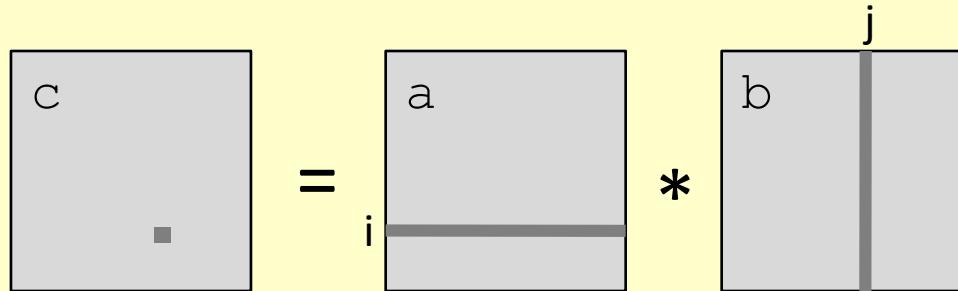


Today

- Cache organization and operation
- Performance impact of caches
 - The memory mountain
 - Rearranging loops to improve spatial locality
 - Using blocking to improve temporal locality

Example: Matrix Multiplication

```
c = (double *) calloc(sizeof(double), n*n);  
  
/* Multiply n x n matrices a and b */  
void mmm(double *a, double *b, double *c, int n) {  
    int i, j, k;  
    for (i = 0; i < n; i++)  
        for (j = 0; j < n; j++)  
            for (k = 0; k < n; k++)  
                c[i*n+j] += a[i*n + k]*b[k*n + j];  
}
```



Cache Miss Analysis

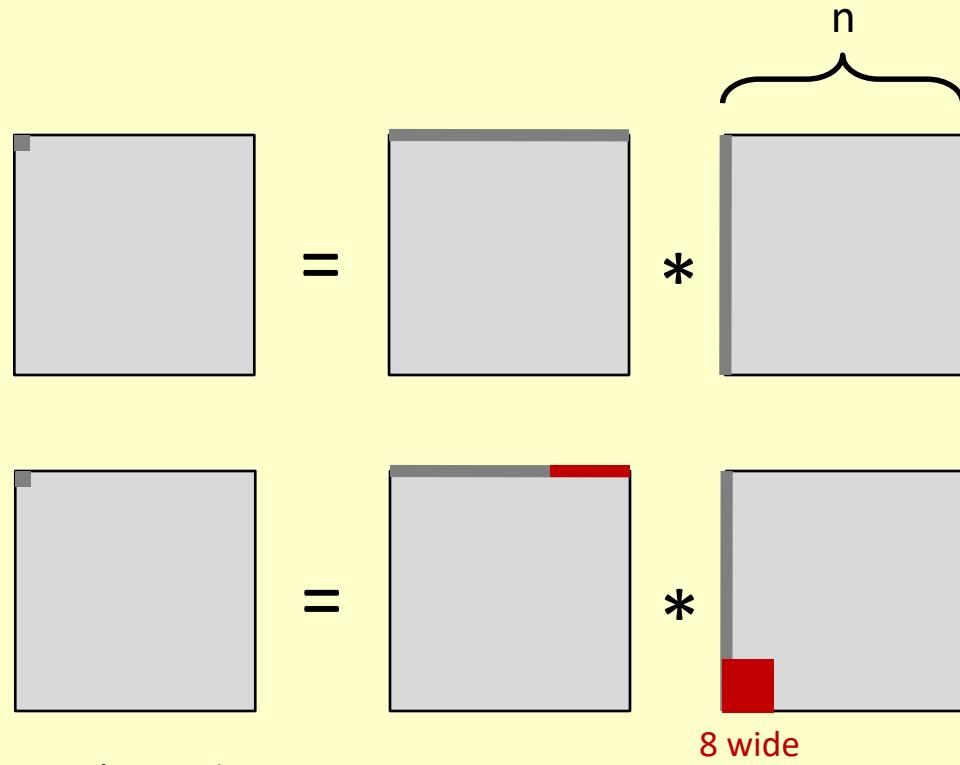
■ Assume:

- Matrix elements are doubles
- Cache block = 8 doubles
- Cache size $C \ll n$ (much smaller than n)

■ First iteration:

- $n/8 + n = 9n/8$ misses

- Afterwards **in cache:**
(schematic)



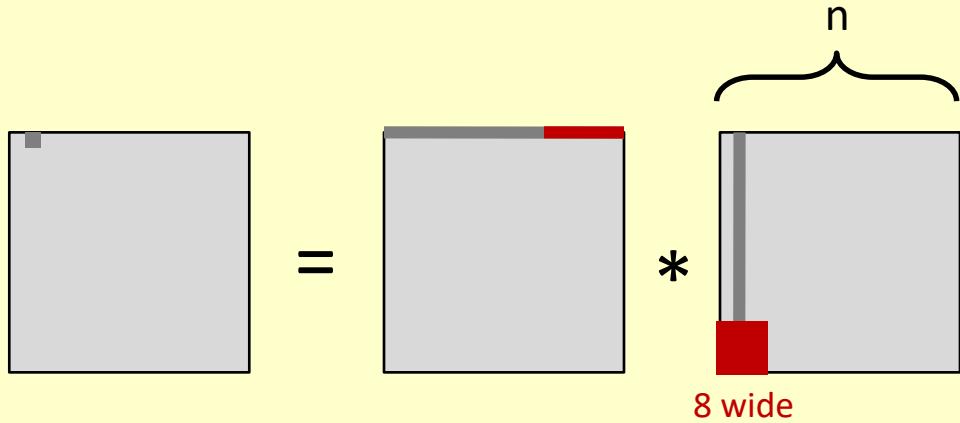
Cache Miss Analysis

■ Assume:

- Matrix elements are doubles
- Cache block = 8 doubles
- Cache size $C \ll n$ (much smaller than n)

■ Second iteration:

- Again:
 $n/8 + n = 9n/8$ misses



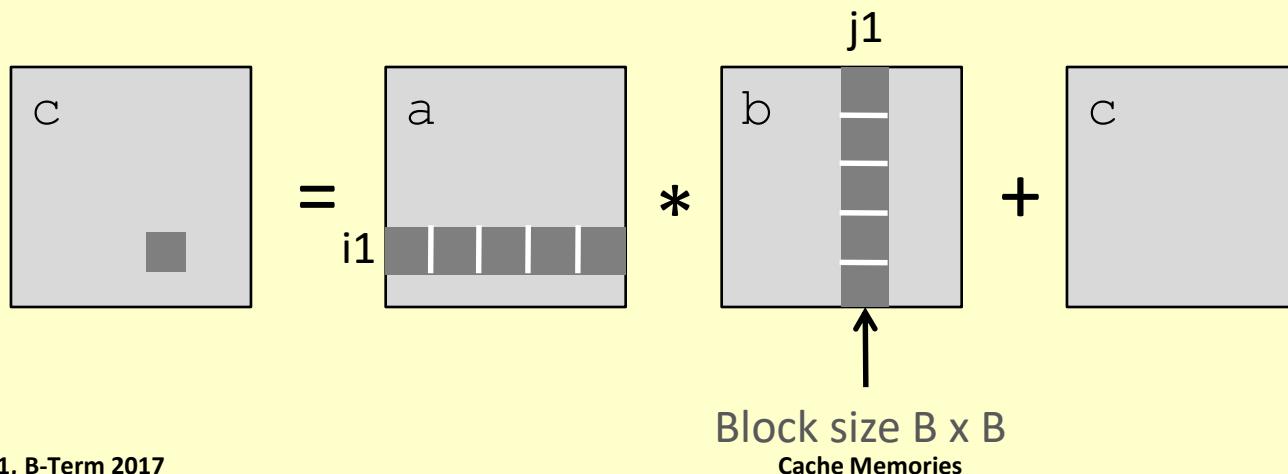
■ Total misses:

- $9n/8 * n^2 = (9/8) * n^3$

Blocked Matrix Multiplication

```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i+=B)
        for (j = 0; j < n; j+=B)
            for (k = 0; k < n; k+=B)
                /* B x B mini matrix multiplications */
                for (i1 = i; i1 < i+B; i++)
                    for (j1 = j; j1 < j+B; j++)
                        for (k1 = k; k1 < k+B; k++)
                            c[i1*n+j1] += a[i1*n + k1]*b[k1*n + j1];
}
```



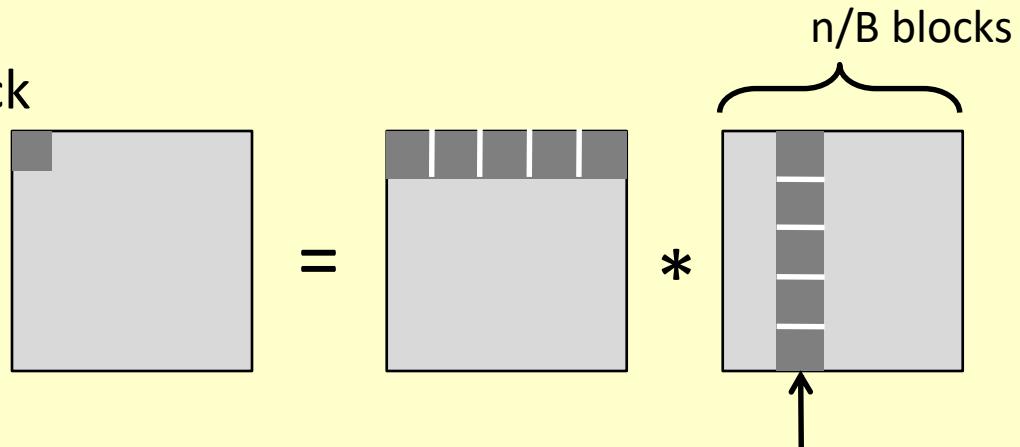
Cache Miss Analysis

■ Assume:

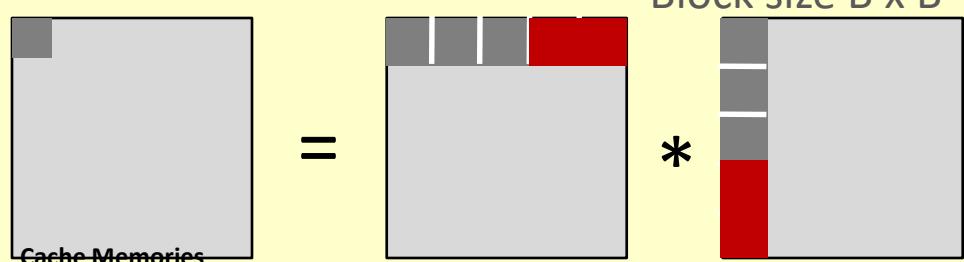
- Cache block = 8 doubles
- Cache size $C \ll n$ (much smaller than n)
- Three blocks fit into cache: $3B^2 < C$

■ First (block) iteration:

- $B^2/8$ misses for each block
- $2n/B * B^2/8 = nB/4$
(omitting matrix c)



- Afterwards in cache
(schematic)



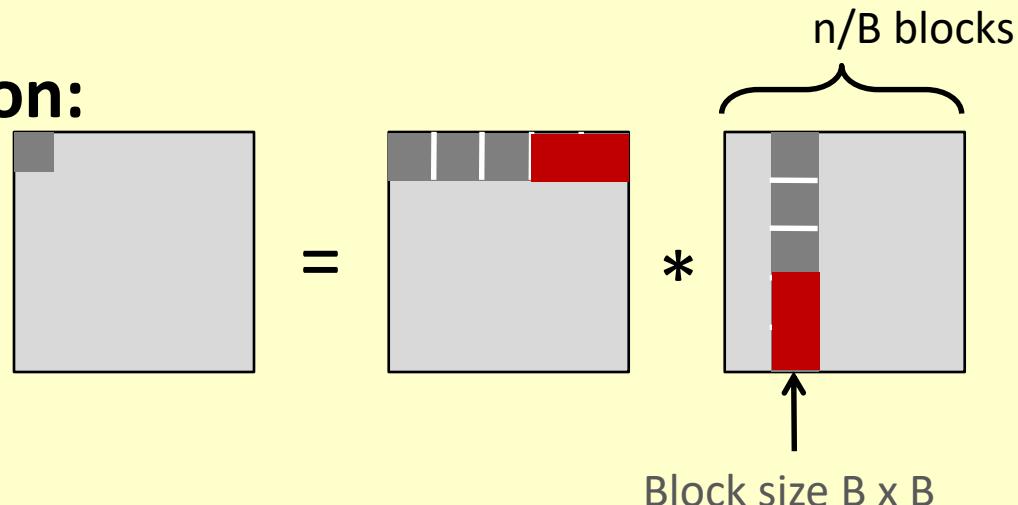
Cache Miss Analysis

■ Assume:

- Cache block = 8 doubles
- Cache size $C \ll n$ (much smaller than n)
- Three blocks fit into cache: $3B^2 < C$

■ Second (block) iteration:

- Same as first iteration
- $2n/B * B^2/8 = nB/4$



■ Total misses:

- $nB/4 * (n/B)^2 = n^3/(4B)$

Summary

- No blocking: $(9/8) * n^3$
- Blocking: $1/(4B) * n^3$
- Suggest largest possible block size B, but limit $3B^2 < C!$
- Reason for dramatic difference:
 - Matrix multiplication has inherent temporal locality:
 - Input data: $3n^2$, computation $2n^3$
 - Every array element used $O(n)$ times!
 - But program has to be written properly

Cache Summary

- Cache memories can have significant performance impact
- You can write your programs to exploit this!
 - Focus on the inner loops, where bulk of computations and memory accesses occur.
 - Try to maximize spatial locality by reading data objects sequentially with stride 1.
 - Try to maximize temporal locality by using a data object as often as possible once it's read from memory.

Questions?

One more note about caches

- Suppose a multi-level cache
- L1
 - Access time = 4 cycles
 - Miss rate = m_1
 - Miss penalty = p_1 = average access time of L2
- L2
 - Access time = 11 cycles
 - Miss rate = m_2
 - Miss penalty = p_2 = average access time of L3
- ...

One more note about caches (continued)

- **Average access time_{L1} =**
 - $4 + m_1 \times \text{Average access time}_{L2} =$
 - $4 + m_1 \times (11 + m_2 \times \text{Average access time}_{L3})$
- **Suppose**
 - Average access time_{L3} = 100 cycles
 - $m_1 = 10\%$, $m_2 = 4\%$
- **Then Average access time_{L1} =**
$$\begin{aligned} & 4 + 0.1 \times (11 + .04 \times 100) \\ & = 4 + 0.1 \times 15 \\ & = 4 + 1.5 = 5.5 \text{ cycles} \end{aligned}$$

Multi-level memories

- Same analysis can be applied to multiple levels
- Three levels of on-chip cache (Core i7)
- DRAM
 - Possibly multi-level in cluster systems
- Disk
 - Holding virtual memory

Questions?

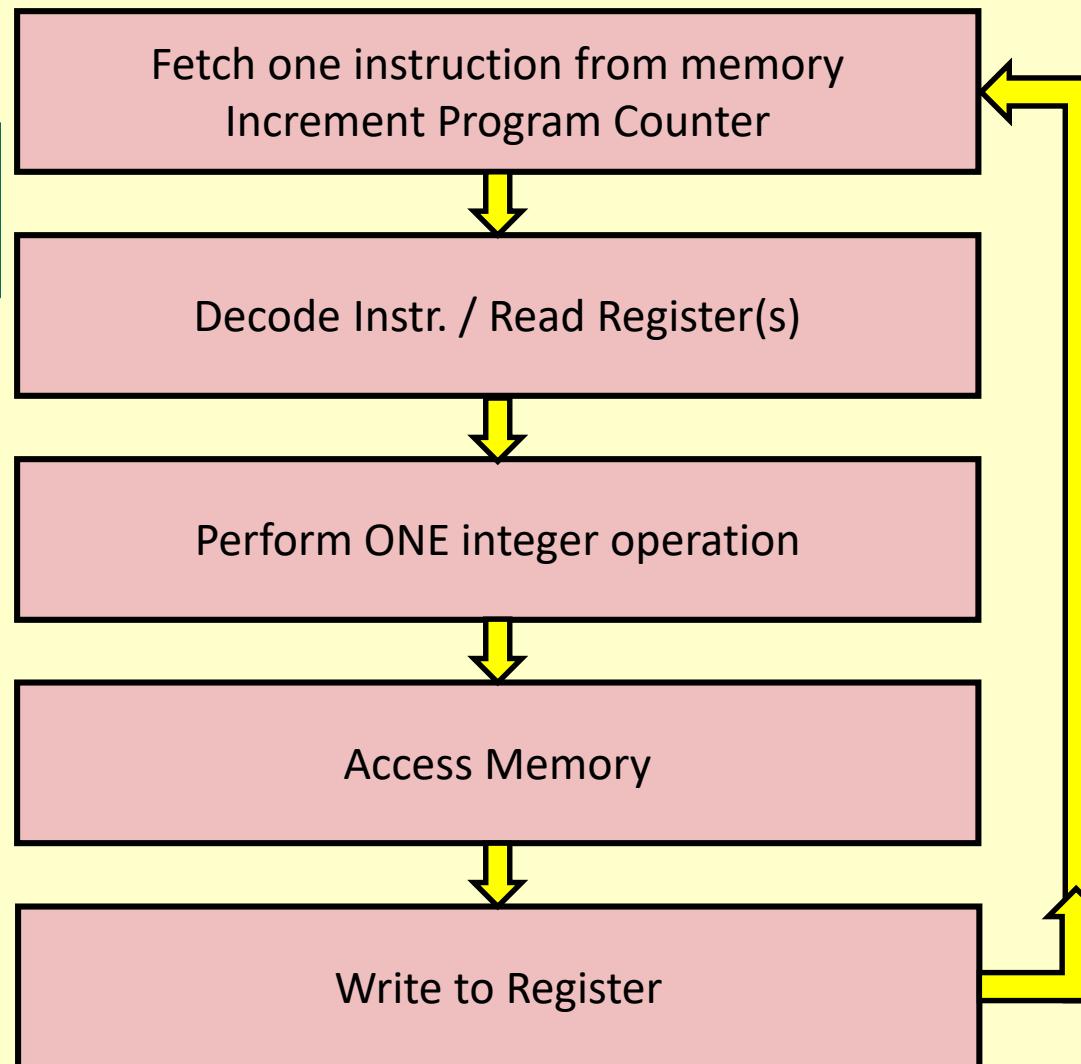
The Memory Hierarchy

Professor Hugh C. Lauer
CS-2011, Machine Organization and Assembly Language

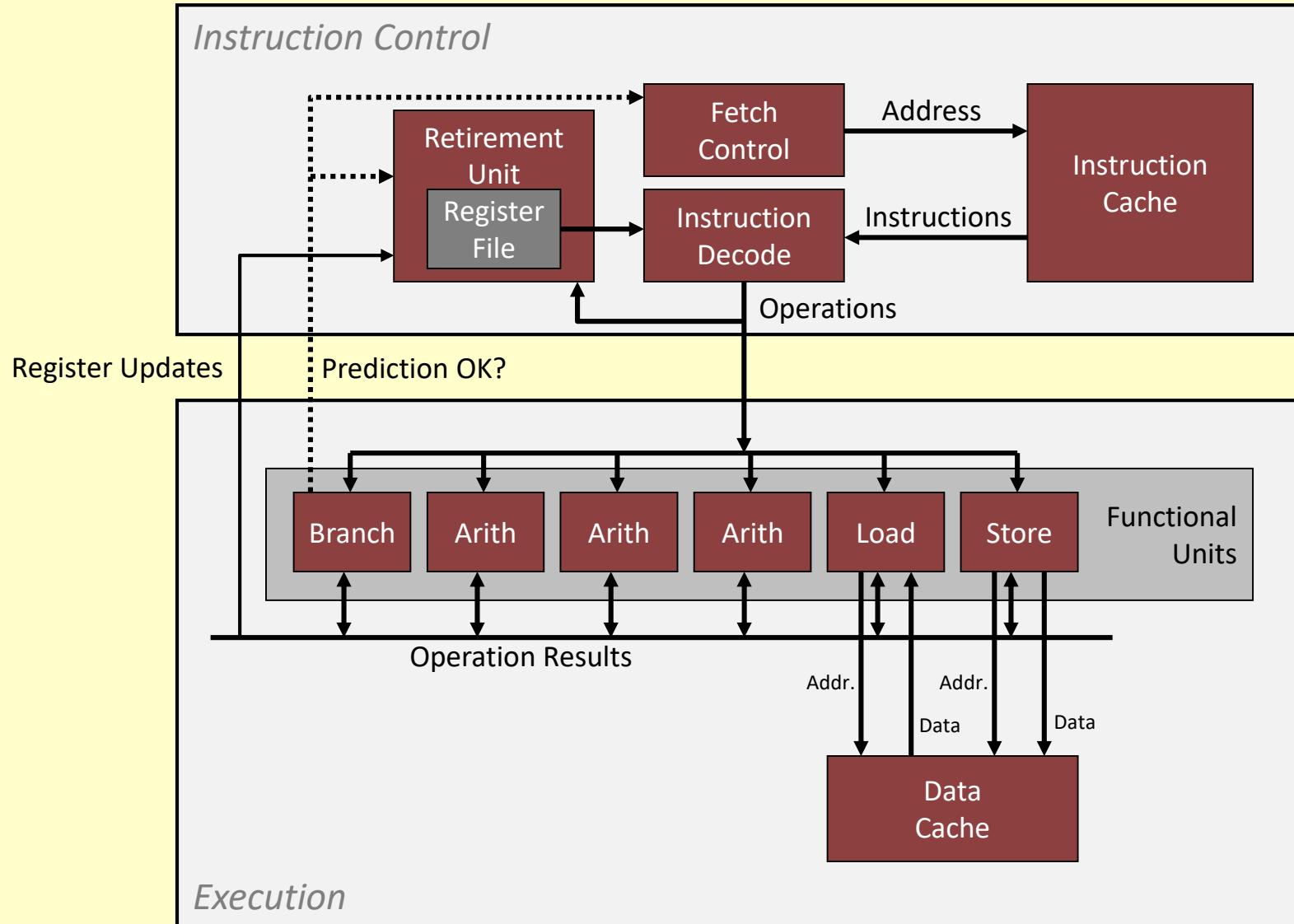
(Slides include copyright materials from *Computer Systems: A Programmer's Perspective*, by Bryant and O'Hallaron, and from *The C Programming Language*, by Kernighan and Ritchie)

Execution Model for Modern Computers

A highly idealized view



Modern Processor Design

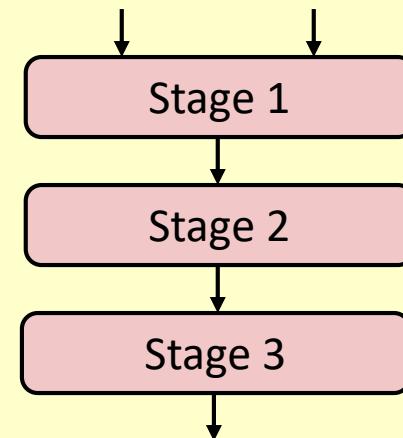


Superscalar Processor

- **Definition:** A superscalar processor can issue and execute *multiple instructions in one cycle*. The instructions are retrieved from a sequential instruction stream and are usually scheduled dynamically.
- **Benefit:** without programming effort, superscalar processor can take advantage of the *instruction level parallelism* that most programs have
- Most modern processors are superscalar.
- Intel: since Pentium (1993)

Pipelined Functional Units

```
long mult_eg(long a, long b, long c) {
    long p1 = a*b;
    long p2 = a*c;
    long p3 = p1 * p2;
    return p3;
}
```



	Time							
	1	2	3	4	5	6	7	
Stage 1	$a*b$	$a*c$			$p1*p2$			
Stage 2		$a*b$	$a*c$			$p1*p2$		
Stage 3			$a*b$	$a*c$				$p1*p2$

- Divide computation into stages
- Pass partial computations from stage to stage
- Stage i can start on new computation once values passed to $i+1$
- E.g., complete 3 multiplications in 7 cycles, even though each requires 3 cycles

More to say about this later in course!

Much more in CS-4515, Computer Architecture
Spring 2019

Problem

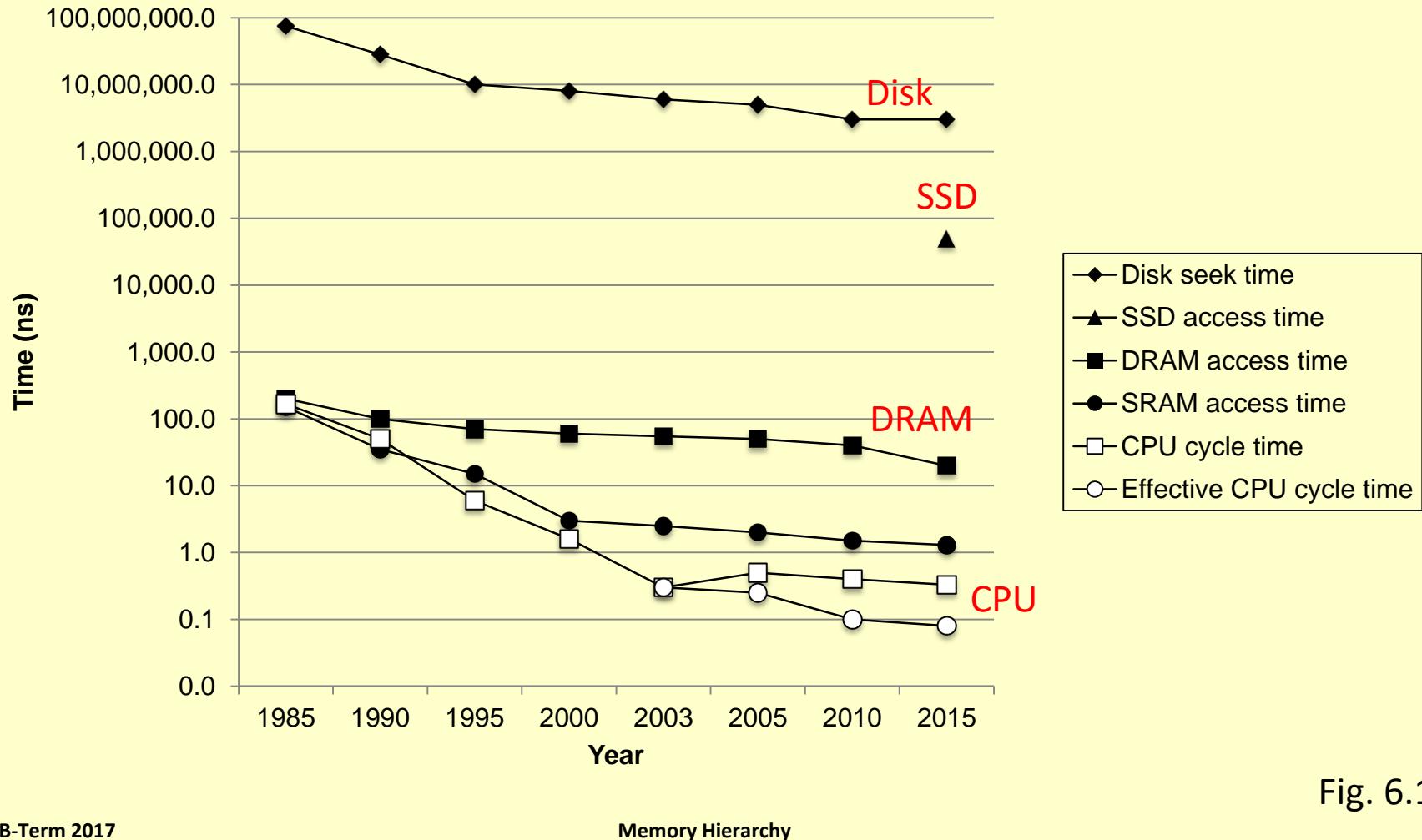
- **addq %rdx, %rax**
 - Requires *one* cycle (register to register)
- **addq 8(%rsp), %rax**
 - Requires *many* cycles
 - Depending upon type of memory!
- **On-chip**
 - 2–10 cycles to local SRAM
- **Off-chip**
 - 100+ cycles to DRAM

Problem (continued)

- All that compute power is wasted without quick access to memory
- Large on-chip memories impractical
 - On-chip DRAMs technologically challenging

The Processor-Memory Gap

The gap widens between DRAM, disk, and Processor speeds.



Solution

The key to bridging this CPU-Memory gap is a fundamental property of computer programs known as **locality**

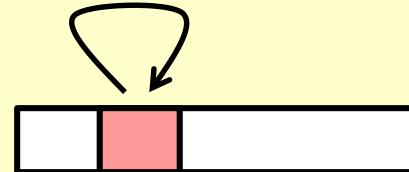
Exploit locality in the form of caches

Today

- Storage technologies and trends
- Locality of reference
- Caching in the memory hierarchy

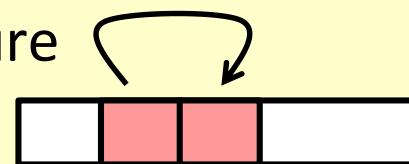
Locality

- **Principle of Locality:** Programs tend to use data and instructions with addresses near those they have used recently



- **Temporal locality:**

- Recently referenced items are likely to be referenced again in the near future



- **Spatial locality:**

- Items with nearby addresses tend to be referenced close together in time

Locality Example

```
sum = 0;  
for (i = 0; i < n; i++)  
    sum += a[i];  
return sum;
```

■ Data references

- Reference array elements in succession (stride-1 reference pattern). Spatial locality
- Reference variable **sum** each iteration. Temporal locality

■ Instruction references

- Reference instructions in sequence. Spatial locality
- Cycle through loop repeatedly. Temporal locality

Qualitative Estimates of Locality

- **Claim:** Being able to look at code and get a qualitative sense of its locality is a key skill for a professional programmer.
- **Question:** Does this function have good locality with respect to array a?

```
int sum_array_rows(int a[M] [N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i] [j];
    return sum;
}
```

I.e., hold row number constant while looping thru columns

Locality Example

- **Question:** What about this function?
- Does it have good locality with respect to array a?

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

I.e., hold column number constant while looping thru rows

Locality Example

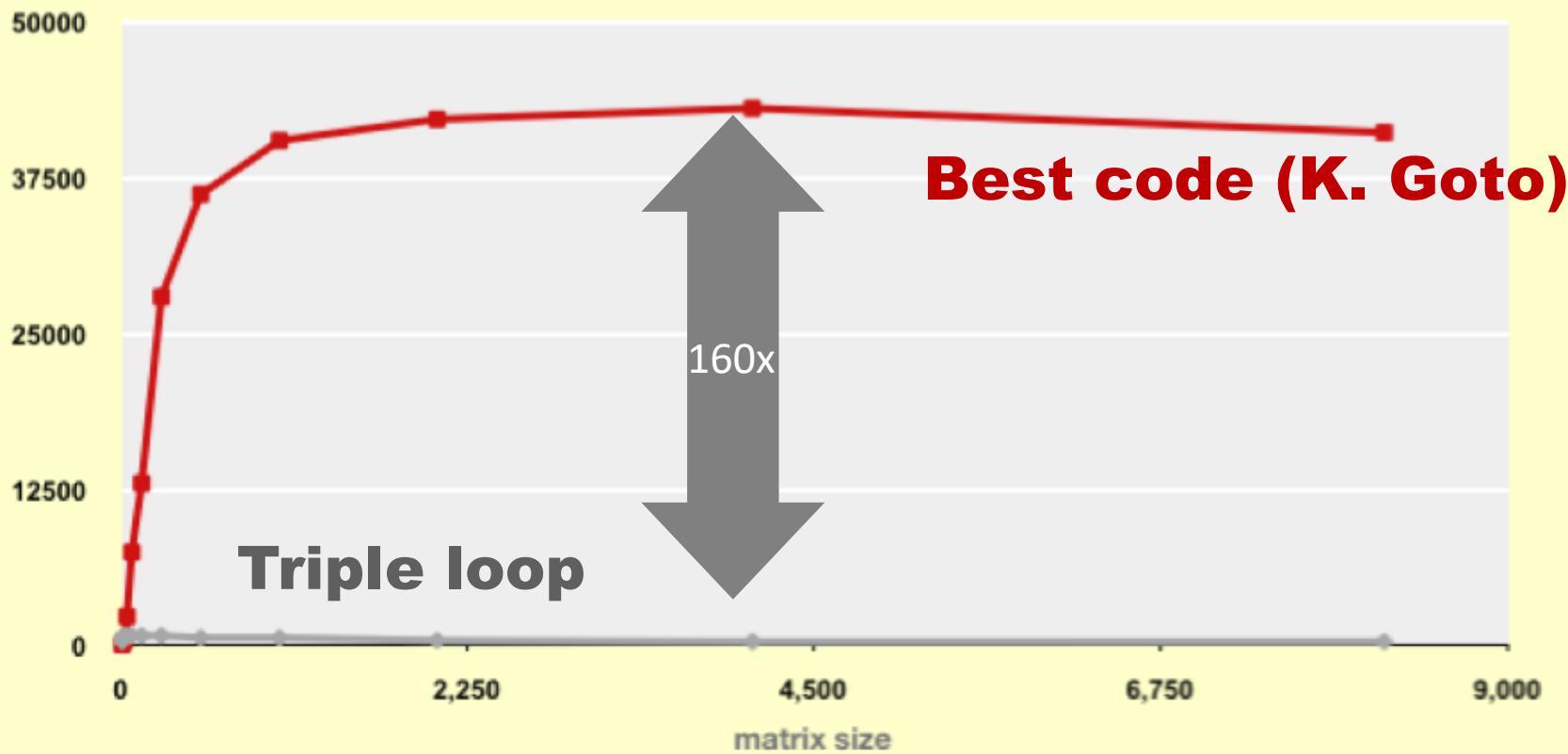
- **Question:** Can you permute the loops so that the function scans the 3-d array `a` with a stride-1 reference pattern (and thus has good spatial locality)?

```
int sum_array_3d(int a[M] [N] [N] )
{
    int i, j, k, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            for (k = 0; k < N; k++)
                sum += a[k] [i] [j];
    return sum;
}
```

Example Matrix Multiplication

Matrix-Matrix Multiplication (MMM) on 2 x Core 2 Duo 3 GHz (double precision)
Gflop/s



- Standard desktop computer, vendor compiler, using optimization flags
- Both implementations have **exactly the same operations count ($2n^3$)**
- **What is going on?**

Locality Example

- **Question:** Can you permute the loops so that the function scans the 3-d array `a` with a stride-1 reference pattern (and thus has good spatial locality)?

```
int sum_array_3d(int a[M] [N] [N] )
{
    int i, j, k, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            for (k = 0; k < N; k++)
                sum += a[k] [i] [j];
    return sum;
}
```

Memory Hierarchies

- Some fundamental and enduring properties of hardware and software:
 - Fast storage technologies cost more per byte, have less capacity, and require more power (heat!).
 - Widening gap between processor and main memory speeds.
 - Well-written programs tend to exhibit good locality.
- These fundamental properties complement each other beautifully.
- They suggest an approach for organizing memory and storage systems known as a **memory hierarchy**.

Today

- Storage technologies and trends
- Locality of reference
- ~~Caching in the memory hierarchy~~
- Caches and caching principles

Definition:- Cache

- A small fast memory that holds a (frequently accessed) subset of items from a much larger, slower memory

- Reason:-
To approximate the performance of the fast memory while retaining the size and economic benefits of the larger memory

Basic Idea

- Design a cache system so that ...
- ... most of the accesses go to the small, fast memory, ...
- ... and those that don't go to the small fast memory occur with sufficiently low probability that the extra cost of access does not hurt, ...
- ... at least, not very much

Caches occur *everywhere* in computing

■ Transaction processing

- Keep records of today's departures in RAM or local storage while records of future flights are on remote database

■ Program execution

- Keep the bytes near the current program counter in on-chip SRAM memory while rest of program is in DRAM

■ File management

- Keep disk maps of open files in RAM while retaining maps of *all* files on disk

■ Game design

- Keep details of nearby environment in cache of each character

■ ...

In fact, ...

**Caching is, by far, THE MOST IMPORTANT TOPIC pertaining
to the performance of ANY hardware or software or
distributed system!**

Example — Virtual Memory

- **Definition:**— Virtual memory is the *illusion* that each running program has its own memory space
 - Separate from those of all other running programs ...
 - ... on the same computer or different computers ...
 - ... belonging to the same user or different users
- **In reality, virtual memories are stored on disk**
 - Yes, really slow disks!
 - RAM is a *cache* of the frequently used “pages” of virtual memory

Properties of all caches

- A mechanism to recognize when something is already in the cache ...
 - ... and use it
 - Called a Cache Hit
- A mechanism to recognize when something needed is *not* in the cache ...
 - ... and to fetch it from the underlying large memory into the cache
 - Called a Cache Miss
- A policy for throwing something out of a cache ...
 - ... to make space for cache misses
- A mechanism for updating the underlying memory when cached objects change ...
 - ... or (especially) when they are thrown out!

Caches can be layered

Processor
designer's
view

- L1 is a *cache* of L2 memory
 - 32 k-bytes, on-chip, 4 cycle access (Core i7)
- L2 is a *cache* of L3 memory
 - 256 k-bytes, on-chip, 11 cycle access
- L3 is a *cache* of DRAM
 - 8 megabytes, shared among 4 cores; 30-40 cycle access
- DRAM is a *cache* of virtual memory
 - N gigabytes, shared among all processes; 100's of cycles

Caches can be layered

Programmer's view

- L1 is a *cache* of L2 memory
 - 32 k-bytes, on-chip, 4 cycle access (Core i7)
- L2 is a *cache* of L3 memory
 - 256 k-bytes, on-chip, 11 cycle access
- L3 is a *cache* of DRAM
 - 8 megabytes, shared among 4 cores; 30-40 cycle access
- DRAM is a *cache* of virtual memory
 - N gigabytes, shared among all processes; 100's of cycles

Memory Hierarchy — layered caches

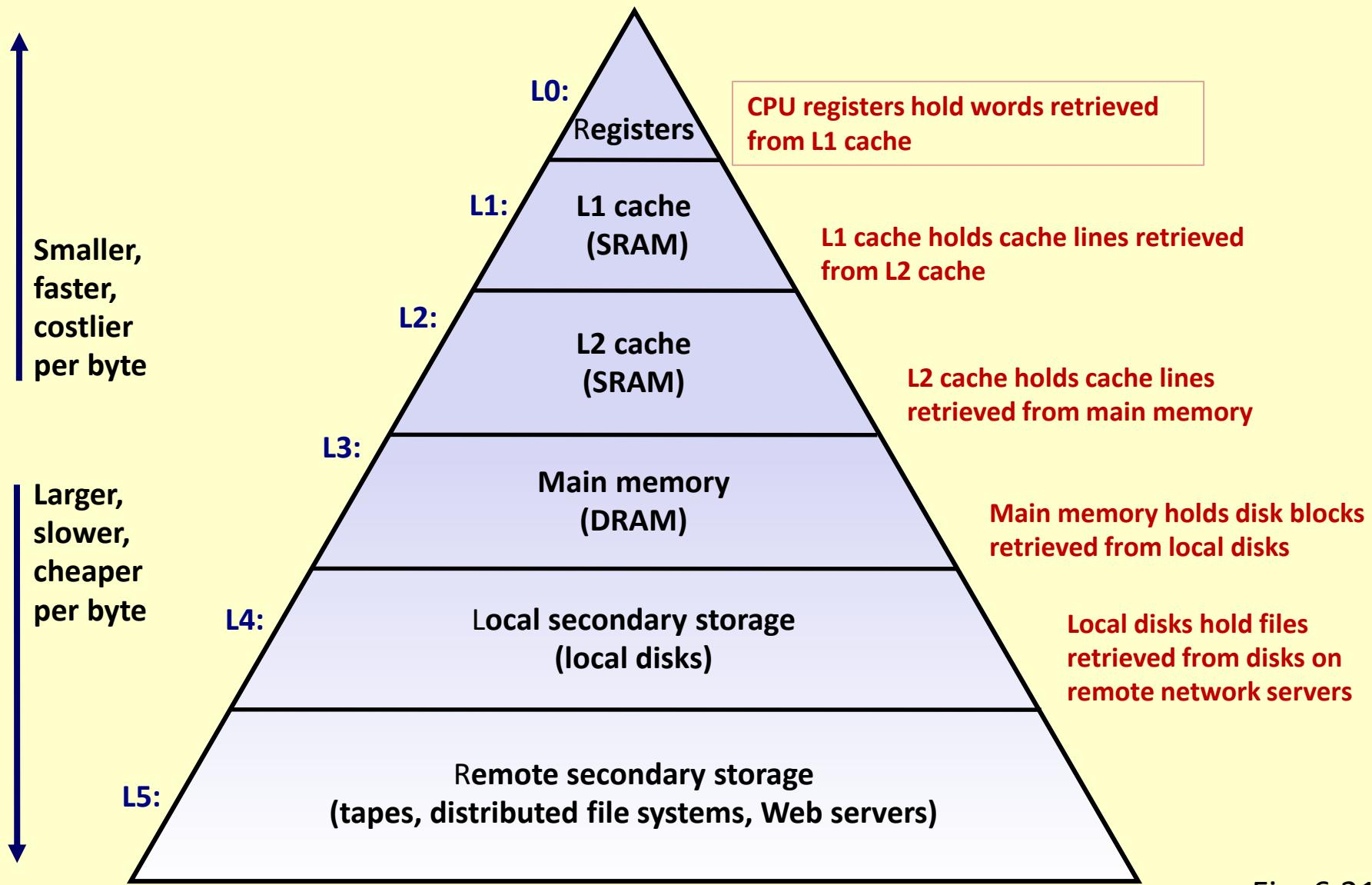


Fig. 6.21

Caches and Memory Hierarchies

■ Fundamental idea of a memory hierarchy

- For each k , the faster, smaller device at level k serves as a *cache* for the larger, slower device at level $k+1$

■ Why do memory hierarchies work?

- Because of *locality*, programs tend to access the data at level k more often than they access the data at level $k+1$
- Thus, the storage at level $k+1$ can be slower, and thus larger and cheaper per bit

■ *Big Idea:* The memory hierarchy creates illusion of a large pool of storage ...

- ... as big and (nearly) as cheap as the bottom layer
- ... as (nearly) fast as the top layer

Caching is all about performance ...

... and probabilities

Cache Performance

■ Miss Rate

- Fraction of memory references not found in cache (misses / accesses)
 $= 1 - \text{hit rate}$
- Typical numbers (in percentages):
 - 3-10% for L1
 - can be quite small (e.g., < 1%) for L2, depending on size, etc.

■ Hit Time

- Time to deliver a block in the cache to the processor
 - includes time to determine whether the block is in the cache
- Typical numbers:
 - 1-4 clock cycle for L1
 - 10-20 clock cycles for L2

■ Miss Penalty

- Additional time required because of a miss
 - typically 50-200 cycles for main memory (Trend: increasing!)

Cache Performance (continued)

■ Average access time =

- Hit time + miss_rate × miss penalty

■ Example

- Hit time for L1 cache = 1 cycle
- Miss penalty for L1 cache = 10 cycles
- Miss rate = 10%
- ⇒ Average access time = $1 + 0.1 * 10 = 2$

■ Example 2

- Miss rate = 1%
- ⇒ Average access time = $1 + 0.01 * 10 = 1.1$

Think about those numbers

- Huge difference between a hit and a miss
 - Could be 100x, if just L1 and main memory
- Would you believe 99% hits is twice as good as 97%?
 - Consider:
cache hit time of 1 cycle
miss penalty of 100 cycles
- Average access time:
 - 97% hits: 1 cycle + 0.03 * 100 cycles = **4 cycles**
 - 99% hits: 1 cycle + 0.01 * 100 cycles = **2 cycles**
- This is why “miss rate” is used instead of “hit rate”

Writing Cache Friendly Code

See especially: §6.5

■ Make the common case go fast

- Focus on the inner loops of the core functions

■ Minimize the misses in the inner loops

- Repeated references to variables are good (**temporal locality**)
- Stride-1 reference patterns are good (**spatial locality**)

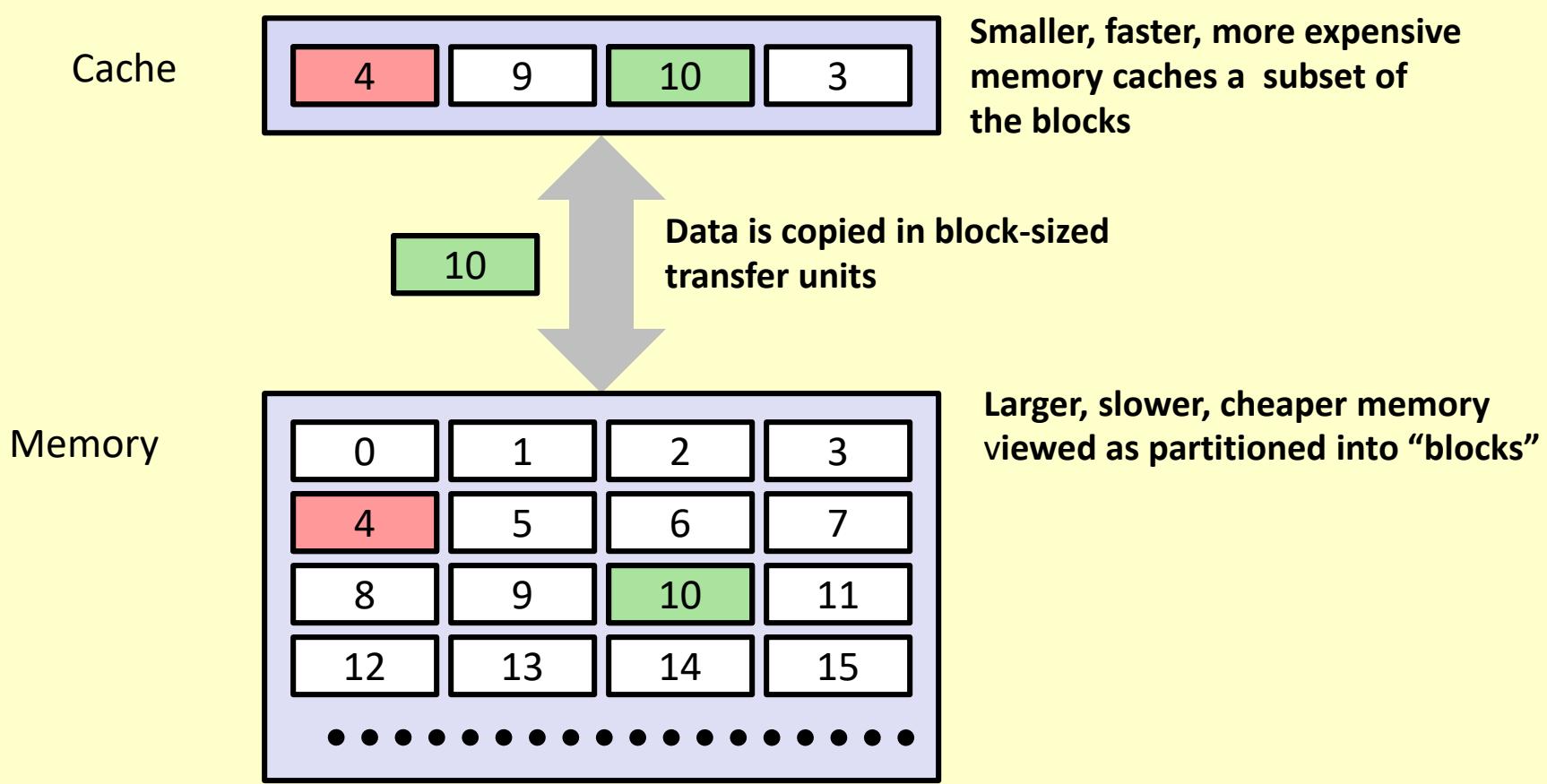
Key idea: Our qualitative notion of locality is quantified through our understanding of cache memories.

Note about hit-miss probabilities

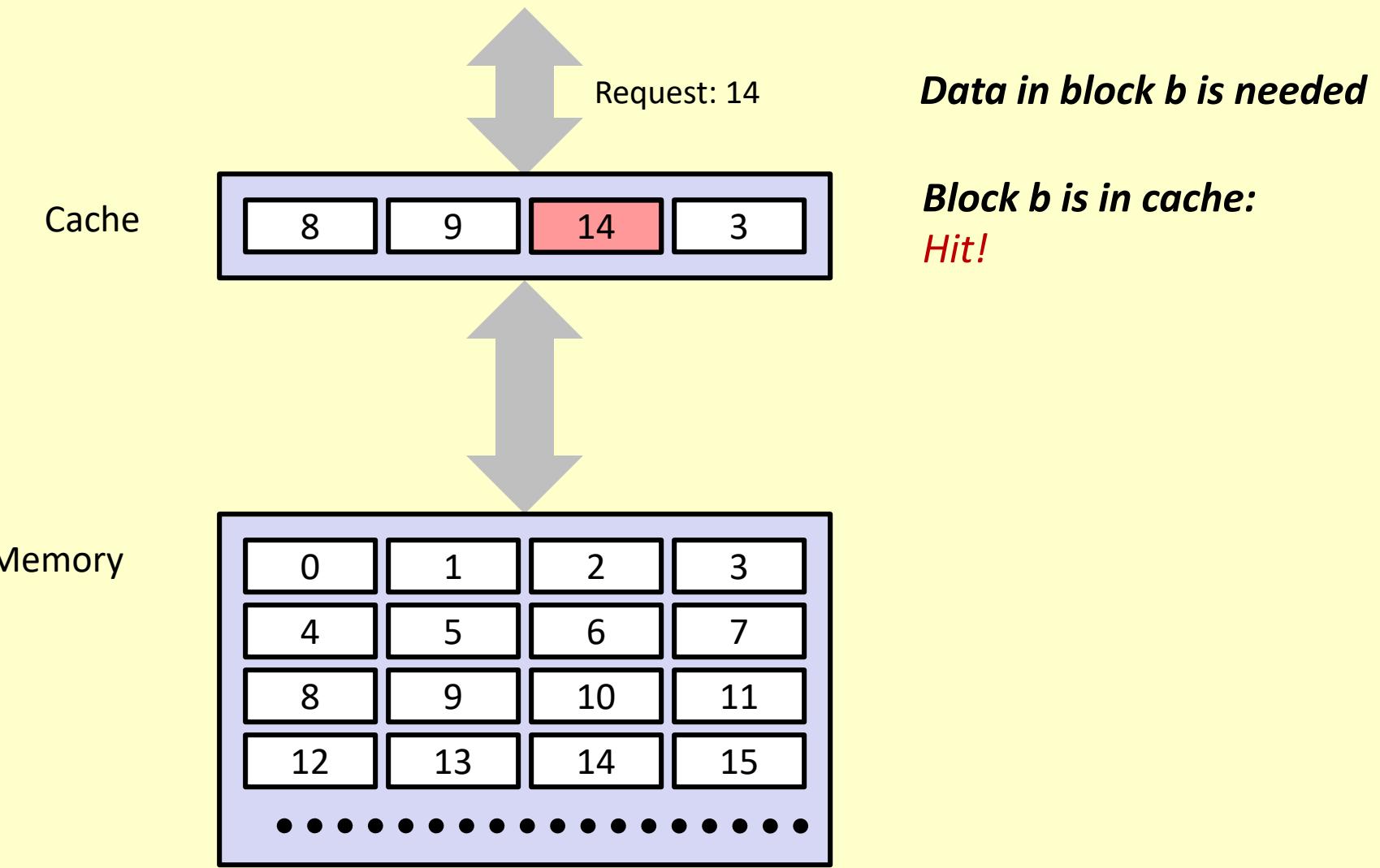
- No *a priori* methods to predict probabilities of caches hits and misses
- I.e., no way to tell in advance how well a particular caching strategy will work
 - Must determine experimentally!
- **Benchmarks**
 - Standardized suites of real programs/applications for measuring performance
- **Used in**
 - Hardware design, OS design, Database design, etc.

Questions?

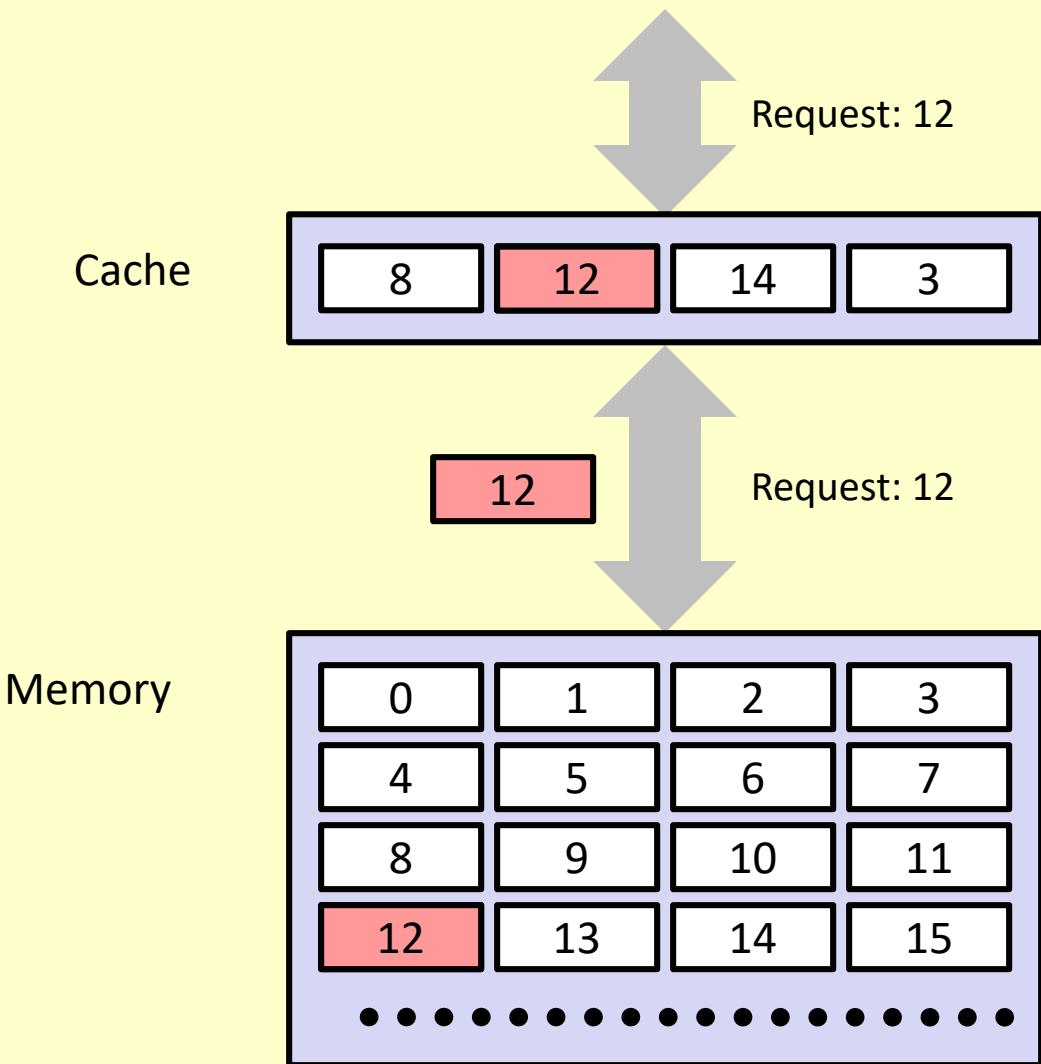
Caches in Microprocessors



General Cache Concepts: Hit



General Cache Concepts: Miss



Data in block b is needed

***Block b is not in cache:
Miss!***

***Block b is fetched from
memory***

Block b is stored in cache

- **Placement policy:** determines where b goes
- **Replacement policy:** determines which block gets evicted (victim)

Types of Cache Misses

■ Cold (compulsory) miss

- Cold misses occur because the cache is empty.

■ Conflict miss

- Most caches limit blocks at level $k+1$ to a small subset (sometimes a singleton) of the block positions at level k .
 - E.g. Block i at level $k+1$ must be placed in block $(i \bmod 4)$ at level k .
- Conflict misses occur when the level k cache is large enough, but multiple data objects all map to the same level k block.
 - E.g. Referencing blocks 0, 8, 0, 8, 0, 8, ... would miss every time.

■ Capacity miss

- Occurs when the set of active cache blocks (**working set**) is larger than the cache.

Examples of Caching in the Hierarchy

Cache Type	What is Cached?	Where is it Cached?	Latency (cycles)	Managed By
Registers	4-8 bytes words	Processor core	0	Compiler
TLB	Address translations	On-Chip TLB	0	Hardware
L1 cache	64-bytes block	On-Chip L1	1	Hardware
L2 cache	64-bytes block	On/Off-Chip L2	10	Hardware
Virtual Memory	4-KB page	Main memory	100	Hardware + OS
Buffer cache	Parts of files	Main memory	100	OS
Disk cache	Disk sectors	Disk controller	100,000	Disk firmware
Network buffer cache	Parts of files	Local disk	10,000,000	AFS/NFS client
Browser cache	Web pages	Local disk	10,000,000	Web browser
Web cache	Web pages	Remote server disks	1,000,000,000	Web proxy server

Summary

- The speed gap between CPU, memory and mass storage continues to widen.
- Well-written programs exhibit a property called locality.
- Memory hierarchies based on caching close the gap by exploiting locality.

Questions?

Introduction to the Architecture of Computers

Professor Hugh C. Lauer
CS-2011, Machine Organization and Assembly Language

(Slides include copyright materials from *Computer Systems: A Programmer's Perspective*, by Bryant and O'Hallaron, and from *The C Programming Language*, by Kernighan and Ritchie)

Today

- Before electronic computers
- Logic and gates
- Latches and Registers

Before electronic computers



Data values represented by positions of beads

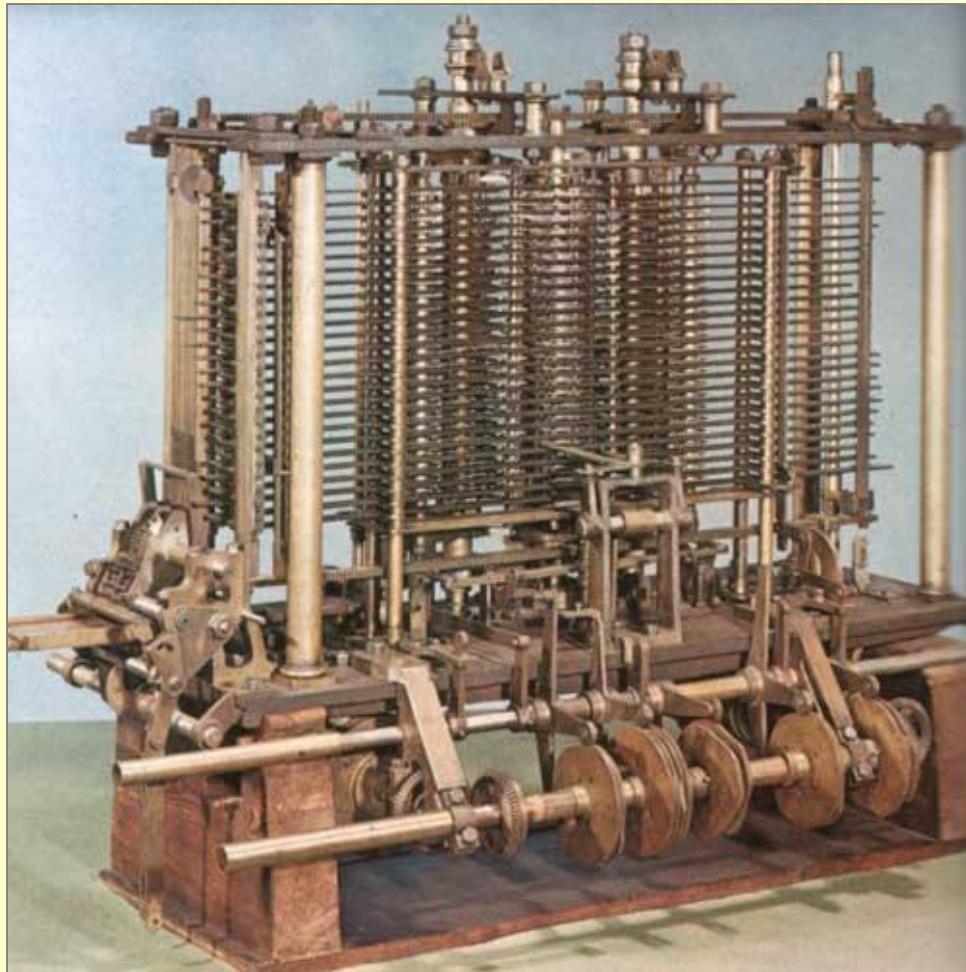
Arithmetic by manual algorithm



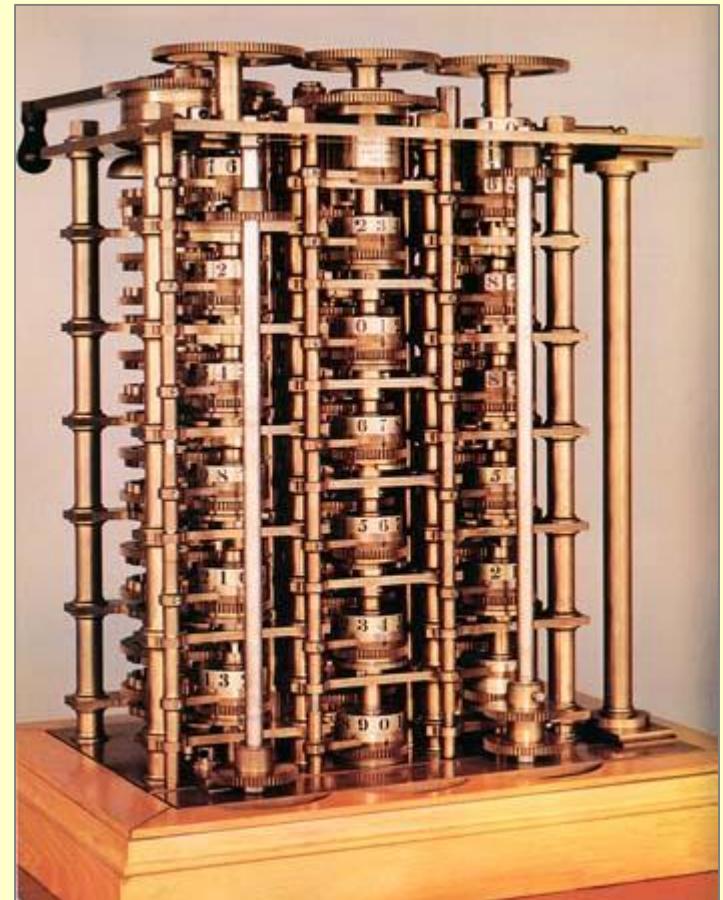
Data values represented by rotational positions of wheels and dials

Arithmetic by rotating wheels and gears

Charles Babbage “engines”



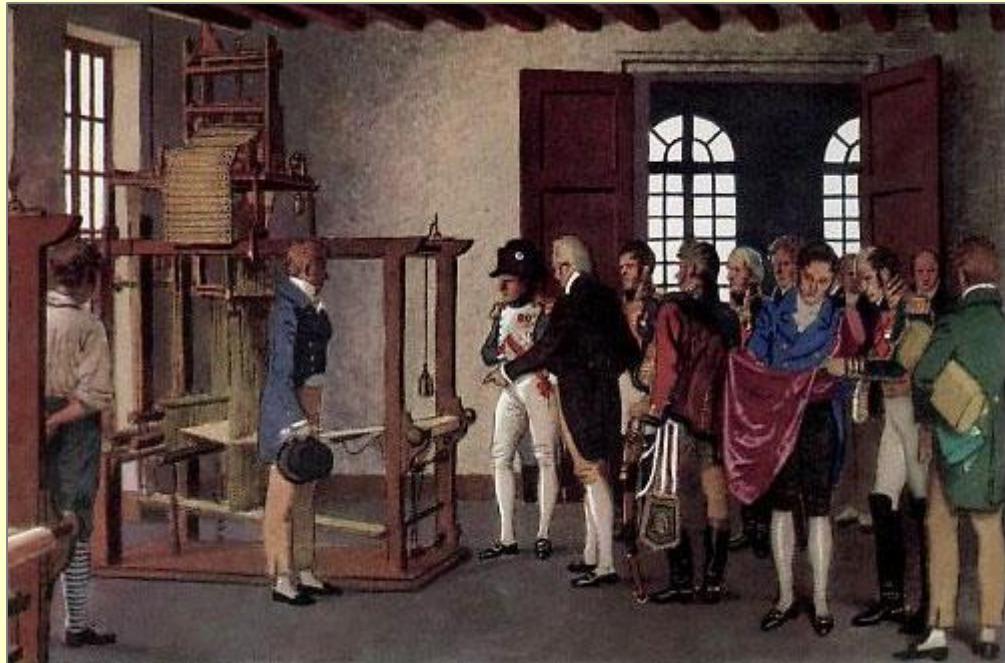
Analytical engine



Difference engine

Data values represented
by rotational positions of
wheels and dials

Jacquard Loom



Punched cards for controlling
patterns of woven cloth



Punched cards were part of
Babbage's design for data
entry and program control

Punched card tabulating equipment



Late 19th century



Mid 20th century

Data stored in trays (i.e., “files”) of punched cards
algorithms coded into plug-boards to operate on
data, punch new cards, etc.

Today

- Before electronic computers
- Logic and gates
- Latches and Registers

Reading Assignment: §4.2

Overview of Logic Design

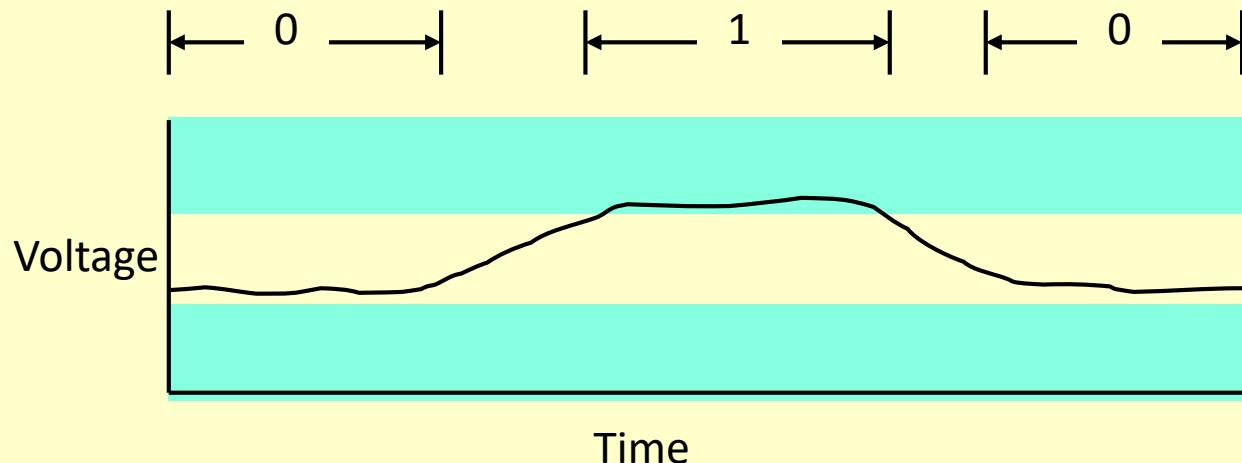
■ Fundamental Hardware Requirements

- Communication
 - How to get values from one place to another
- Computation
- Storage

■ Bits are Our Friends

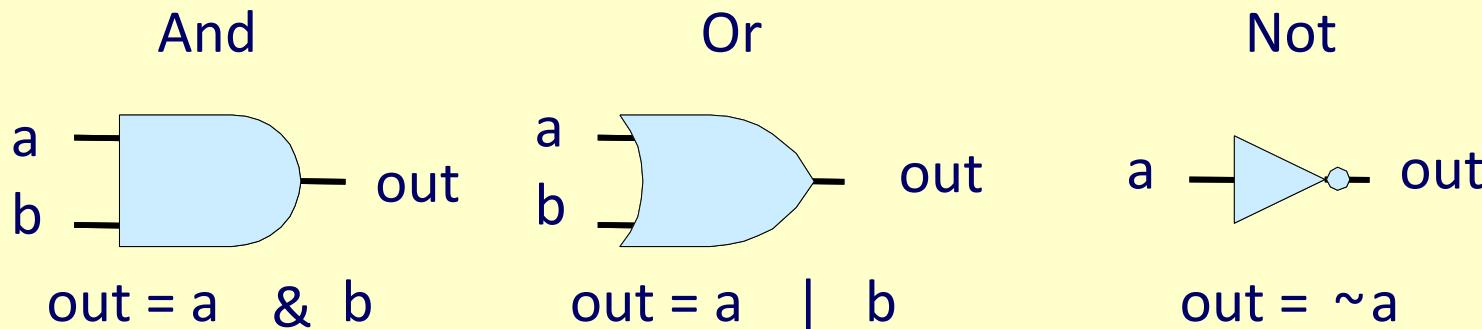
- Everything expressed in terms of values 0 and 1
- Communication
 - Low or high voltage on wire
- Computation
 - Compute Boolean functions
- Storage
 - Store bits of information

Digital Signals

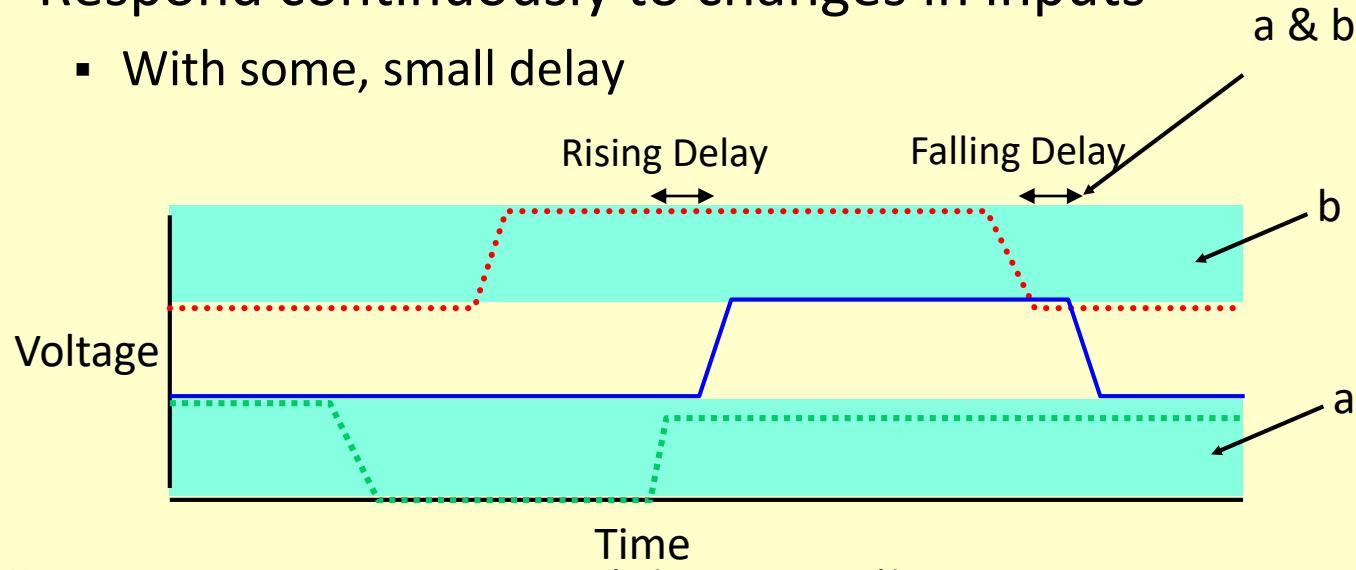


- Use voltage thresholds to extract discrete values from continuous signal
- Simplest version: 1-bit signal
 - Either high range (1) or low range (0)
 - With guard range between them
- Not strongly affected by noise or low quality circuit elements
 - Can make circuits simple, small, and fast

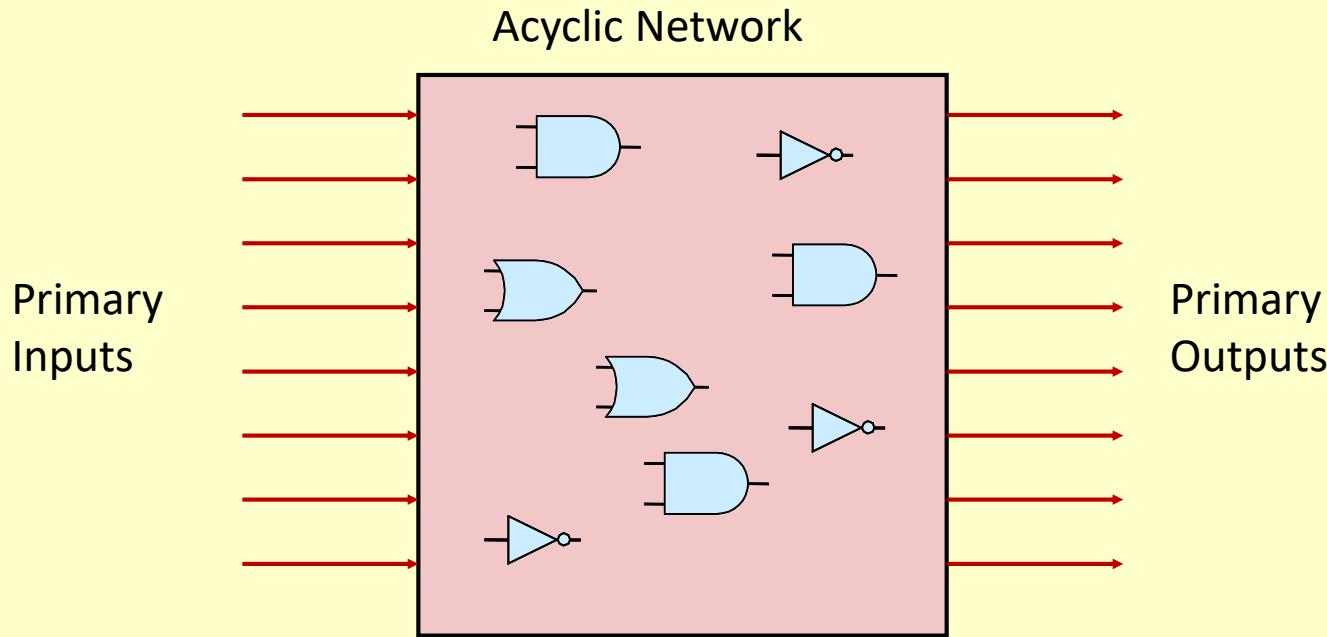
Computing with Logic Gates



- Outputs are Boolean functions of inputs
- Respond continuously to changes in inputs
 - With some, small delay



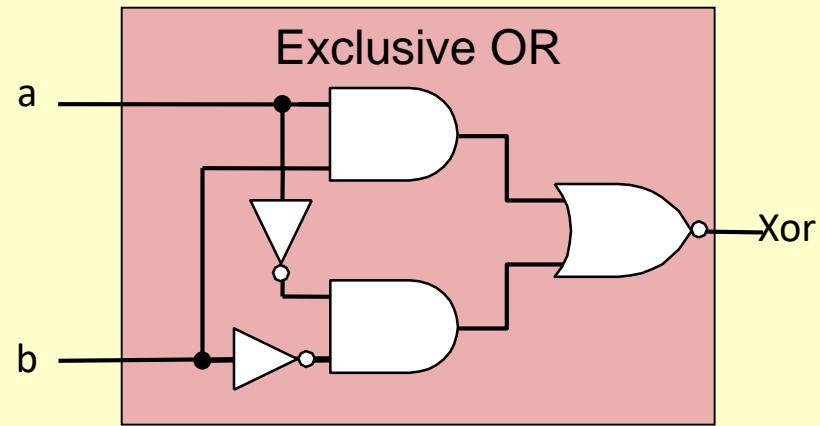
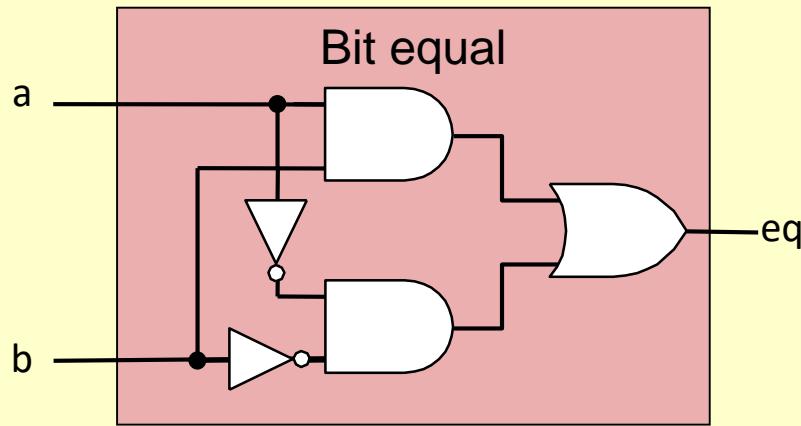
Combinational Circuits



■ Acyclic Network of Logic Gates

- Continuously responds to changes on primary inputs
- Primary outputs become (after some delay) Boolean functions of primary inputs

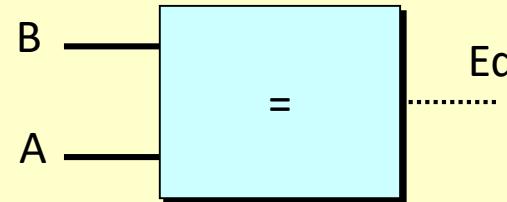
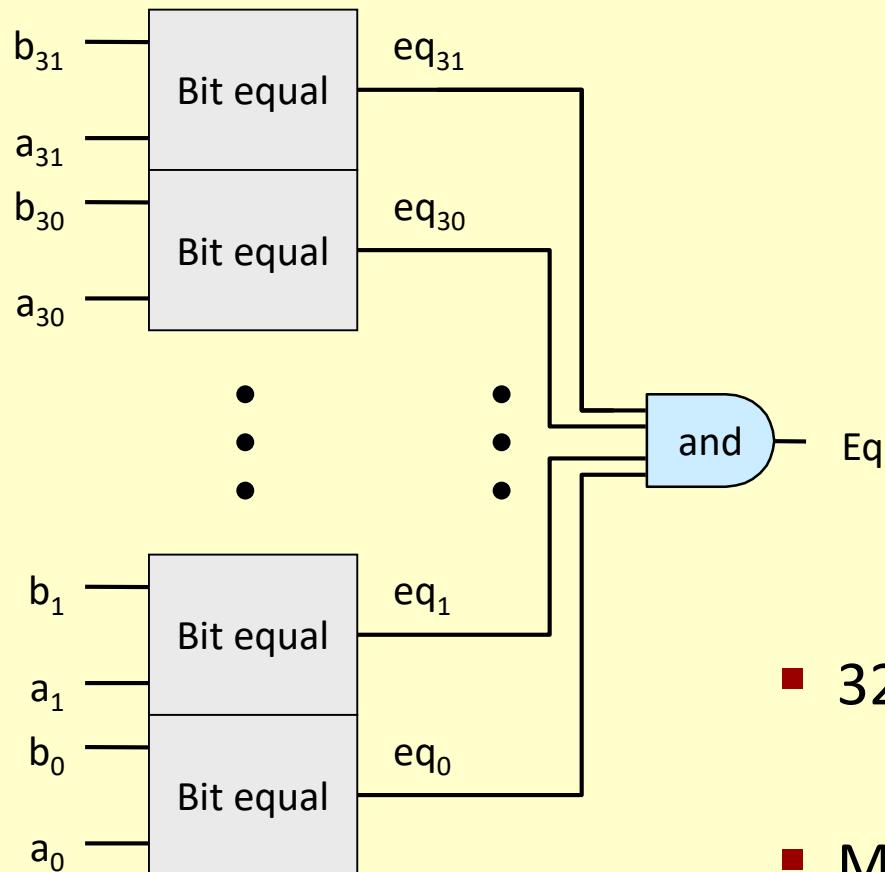
Bit Equality and Exclusive OR



- Generate 1 if a and b are equal
- Generate 1 if a and b are *not* equal

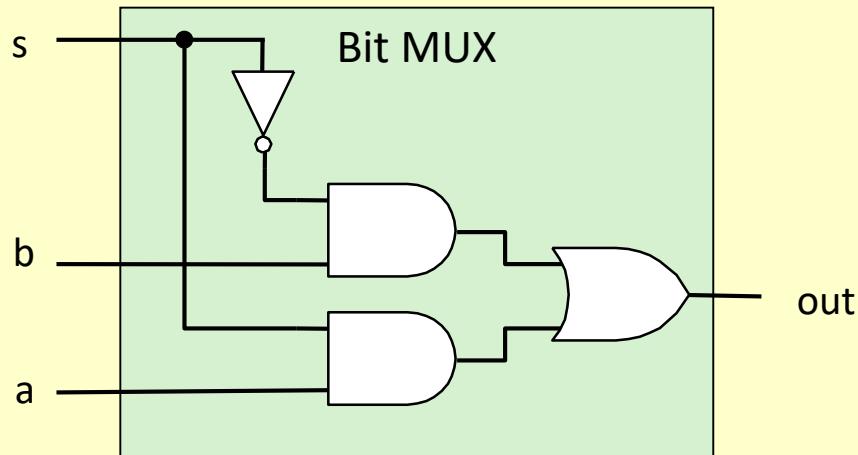
Word Equality

Word-Level Representation



- 32-bit word size
- May be adapted to any word size

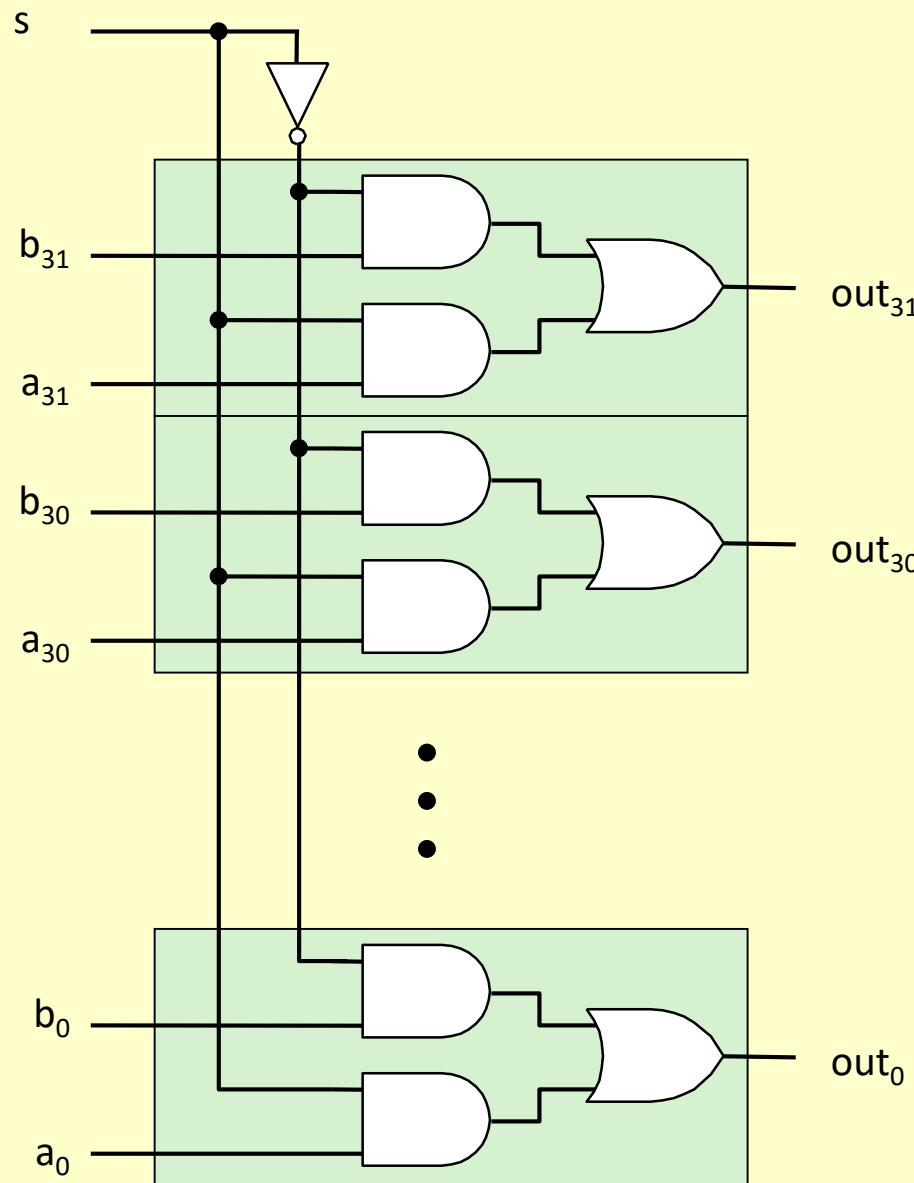
Bit-Level Multiplexor



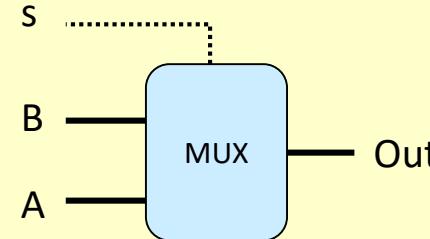
```
bool out = (s&a) || (!s&b)
```

- Control signal s
- Data signals a and b
- Output a when $s=1$, b when $s=0$

Word Multiplexor



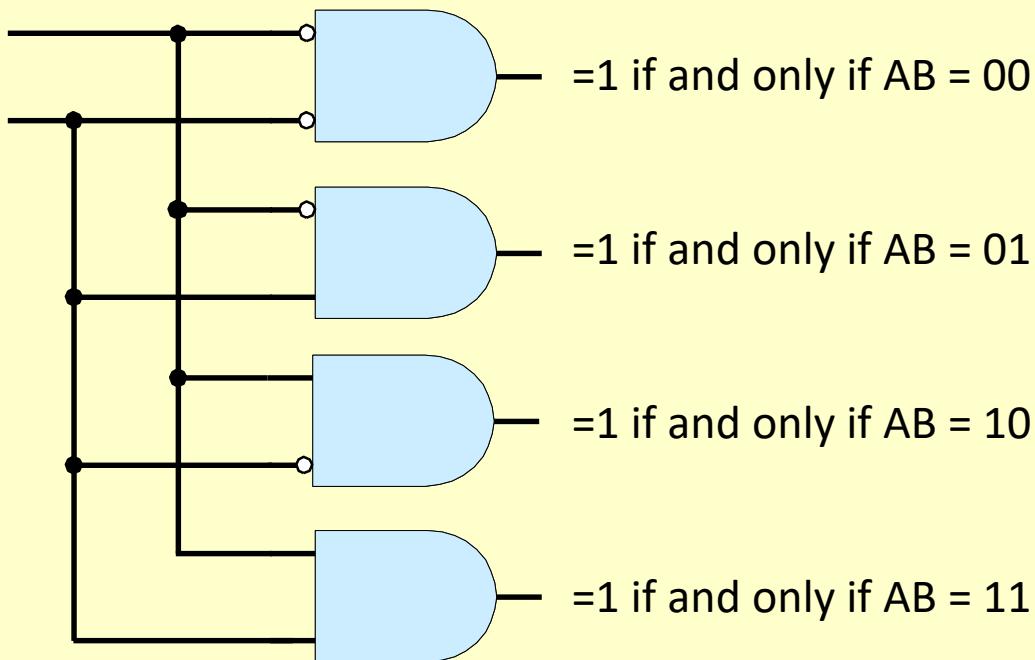
Word-Level Representation



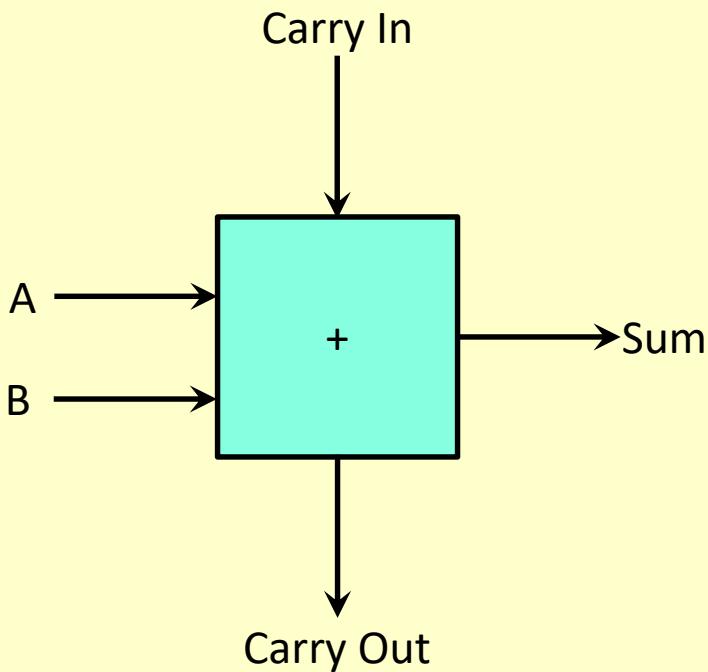
- Select input word A or B depending on control signal s

Decoder

- Opposite of Multiplexor
- Selects one of 2^n outputs from n inputs



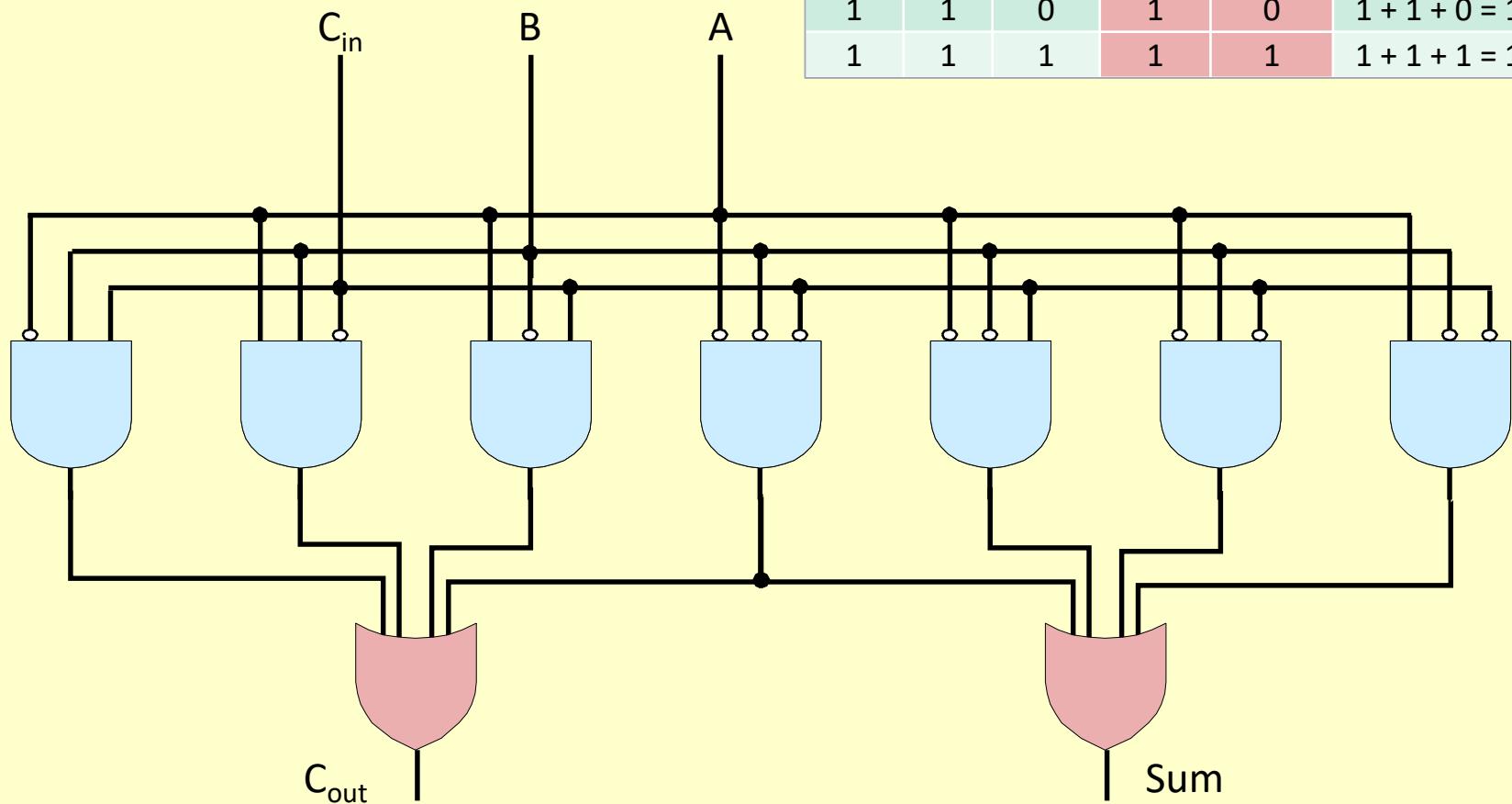
Single-bit adder



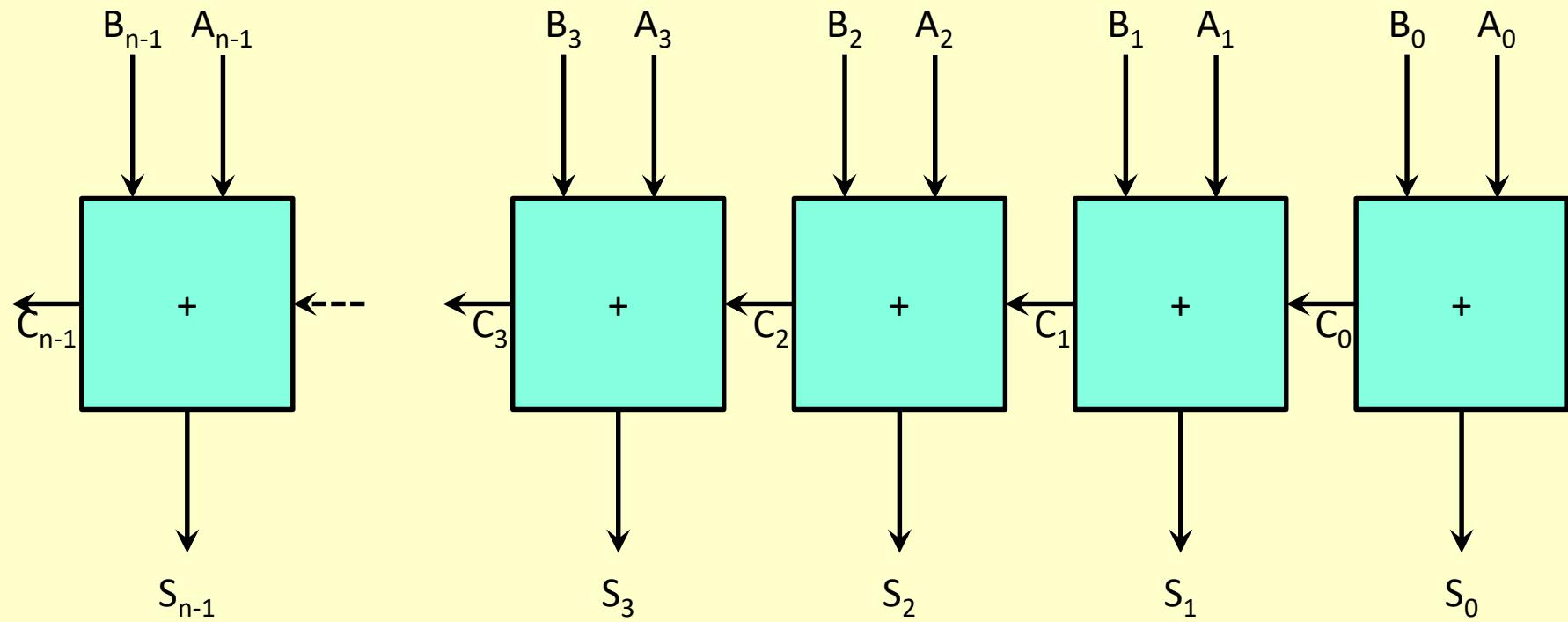
A	B	Carry In	Carry Out	Sum	
0	0	0	0	0	$0 + 0 + 0 = 00_2$
0	0	1	0	1	$0 + 0 + 1 = 01_2$
0	1	0	0	1	$0 + 1 + 0 = 01_2$
0	1	1	1	0	$0 + 1 + 1 = 10_2$
1	0	0	0	1	$1 + 0 + 0 = 01_2$
1	0	1	1	0	$1 + 0 + 1 = 10_2$
1	1	0	1	0	$1 + 1 + 0 = 10_2$
1	1	1	1	1	$1 + 1 + 1 = 11_2$

Single-bit adder (cont.)

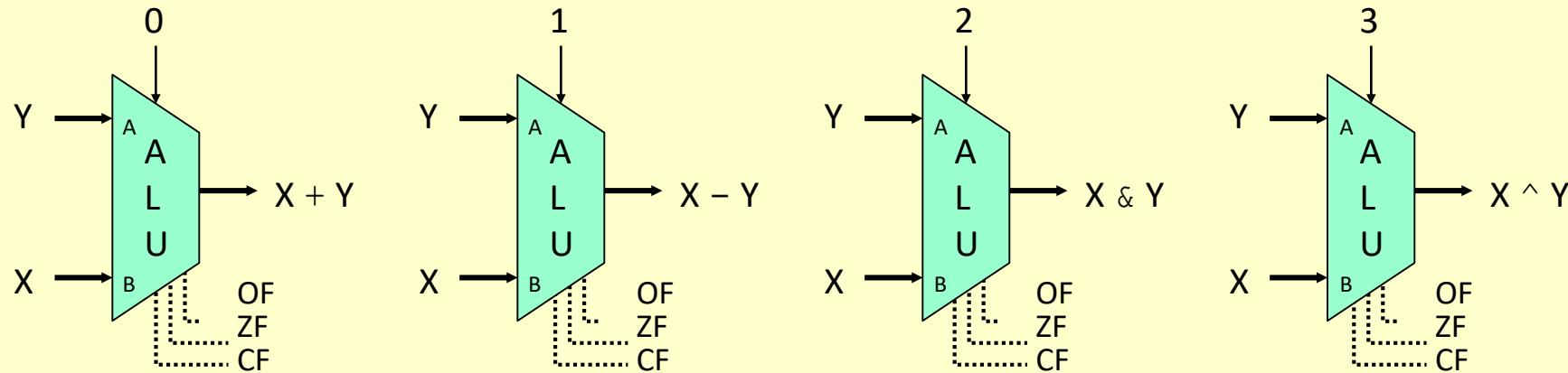
A	B	Carry In	Carry Out	Sum	
0	0	0	0	0	$0 + 0 + 0 = 00_2$
0	0	1	0	1	$0 + 0 + 1 = 01_2$
0	1	0	0	1	$0 + 1 + 0 = 01_2$
0	1	1	1	0	$0 + 1 + 1 = 10_2$
1	0	0	0	1	$1 + 0 + 0 = 01_2$
1	0	1	1	0	$1 + 0 + 1 = 10_2$
1	1	0	1	0	$1 + 1 + 0 = 10_2$
1	1	1	1	1	$1 + 1 + 1 = 11_2$



Multi-bit adder



Arithmetic Logic Unit (single-bit example)



- Combinational logic
 - Continuously responding to inputs
- Control signal selects function computed
 - Corresponding to four basic arithmetic/logical operations
- Also computes values for condition codes

Figure 4.15
(Not in our textbook)

Modern Arithmetic-Logic Unit (ALU)

■ Combines

- Add
- Subtract
- And
- Or
- Not
- Xor
- Equality
- <
- >
- <<
- >>
- ...

■ Outputs

- Result
- CF — Carry flag
- ZF — Zero flag
- SF — Sign flag
- OF — Overflow flag

Result developed within one cycle (300 ps)

Conspicuously absent:— multiplication & division!

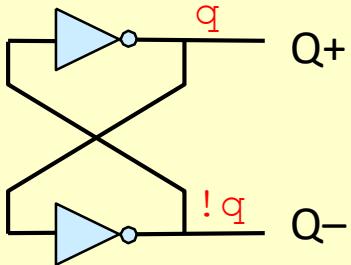
Questions?

Today

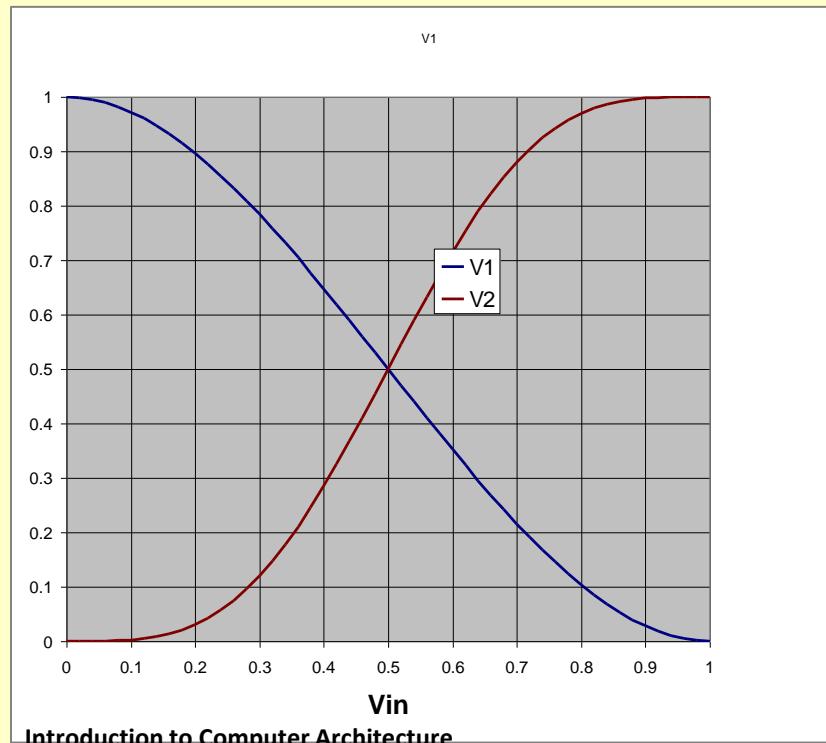
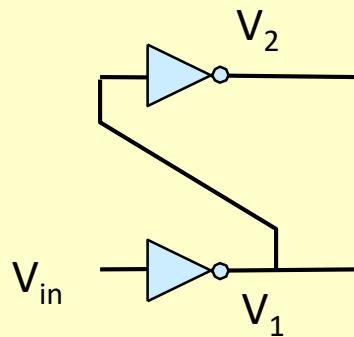
- Before electronic computers
- Logic and gates
- Latches and Registers

Storing 1 Bit

Bistable Element

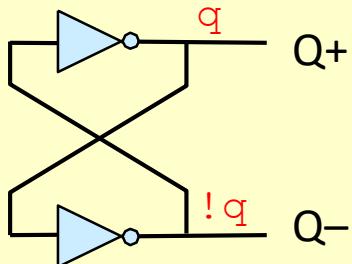


\bar{q} = 0 or 1

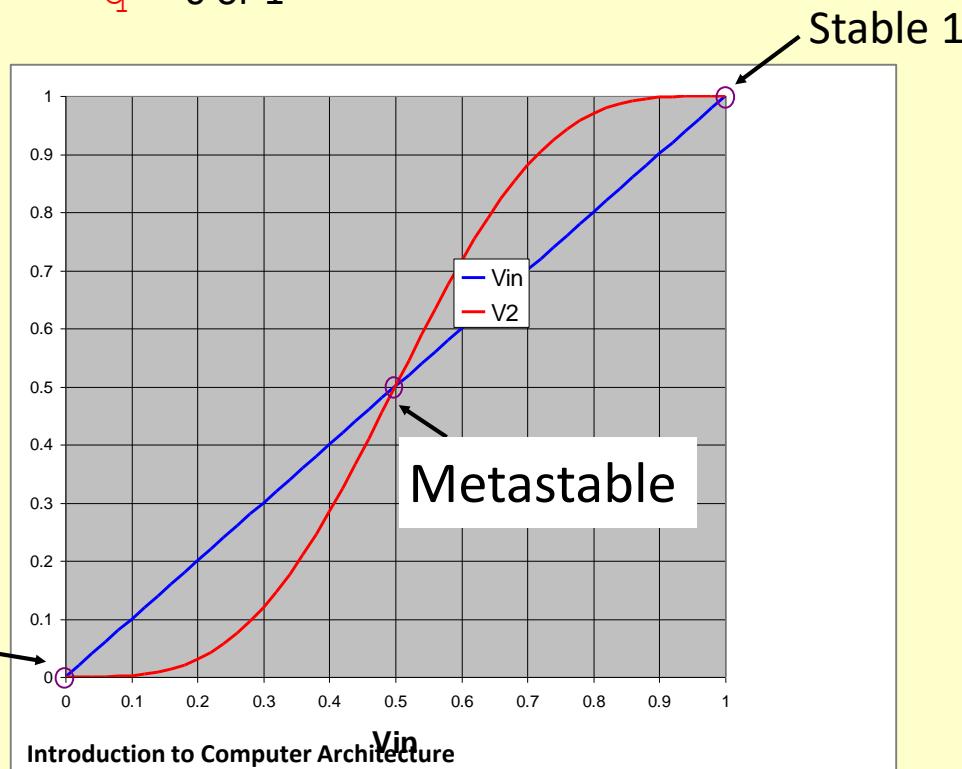
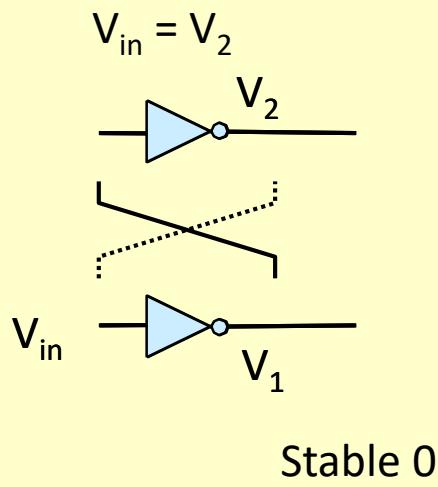


Storing 1 Bit (continued)

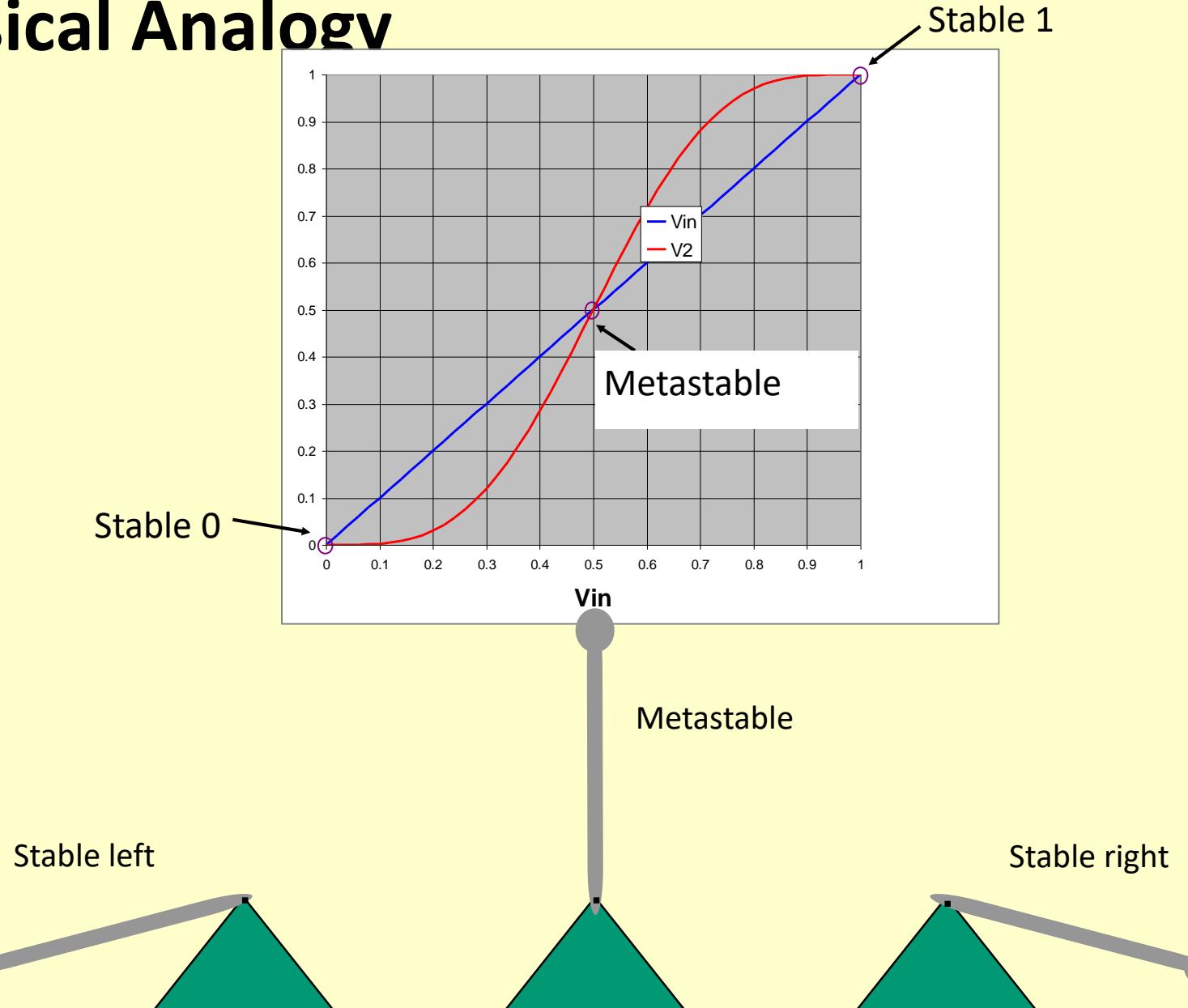
Bistable Element



$q = 0 \text{ or } 1$

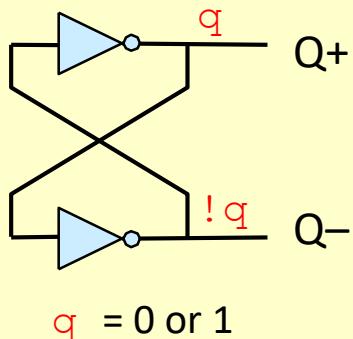


Physical Analogy

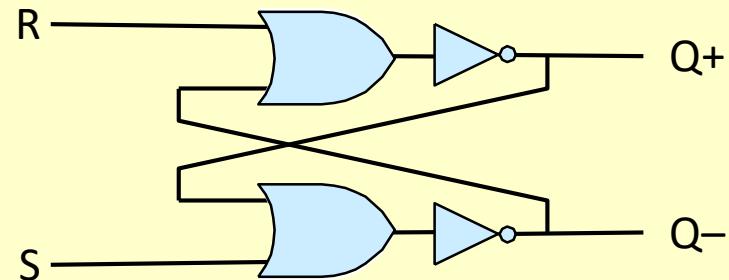


Storing and Accessing 1 Bit

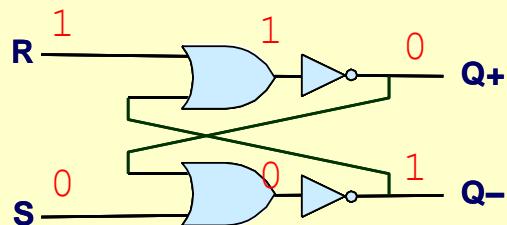
Bistable Element



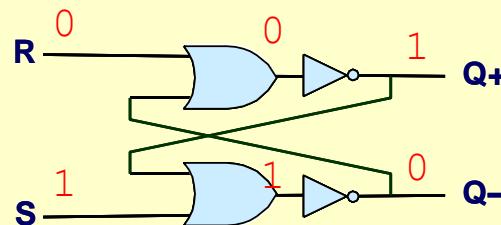
R-S Latch



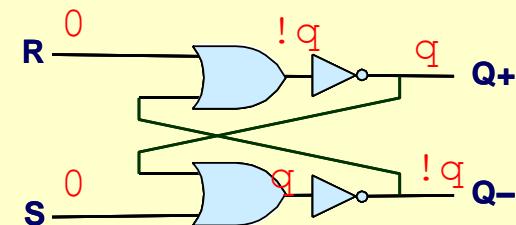
Resetting



Setting

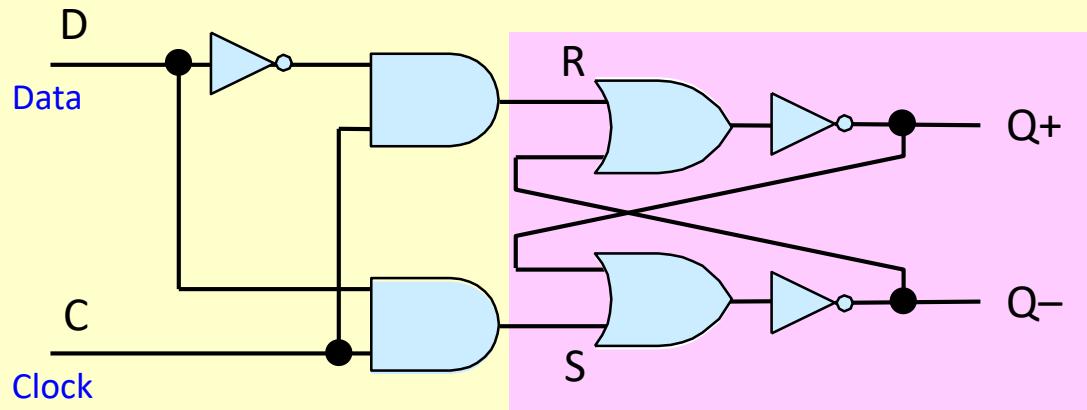


Storing

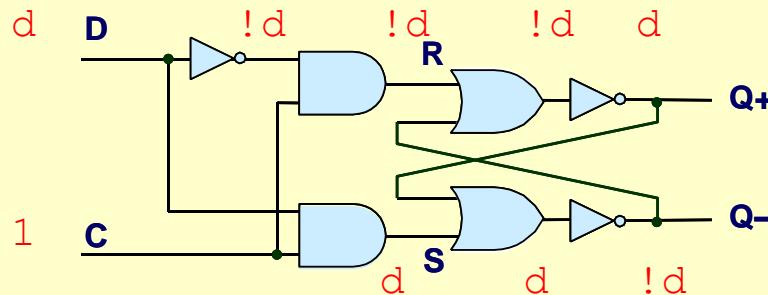


1-Bit Latch

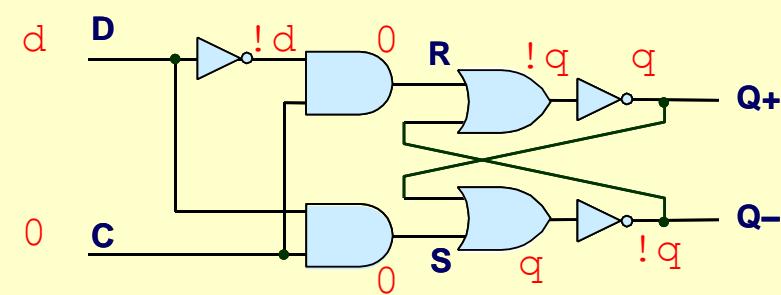
D Latch



Latching



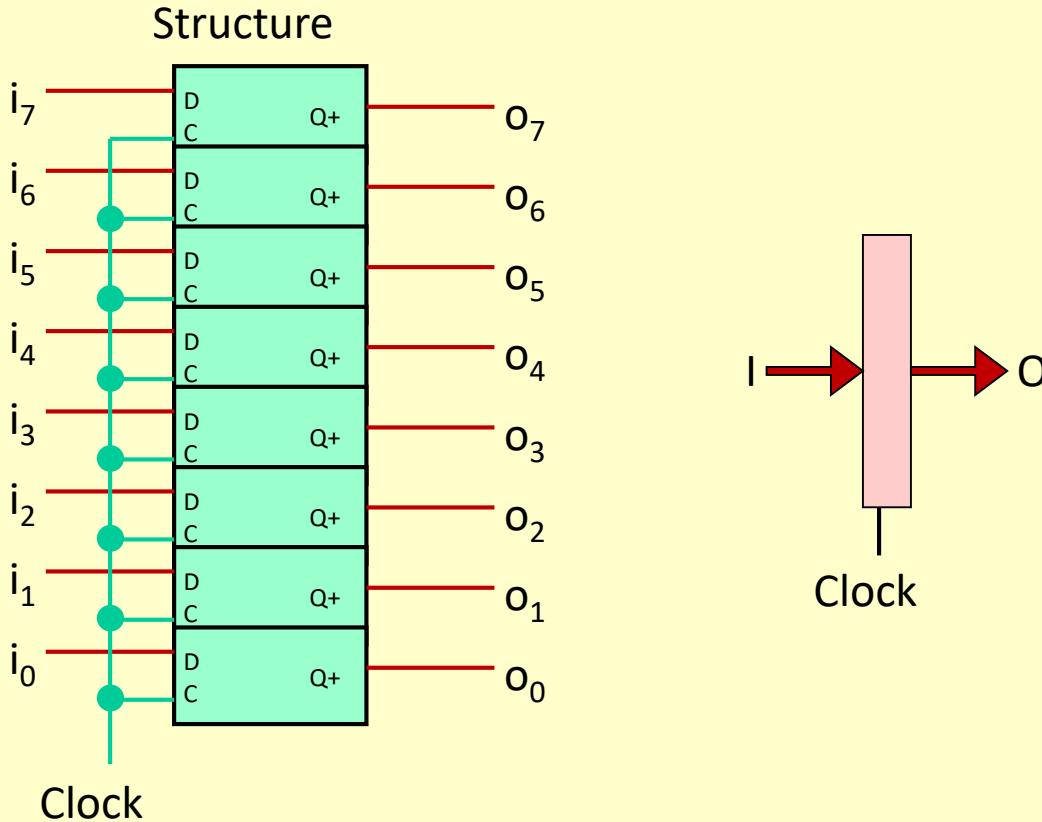
Storing



Definition — *Clock*

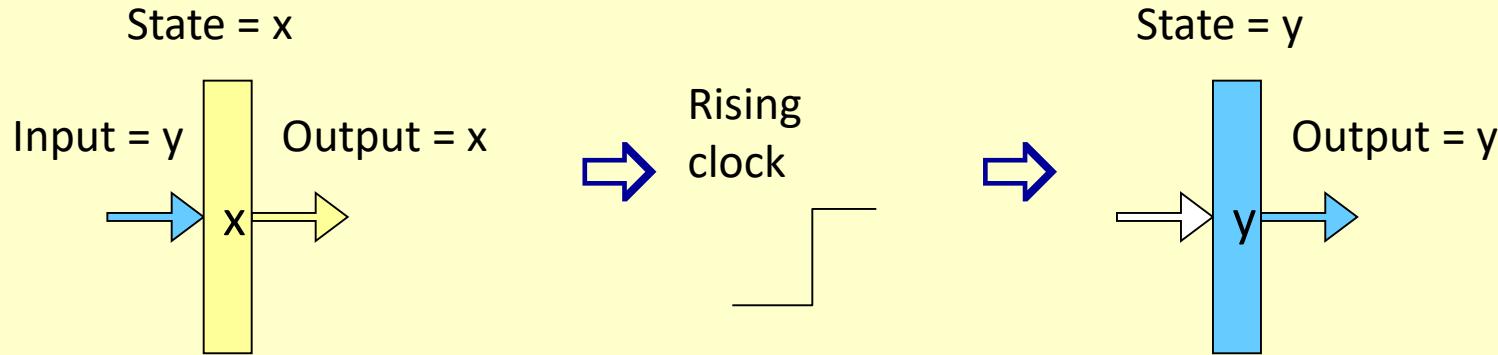
- A periodic signal consisting of an alternating sequence of ones and zeros at regular intervals
 - E.g., 333 picoseconds in a modern 3 GHz processor
- Purpose:— to control *when* to capture the result of a logic circuit
 - E.g., an ALU consisting of AND's, OR's, and NOT's
 - E.g., a register or memory or other device

Register



- Stores word of data
 - Different from *program registers* seen in assembly code
- Collection of edge-triggered latches
- Loads input on rising edge of clock

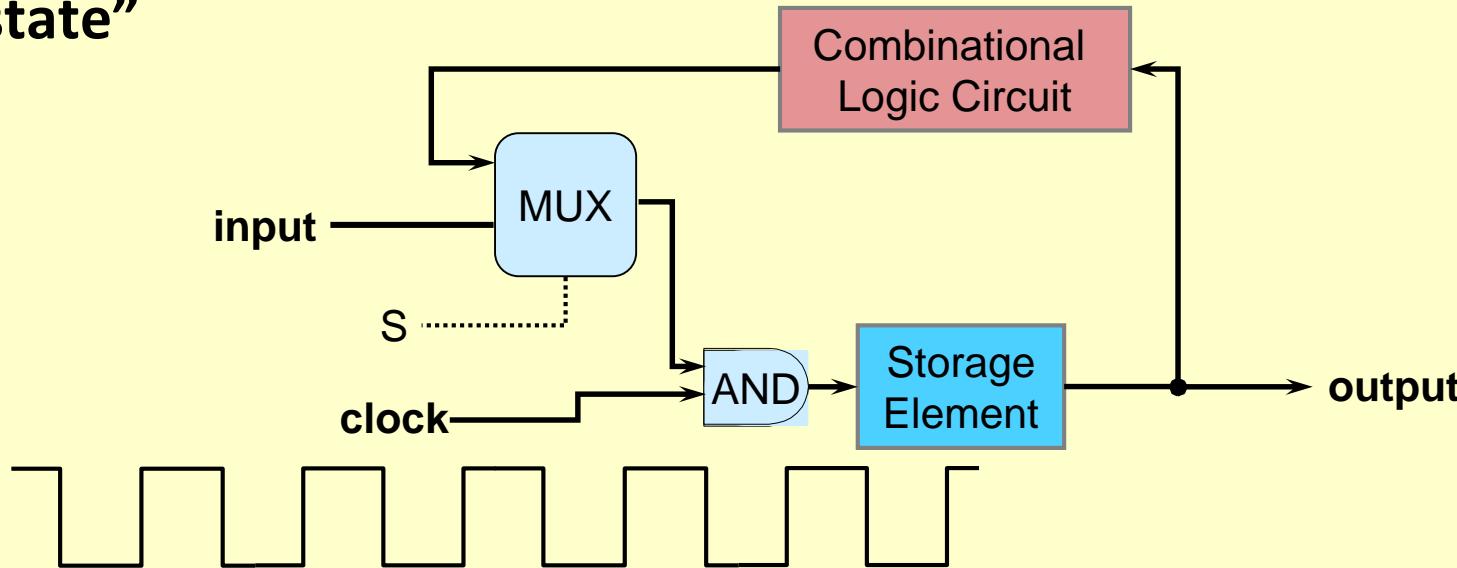
Register Operation



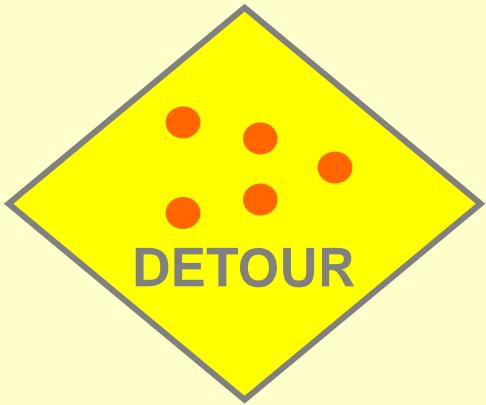
- Stores data bits
- For most of time acts as barrier between input and output
- As clock rises, loads input

(Finite) State Machine

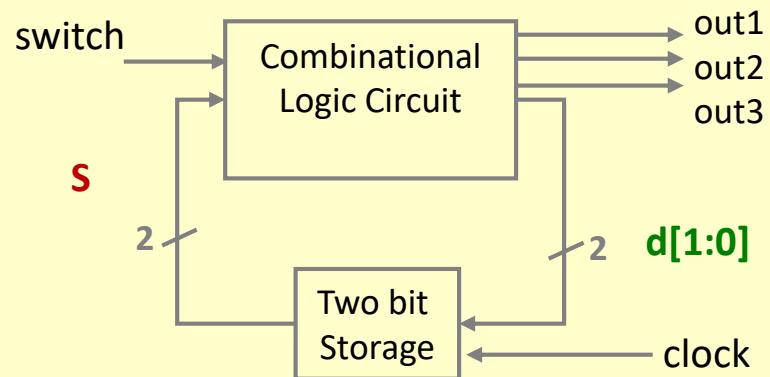
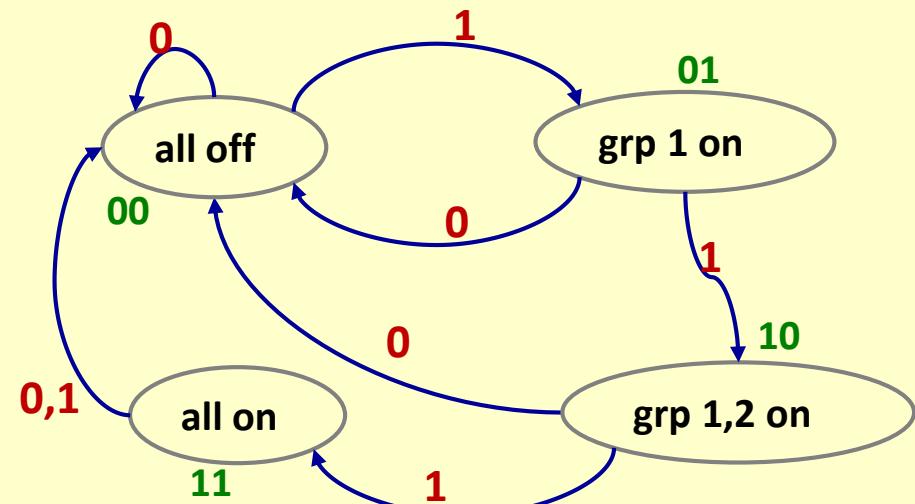
- A register or latch of n bits representing the “state” of the circuit
- An acyclic network of combinatorial logic to compute a new value of n bits based on the existing value of n bits
- A clock signal to effect the update of the “state” to a new “state”



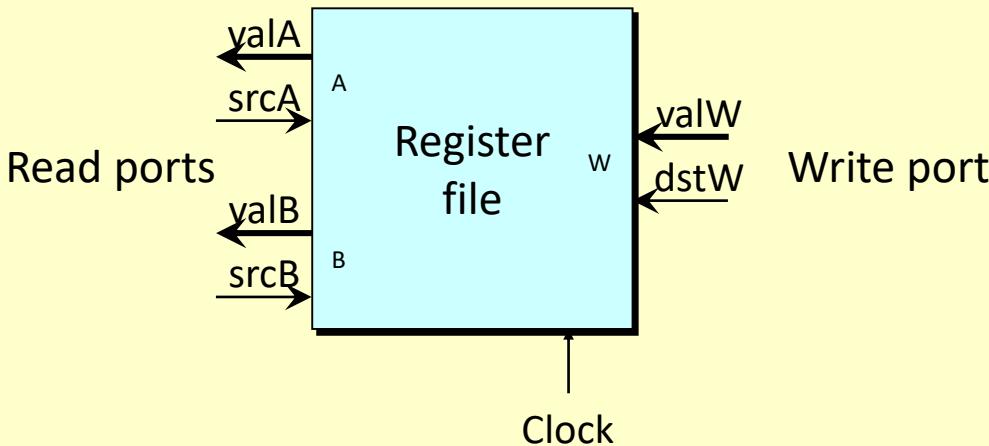
Finite State Machine Example



- Three groups of lights to be lit in a sequence: group 1 on, groups 1 & 2 on, all groups on, all off.
- The lights are on only if the main switch is on.
- Four states: so we need two bits to identify each state.



Register File



- Stores multiple words of memory
 - Address input specifies which word to read or write
- Register file
 - Holds values of program registers
 - **%rax, %rsp**, etc.
 - Register identifier serves as address
 - ID 15 (0xF) implies no read or write performed
- Multiple Ports
 - Can read and/or write multiple words in one cycle
 - Each has separate address and data input/output



Summary

- **Data values stored as bits**
 - On wires, in memory cells, etc.
- **Gates are logic elements that combine values of bits to produce other bits**
 - And, Or, Xor, addition, subtraction, comparison, etc.
- **Latches capture bit values on wires and keep them until reset**
 - So long as power stays on
- **Setting of latches is triggered by a clock, which allow data into the latches only when the results of combinatorial logic elements has stabilized.**

Reading Assignment: §4.2

Questions?

Intro to Architecture of Computers – III

Pipelining

Hugh C. Lauer

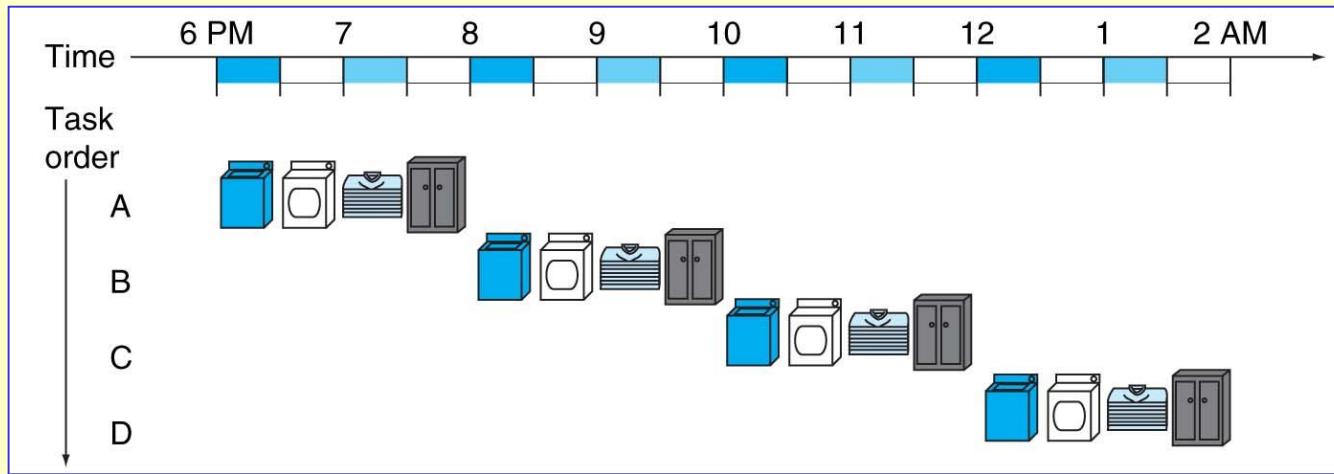
Department of Computer Science

(Slides shamelessly adapted from Bryant & O'Hallaron, with additional materials from Patterson & Hennessey, "Computer Organization & Design," revised 4th ed. and from Hennessey & Patterson, "Computer Architecture: A Quantitative Approach," 4th ed.)

Today

- Analogies in real world
- Pipelining in modern processors — MIPS
- Pipelining in Y86
- Hazards

Clothes-washing analogy



Time per load:-
2 hours (still)

Time for 4 loads:-
3.5 hours!

2 loads per hour
sustained rate!

Real-world pipelines: car washes

Sequential



Parallel



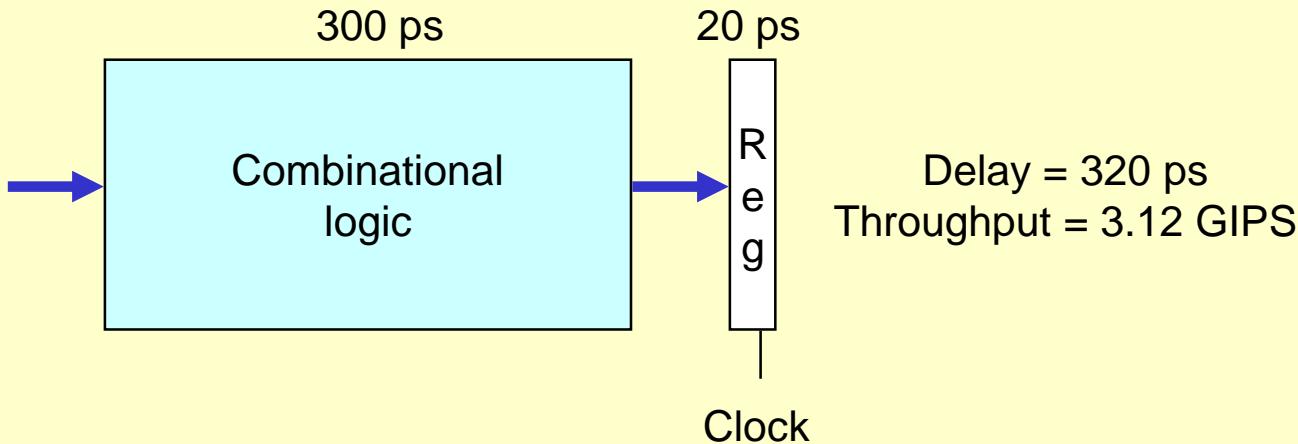
Pipelined



■ Idea

- Divide process into independent stages
- Move objects through stages in sequence
- At any given times, multiple objects being processed

Computational example

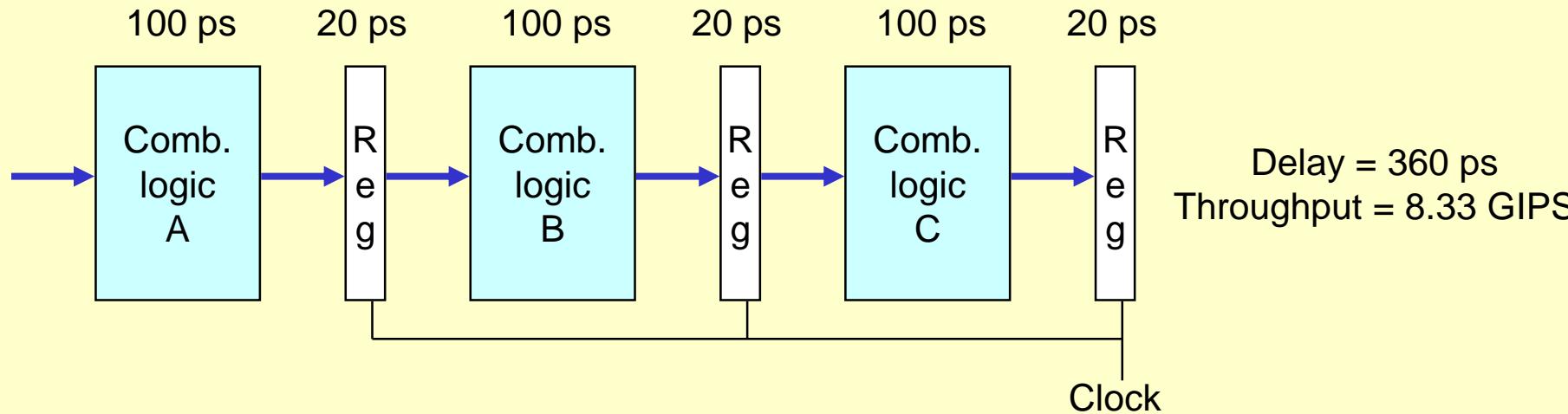


■ System

- Computation requires total of 300 picoseconds
- Additional 20 picoseconds to save result in register
- Must have clock cycle of at least 320 ps

- Can begin a new operation every cycle — i.e., every 320 ps

3-way pipelined version

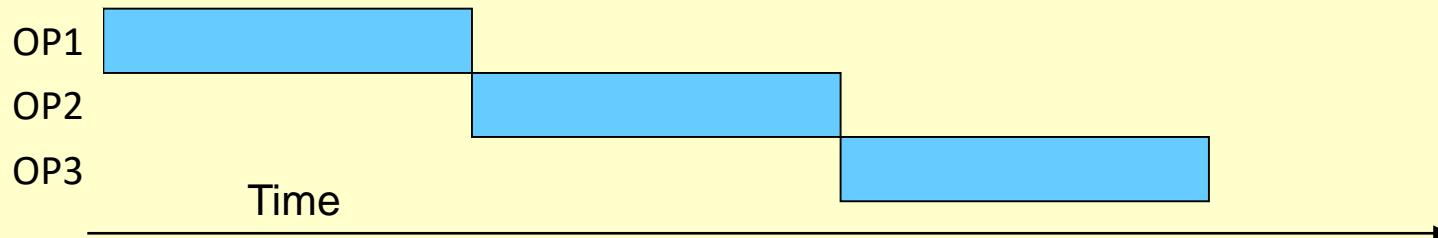


■ System

- Divide combinational logic into 3 blocks of 100 ps each
- Can begin new operation as soon as previous passes through stage A.
 - Begin new operation every 120 ps
- Overall latency per operation increases
 - 360 ps from start to finish

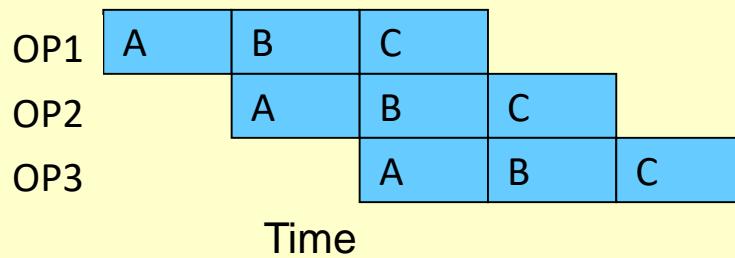
Pipeline diagrams

■ Unpipelined



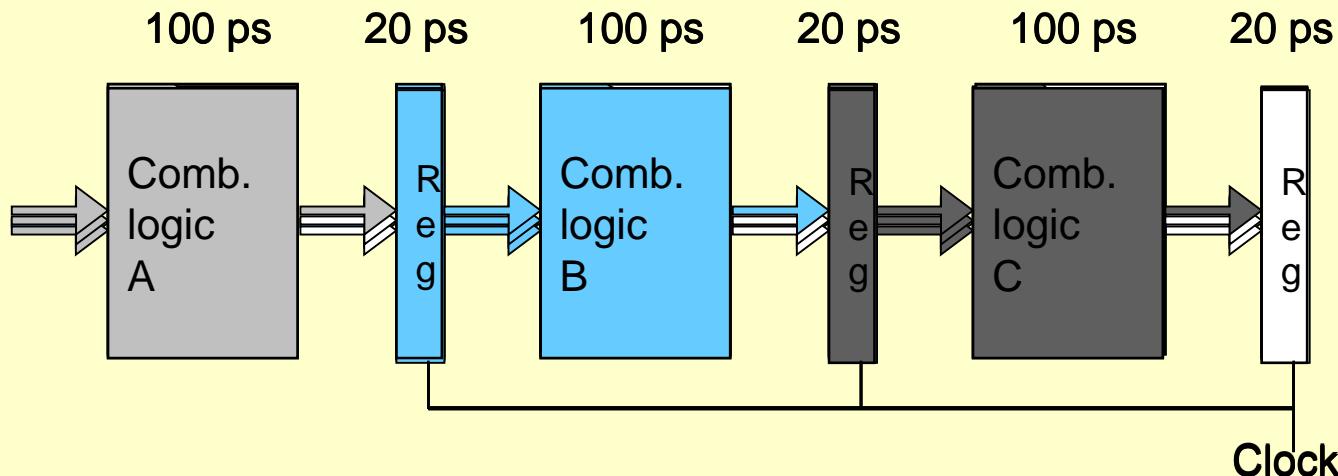
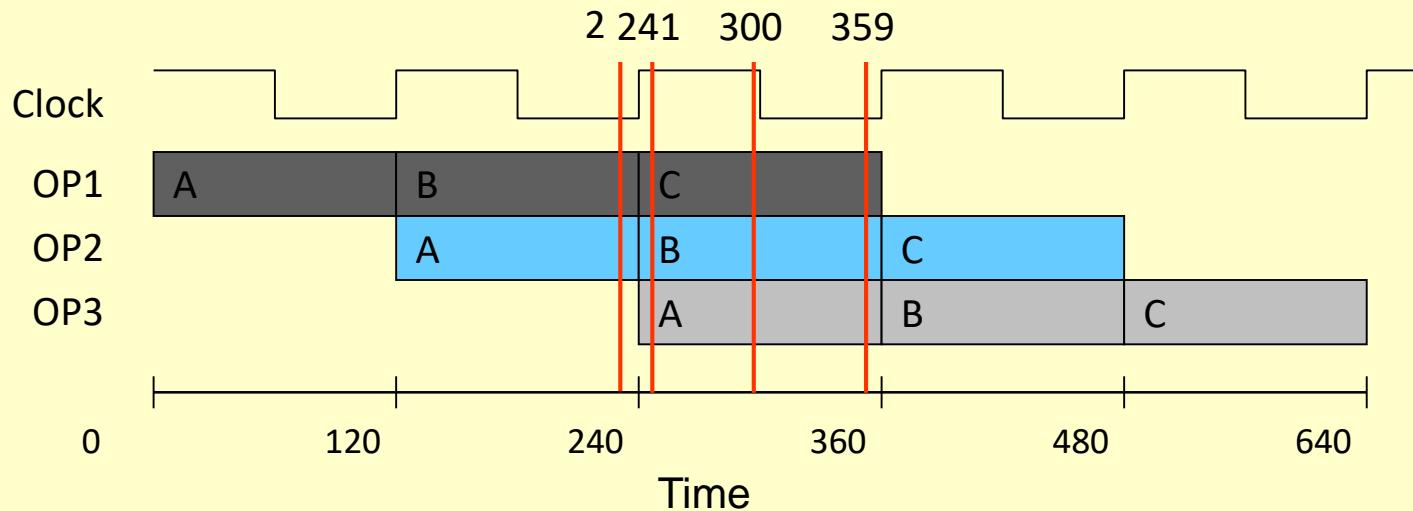
- Cannot start new operation until previous one completes

■ 3-Way Pipelined

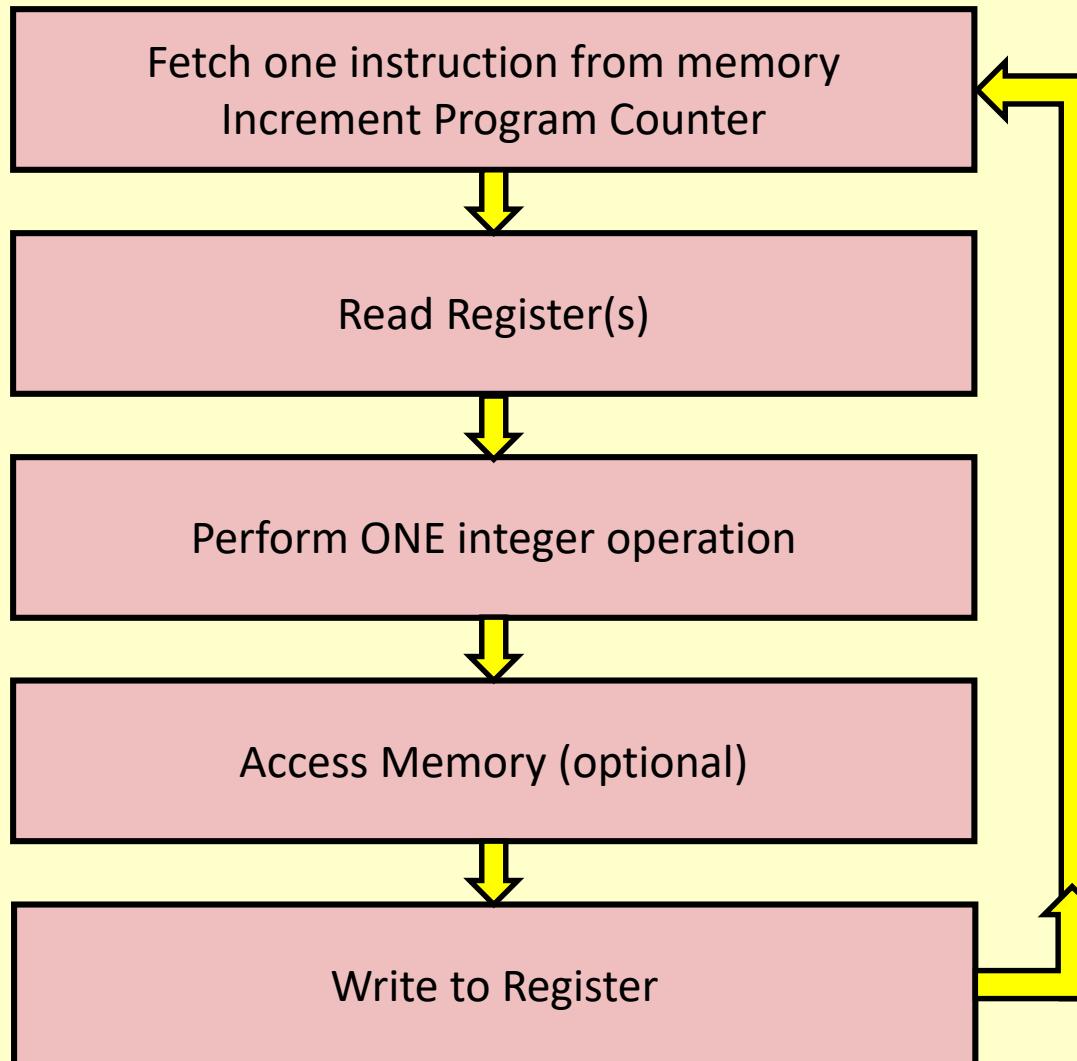


- Up to 3 operations in progress simultaneously

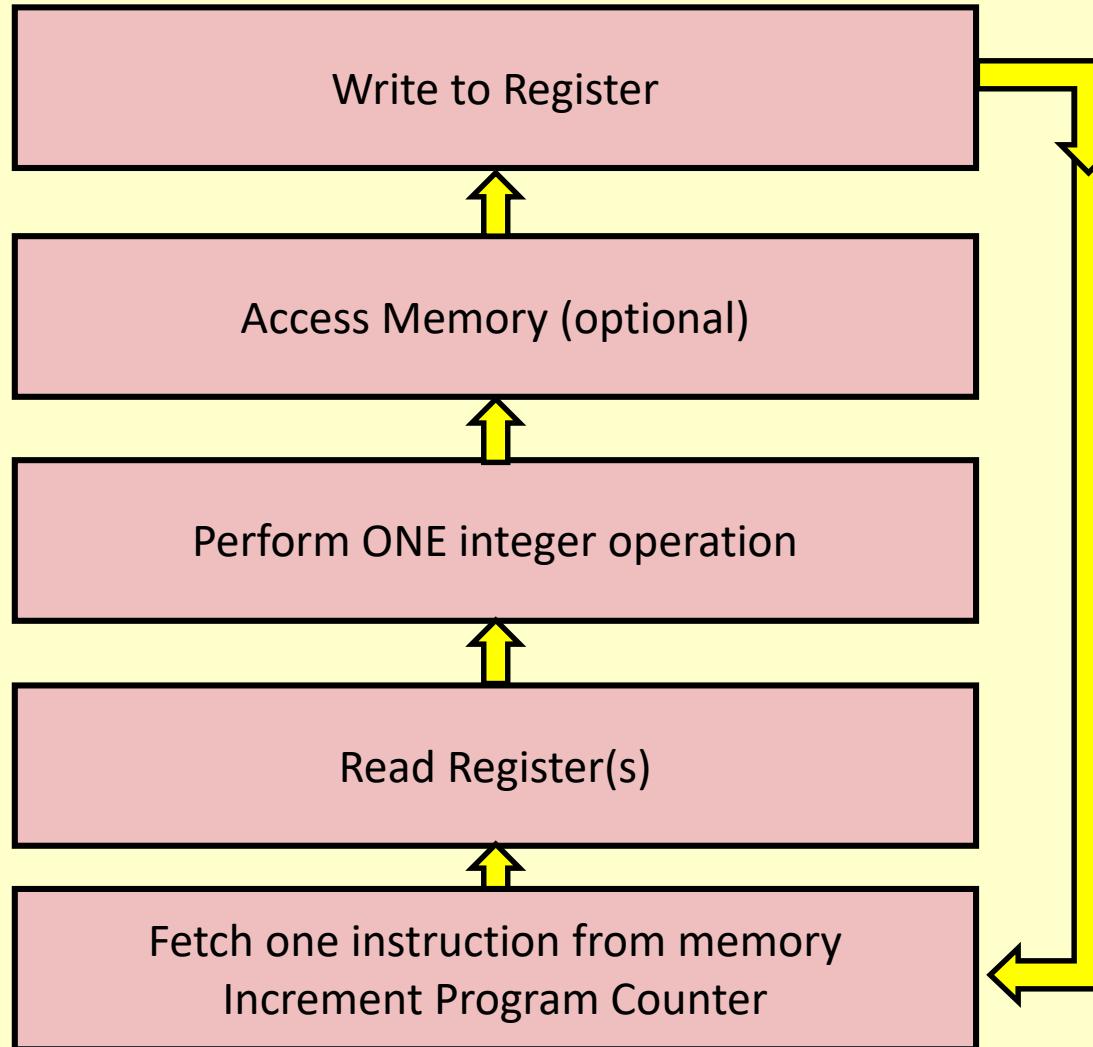
Operating a pipeline



Execution model for modern computers

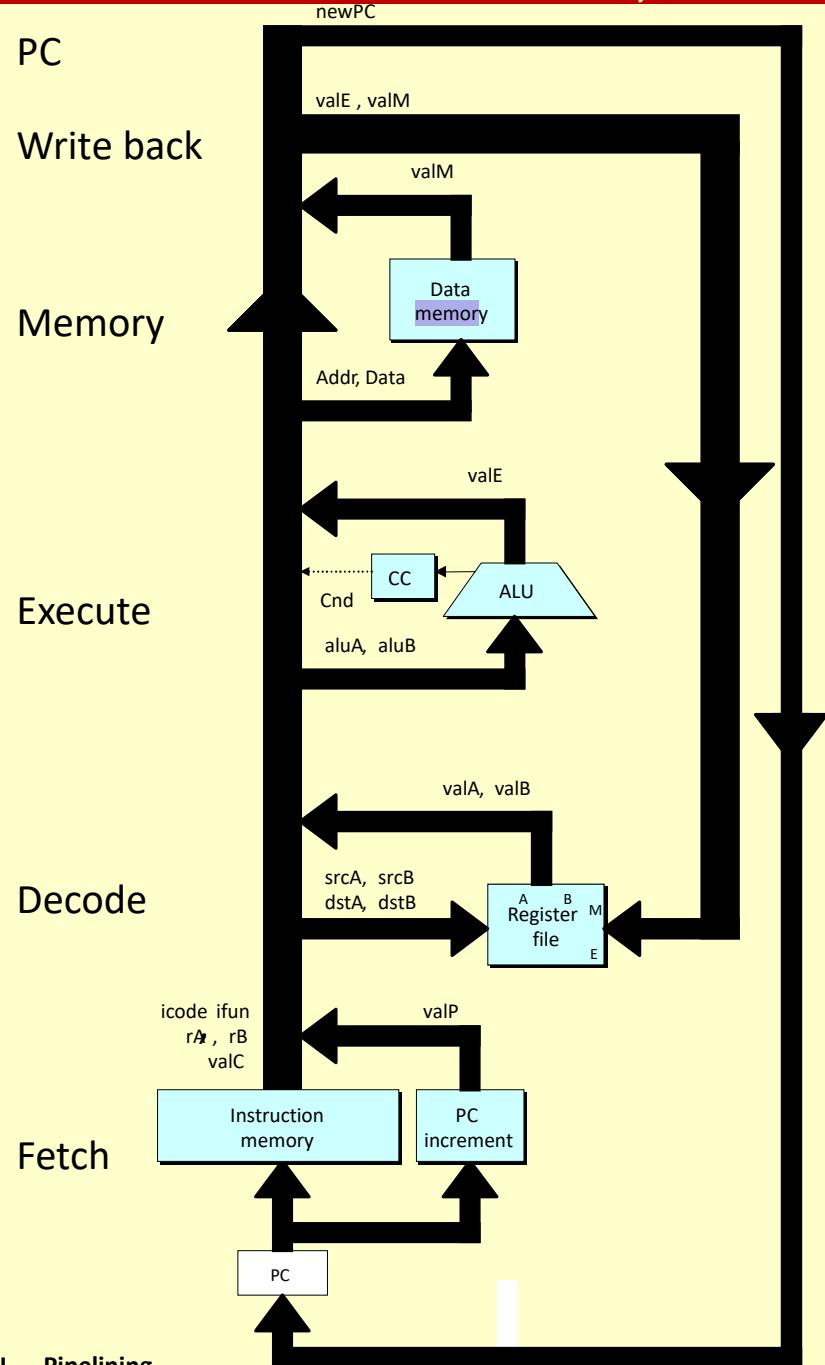


Execution model for modern computers (flipped vertically)



Sequential stages

- **Fetch**
 - Read instruction from instruction memory
 - Increment program counter (`%rip`)
- **Decode**
 - Read registers named in instruction
- **Execute**
 - Compute value or address
- **Memory**
 - Read or write data
- **Write Back**
 - Write program registers
- **PC**
 - Update program counter (incremented or jump/call/ret target)



Sequential stages

Fetch

- Read instruction from instruction memory
- Increment program counter (`%rip`)

I.e., five different instructions “in flight” at the same time!

Execute

- Compute value or address

Memory

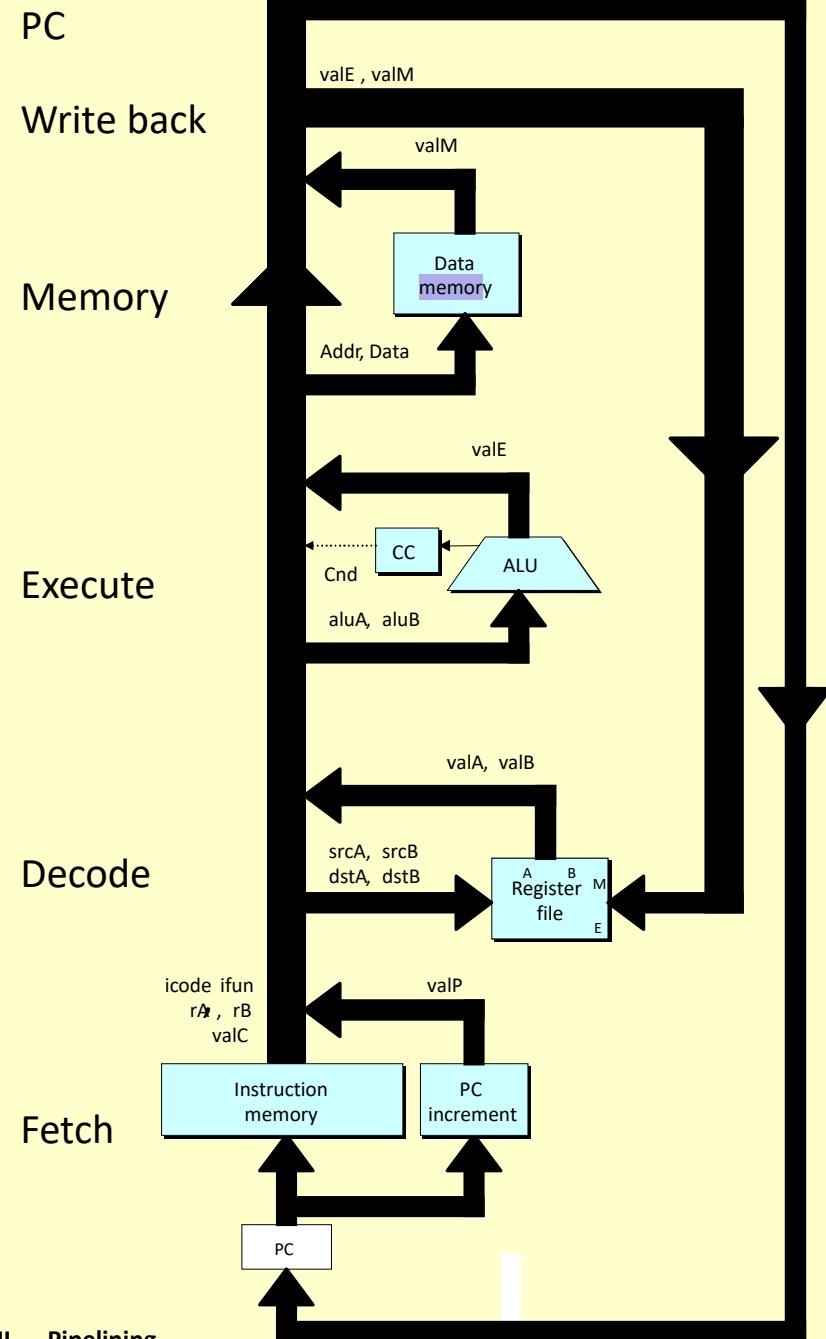
- Read or write data

Write Back

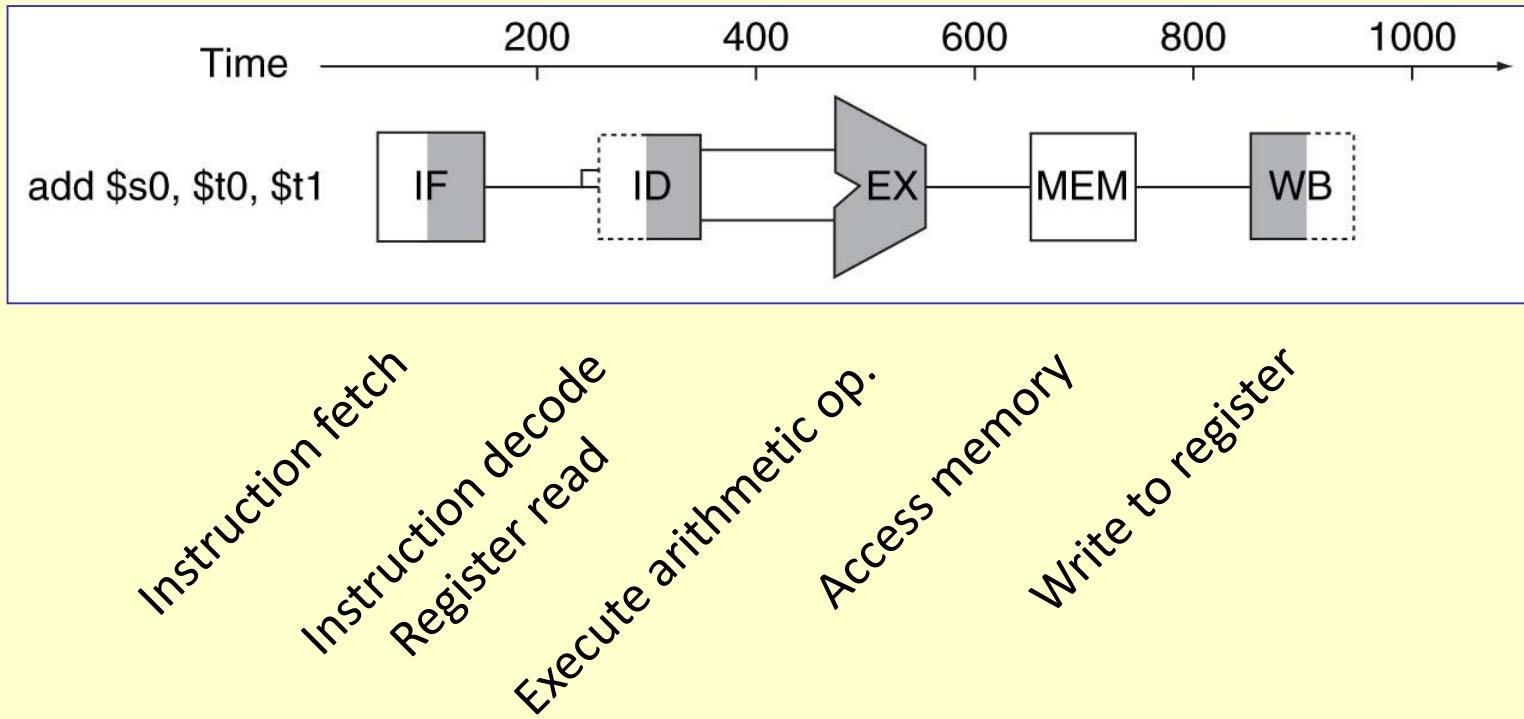
- Write program registers

PC

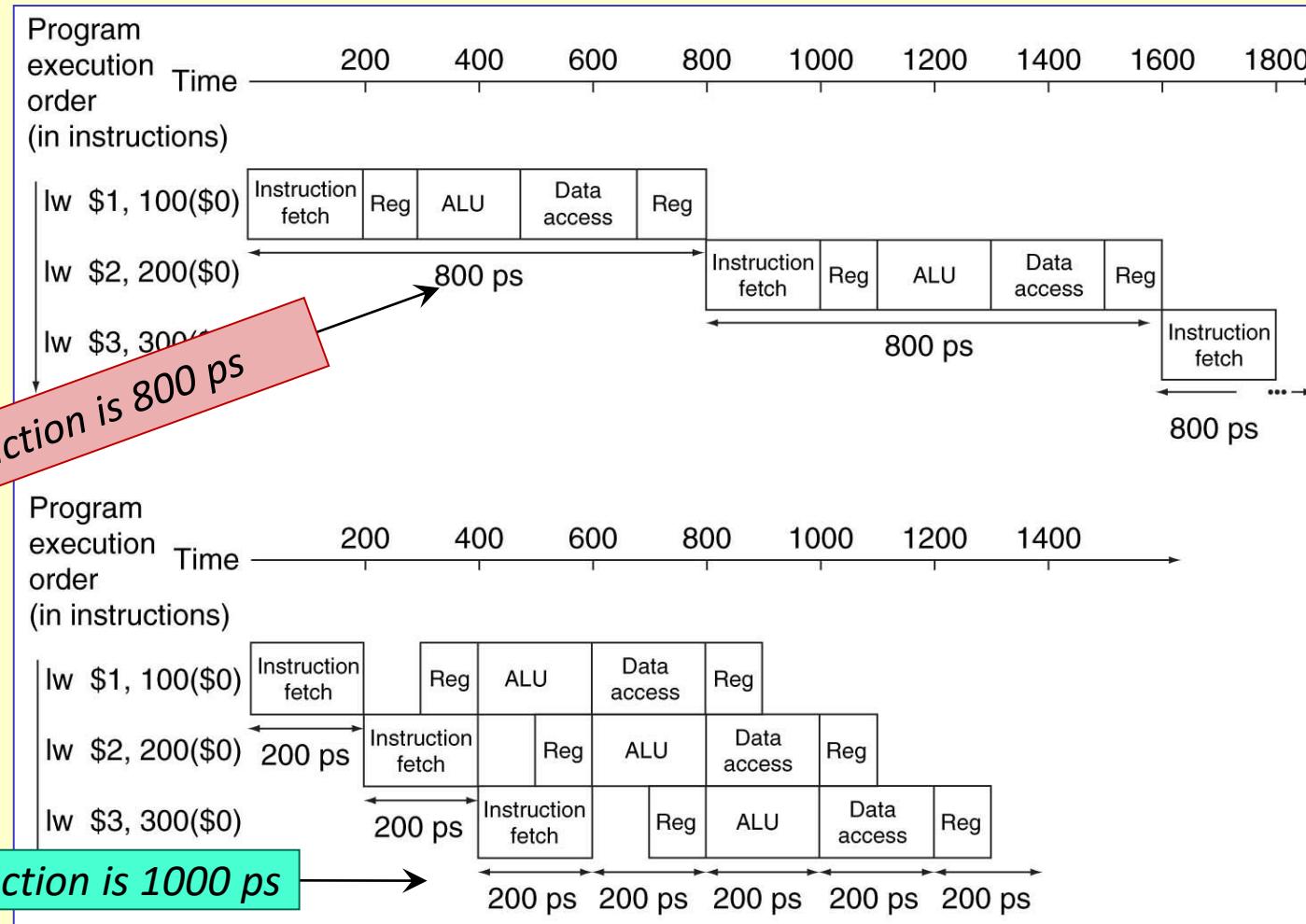
- Update program counter (incremented or jump/call/ret target)



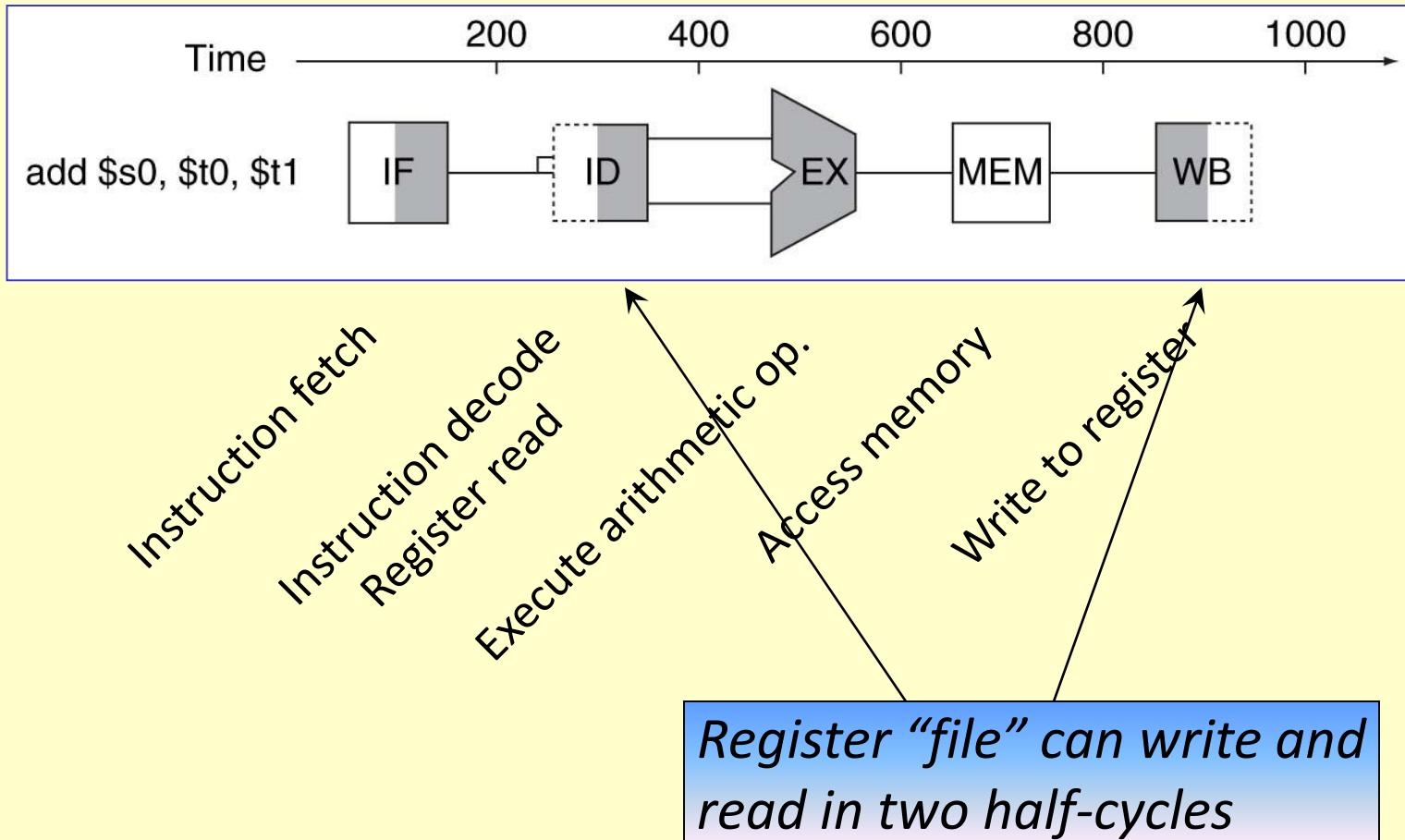
Stylized processor execution stages



Stylized processor — not pipelined vs. pipelined



Stylized processor pipeline

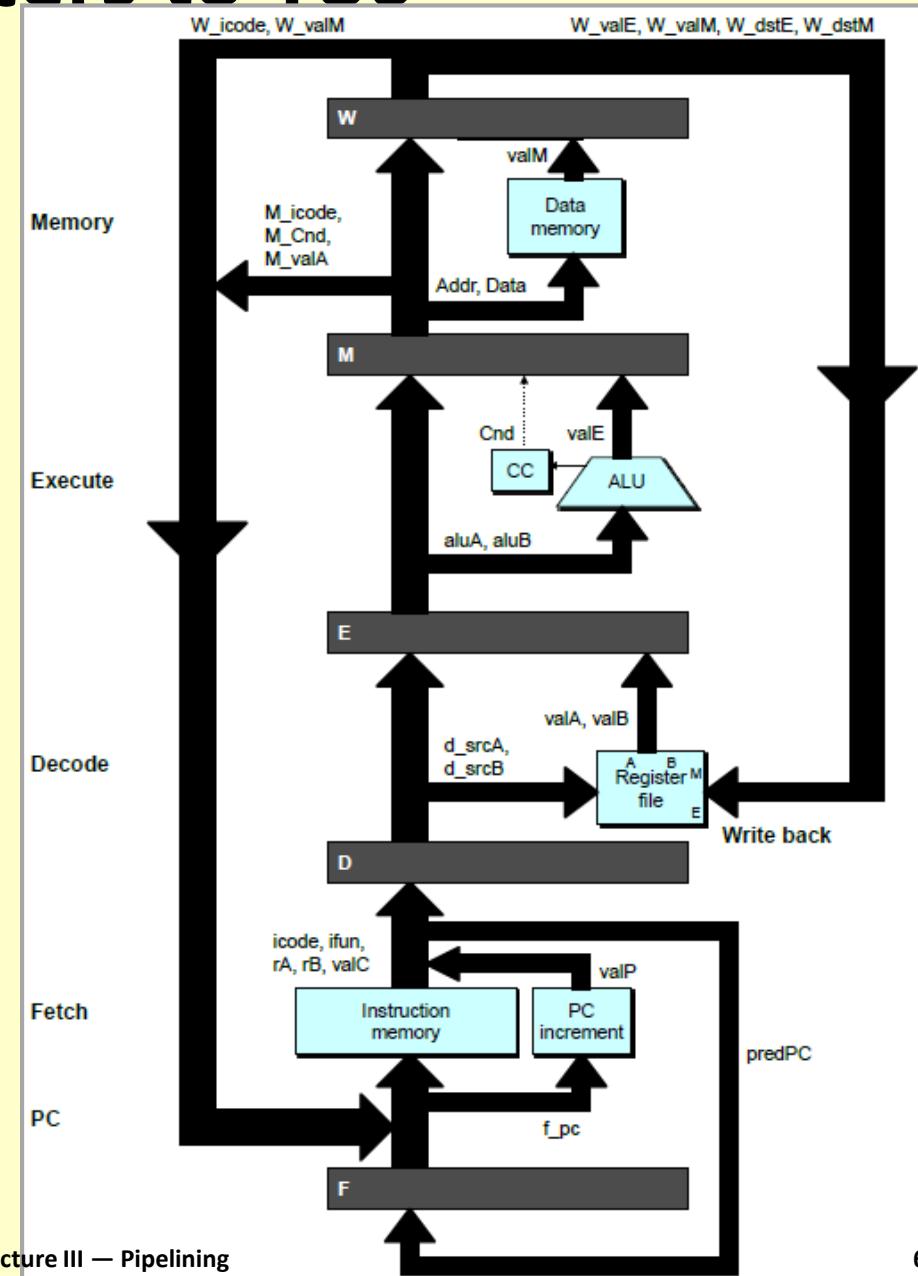
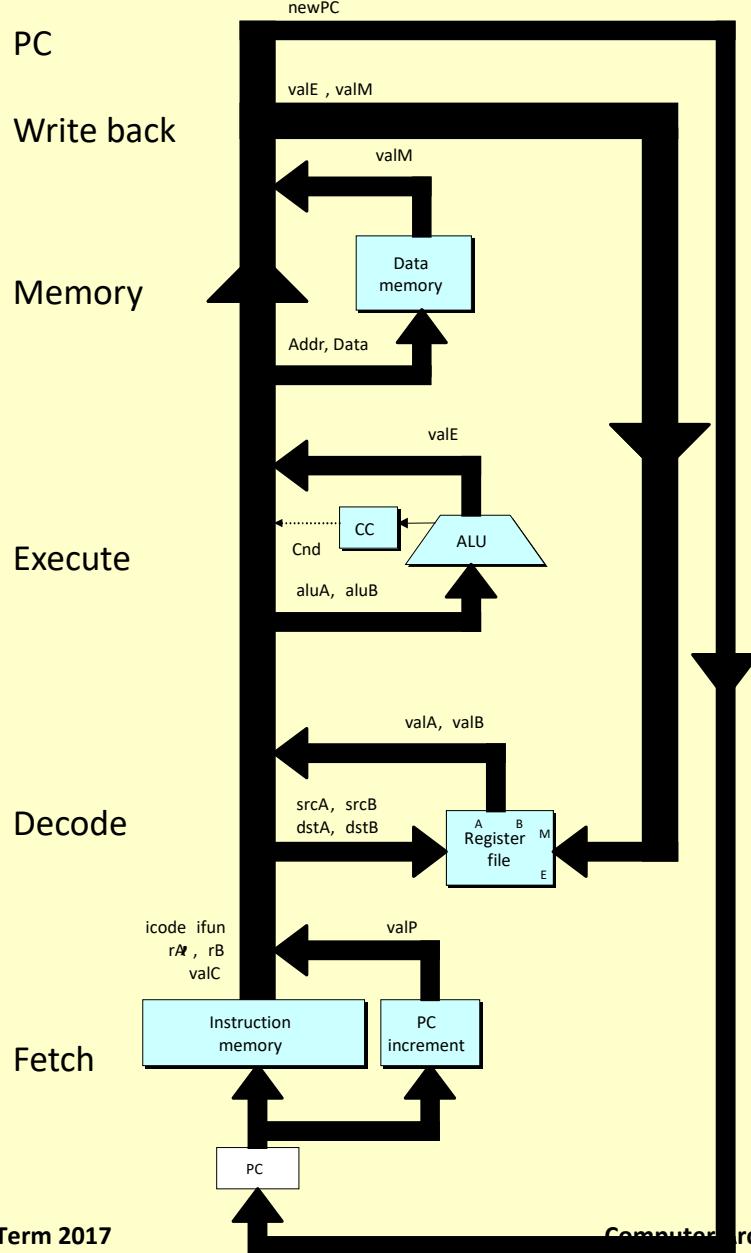


Questions?

Today

- Analogies in real world
- Pipelining in modern processors — MIPS
- Pipelining in Y86
- Hazards

Adding pipeline registers to Y86



Pipeline stages

Fetch

- Select current PC
- Read instruction
- Compute incremented PC

Decode

- Read program registers

Execute

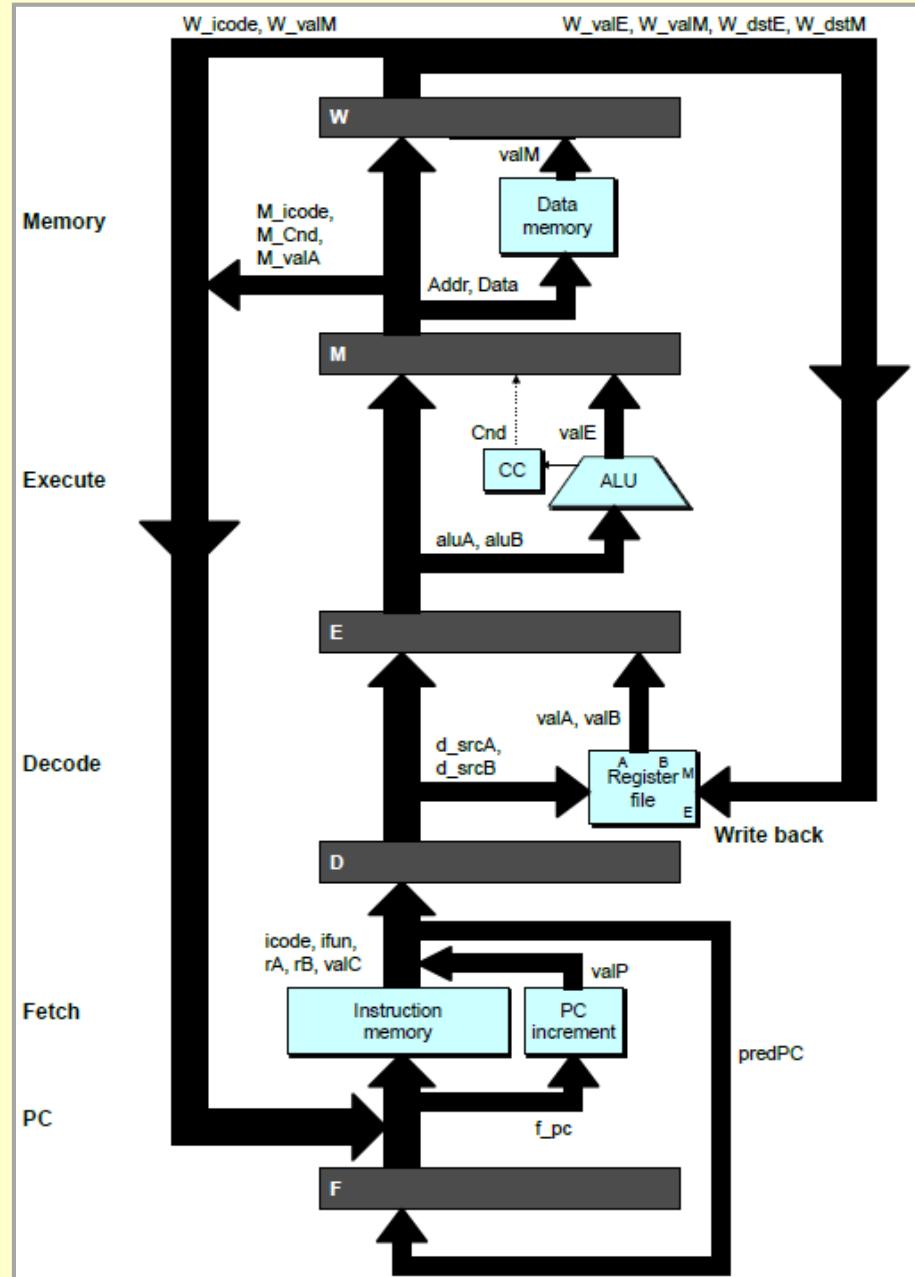
- Operate ALU

Memory

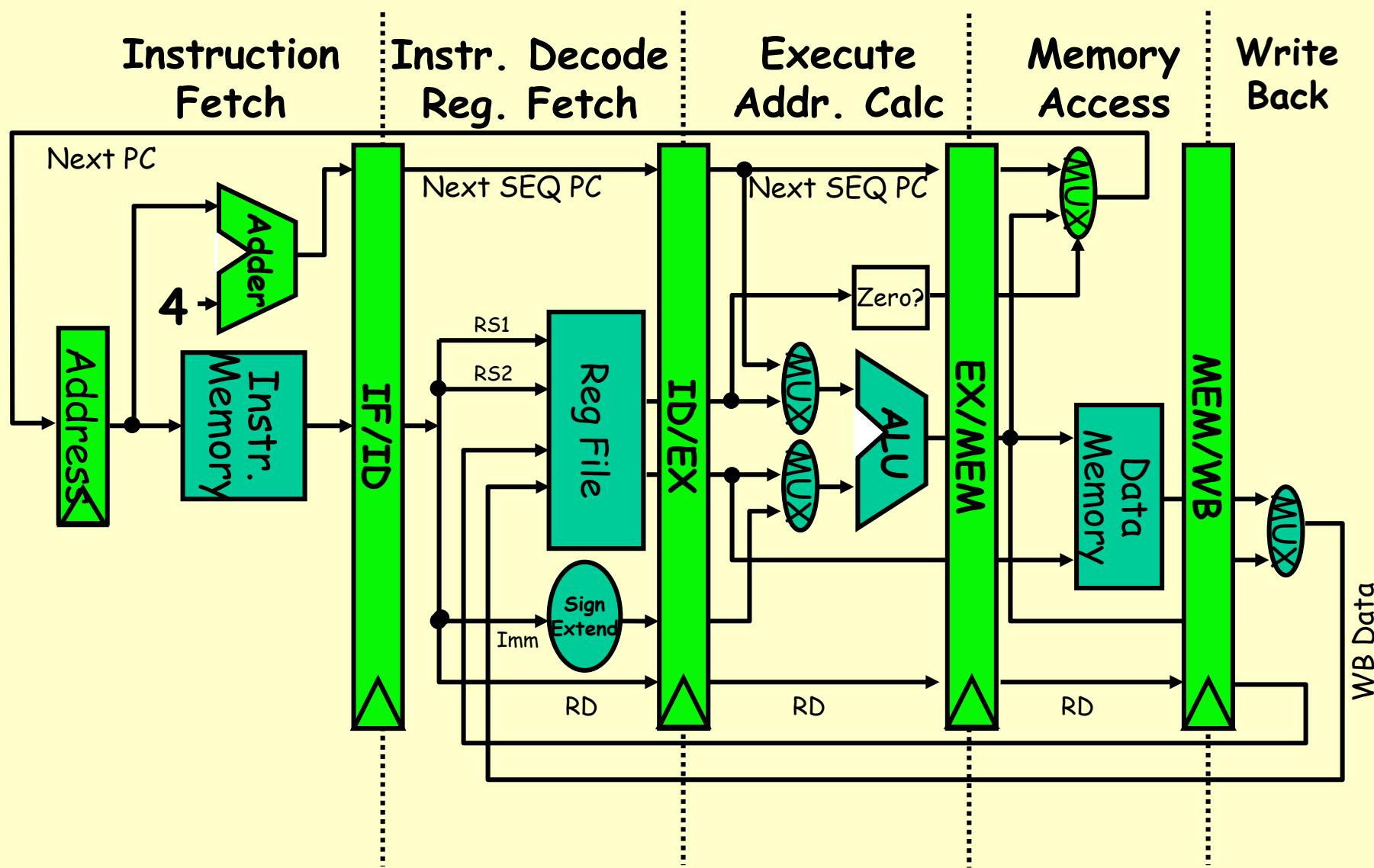
- Read or write data memory

Write Back

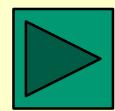
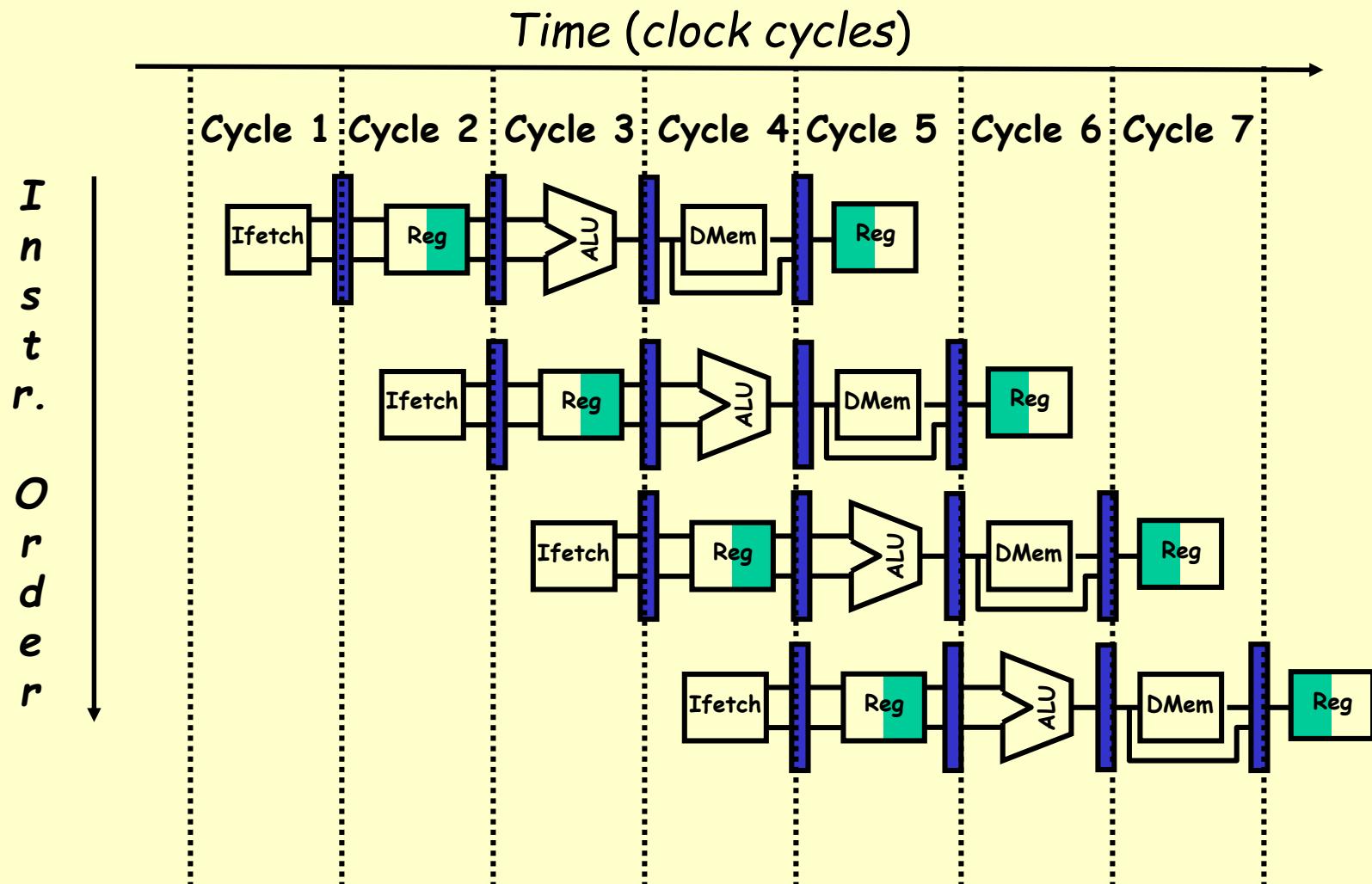
- Update register file



Alternate view (Hennessey & Patterson)



Visualizing pipeline behavior



Stylized pipeline performance

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (lw)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (sw)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (add, sub, AND, OR, slt)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (beq)	200 ps	100 ps	200 ps			500 ps

Stylized pipeline performance

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (lw)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (sw)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (add, sub, AND, OR, slt)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (beq)	200 ps	100 ps	200 ps			500 ps

Assumes L1 D-cache

Assumes L1 I-cache

Quantifying the speedup

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (lw)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (sw)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (add, sub, AND, OR, slt)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (beq)	200 ps	100 ps	200 ps			500 ps

■ Assume

- 40% ALU operations
- 20% branches
- 30% Load operations
- 10% Store operations
- No pipeline penalty

■ Average (non-pipelined) instruction duration = 650 ps

$$0.4 \times 600 + 0.2 \times 500 + 0.3 \times 800 + 0.1 \times 700 = 650$$

Quantifying the speedup (continued)

$$\text{Speedup} = \frac{\text{AveUnPipelinedTime}}{\text{AvePipelinedTime}} = \frac{650}{200} = 3.25$$

- Unpipelined architecture would allow variable duration instructions
 - Branches much faster than Loads
- Pipelined architecture requires every cycle to take exactly the same time
 - Some wasted time in Branch, ALU, and Store operations
- Speedup is less if there is a penalty for pipelining

Quantifying the speedup (continued)

$$\text{Speedup} = \frac{\text{AveUnPipelinedTime}}{\text{AvePipelinedTime}} = \frac{650}{200} = 3.25$$

- Unpipelined architecture would allow variable duration instructions

- Branches much faster than Load

Note: If there is a penalty for including pipelining (vs. non-pipelined design), speed-up decreases

- Pipelined architecture requires every cycle to take exactly the same time
 - Some wasted time in Branch, ALU, and Store operations
- Speedup is less if there is a penalty for pipelining

Life is never that simple!

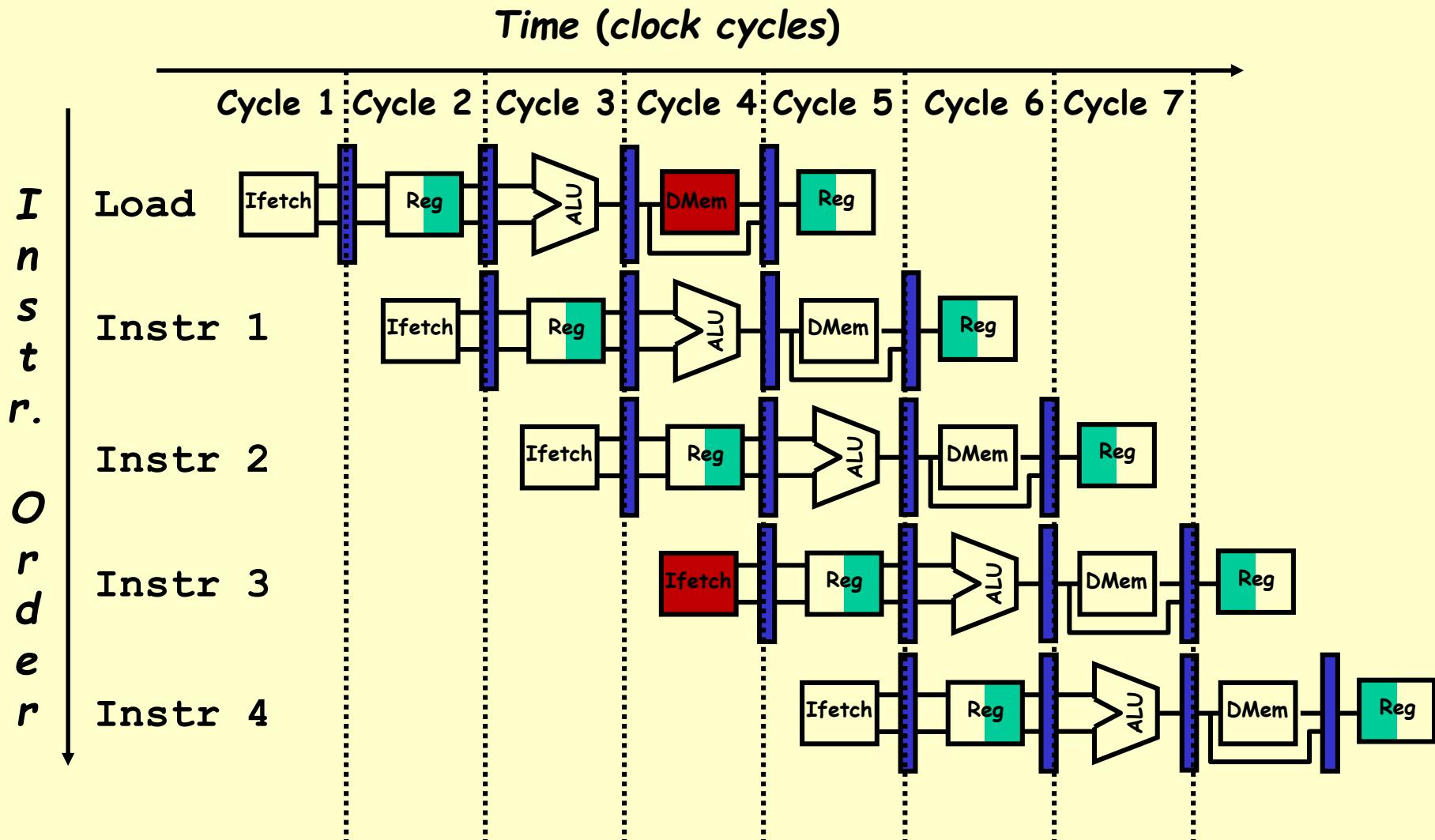
- Introducing *hazards*
- I.e., factors that interfere with full and efficient pipelining
- Prevent instruction from starting or continuing on next cycle

Definition!

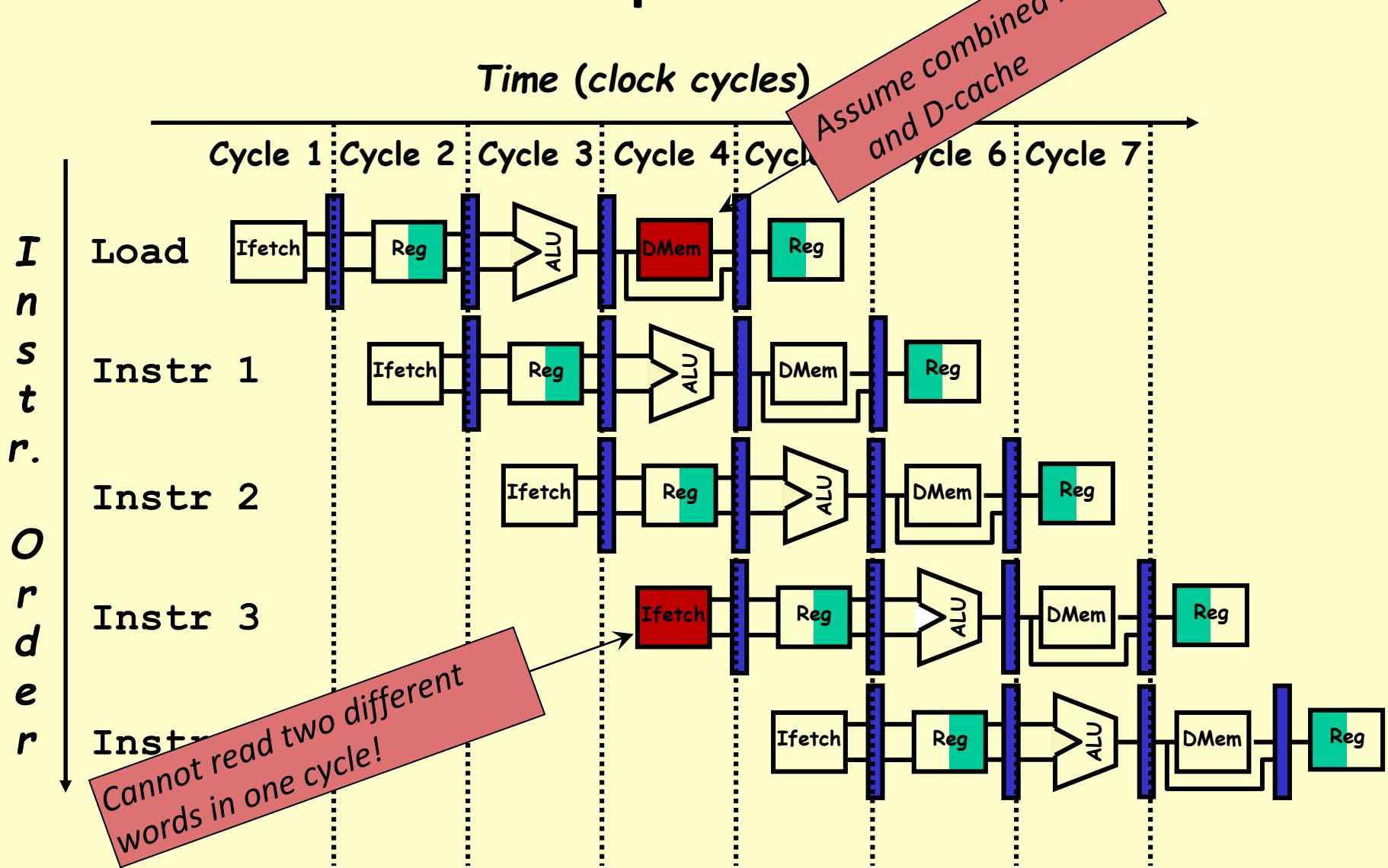
Hazards

- ***Structural* – resource conflicts among successive instructions**
 - Hardware cannot support all possible combinations of instructions in rapid succession
- ***Data* – dependencies of instructions on results of other instructions**
 - $x = a * b + c$ Cannot add until multiply is done
- ***Control* – branches that change instruction fetch order**
 - Will affect instructions that have already been fetched!

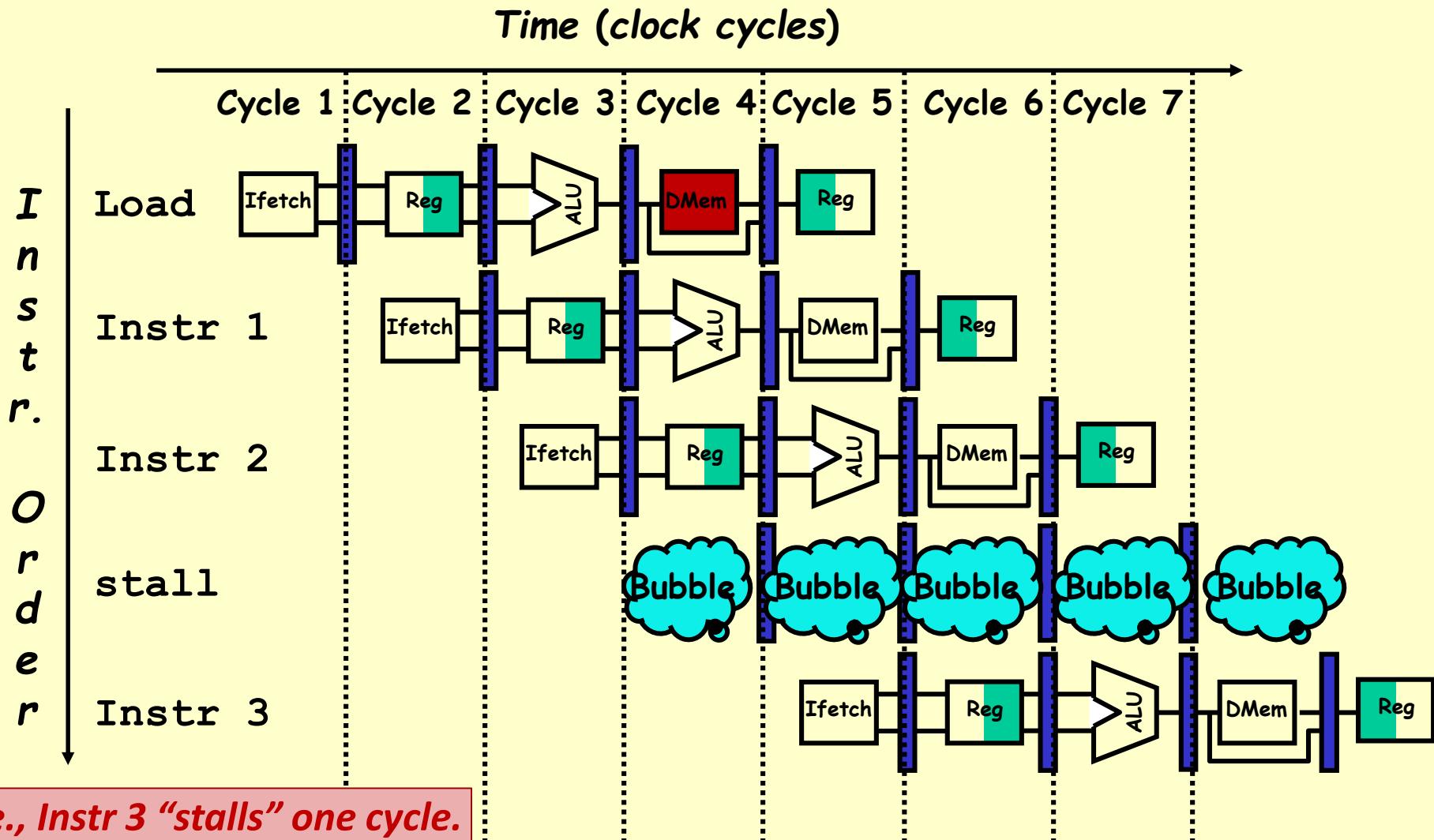
Structural hazard example:— one memory port



Structural hazard example:— one mem. port



Structural hazard example:— one memory port



“Harvard Architecture”

- (Originally) physically separate storage and signal pathways for instructions and data
- (Today) separate I-caches and D-caches

Other structural hazards (examples)

■ Multiply

- Expensive in gates to make fully pipelined

■ Floating-point divide

- Very expensive to make fully pipelined

■ Floating-point square-root

- Prohibitively expensive to pipeline at all

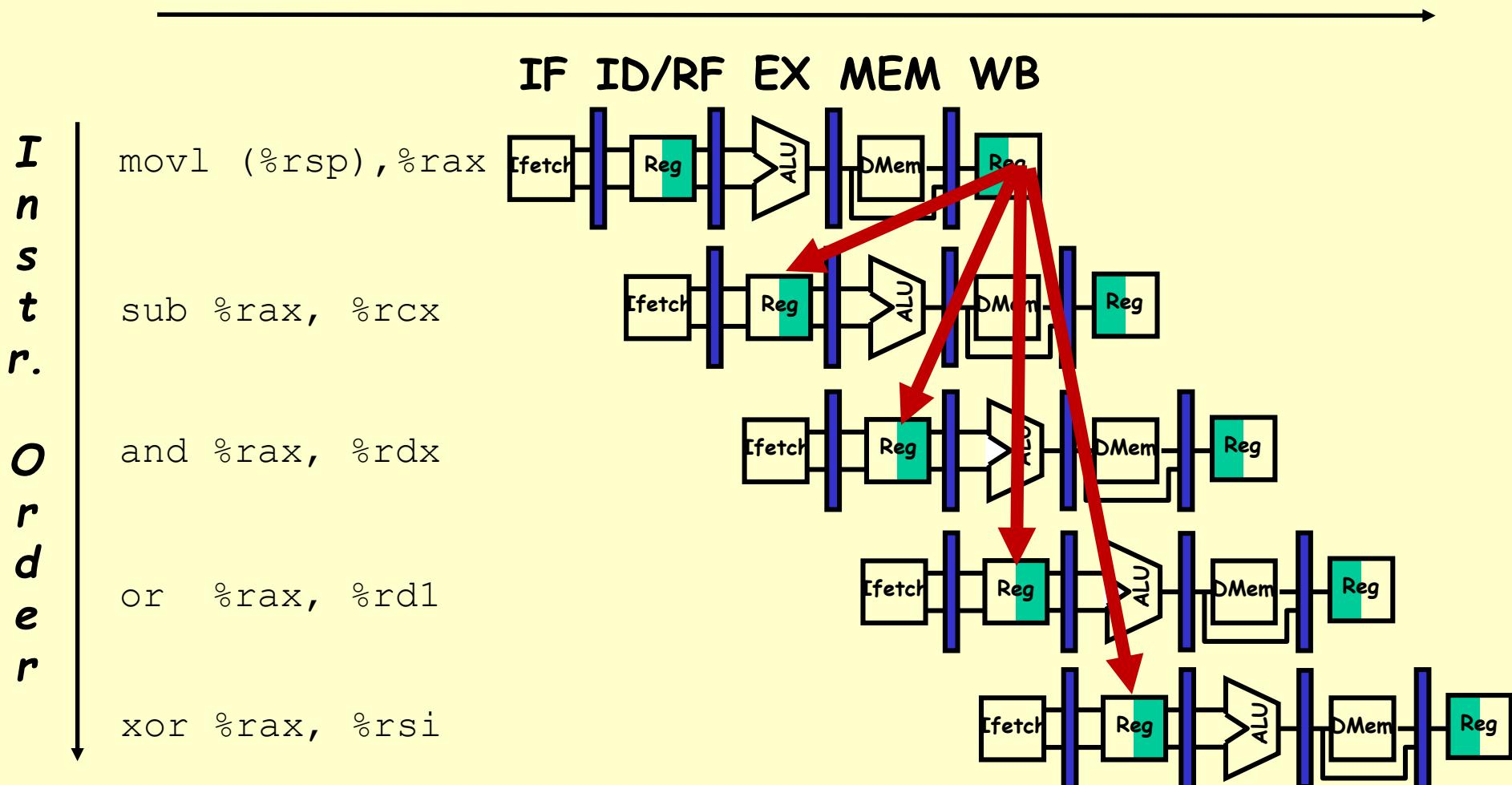
Structural hazards (conclusion)

- Processor hardware *must check*
 - Introduce “stalls”
- Compiler *should be aware*
 - Re-arrange compiled code

Questions?

Data hazard on %eax/%rax

Time (clock cycles)

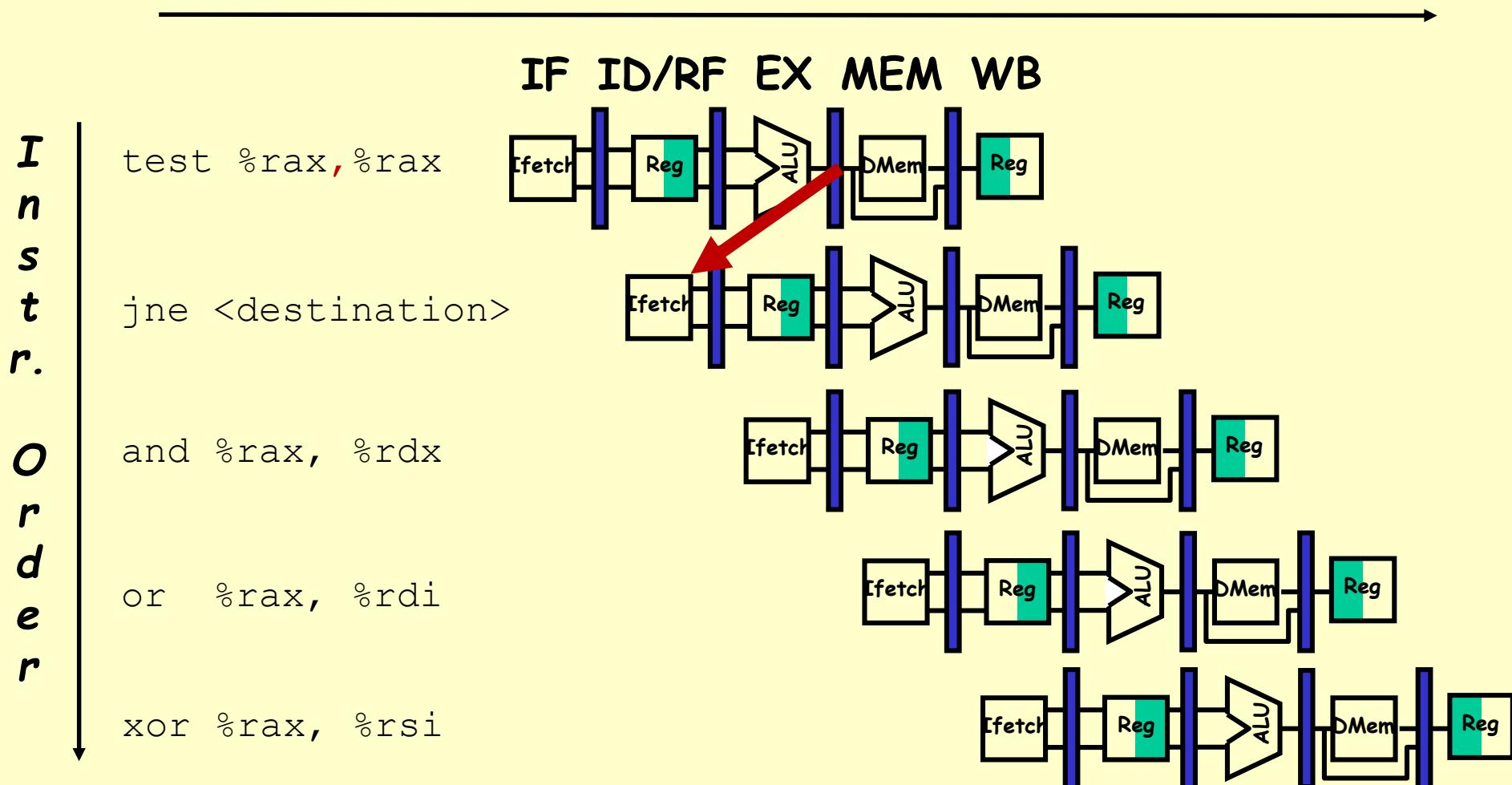


Data hazards

- **Extremely common**
- **Some hardware solutions**
 - Special forwarding data paths
 - Other extraordinary solutions
- **Compilers need to beware!**
 - Move code to minimize
 - Be aware during register allocation

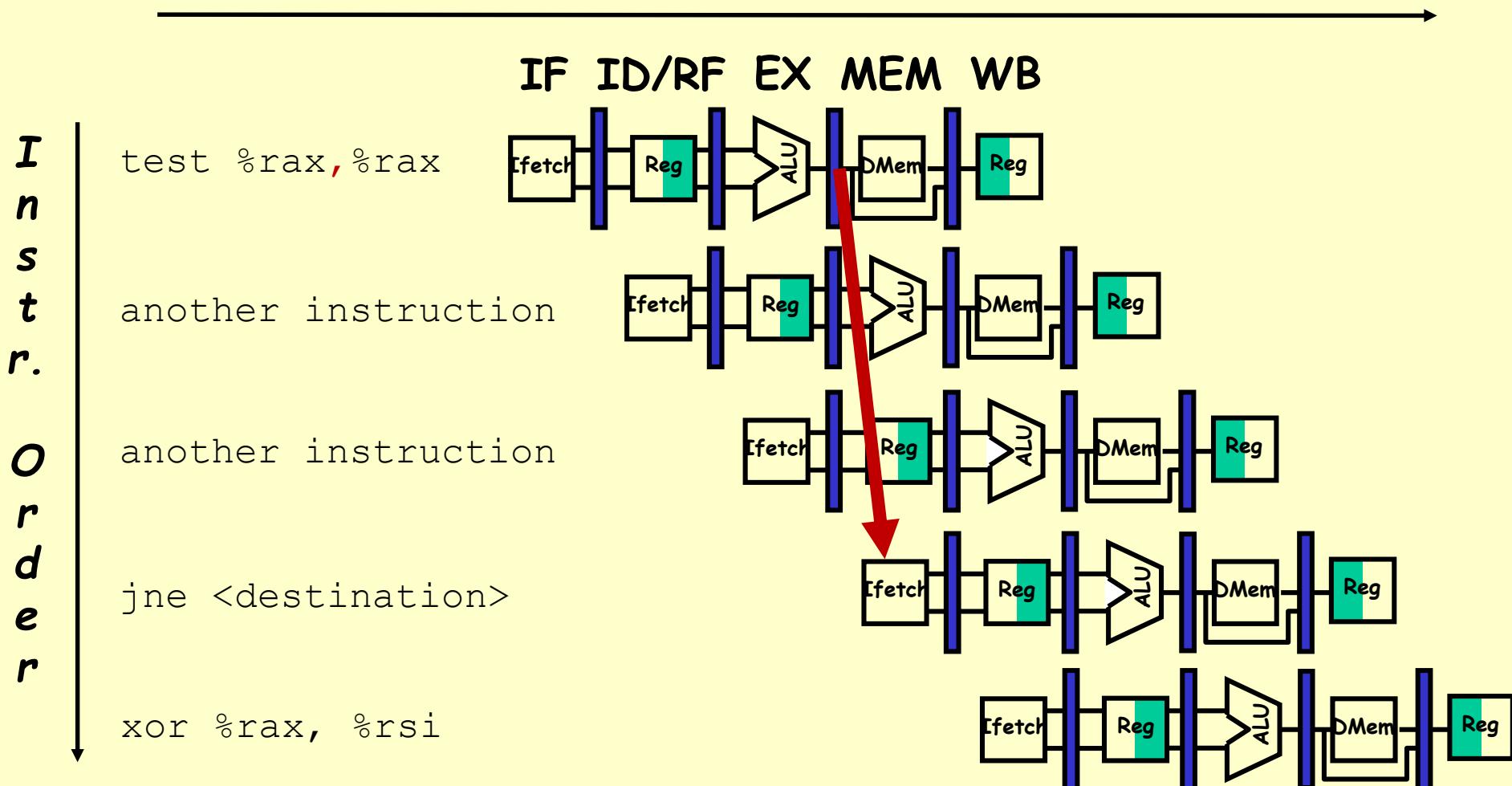
Control hazard on jumps

Time (clock cycles)



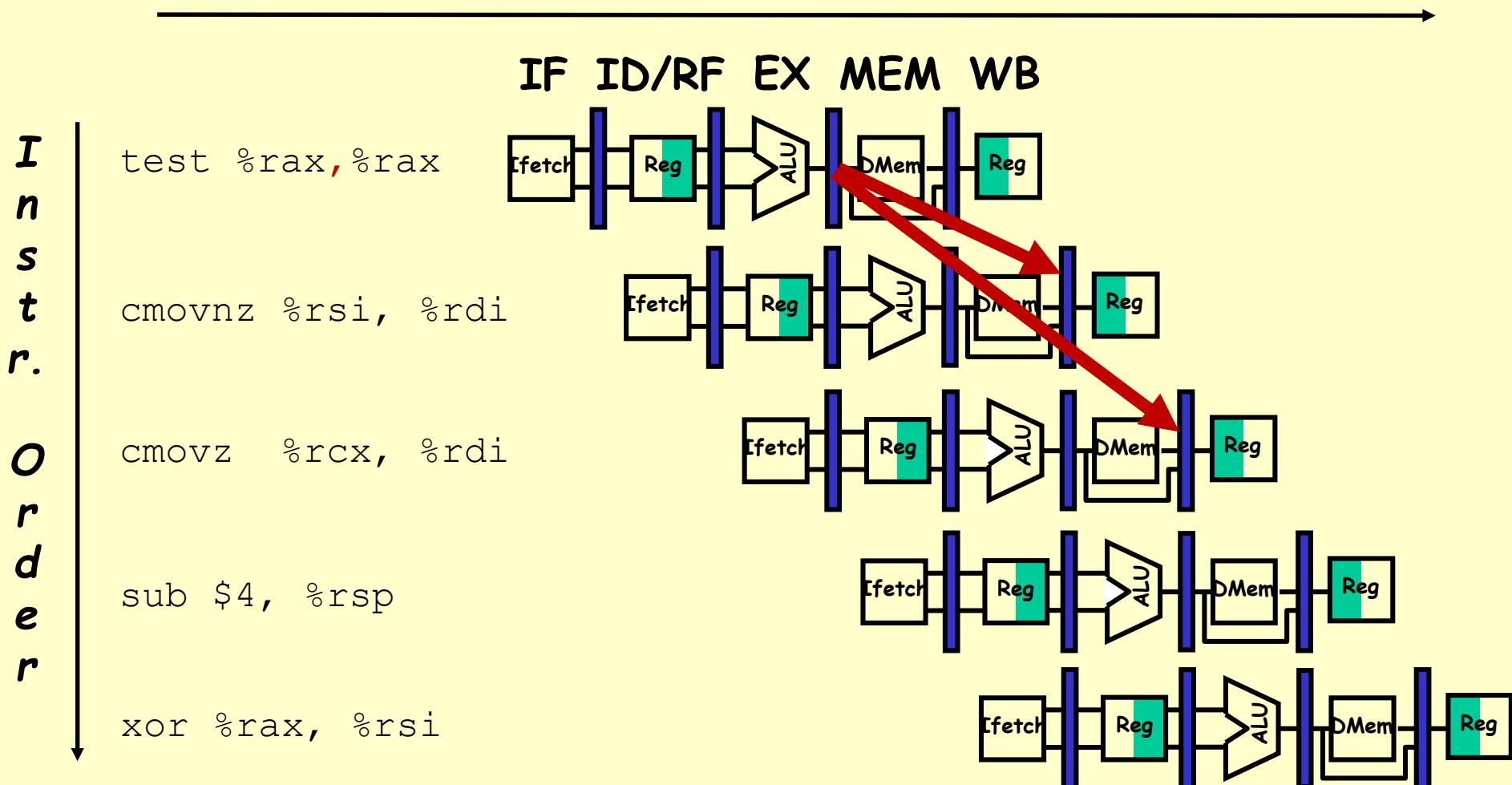
Control hazard on jumps — better

Time (clock cycles)



Control hazard on jumps — even better

Time (clock cycles)



Other control hazards

- Call — PC changes on very next instruction
- Ret — ditto
 - Plus data hazards on stack
- Architectural solutions
 - Branch delay
- Compiler solutions
 - Careful code motion
- Hardware solutions
 - Speculative execution

Questions?

Linking and Loading

Professor Hugh C. Lauer
CS-2011, Machine Organization and Assembly Language

(Slides include copyright materials from *Computer Systems: A Programmer's Perspective*, by Bryant and O'Hallaron, and from *The C Programming Language*, by Kernighan and Ritchie)

Today

- **Linking**
- **Case study: Library interpositioning**

Reading Assignment: §7.1 – §7.12

Example C Program

main.c

```
int buf[2] = {1, 2};

int main()
{
    swap();
    return 0;
}
```

swap.c

```
extern int buf[];

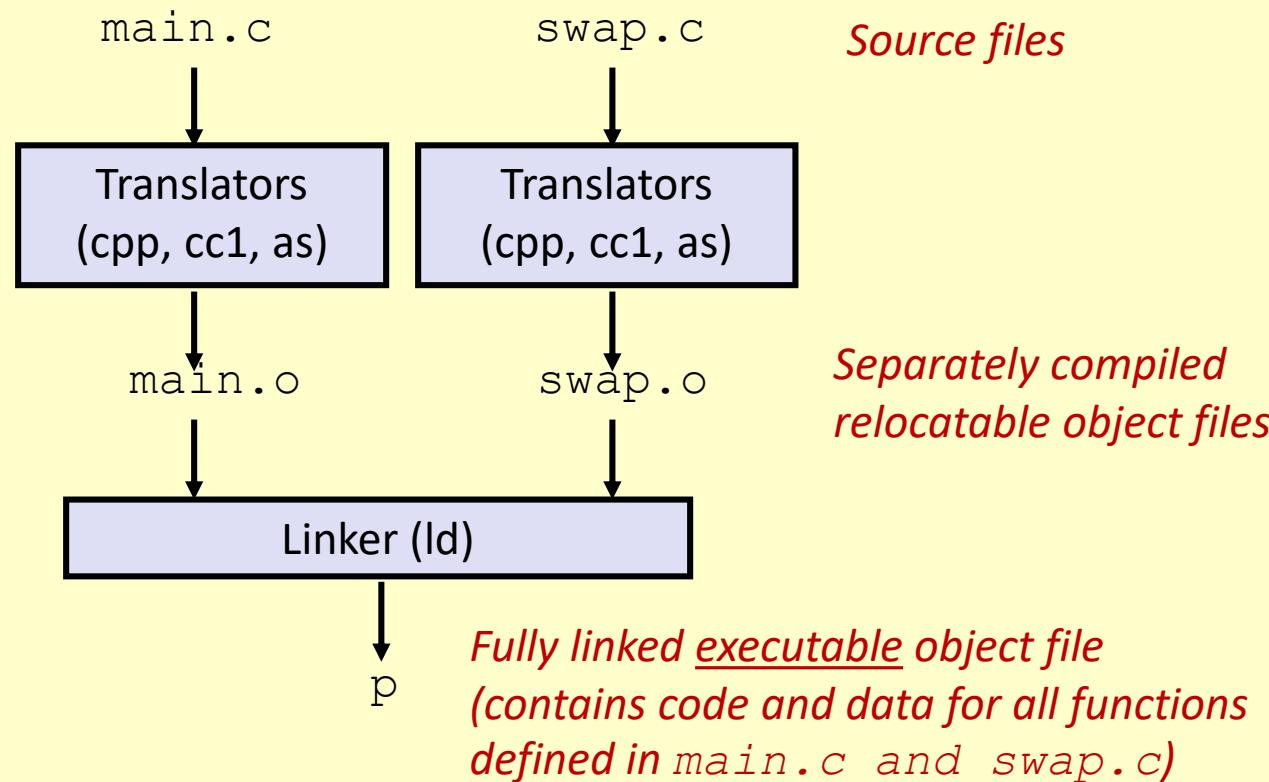
int *bufp0 = &buf[0];
static int *bufp1;

void swap()
{
    int temp;

    bufp1 = &buf[1];
    temp = *bufp0;
    *bufp0 = *bufp1;
    *bufp1 = temp;
}
```

Static Linking

- Programs are translated and linked using a *compiler driver*:
 - `linux> gcc -O2 -g -o p main.c swap.c`
 - `linux> ./p`



Why Linkers?

■ Reason 1: Modularity

- Program can be written as a collection of smaller source files, rather than one monolithic mass.
- Especially amenable to team development
- Can build libraries of common functions (more on this later)
 - e.g., Math library, standard C library

Why Linkers? (continued)

■ Reason 2: Efficiency

- Time: Separate compilation
 - Change one source file, compile, and then relink.
 - No need to recompile other source files.
- Space: Libraries
 - Common functions can be aggregated into a single file...
 - Yet executable files and running memory images contain only code for the functions they actually use.

What Do Linkers Do?

■ Symbol Resolution

- I.e., connect declared/defined objects with references to them in other modules

■ Relocation

- I.e., reposition code within executable image and change values of internal pointers to match

What Do Linkers Do?

■ Step 1. Symbol resolution

- Programs define and reference *symbols* (variables and functions):
 - `void swap() { ... } /* define symbol "swap" */`
 - `swap(); /* reference symbol "swap" */`
 - `int *xp = &x; /* define symbol "xp", reference symbol "x" */`
- Symbol definitions are stored (by compiler) in *symbol table*.
 - Symbol table is an array of **structs**
 - Each entry includes name, size, and location of whatever the symbol refers to.
- **During symbol resolution step, the linker associates each symbol reference with exactly one symbol definition.**

What Do Linkers Do? (continued)

■ Step 2. Relocation

- Merge separate code and data sections into single sections
- Relocate symbols from their relative locations in the .o files to their final absolute memory locations in the executable.
- Update all references to these symbols to reflect their new positions.

Three Kinds of Object Files (Modules)

■ Relocatable object file (.o file)

- Contains code and data in a form that can be combined with other relocatable object files to form executable object file.
 - Each .o file is produced from exactly one source (.c) file

■ Executable object file (a .out file)

- Contains code and data in a form that can be copied directly into memory and then executed.

■ Shared object file (.so file)

- Special type of relocatable object file that can be loaded into memory and linked dynamically, at either load time or run-time.
- Called *Dynamic Link Libraries* (DLLs) by Windows

Executable and Linkable Format (ELF)

- **Standard binary format for object files**
 - Originally proposed by AT&T System V Unix
 - Later adopted by BSD Unix variants and Linux
- **One unified format for**
 - Relocatable object files (**.o**),
 - Executable object files (**a .out**)
 - Shared object files (**.so**)
- **Generic name: ELF binaries**

ELF Object File Format

■ Elf header

- Word size, byte ordering, file type (.o, exec, .so), machine type, etc.

■ Segment header table

- Page size, virtual addresses memory segments (sections), segment sizes.

■ .text section

- Code

■ .rodata section

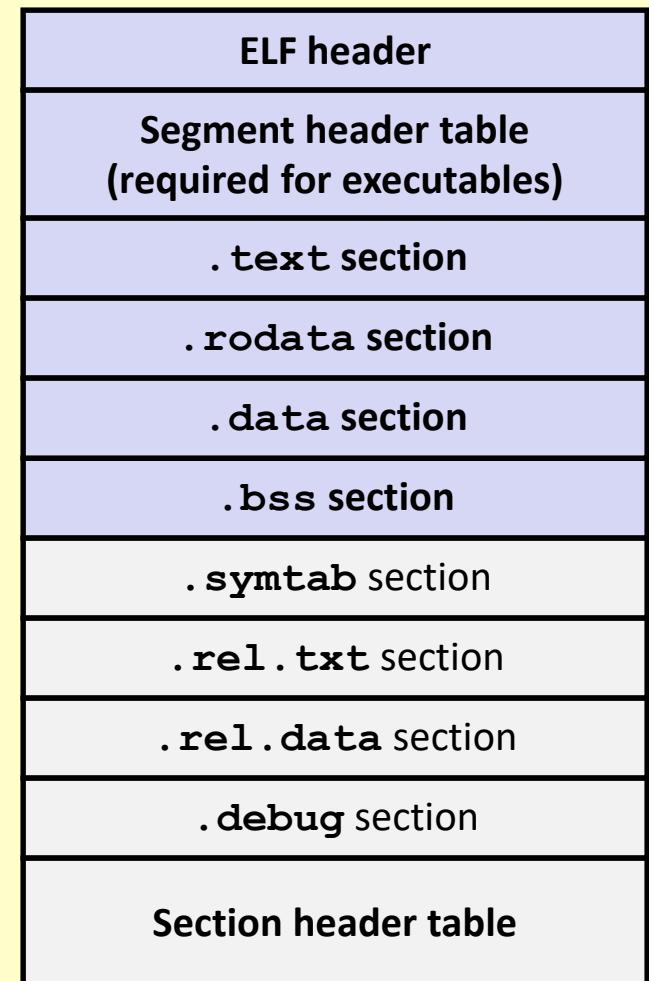
- Read Only data: jump tables, vtables, etc., ...

■ .data section

- Initialized global & static variables

■ .bss section

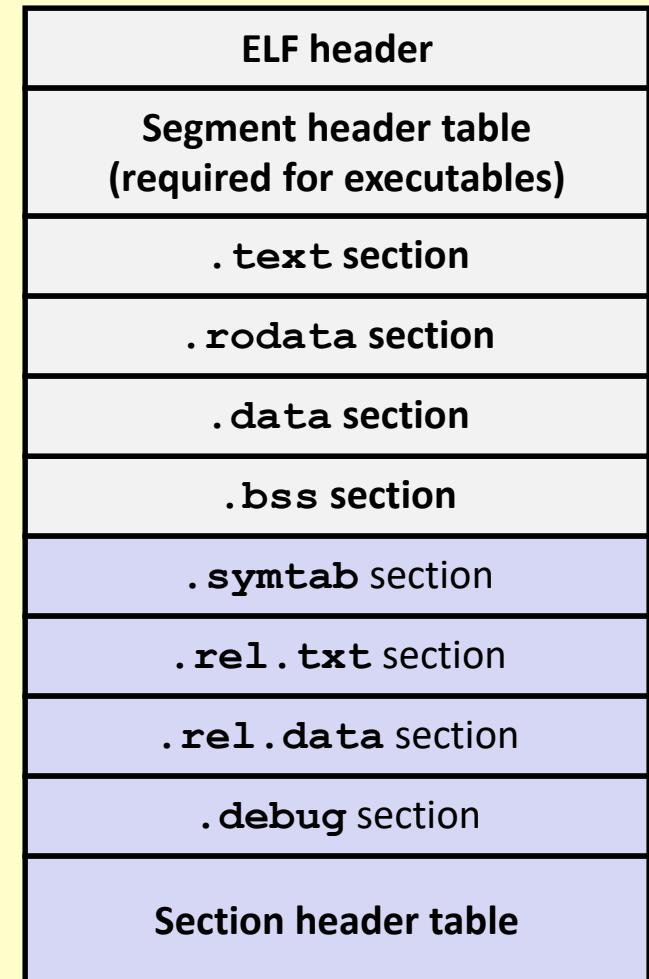
- Uninitialized global & static variables
- “Block Storage Start”
- “Better Save Space”
- Has section header but occupies no space



See §7.4, p. 674

ELF Object File Format (continued)

- **.syntab section**
 - Symbol table
 - Procedure and static variable names
 - Section names and locations
- **.rel.text section**
 - Relocation info for **.text** section
 - Addresses of instructions that will need to be modified in the executable
 - Instructions for modifying.
- **.rel.data section**
 - Relocation info for **.data** section
 - Addresses of pointer data that will need to be modified in the merged executable
- **.debug section**
 - Info for symbolic debugging (`gcc -g`)
- **Section header table**
 - Offsets and sizes of each section



Linker Symbols

■ Global symbols

- Symbols defined by module m that can be **referenced** by other modules.
- E.g.: non-**static** C functions and non-**static** global variables.

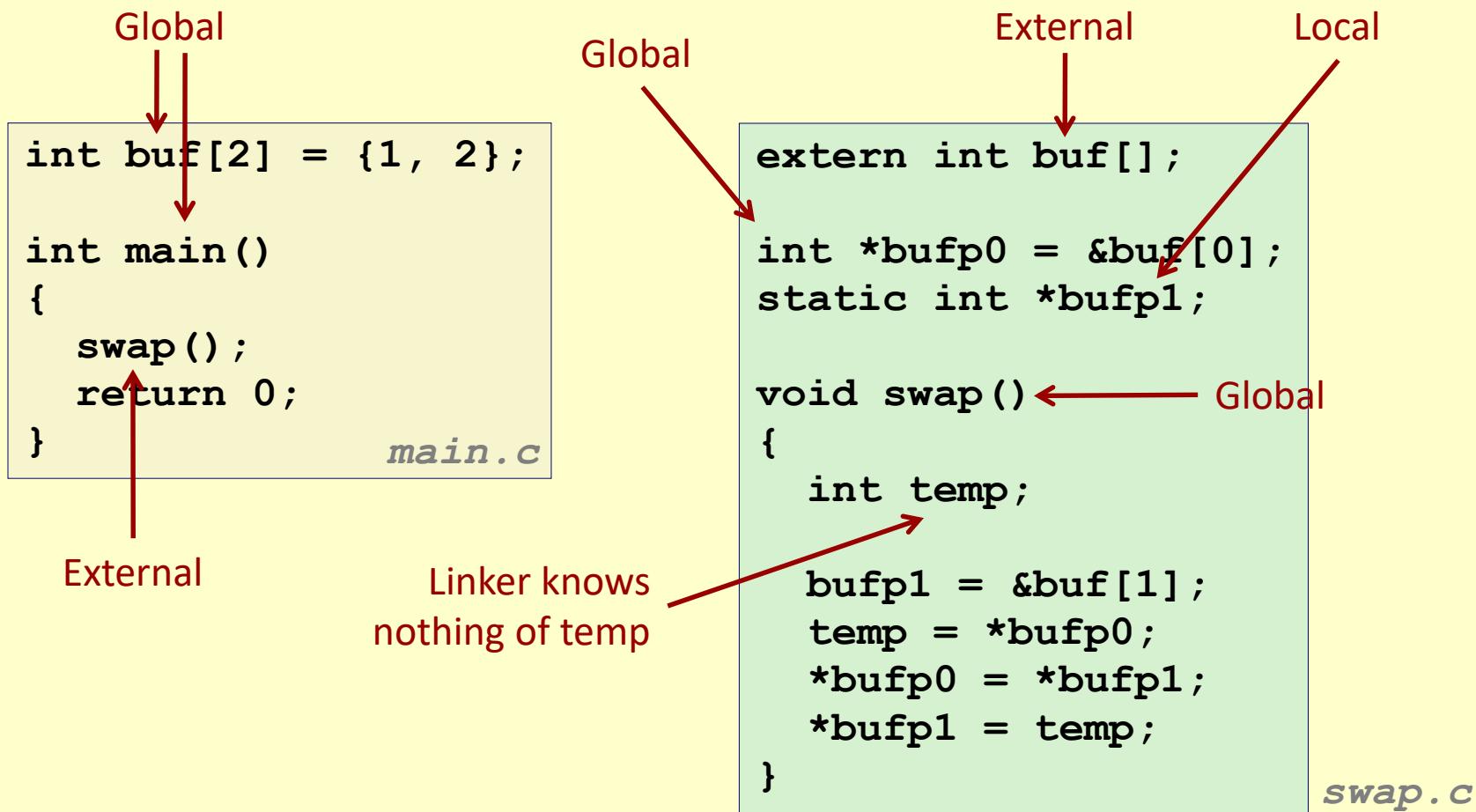
■ External symbols

- Global symbols that are referenced by module m but **defined** by some other module.

■ Local symbols

- Symbols that are defined and referenced exclusively by module m .
- E.g.: C functions and variables defined with the **static** attribute.
- **Local linker symbols are *not* local program variables**

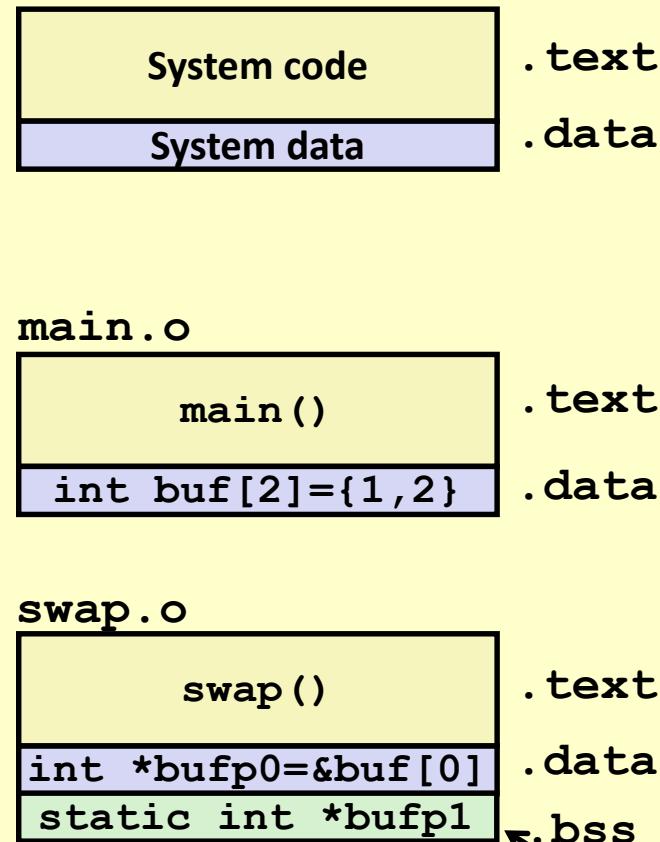
Resolving Symbols



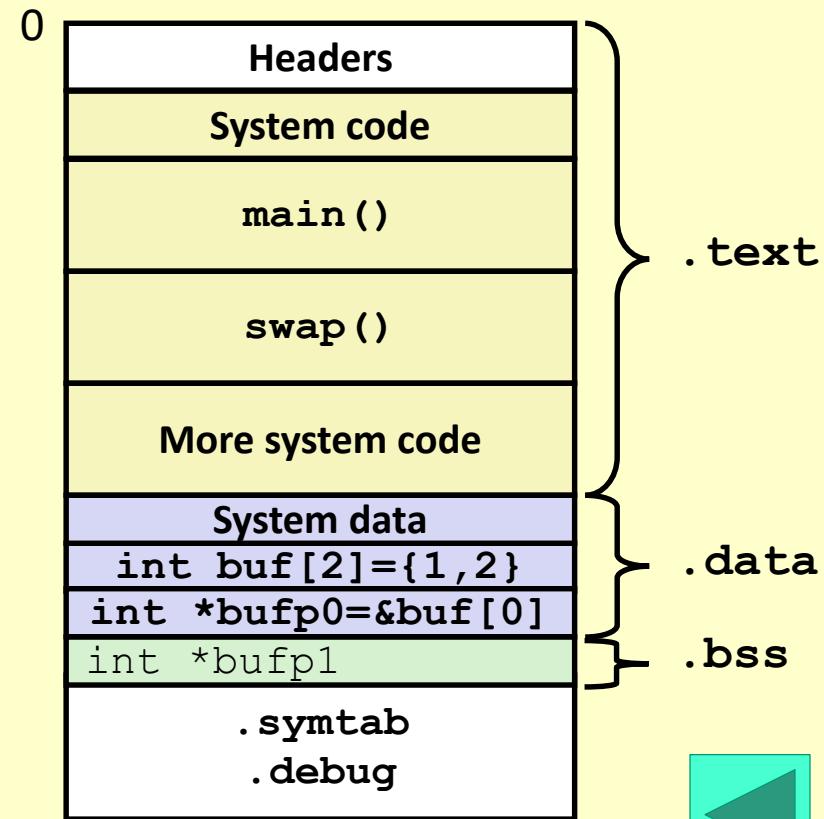
Questions?

Relocating Code and Data

Relocatable Object Files



Executable Object File



Relocation Info (main)

main.c

```
int buf[2] = {1,2};

int main()
{
    swap();
    return 0;
}
```

main.o

0000000 <main>:			
0:	8d 4c 24 04	lea	0x4(%esp),%ecx
4:	83 e4 f0	and	\$0xffffffff0,%esp
7:	ff 71 fc	pushl	0xfffffff(%ecx)
a:	55	push	%ebp
b:	89 e5	mov	%esp,%ebp
d:	51	push	%ecx
e:	83 ec 04	sub	\$0x4,%esp
11:	e8 fc ff ff ff	call	12 <main+0x12>
		12: R_386_PC32	swap
16:	83 c4 04	add	\$0x4,%esp
19:	31 c0	xor	%eax,%eax
1b:	59	pop	%ecx
1c:	5d	pop	%ebp
1d:	8d 61 fc	lea	0xfffffff(%ecx),%esp
20:	c3	ret	

Disassembly of section .data:

Source: objdump -r -d

0000000 <buf>:	
0:	01 00 00 00 00 02 00 00 00

Relocation Info (main – x86_64)

main.c

```
int buf[2] =
{0xdeadbeef,
 1};

int main()
{
    swap();
    return 0;
}
```

main.o

Disassembly of section .text:

0000000000000000 <main>:

0:	55	push %rbp
1:	48 89 e5	mov %rsp,%rbp
4:	b8 00 00 00 00	mov \$0x0,%eax
9:	e8 00 00 00 00	callq e <main+0xe>
	a: R_X86_64_PC32	swap-0x4
e:	b8 00 00 00 00	mov \$0x0,%eax
13:	5d	pop %rbp
14:	c3	retq

Disassembly of section .data:

0000000000000000 <buf>:

0:	ef	out %eax,(%dx)
1:	be ad de 01 00	mov \$0x1dead,%esi

Source: objdump -r -d

Relocation Info (swap, .text)

swap.c

```

extern int buf[];

int
*bufp0 = &buf[0];

static int *bufp1;

void swap()
{
    int temp;

    bufp1 = &buf[1];
    temp = *bufp0;
    *bufp0 = *bufp1;
    *bufp1 = temp;
}

```

swap.o

Disassembly of section .text:

00000000 <swap>:			
0: 8b 15 00 00 00 00	mov	0x0,%edx	
2: R_386_32	buf		
6: a1 04 00 00 00	mov	0x4,%eax	
7: R_386_32	buf		
b: 55	push	%ebp	
c: 89 e5	mov	%esp,%ebp	
e: c7 05 00 00 00 00 04	movl	\$0x4,0x0	
15: 00 00 00			
10: R_386_32	.bss		
14: R_386_32	buf		
18: 8b 08	mov	(%eax),%ecx	
1a: 89 10	mov	%edx,(%eax)	
1c: 5d	pop	%ebp	
1d: 89 0d 04 00 00 00	mov	%ecx,0x4	
1f: R_386_32	buf		
23: c3	ret		

Relocation Info (swap, .text - x86_64)

swap.c

```

extern int buf[];

int
*bufp0 = &buf[0];

static int *bufp1;

void swap()
{
    int temp;

    bufp1 = &buf[1];
    temp = *bufp0;
    *bufp0 = *bufp1;
    *bufp1 = temp;
}

```

swap.o

0000000000000000 <swap>:					
0:	55		push	%rbp	
1:	48 89 e5		mov	%rsp,%rbp	
4:	48 c7 05 00 00 00 00 00		movq	\$0x0,0x0(%rip) # f <swap+0xf>	
b:	00 00 00 00				
			7: R_X86_64_PC32	.bss-0x8	
			b: R_X86_64_32S	buf+0x4	
f:	48 8b 05 00 00 00 00 00		mov	0x0(%rip),%rax	# 16 <swap+0x16>
			12: R_X86_64_PC32	bufp0-0x4	
16:	8b 00		mov	(%rax),%eax	
18:	89 45 fc		mov	%eax,-0x4(%rbp)	
1b:	48 8b 05 00 00 00 00 00		mov	0x0(%rip),%rax	# 22 <swap+0x22>
			1e: R_X86_64_PC32	bufp0-0x4	
22:	48 8b 15 00 00 00 00 00		mov	0x0(%rip),%rdx	# 29 <swap+0x29>
			25: R_X86_64_PC32	.bss-0x4	
29:	8b 12		mov	(%rdx),%edx	
2b:	89 10		mov	%edx,(%rax)	
2d:	48 8b 05 00 00 00 00 00		mov	0x0(%rip),%rax	# 34 <swap+0x34>
			30: R_X86_64_PC32	.bss-0x4	
34:	8b 55 fc		mov	-0x4(%rbp),%edx	
37:	89 10		mov	%edx,(%rax)	
39:	90		nop		
3a:	5d		pop	%rbp	
3b:	c3		retq		

Relocation Info (swap, .data)

swap.c

```
extern int buf[];

int *bufp0 =
    &buf[0];
static int *bufp1;

void swap()
{
    int temp;

    bufp1 = &buf[1];
    temp = *bufp0;
    *bufp0 = *bufp1;
    *bufp1 = temp;
}
```

Disassembly of section .data:

00000000 <bufp0>:
0: 00 00 00 00

0: R_386_32 buf



Executable Before/After Relocation (.text)

0000000 <main>:

```

    . . .
e:   83 ec 04           sub    $0x4,%esp
11:  e8 fc ff ff ff    call   12 <main+0x12>
                12: R_386_PC32 swap
16:  83 c4 04           add    $0x4,%esp
    . . .

```

$0x8048396 + 0x1a$
 $= 0x80483b0$

08048380 <main>:

8048380:	8d 4c 24 04	lea 0x4(%esp),%ecx
8048384:	83 e4 f0	and \$0xffffffff0,%esp
8048387:	ff 71 fc	pushl 0xfffffff(%ecx)
804838a:	55	push %ebp
804838b:	89 e5	mov %esp,%ebp
804838d:	51	push %ecx
804838e:	83 ec 04	sub \$0x4,%esp
8048391:	e8 b0 48 83 08	call 80483b0 <swap>
8048396:	83 c4 04	add \$0x4,%esp
8048399:	31 c0	xor %eax,%eax
804839b:	59	pop %ecx
804839c:	5d	pop %ebp
804839d:	8d 61 fc	lea 0xfffffff(%ecx),%esp
80483a0:	c3	ret

0:	8b 15 00 00 00 00	mov	0x0, %edx
	2: R_386_32	buf	
6:	a1 04 00 00 00	mov	0x4, %eax
	7: R_386_32	buf	
...			
e:	c7 05 00 00 00 00 04	movl	\$0x4, 0x0
15:	00 00 00		
	10: R_386_32	.bss	
	14: R_386_32	buf	
...			
1d:	89 0d 04 00 00 00	mov	%ecx, 0x4
	1f: R_386_32	buf	
23:	c3	ret	

080483b0 <swap>:

80483b0:	8b 15 20 96 04 08	mov	0x8049620, %edx
80483b6:	a1 24 96 04 08	mov	0x8049624, %eax
80483bb:	55	push	%ebp
80483bc:	89 e5	mov	%esp, %ebp
80483be:	c7 05 30 96 04 08 24	movl	\$0x8049624, 0x8049630
80483c5:	96 04 08		
80483c8:	8b 08	mov	(%eax), %ecx
80483ca:	89 10	mov	%edx, (%eax)
80483cc:	5d	pop	%ebp
80483cd:	89 0d 24 96 04 08	mov	%ecx, 0x8049624
80483d3:	c3	ret	

Executable After Relocation (.data)

```
Disassembly of section .data:
```

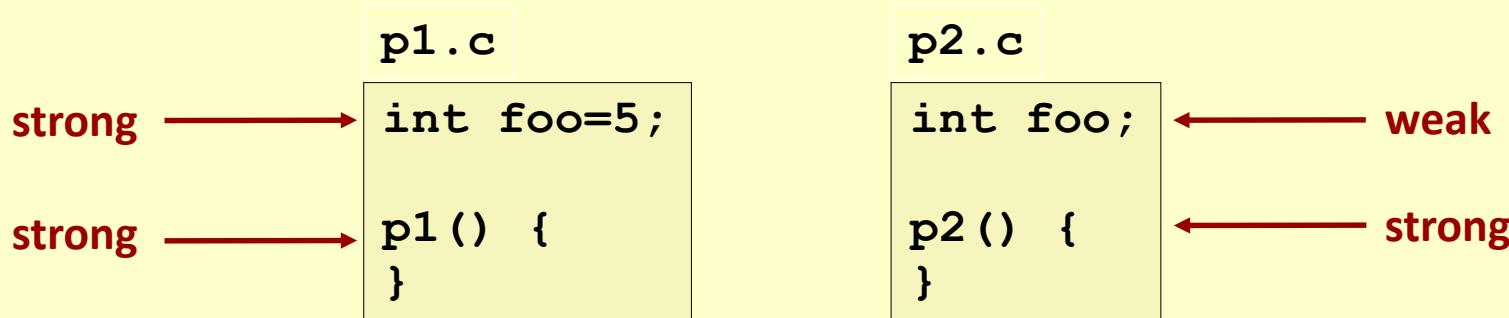
```
08049620 <buf>:  
 8049620: 01 00 00 00 02 00 00 00  
  
08049628 <bufp0>:  
 8049628: 20 96 04 08
```



Questions?

Strong and Weak Symbols

- Program symbols are either strong or weak
 - **Strong**: procedures/functions and initialized globals
 - **Weak**: uninitialized globals



Linker's Symbol Rules

- **Rule 1: Multiple strong symbols *of same name* are not allowed**
 - Each item can be defined only once
 - Otherwise: Linker error
- **Rule 2: Given one strong symbol and multiple weak symbols *of same name*, choose the strong symbol**
 - References to the weak symbol resolve to the strong symbol
- **Rule 3: If there are multiple weak symbols, pick an arbitrary one**
 - Can override this with `gcc -fno-common`

Linker Puzzles

```
int x;
p1() {}
```

```
p1() {}
```

Link time error: two strong symbols (p1)

```
int x;
p1() {}
```

```
int x;
p2() {}
```

References to `x` will refer to the same uninitialized int. *Is this what you really want?*

```
int x;
int y;
p1() {}
```

```
double x;
p2() {}
```

Writes to `x` in `p2` *might* overwrite `y`!
Evil!

Choice of `x` is arbitrary!

```
int x=7;
int y=5;
p1() {}
```

```
double x;
p2() {}
```

Writes to `x` in `p2` *will* overwrite `y`!
Nasty!

```
int x=7;
p1() {}
```

```
int x;
p2() {}
```

References to `x` will refer to the same initialized variable.

Nightmare scenario: two identical weak structs, compiled by different compilers with different alignment rules.

Role of .h Files

c1.c

```
#include "global.h"

int f() {
    return g+1;
}
```

c2.c

```
#include <stdio.h>
#include "global.h"

int main() {
    if (!init)
        g = 37;
    int t = f();
    printf("Calling f yields %d\n", t);
    return 0;
}
```

global.h

```
#ifdef INITIALIZE
int g = 23;
static int init = 1;
#else
int g;
static int init = 0;
#endif
```

Running Preprocessor

c1.c

```
#include "global.h"

int f() {
    return g+1;
}
```

global.h

```
#ifdef INITIALIZE
int g = 23;
static int init = 1;
#else
int g;
static int init = 0;
#endif
```

-DINITIALIZE

```
int g = 23;
static int init = 1;
int f() {
    return g+1;
}
```

no initialization

```
int g;
static int init = 0;
int f() {
    return g+1;
}
```

#include causes C preprocessor to insert file verbatim

Role of .h Files (continued)

c1.c

```
#include "global.h"

int f() {
    return g+1;
}
```

c2.c

```
#include <stdio.h>
#include "global.h"

int main() {
    if (!init)
        g = 37;
    int t = f();
    printf("Calling f yields %d\n", t);
    return 0;
}
```

global.h

```
#ifdef INITIALIZE
int g = 23;
static int init = 1;
#else
int g;
static int init = 0;
#endif
```

What happens:

```
gcc -o p c1.c c2.c
```

??

```
gcc -o p c1.c c2.c \
```

-DINITIALIZE

??

Global Variables

- **Avoid if you can**
- **Otherwise**
 - Use **static** if you can
 - I.e., when it is not shared with other modules
 - Initialize if you define a global variable
 - Always, always, always ...
 - Use **extern** whenever you want access to an external global variable
 - Helps avoid surprises

Questions?

Packaging Commonly Used Functions

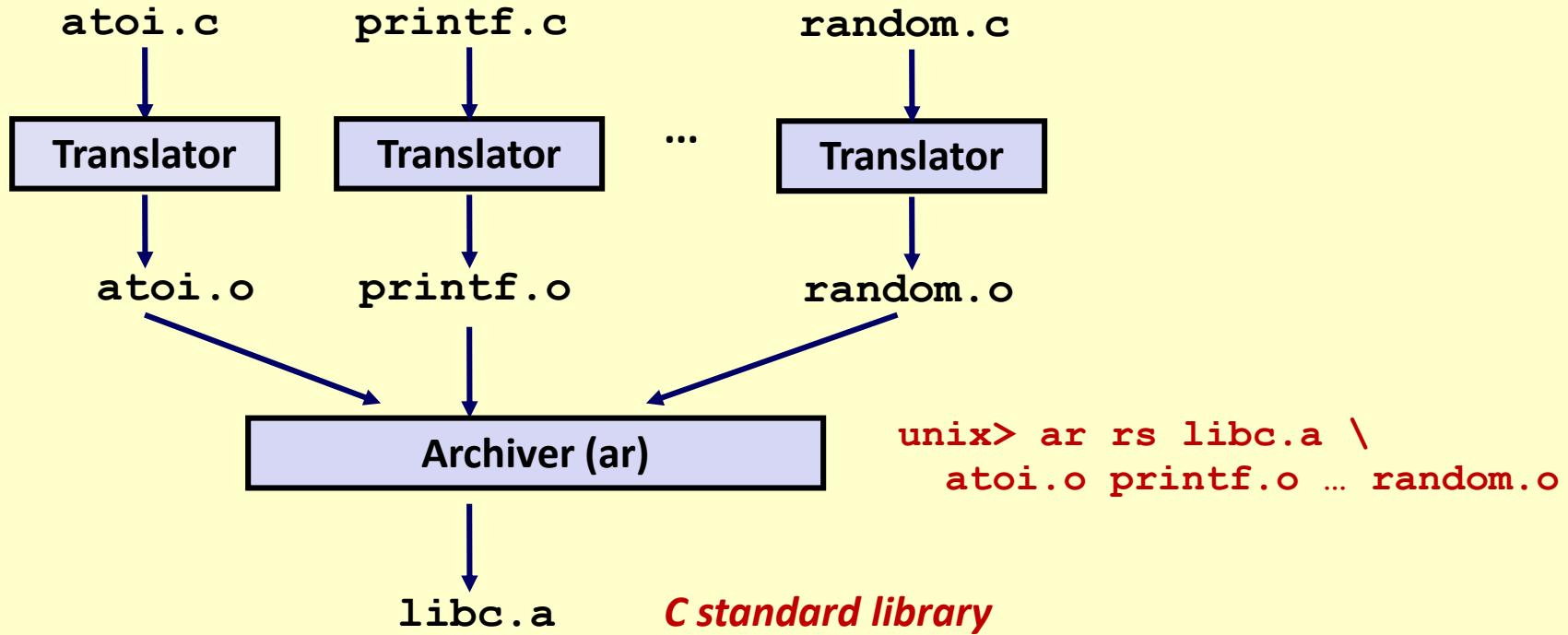
- How to package functions commonly used by programmers?
 - Math, I/O, memory management, string manipulation, etc.
- Awkward, given the linker framework so far:
 - **Option 1:** Put all functions into a single source file
 - Programmers link big object file into their programs
 - Space and time inefficient
 - **Option 2:** Put each function in a separate source file
 - Programmers explicitly link appropriate binaries into their programs
 - More efficient, but burdensome on the programmer

Solution: Static Libraries

■ **Static libraries (.a archive files)**

- Concatenate related relocatable object files into a single file with an index (called an *archive*).
- Enhance linker so that it tries to resolve unresolved external references by looking for the symbols in one or more archives.
- If an archive member file resolves reference, link it into the executable.

Creating Static Libraries



- Archiver allows incremental updates
- Recompile function that changes and replace .o file in archive.

Commonly Used Libraries

libc.a (the C standard library)

- 8 MB archive of 1392 object files.
- I/O, memory allocation, signal handling, string handling, data and time, random numbers, integer math

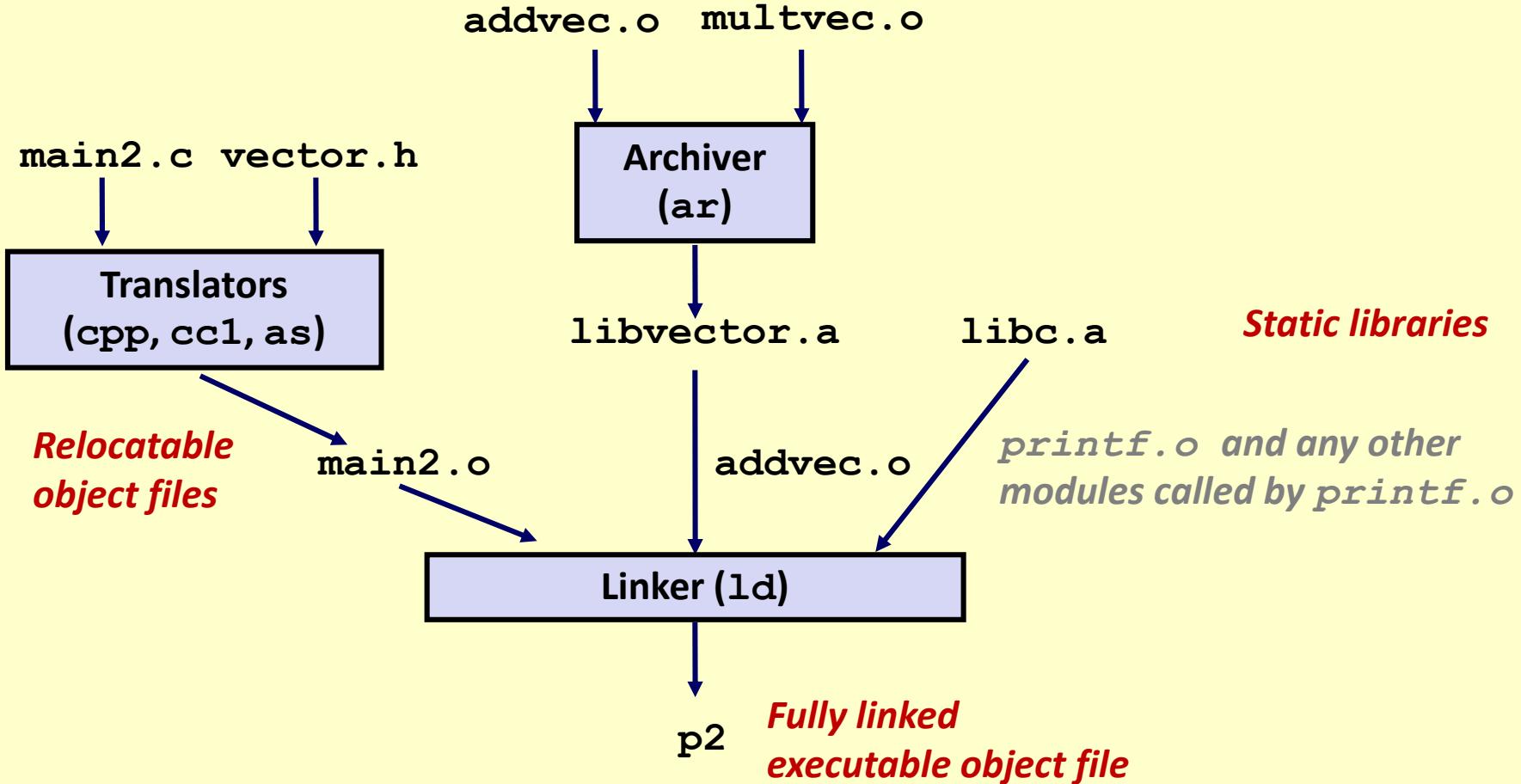
libm.a (the C math library)

- 1 MB archive of 401 object files.
- floating point math (sin, cos, tan, log, exp, sqrt, ...)

```
% ar -t /usr/lib/libc.a | sort
...
fork.o
...
fprintf.o
fpu_control.o
fputc.o
freopen.o
fscanf.o
fseek.o
fstab.o
...
```

```
% ar -t /usr/lib/libm.a | sort
...
e_acos.o
e_acosf.o
e_acosh.o
e_acoshf.o
e_acoshl.o
e_acosl.o
e_asin.o
e_asinf.o
e_asinl.o
...
```

Linking with Static Libraries



Using Static Libraries

■ Linker's algorithm for resolving external references:

- Scan `.o` files and `.a` files in the *command line order*
- During the scan, keep a list of the current unresolved references
- As each new `.o` or `.a` file, *obj*, is encountered, try to resolve each unresolved reference in the list against the symbols defined in *obj*
- If any entries in the unresolved list at end of scan, then error

■ Problem:

- Command line order matters!
- Moral: put libraries at the end of the command line.

```
unix> gcc -L. libtest.o -lmine
unix> gcc -L. -lmine libtest.o
libtest.o: In function `main':
libtest.o(.text+0x4): undefined reference to `libfun'
```

Questions?

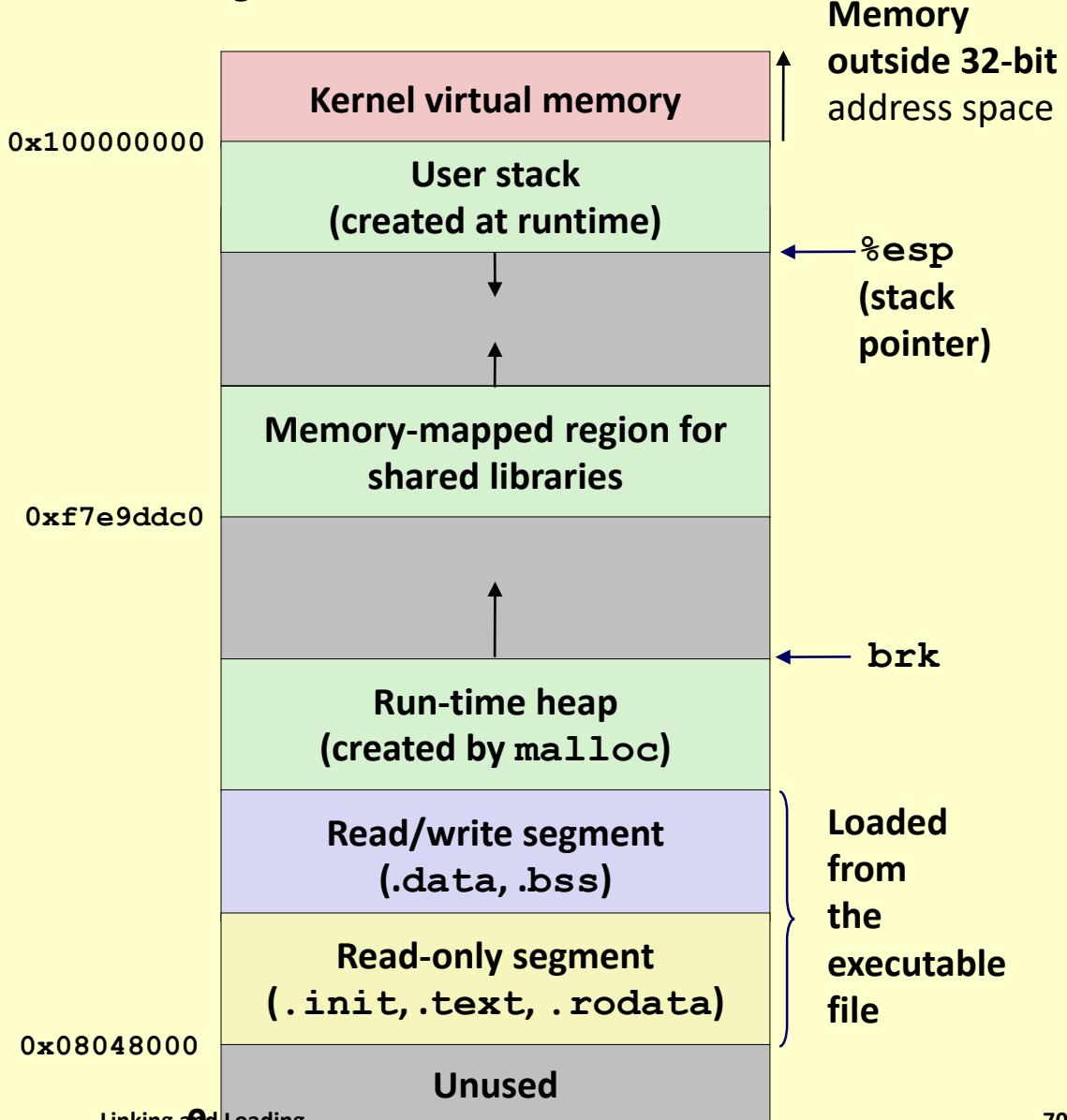
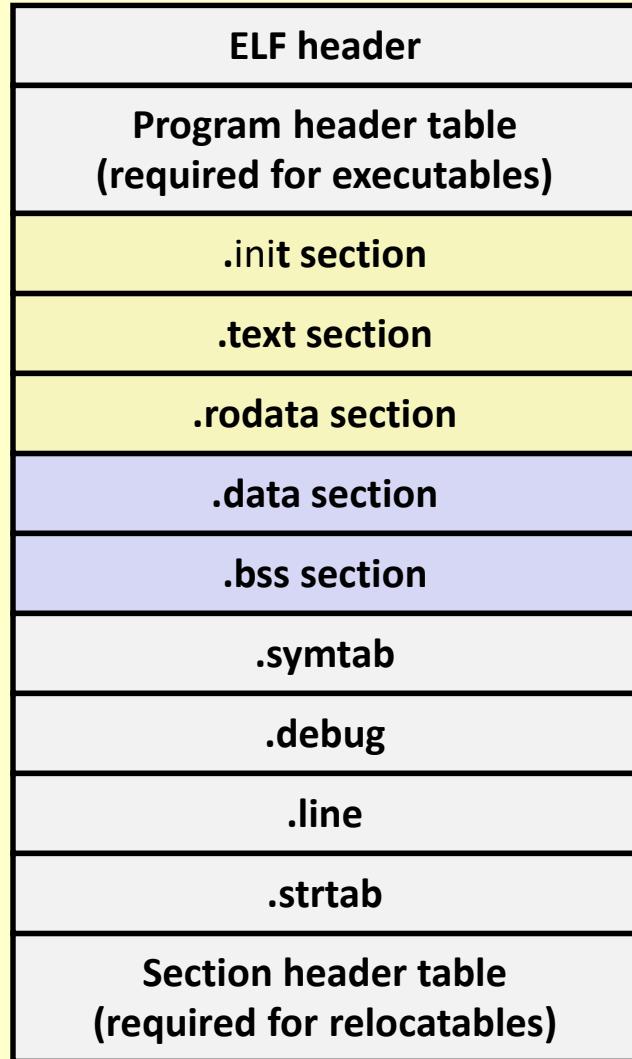
Quiz question

- **What does “undefined reference” mean?**
 - I.e., when compiling your C or C++ program

- **How do you fix it?**

Loading Executable Object Files

Executable Object File



Shared Libraries

■ Static libraries have the following disadvantages:

- Duplication in the stored executables (every function need std **libc**)
- Duplication in the running executables
- Minor bug fixes of system libraries require each application to explicitly relink

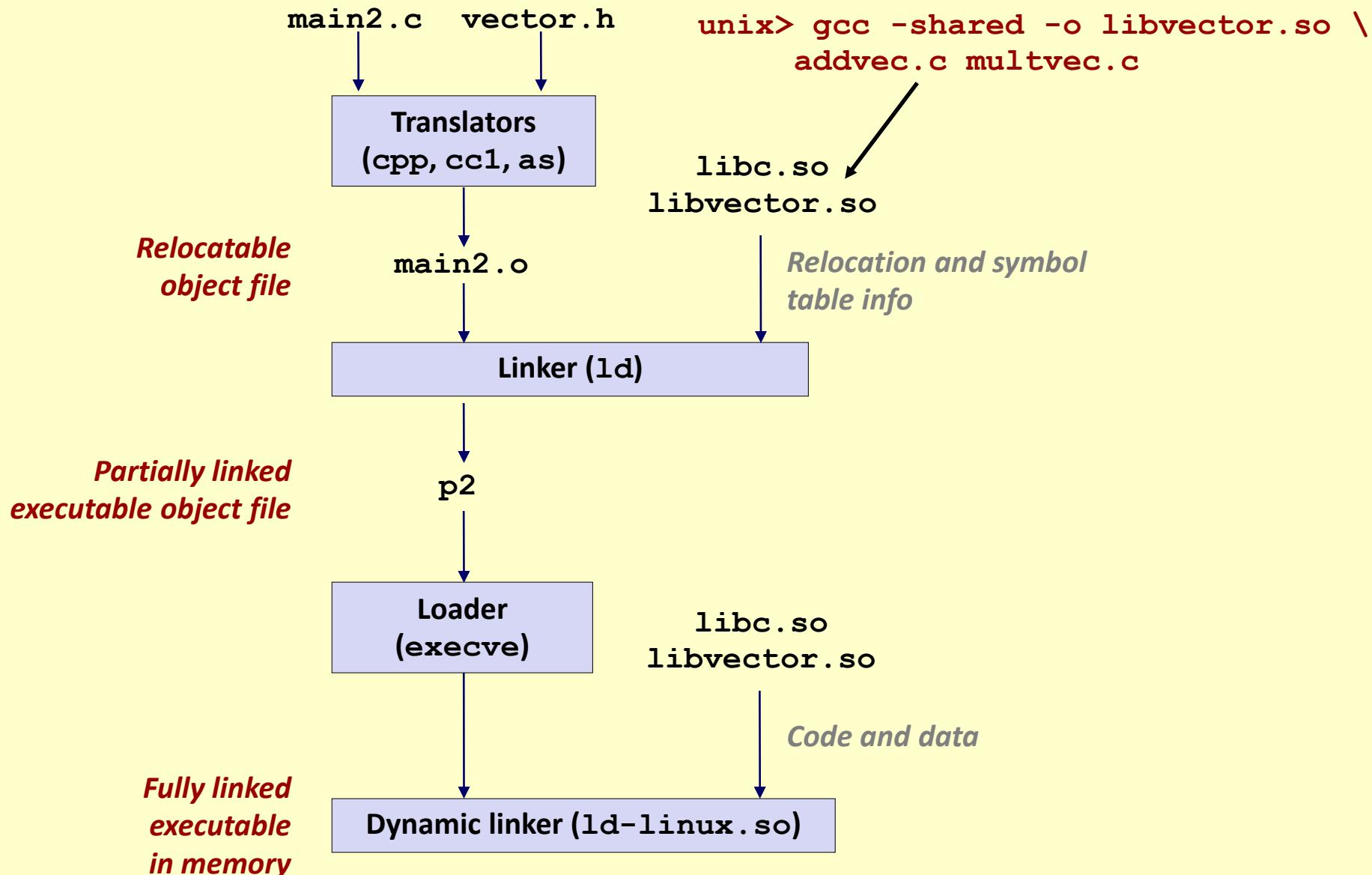
■ Modern solution: Shared Libraries

- Object files that contain code and data that are loaded and linked into an application *dynamically*, at either *load-time* or *run-time*
- Also called: dynamic link libraries, DLLs, .**so** files

Shared Libraries (continued)

- **Dynamic linking can occur when executable is first loaded and run (load-time linking).**
 - Common case for Linux, handled automatically by the dynamic linker (`ld-linux.so`).
 - Standard C library (`libc.so`) usually dynamically linked.
- **Dynamic linking can also occur after program has begun (run-time linking).**
 - In Linux, this is done by calls to the `dlopen()` interface
 - Distributing software.
 - High-performance web servers.
 - Runtime library interpositioning.
- **Shared library routines can be shared by multiple processes.**
 - More on this when we learn about virtual memory (in OS course!)

Dynamic Linking at Load-time



Dynamic Linking at Run-time

```
#include <stdio.h>
#include <dlfcn.h>

int x[2] = {1, 2};
int y[2] = {3, 4};
int z[2];

int main()
{
    void *handle;
    void (*addvec)(int *, int *, int *, int);
    char *error;

    /* dynamically load the shared lib that contains addvec() */
    handle = dlopen("./libvector.so", RTLD_LAZY);
    if (!handle) {
        fprintf(stderr, "%s\n", dlerror());
        exit(1);
    }
```

Continued on next slide

Dynamic Linking at Run-time

```
...  
  
/* get a pointer to the addvec() function we just loaded */  
addvec = dlsym(handle, "addvec");  
if ((error = dlerror()) != NULL) {  
    fprintf(stderr, "%s\n", error);  
    exit(1);  
}  
  
/* Now we can call addvec() just like any other function */  
addvec(x, y, z, 2);  
printf("z = [%d %d]\n", z[0], z[1]);  
  
/* unload the shared library */  
if (dlclose(handle) < 0) {  
    fprintf(stderr, "%s\n", dlerror());  
    exit(1);  
}  
return 0;  
}
```

Questions?

Today

- **Linking**
- **Case study: Library interpositioning**

Case Study: Library Interpositioning

- **Library interpositioning : powerful linking technique that allows programmers to intercept calls to arbitrary functions**

- **Interpositioning can occur at:**
 - Compile time: When the source code is compiled
 - Link time: When the relocatable object files are statically linked to form an executable object file
 - Load/run time: When an executable object file is loaded into memory, dynamically linked, and then executed.

Some Interpositioning Applications

■ Security

- Confinement (sandboxing)
 - Interpose calls to libc functions.
- Behind the scenes encryption
 - Automatically encrypt otherwise unencrypted network connections.

■ Monitoring and Profiling

- Count number of calls to functions
- Characterize call sites and arguments to functions
- Malloc tracing
 - Detecting memory leaks
 - **Generating address traces**

Example program

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>

int main()
{
    free(malloc(10));
    printf("hello, world\n");
    exit(0);
}
```

hello.c

- **Goal: trace the addresses and sizes of the allocated and freed blocks, without modifying the source code.**
- **Three solutions: interpose on the `lib malloc` and `free` functions at**
 - compile time,
 - link time, and/or
 - load/run time.

Compile-time Interpositioning

```
#ifdef COMPILETIME
/* Compile-time interposition of malloc and free using C
 * preprocessor. A local malloc.h file defines malloc (free)
 * as wrappers mymalloc (myfree) respectively.
 */

#include <stdio.h>
#include <malloc.h>

/*
 * mymalloc - malloc wrapper function
 */
void *mymalloc(size_t size, char *file, int line)
{
    void *ptr = malloc(size);
    printf("%s:%d: malloc(%d)=%p\n", file, line, (int)size,
ptr);
    return ptr;
}
```

mymalloc.c

Compile-time Interpositioning

```
#define malloc(size) mymalloc(size, __FILE__, __LINE__ )
#define free(ptr) myfree(ptr, __FILE__, __LINE__ )

void *mymalloc(size_t size, char *file, int line);
void myfree(void *ptr, char *file, int line);

malloc.h
```

```
linux> make helloc
gcc -O2 -Wall -DCOMPILETIME -c mymalloc.c
gcc -O2 -Wall -I. -o helloc hello.c mymalloc.o
linux> make runc
./helloc
hello.c:7: malloc(10)=0x501010
hello.c:7: free(0x501010)
hello, world
```

Link-time Interpositioning

```
#ifdef LINKTIME
/* Link-time interposition of malloc and free using the
static linker's (ld) "--wrap symbol" flag. */

#include <stdio.h>

void * __real_malloc(size_t size);
void __real_free(void *ptr);

/*
 * __wrap_malloc - malloc wrapper function
 */
void * __wrap_malloc(size_t size)
{
    void *ptr = __real_malloc(size);
    printf("malloc(%d) = %p\n", (int)size, ptr);
    return ptr;
}
```

mymalloc.c

Link-time Interpositioning

```
linux> make hellol
gcc -O2 -Wall -DLINKTIME -c mymalloc.c
gcc -O2 -Wall -Wl,--wrap,malloc -Wl,--wrap,free \
-o hellol hello.c mymalloc.o
linux> make runl
./hellol
malloc(10) = 0x501010
free(0x501010)
hello, world
```

- The “**-Wl**” flag passes argument to linker
- Telling linker “**--wrap,malloc**” tells it to resolve references in a special way:
 - Refs to **malloc** should be resolved as **__wrap_malloc**
 - Refs to **__real_malloc** should be resolved as **malloc**

```
#ifdef RUNTIME
/* Run-time interposition of malloc and free based on
 * dynamic linker's (ld-linux.so) LD_PRELOAD mechanism */
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <dlsym.h>

void *malloc(size_t size)
{
    static void *(*mallocp)(size_t size);
    char *error;
    void *ptr;

    /* get address of libc malloc */
    if (!mallocp) {
        mallocp = dlsym(RTLD_NEXT, "malloc");
        if ((error = dlerror()) != NULL) {
            fputs(error, stderr);
            exit(1);
        }
    }
    ptr = mallocp(size);
    printf("malloc(%d) = %p\n", (int)size, ptr);
    return ptr;
}
```

Load/Run-time Interpositioning

mymalloc.c

Load/Run-time Interpositioning

```
linux> make hellor
gcc -O2 -Wall -DRUNTIME -shared -fPIC -o mymalloc.so mymalloc.c
gcc -O2 -Wall -o hellor hello.c
linux> make runr
(LD_PRELOAD="/usr/lib64/libdl.so ./mymalloc.so" ./hellor)
malloc(10) = 0x501010
free(0x501010)
hello, world
```

- The `LD_PRELOAD` environment variable tells the dynamic linker to resolve unresolved refs (e.g., to `malloc`) by looking in `libdl.so` and `mymalloc.so` first.
 - `libdl.so` necessary to resolve references to the `dlopen` functions.

Interpositioning Recap

■ Compile Time

- Apparent calls to malloc/free get macro-expanded into calls to mymalloc/myfree

■ Link Time

- Use linker trick to have special name resolutions
 - malloc → __wrap_malloc
 - __real_malloc → malloc

■ Compile Time

- Implement custom version of malloc/free that use dynamic linking to load library malloc/free under different names

Questions?

More on Memory Hierarchy

Professor Hugh C. Lauer
CS-2011, Machine Organization and Assembly Language

(Slides include copyright materials from *Computer Systems: A Programmer's Perspective*, by Bryant and O'Hallaron, and from *The C Programming Language*, by Kernighan and Ritchie)

Today

Reading Assignment: §6.1 – §6.5

- Storage technologies and trends
- Locality of reference ← First
- Caching in the memory hierarchy ← Preparation for Cachelab

Random-Access Memory (RAM)

■ Key features

- RAM is traditionally packaged as a chip.
- Basic storage unit is normally a cell (one bit per cell).
- Multiple RAM chips form a memory.

■ Static RAM (SRAM)

- Each cell stores a bit with a four or six-transistor circuit.
- Retains value indefinitely, as long as it is kept powered.
- Relatively insensitive to electrical noise (EMI), radiation, etc.
- Faster and more expensive than DRAM.

■ Dynamic RAM (DRAM)

- Each cell stores bit with a capacitor. One transistor is used for access
- Value must be refreshed every 10-100 ms.
- More sensitive to disturbances (EMI, radiation,...) than SRAM.
- Slower and cheaper than SRAM.

SRAM vs DRAM Summary

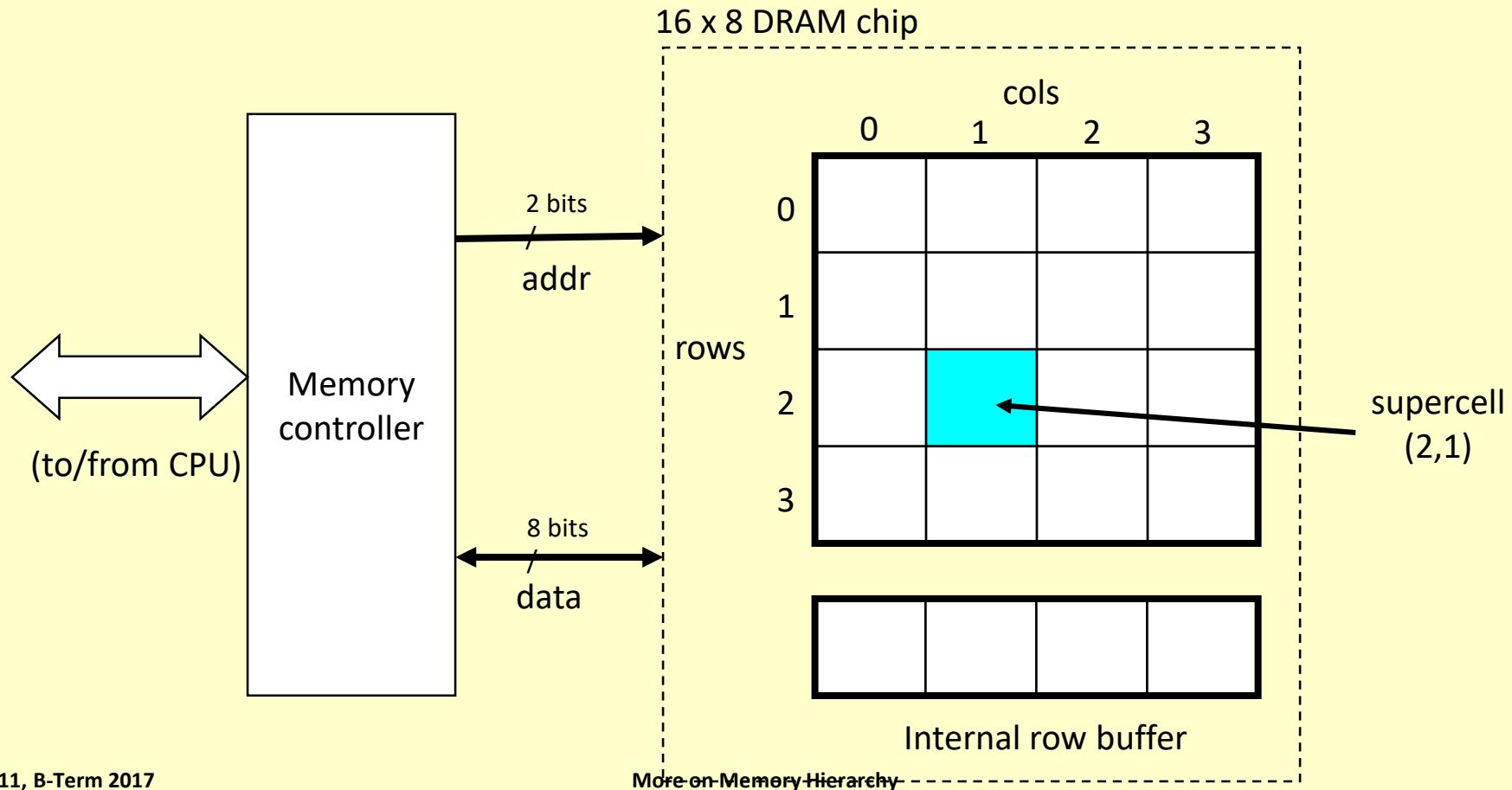
	Trans. per bit	Access time	Needs refresh?	Needs EDC*?	Cost	Applications
SRAM	4 or 6	1X	No	Maybe	100x	Cache memories
DRAM	1	10X	Yes	Yes	1X	Main memories, frame buffers

EDC = Error Detection and Correction

Conventional DRAM Organization

■ $d \times w$ DRAM:

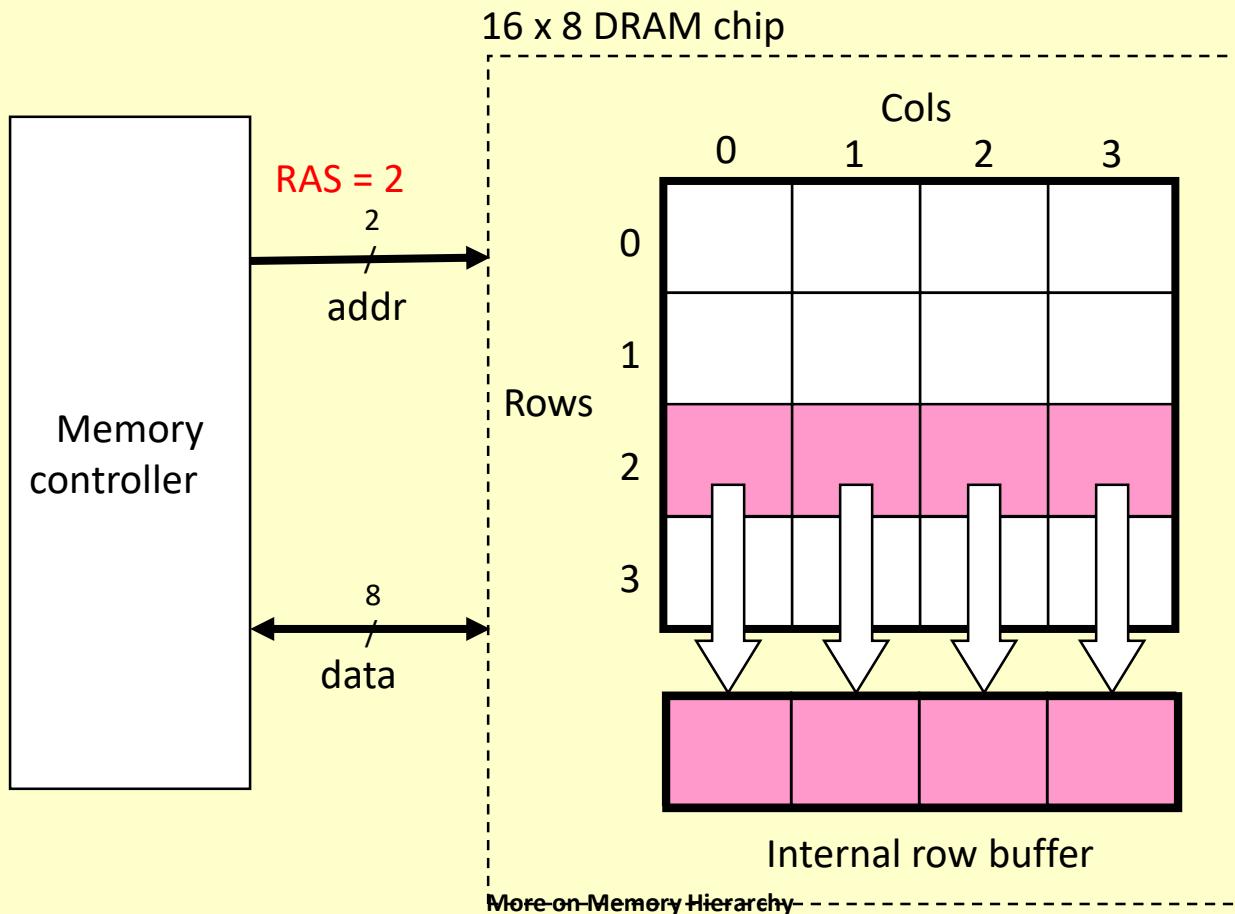
- d^*w total bits organized as d **supercells** of size w bits



Reading DRAM Supercell (2,1)

Step 1(a): Row access strobe (**RAS**) selects row 2.

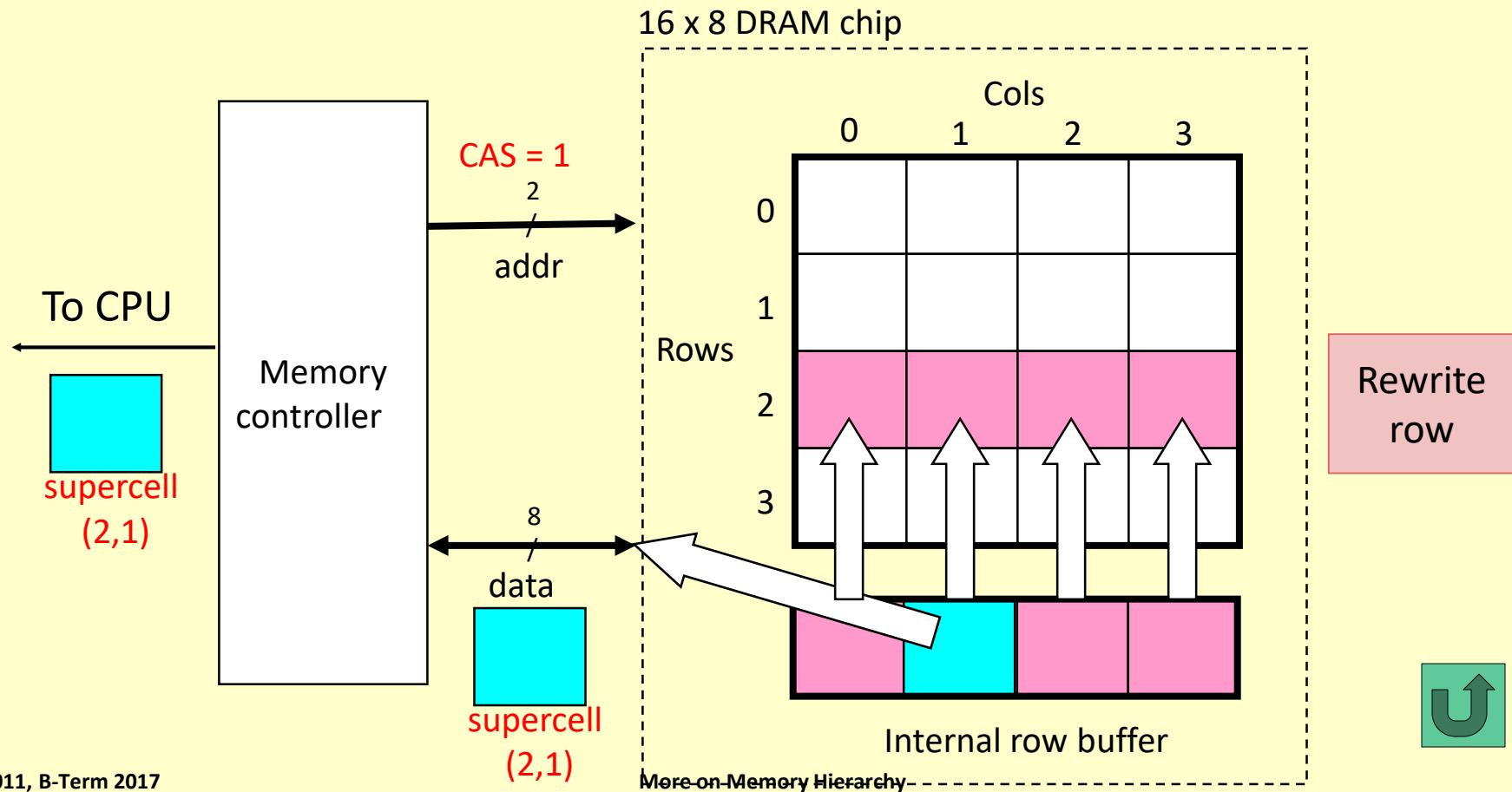
Step 1(b): Row 2 copied from DRAM array to row buffer.



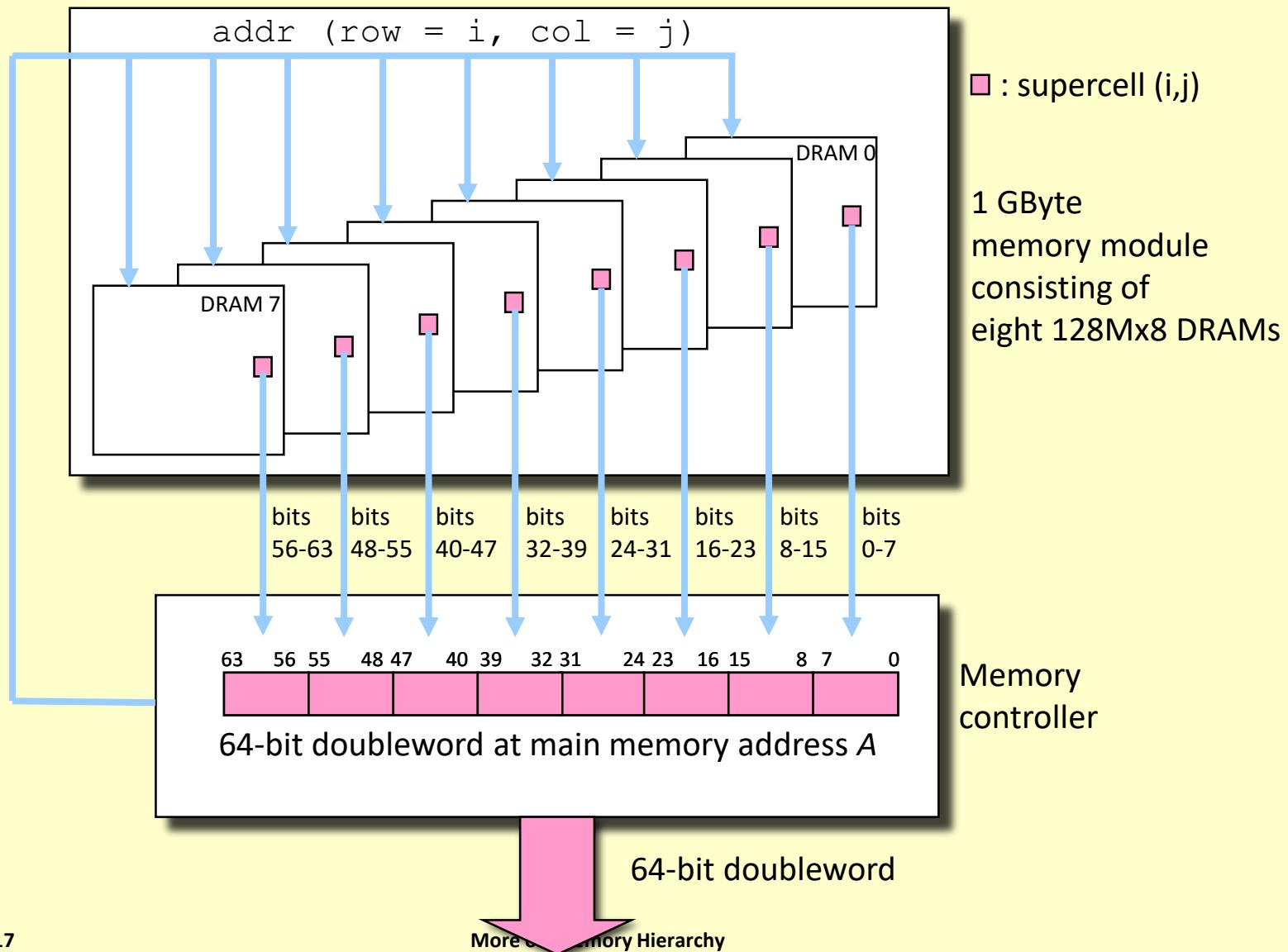
Reading DRAM Supercell (2,1)

Step 2(a): Column access strobe (CAS) selects column 1.

Step 2(b): Supercell (2,1) copied from buffer to data lines, and eventually back to the CPU.



Memory Modules



Enhanced DRAMs

- **Basic DRAM cell has not changed since invention in 1966.**
 - Commercialized by Intel in 1970
 - (except for tinier cells, more bits per sq millimeter)
- **DRAM cores with better interface logic and faster I/O :**
 - Synchronous DRAM (**SDRAM**)
 - Uses a conventional clock signal instead of asynchronous control
 - Allows reuse of the row addresses (e.g., RAS, CAS, CAS, CAS)
 - Double data-rate synchronous DRAM (**DDR SDRAM**)
 - Double edge clocking sends two bits per cycle per pin
 - Different types distinguished by size of small prefetch buffer:
 - **DDR** (2 bits), **DDR2** (4 bits), **DDR3** (8 bits)
 - By 2010, standard for most server and desktop systems
 - Intel Core i7 supports only DDR3 SDRAM

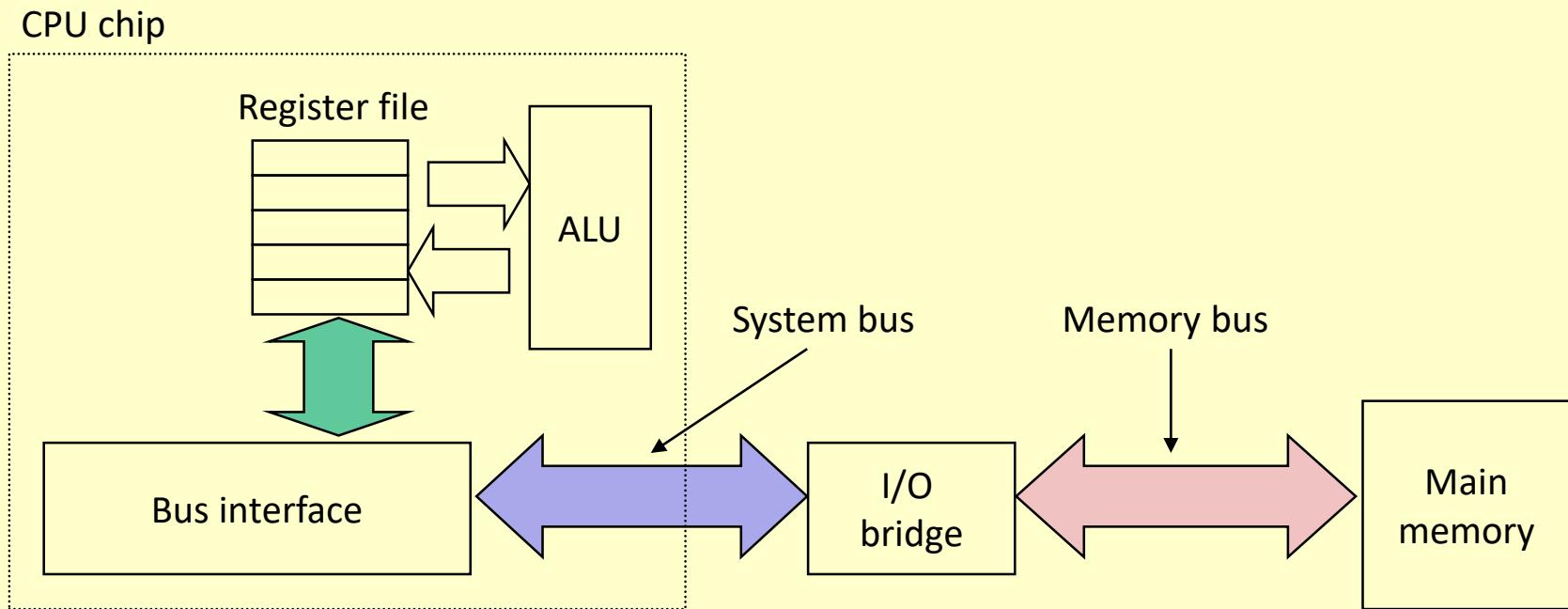


Nonvolatile Memories

- **DRAM and SRAM are volatile memories**
 - Lose information if powered off.
- **Nonvolatile memories retain value even if powered off**
 - Read-only memory (**ROM**): programmed during production
 - Programmable ROM (**PROM**): can be programmed once
 - Eraseable PROM (**EPROM**): can be bulk erased (UV, X-Ray)
 - Electrically eraseable PROM (**EEPROM**): electronic erase capability
 - Flash memory: EEPROMs with partial (sector) erase capability
 - Wears out after about 100,000 erasings.
- **Uses for Nonvolatile Memories**
 - Firmware programs stored in a ROM (BIOS, controllers for disks, network cards, graphics accelerators, security subsystems,...)
 - Solid state disks (replace rotating disks in thumb drives, smart phones, mp3 players, tablets, laptops,...)
 - Disk caches

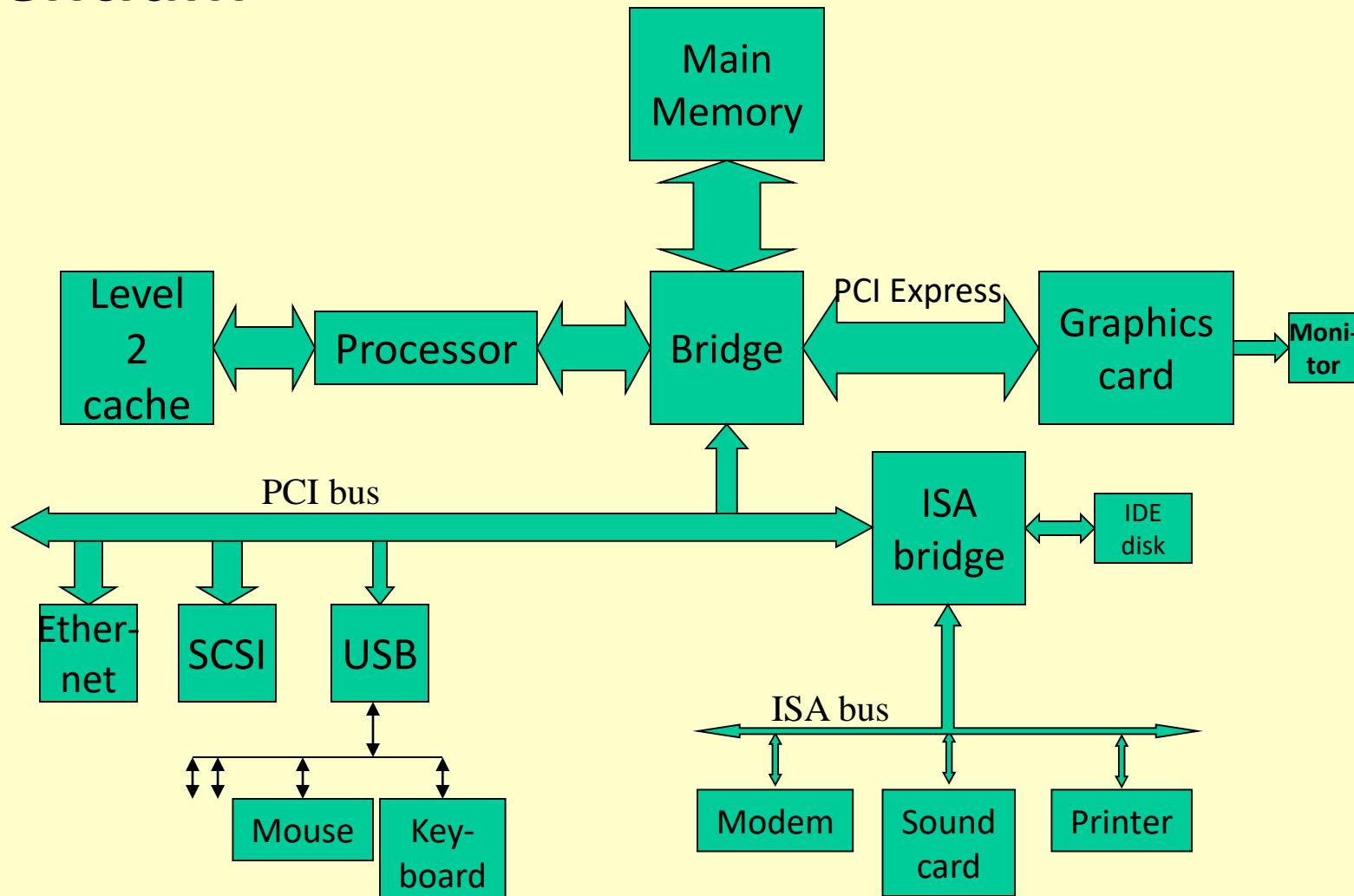
Traditional Bus Structure Connecting CPU and Memory

- A **bus** is a collection of parallel wires that carry address, data, and control signals.
- Buses are typically shared by multiple devices.



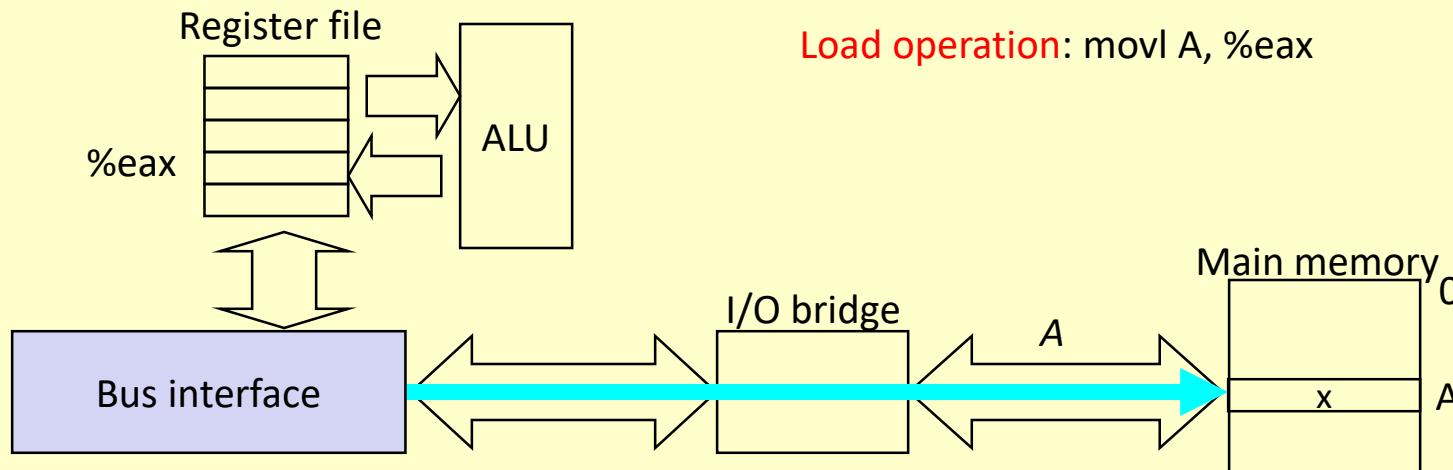
Hardware Organization — 2005 era

Pentium



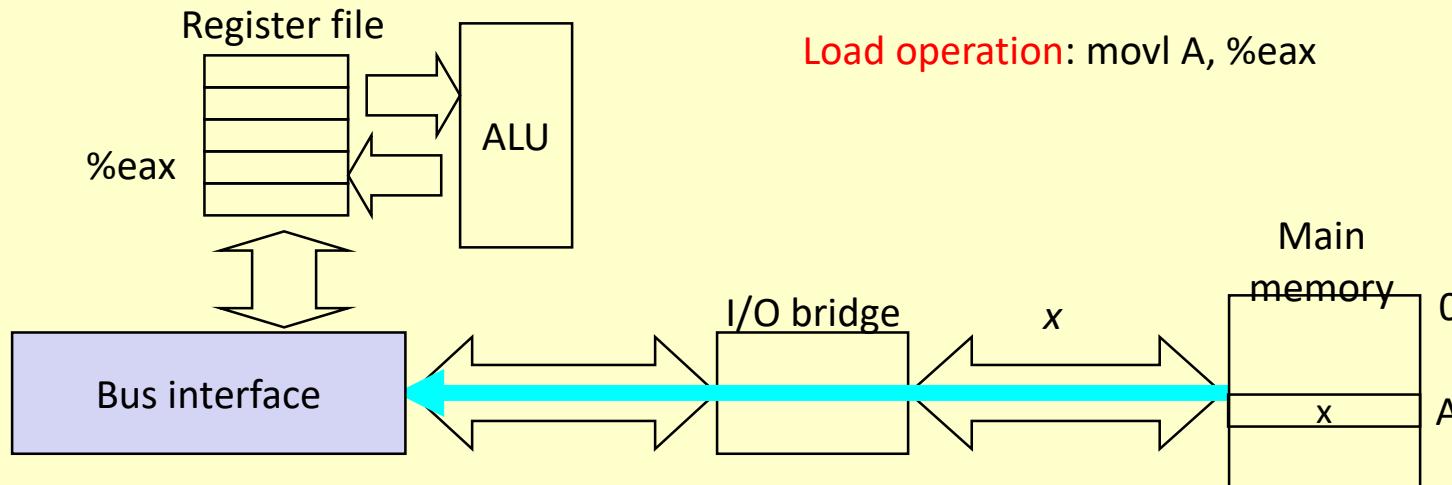
Memory Read Transaction (1)

- CPU places address A on the memory bus.



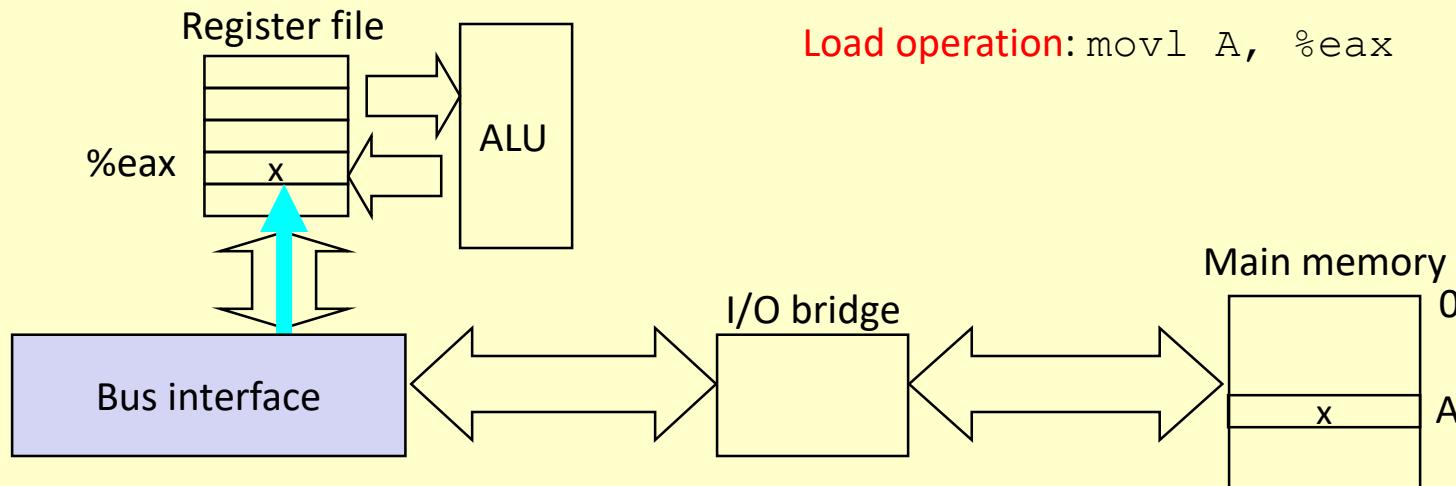
Memory Read Transaction (2)

- Main memory reads A from the memory bus, retrieves word x, and places it on the bus.



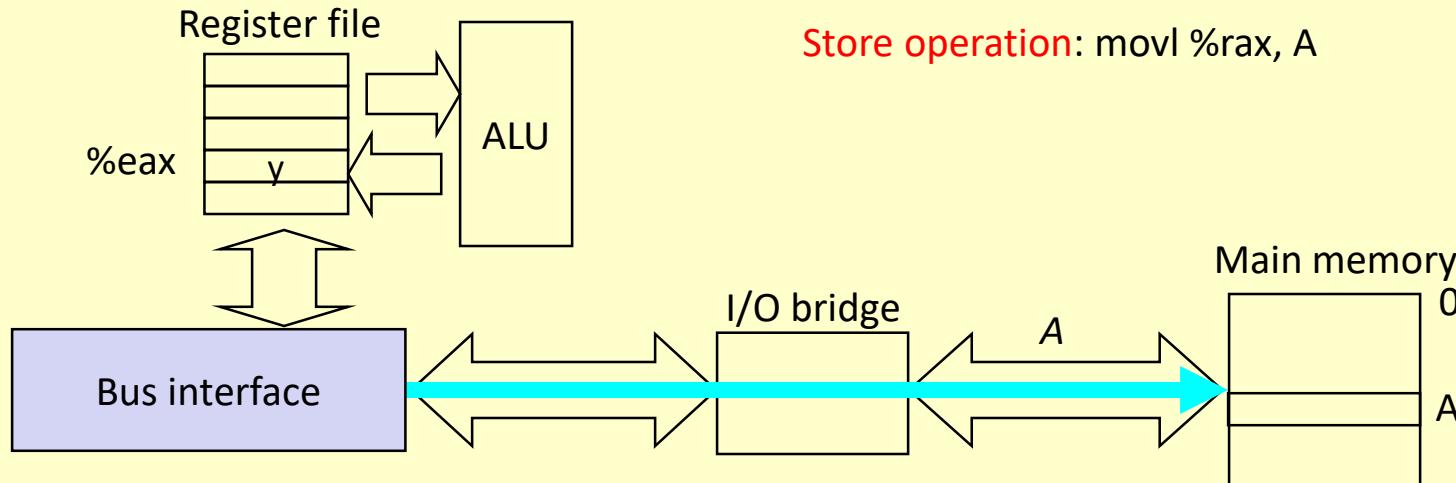
Memory Read Transaction (3)

- CPU read word x from the bus and copies it into register $\%rax$.



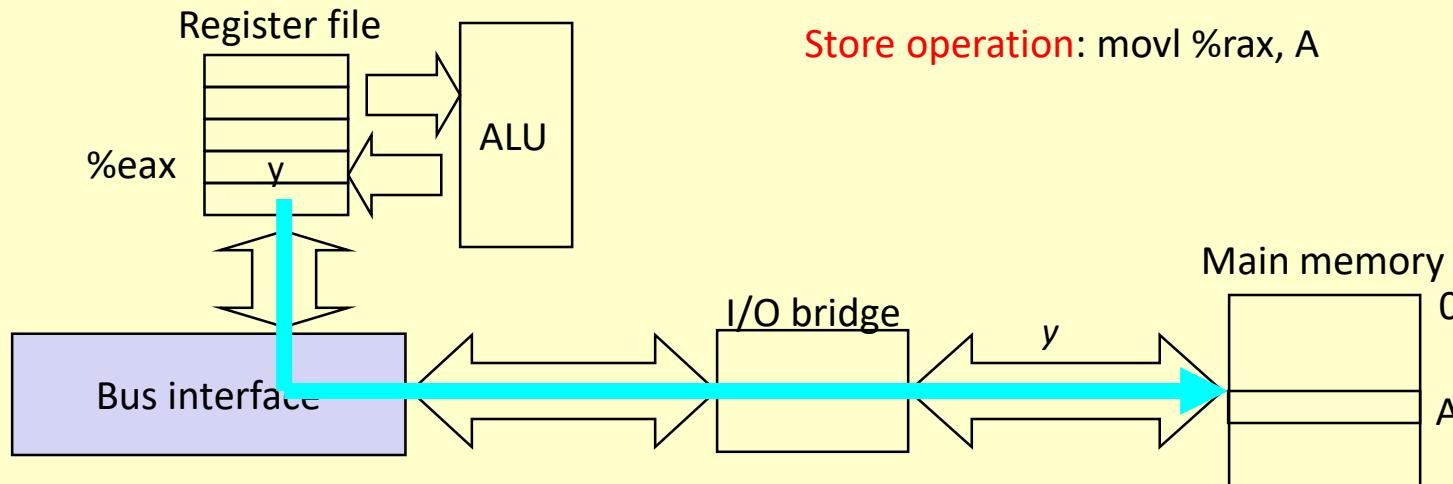
Memory Write Transaction (1)

- CPU places address A on bus. Main memory reads it and waits for the corresponding data word to arrive.



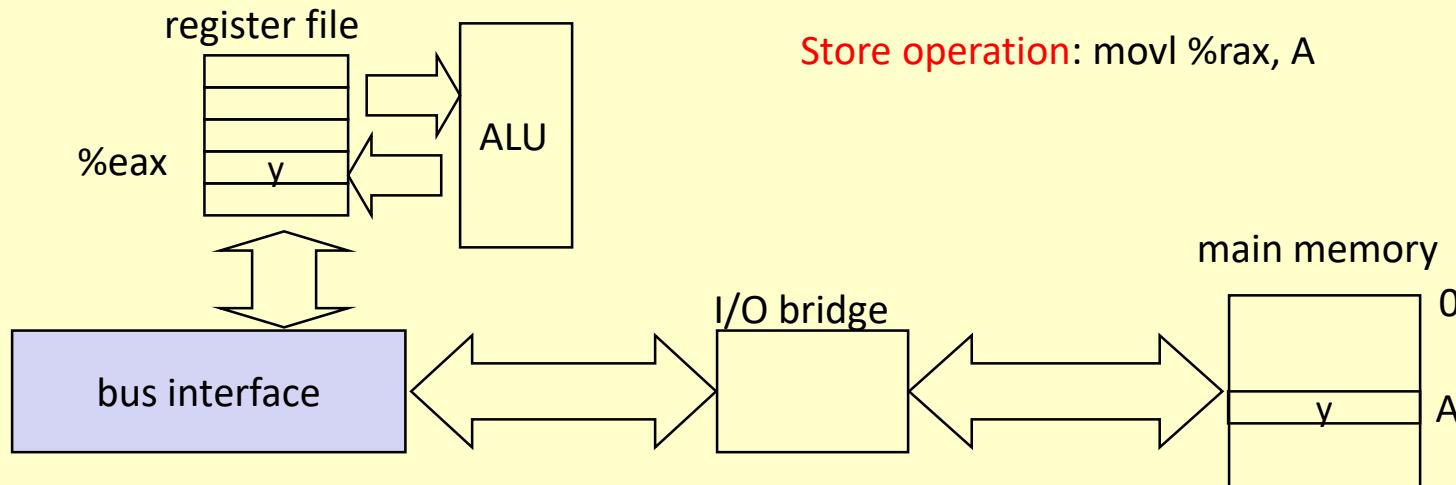
Memory Write Transaction (2)

- CPU places data word y on the bus.



Memory Write Transaction (3)

- Main memory reads data word y from the bus and stores it at address A.



Questions?

What's Inside A Disk Drive?

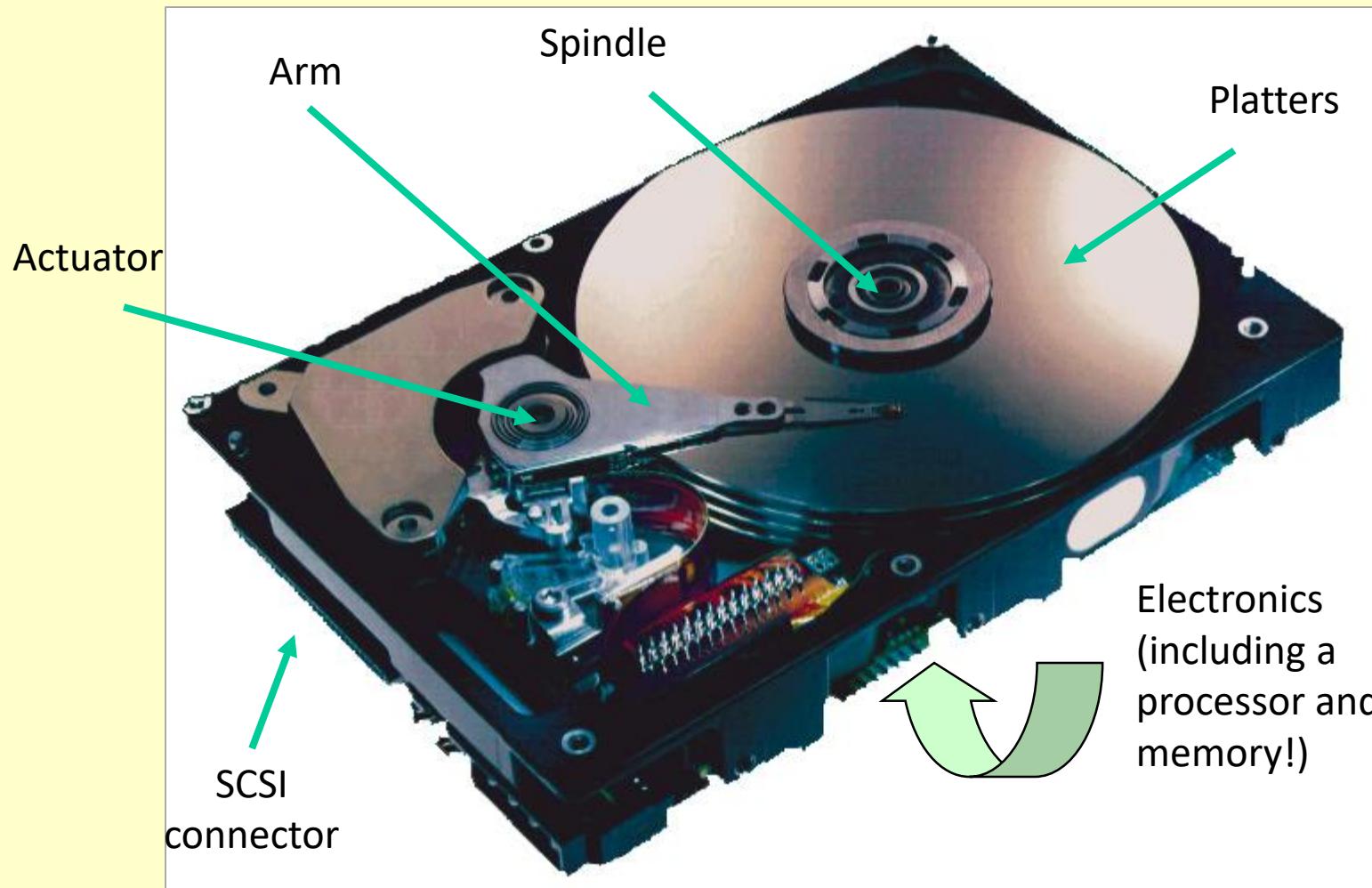
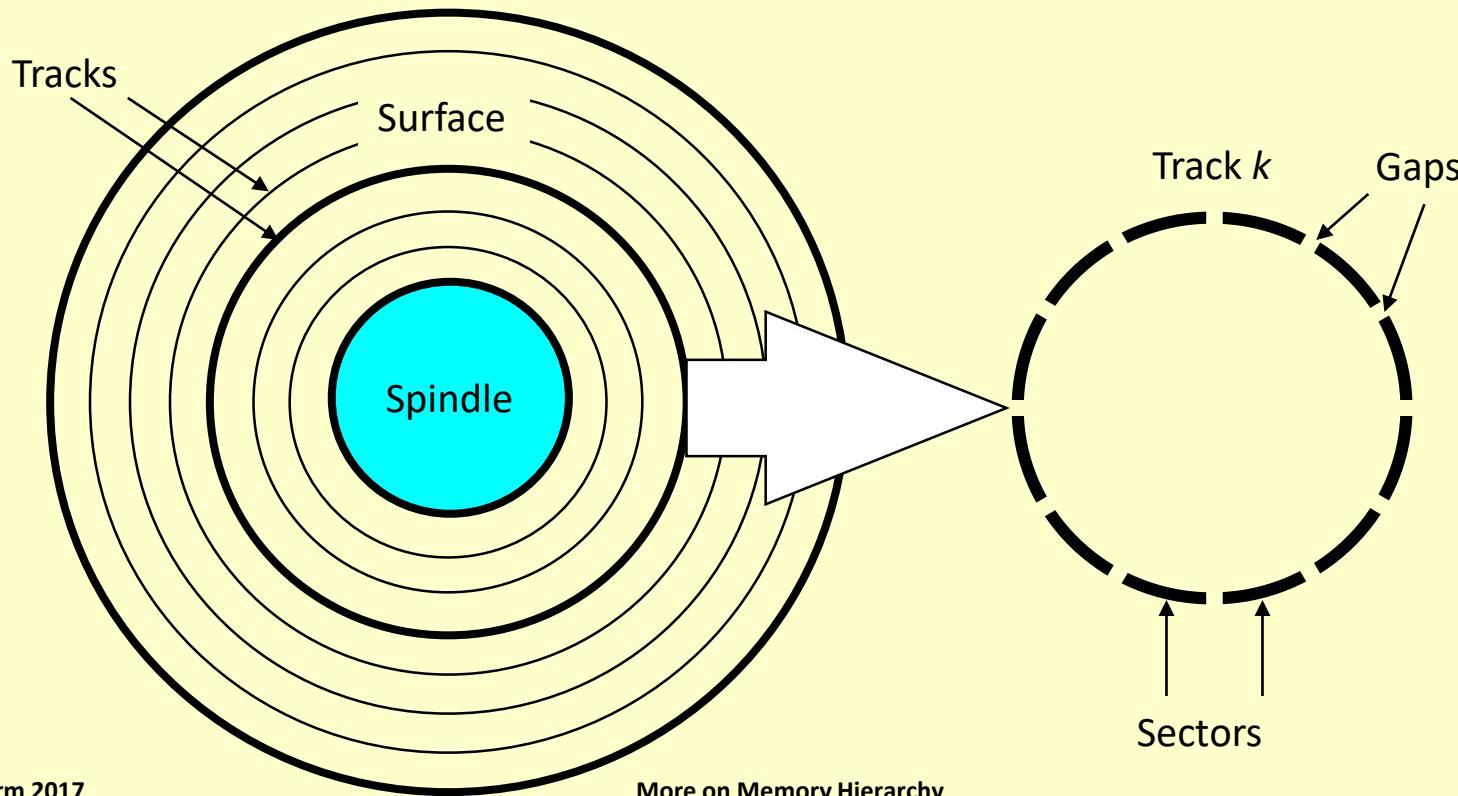


Image courtesy of Seagate Technology

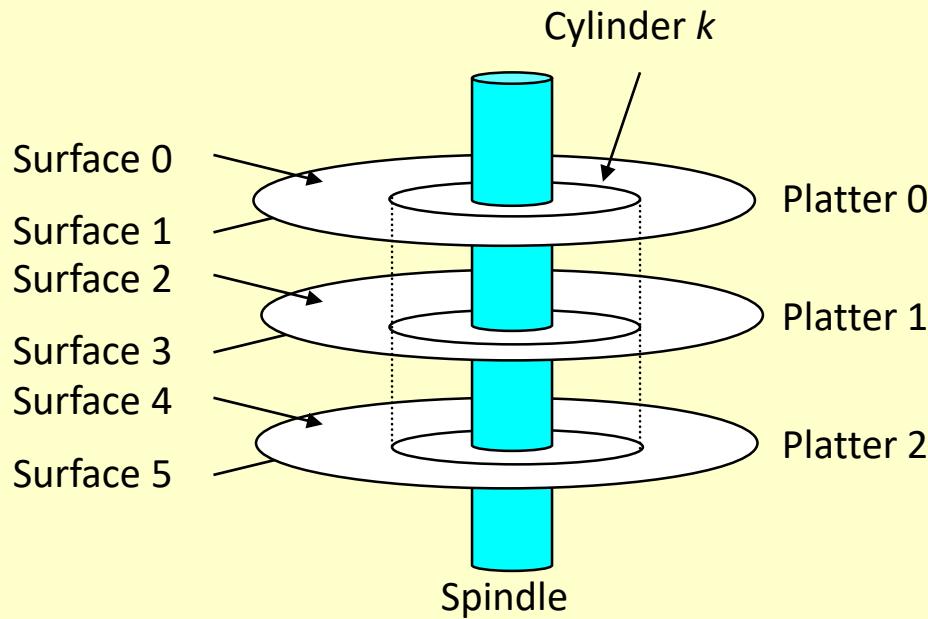
Disk Geometry

- Disks consist of **platters**, each with two **surfaces**.
- Each surface consists of concentric rings called **tracks**.
- Each track consists of **sectors** separated by **gaps**.



Disk Geometry (Multiple-Platter View)

- Aligned tracks form a cylinder.



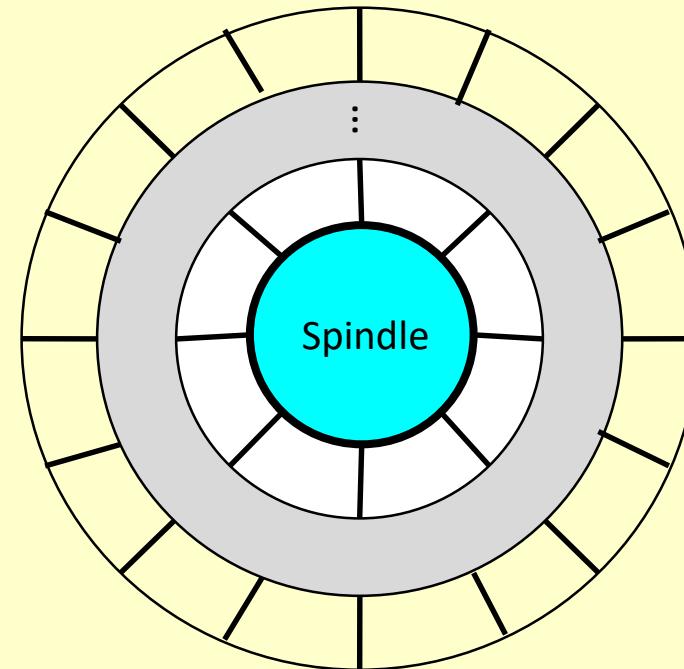
Disk Capacity

- **Capacity:** maximum number of bits that can be stored.
 - Vendors express capacity in units of gigabytes (GB), where $1 \text{ GB} = 10^9 \text{ Bytes}$ (Lawsuit pending! Claims deceptive advertising).
- **Capacity is determined by these technology factors:**
 - **Recording density** (bits/in): number of bits that can be squeezed into a 1 inch segment of a track.
 - **Track density** (tracks/in): number of tracks that can be squeezed into a 1 inch radial segment.
 - **Areal density** (bits/in²): product of recording and track density.
- **Modern disks partition tracks into disjoint subsets called recording zones**
 - Each track in a zone has the same number of sectors, determined by the circumference of innermost track.
 - Each zone has a different number of sectors/track

Recording zones

- Modern disks partition tracks into disjoint subsets called **recording zones**

- Each track in a zone has the same number of sectors, determined by the circumference of innermost track.
- Each zone has a different number of sectors/track, outer zones have more sectors/track than inner zones.
- So we use **average** number of sectors/track when computing capacity.



Computing Disk Capacity

**Capacity = (# bytes/sector) x (avg. # sectors/track) x
(# tracks/surface) x (# surfaces/platter) x
(# platters/disk)**

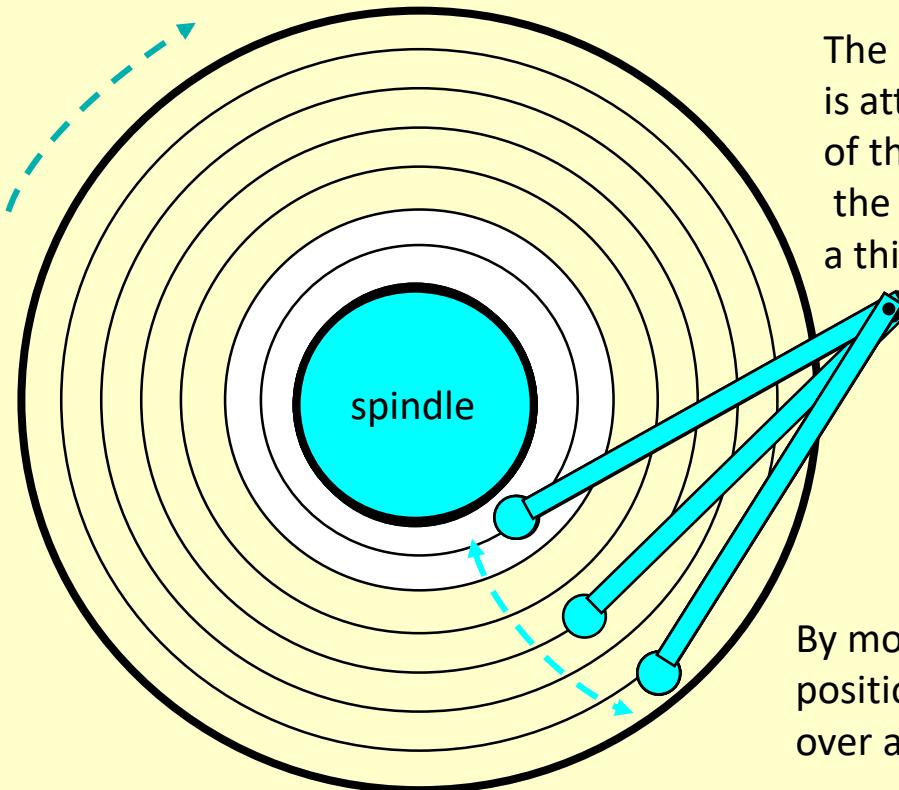
Example:

- 512 bytes/sector
- 300 sectors/track (on average)
- 20,000 tracks/surface
- 2 surfaces/platter
- 5 platters/disk

$$\begin{aligned}\text{Capacity} &= 512 \times 300 \times 20000 \times 2 \times 5 \\ &= 30,720,000,000 \\ &= 30.72 \text{ GB}\end{aligned}$$

Disk Operation (Single-Platter View)

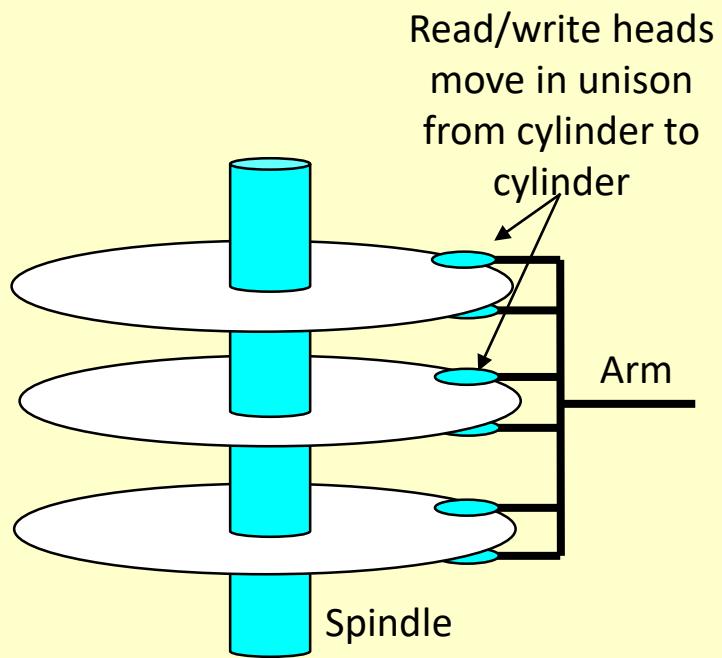
The disk surface spins at a fixed rotational rate



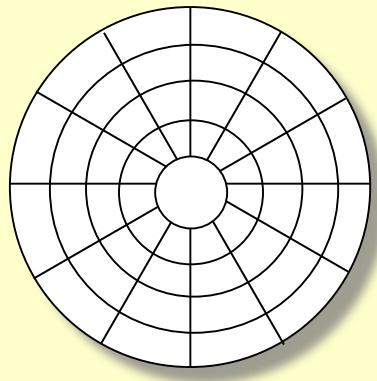
The read/write *head* is attached to the end of the *arm* and flies over the disk surface on a thin cushion of air.

By moving radially, the arm can position the read/write head over any track.

Disk Operation (Multi-Platter View)



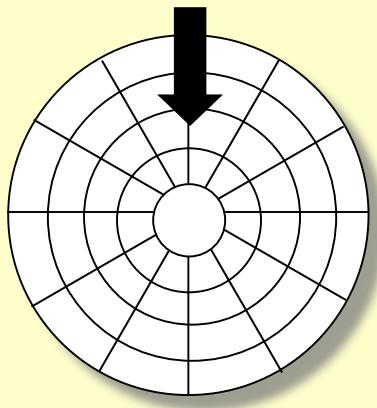
Disk Structure - top view of single platter



Surface organized into tracks

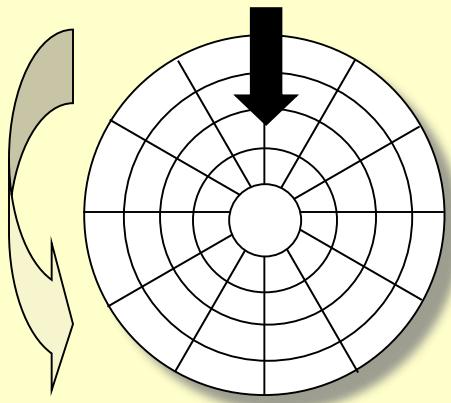
Tracks divided into sectors

Disk Access



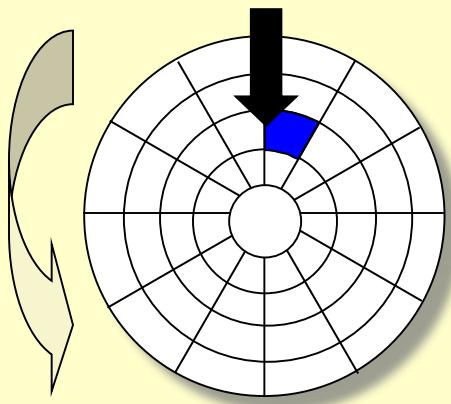
Head in position above a track

Disk Access



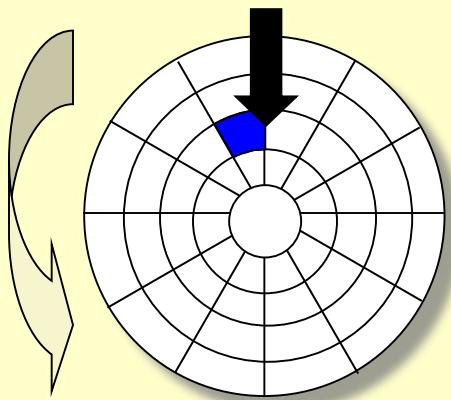
Rotation is counter-clockwise

Disk Access – Read



About to read blue sector

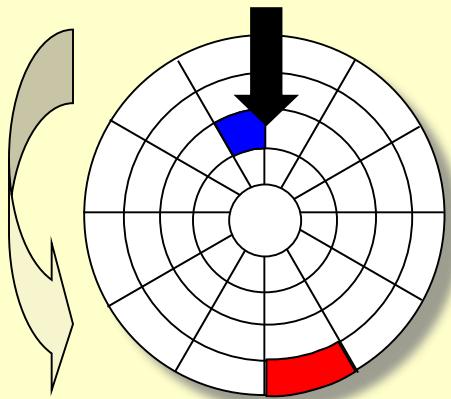
Disk Access – Read



After BLUE read

After reading blue sector

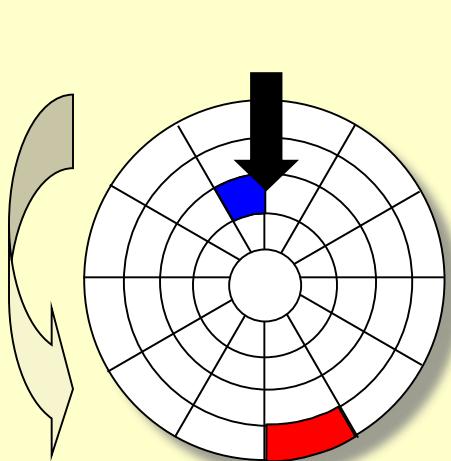
Disk Access – Read



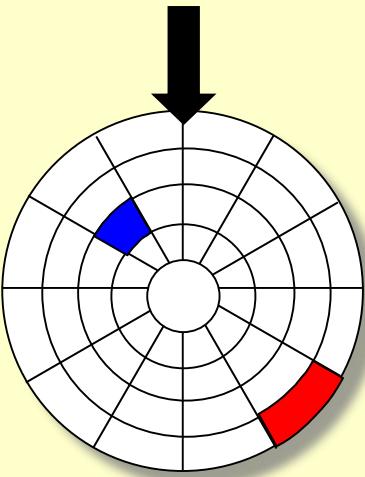
After BLUE read

Red request scheduled next

Disk Access – Seek



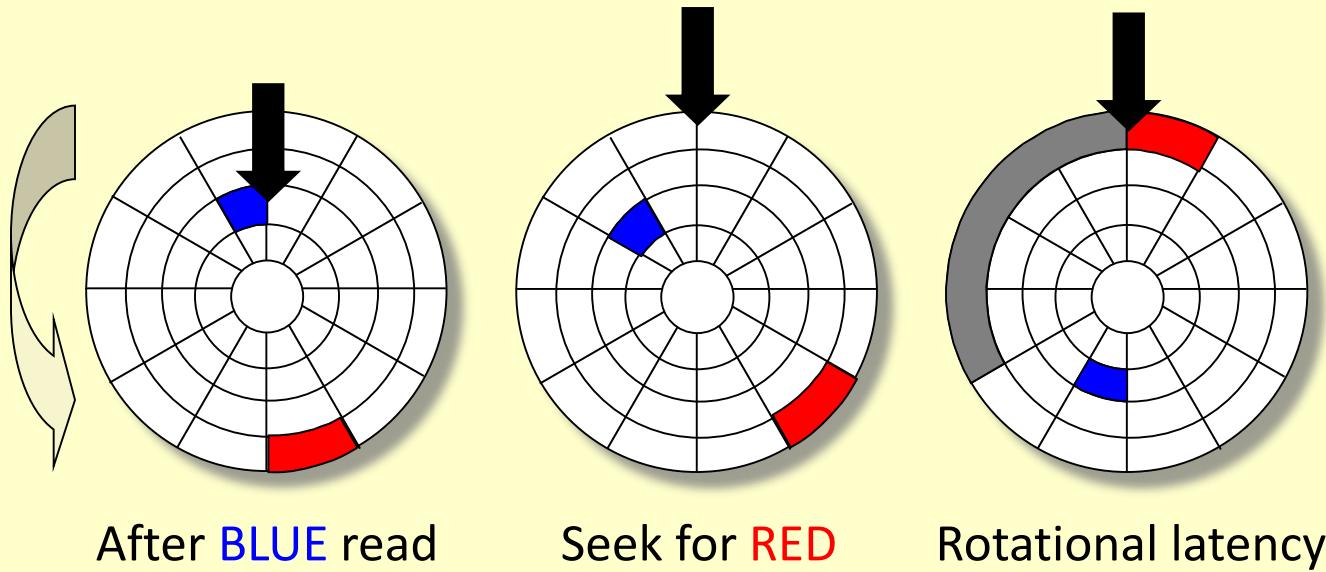
After **BLUE** read



Seek for **RED**

Seek to red's track

Disk Access – Rotational Latency



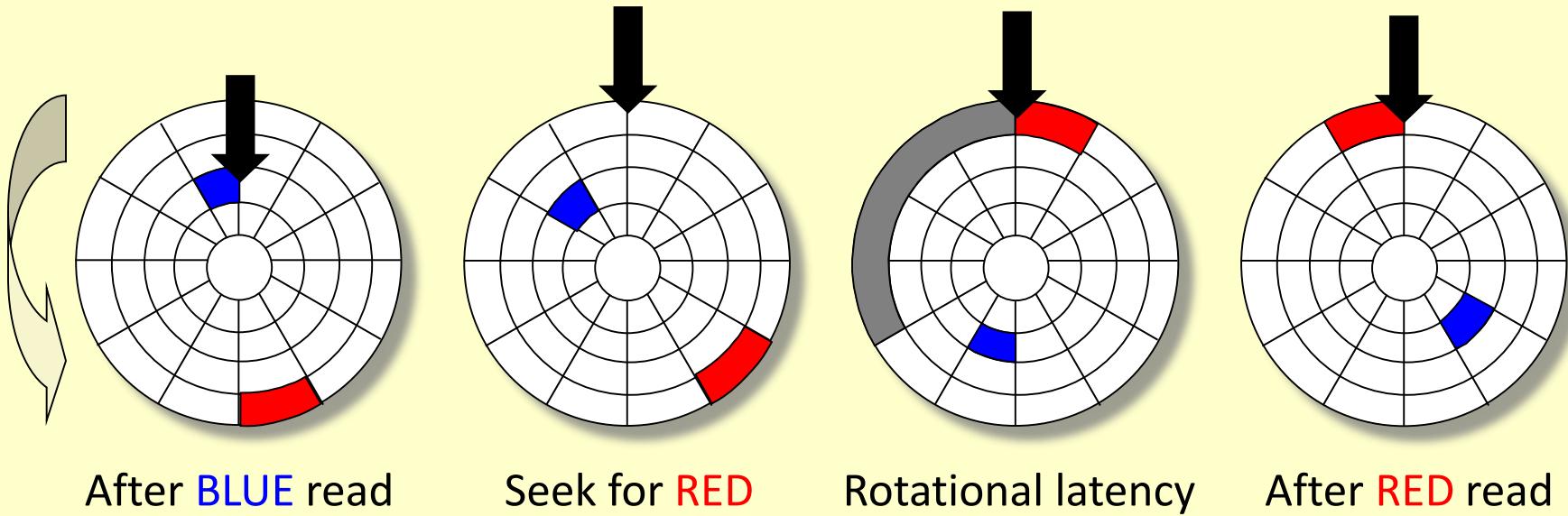
After **BLUE** read

Seek for **RED**

Rotational latency

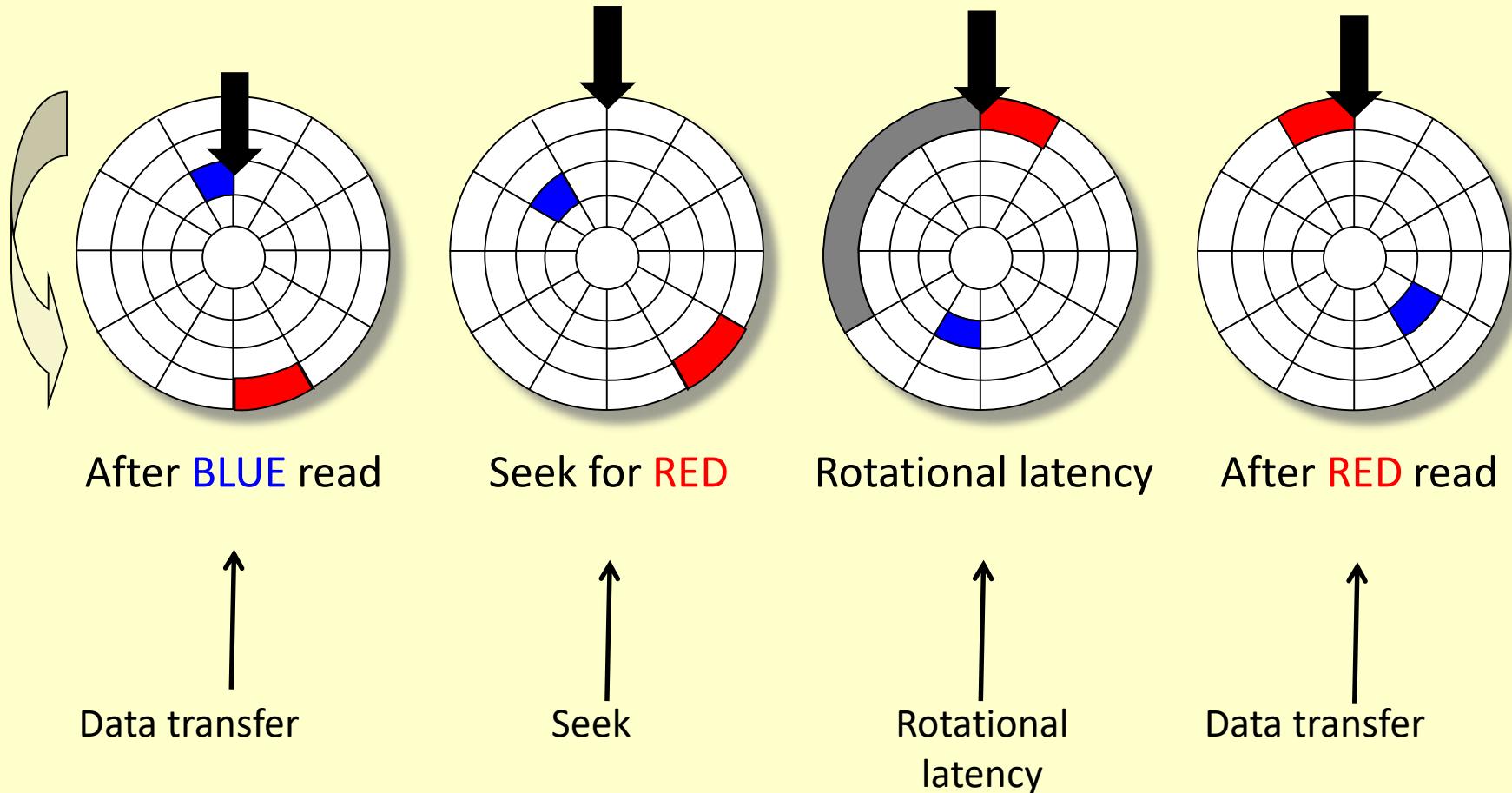
Wait for red sector to rotate around

Disk Access – Read



Complete read of red

Disk Access – Service Time Components



Disk Access Time

- Average time to access some target sector approximated by :
 - $T_{\text{access}} = T_{\text{avg seek}} + T_{\text{avg rotation}} + T_{\text{avg transfer}}$
- Seek time ($T_{\text{avg seek}}$)
 - Time to position heads over cylinder containing target sector.
 - Typical $T_{\text{avg seek}}$ is 3—9 ms
- Rotational latency ($T_{\text{avg rotation}}$)
 - Time waiting for first bit of target sector to pass under r/w head.
 - $T_{\text{avg rotation}} = 1/2 \times 1/\text{RPMs} \times 60 \text{ sec}/1 \text{ min}$
 - Typical $T_{\text{avg rotation}} = 7200 \text{ RPMs}$
- Transfer time ($T_{\text{avg transfer}}$)
 - Time to read the bits in the target sector.
 - $T_{\text{avg transfer}} = 1/\text{RPM} \times 1/(\text{avg # sectors/track}) \times 60 \text{ secs}/1 \text{ min.}$

Disk Access Time Example

■ Given:

- Rotational rate = 7,200 RPM
- Average seek time = 9 ms.
- Avg # sectors/track = 400.

■ Derived:

- $T_{avg\ rotation} = 1/2 \times (60\ secs/7200\ RPM) \times 1000\ ms/sec = 4\ ms.$
- $T_{avg\ transfer} = 60/7200\ RPM \times 1/400\ secs/track \times 1000\ ms/sec = 0.02\ ms$
- $T_{access} = 9\ ms + 4\ ms + 0.02\ ms$

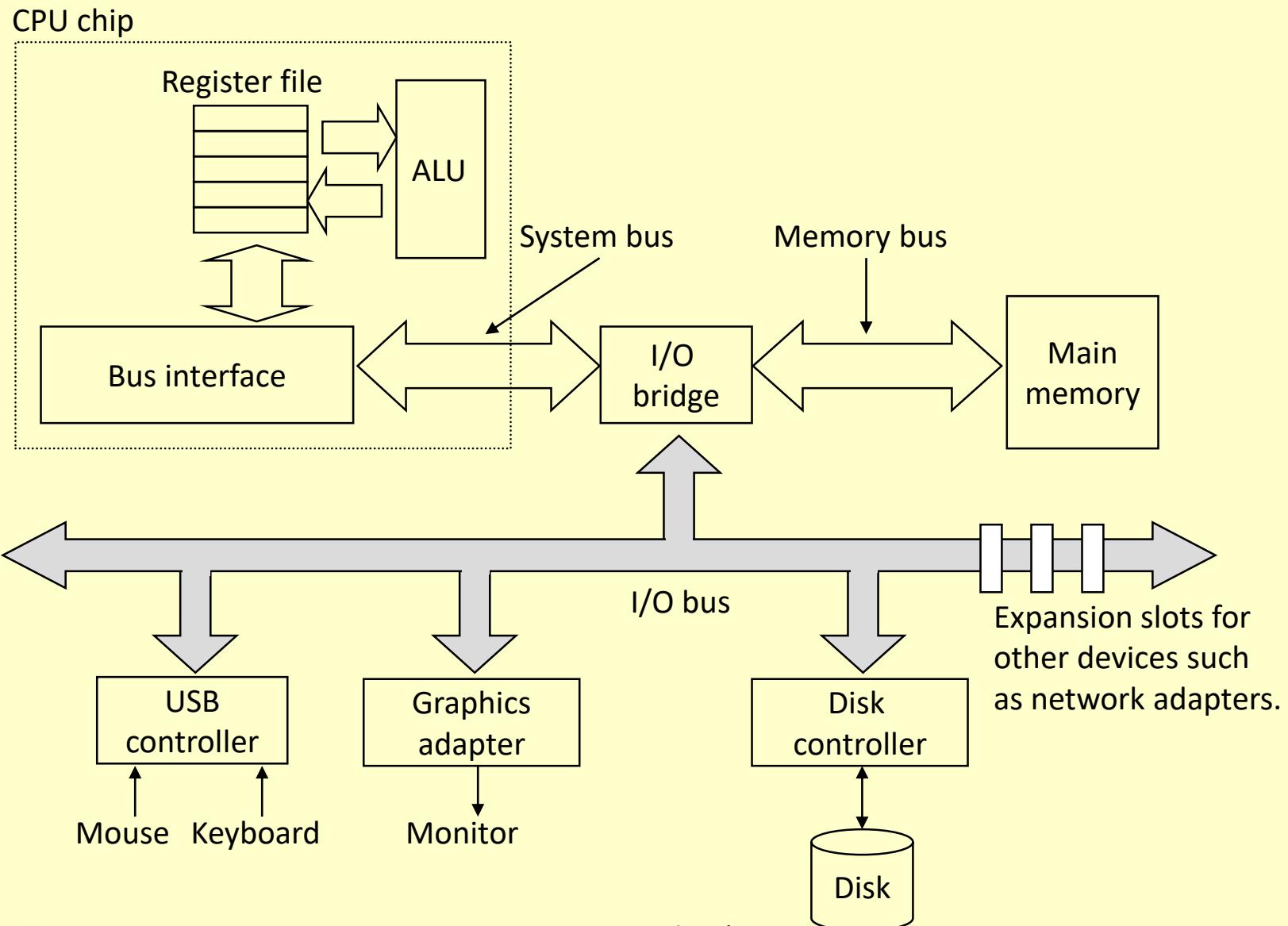
■ Important points:

- Access time dominated by seek time and rotational latency.
- First bit in a sector is the most expensive, the rest are free.
- SRAM access time is about 4 ns/doubleword, DRAM about 60 ns
 - Disk is about 40,000 times slower than SRAM,
 - 2,500 times slower than DRAM.

Logical Disk Blocks

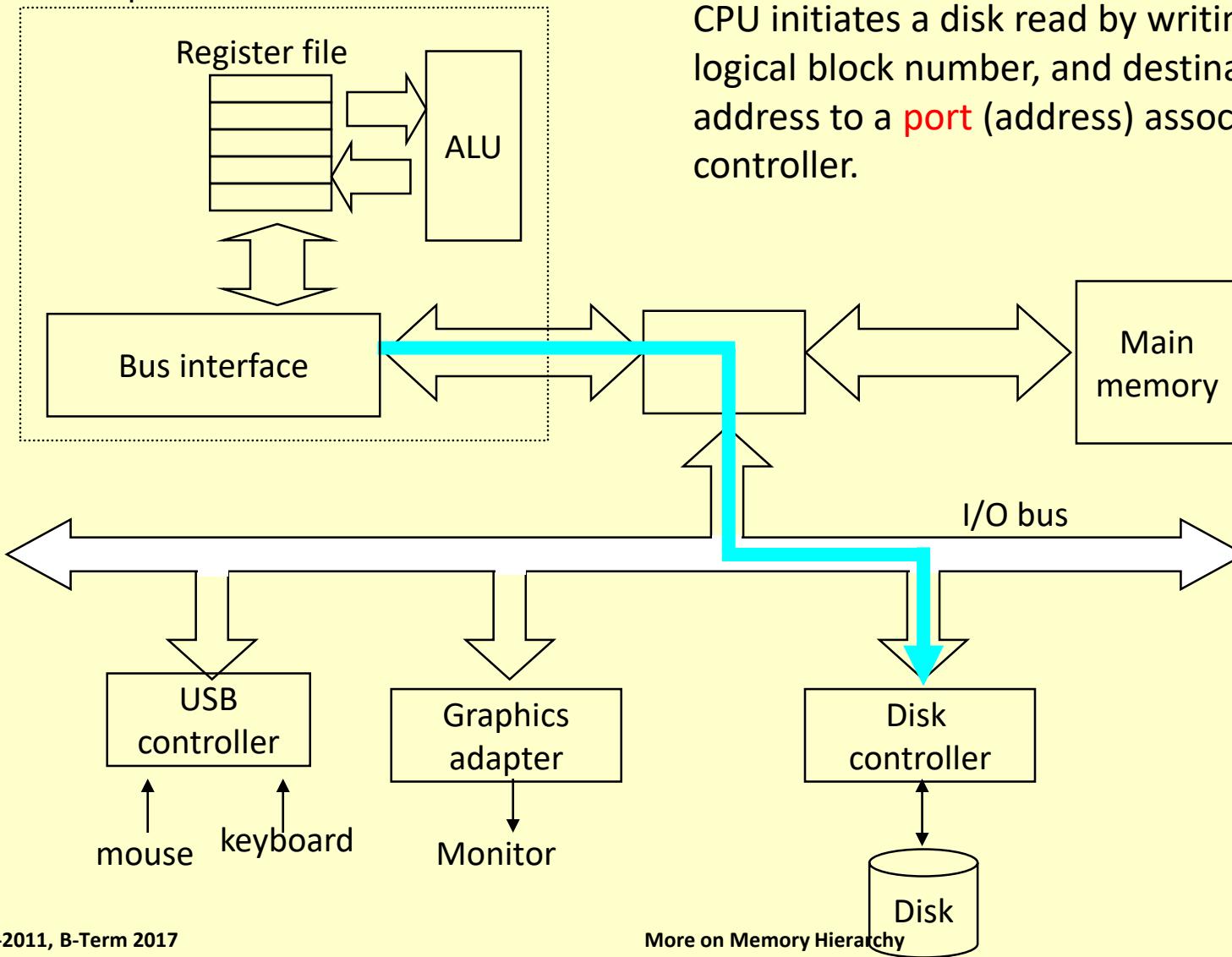
- Modern disks present a simpler abstract view of the complex sector geometry:
 - The set of available sectors is modeled as a sequence of b-sized logical blocks (0, 1, 2, ...)
- Mapping between logical blocks and actual (physical) sectors
 - Maintained by hardware/firmware device called disk controller
 - Converts requests for logical blocks into (surface,track,sector) triples
- Allows controller to set aside spare cylinders for each zone.
 - Accounts for the difference in “formatted capacity” and “maximum capacity”

I/O Bus



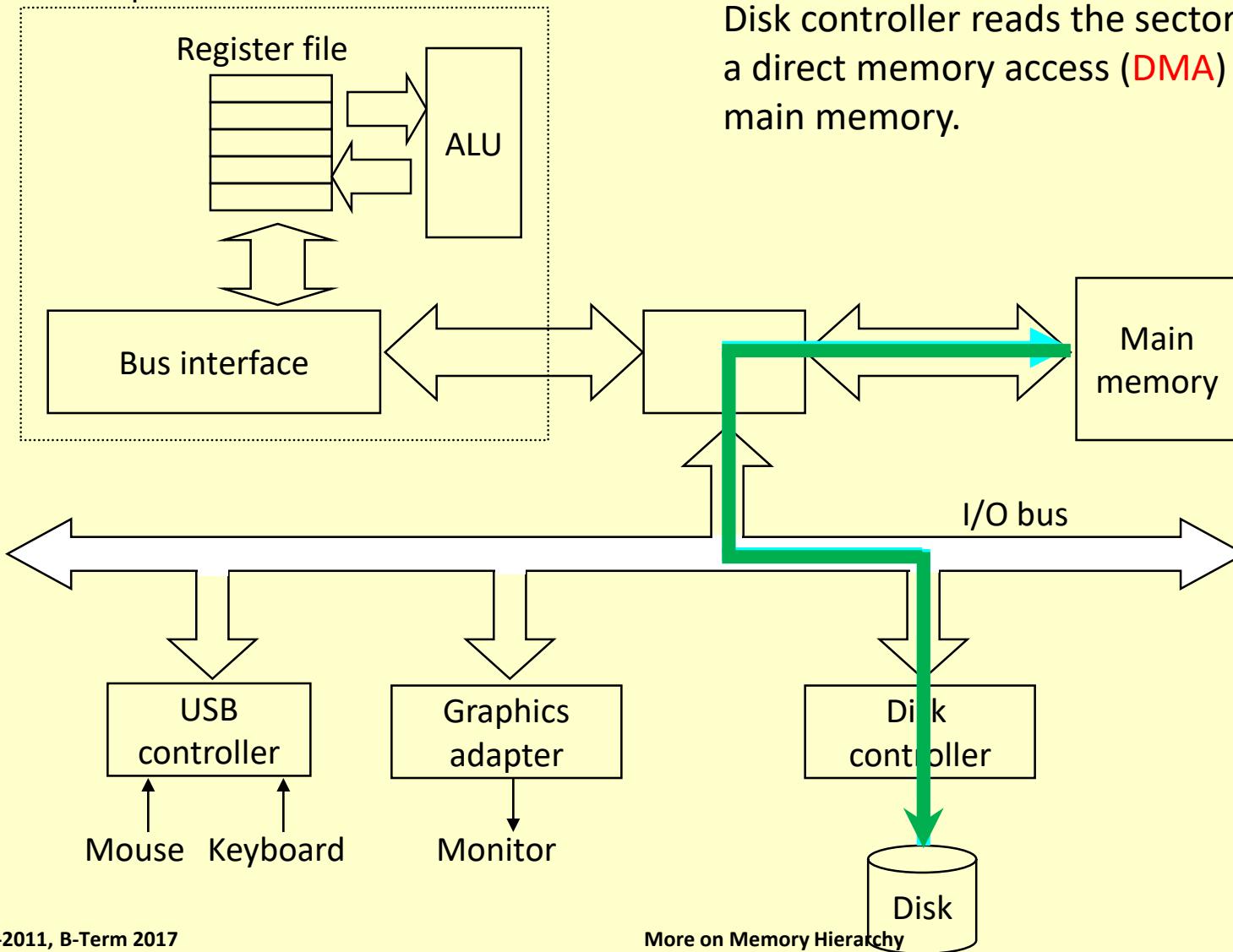
Reading a Disk Sector (1)

CPU chip



Reading a Disk Sector (2)

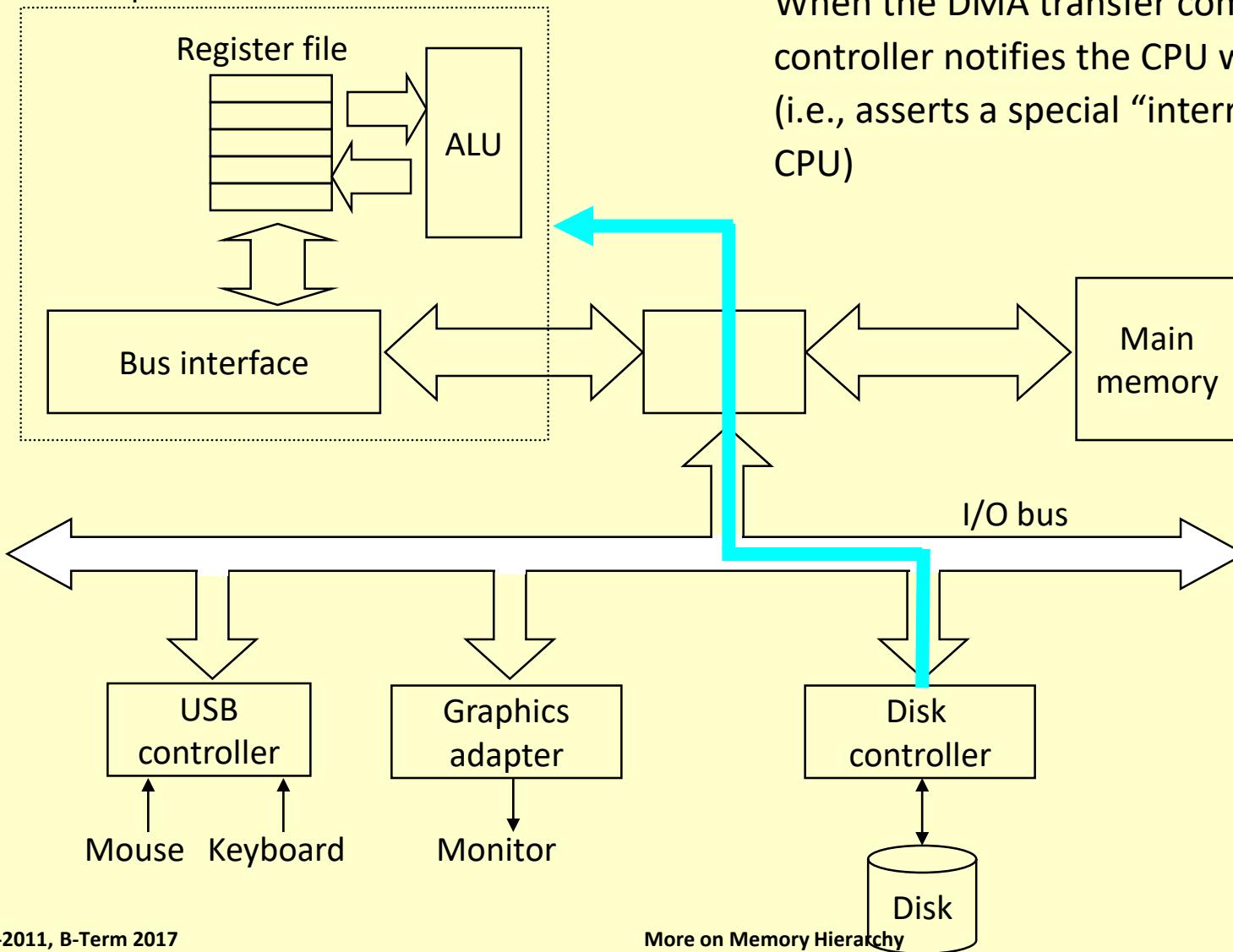
CPU chip



More on Memory Hierarchy

Reading a Disk Sector (3)

CPU chip



Questions?

CS-2011 — Machine Organization and Assembly Language — Recap

Professor Hugh C. Lauer
CS-2011, Machine Organization and Assembly Language

(Slides include copyright materials from *Computer Systems: A Programmer's Perspective*, by Bryant and O'Hallaron, and from *The C Programming Language*, by Kernighan and Ritchie)

Traditional Course in Machine Organization and Assembly Language

- Bits, bytes, gates, logic
 - How the computer works inside
- Von Neumann cycle
 - Instruction fetch and execution
- Machine code and Assembly language
 - Writing out those instructions
- Machine data types
 - Integers, short, long
- A few primitive algorithms
 - Bubblesort in Assembly
- ...

Traditional Course never gets to ...

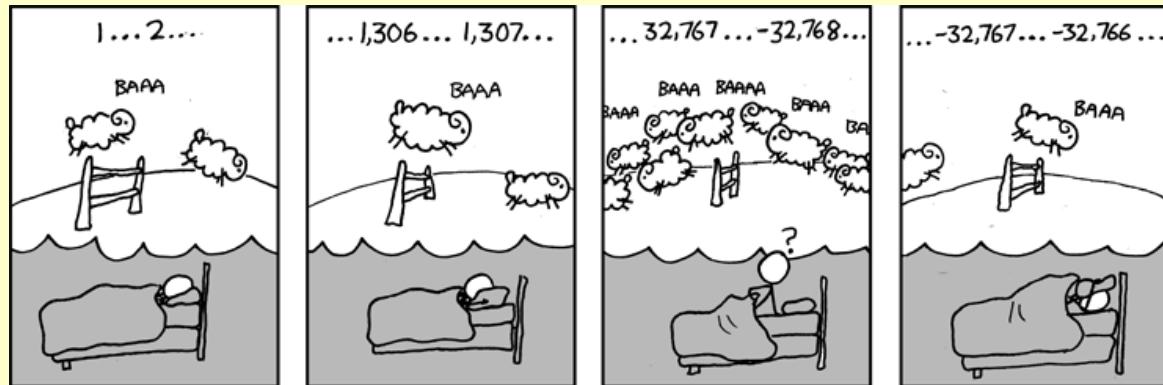
- Bits, bytes, gates, logic
 - How the computer works inside
- Von Neumann cycle
 - Instruction fetch and execution
- Machine code and Assembly language
 - Writing out those instructions
- Machine data types
 - Integers, short, long
- A few primitive algorithms
 - Bubblesort in Assembly
- ...
- Floating point, other data types
 - How computer arithmetic works
- Representation of real programs
 - For-loops, if-else, switches, etc.
 - Functions, stack discipline
 - Parameters and arguments
- Things that matter at runtime ...
 - Memory hierarchy
 - Cache performance
- ... when the abstraction breaks down
 - Buffer overflow

Great Reality #1:

Ints are not integers, floats are not reals

■ Example 1: Is $x^2 \geq 0$?

- Float's: Yes!



- Int's:

- $40000 * 40000 \rightarrow 1,600,000,000$
- $50000 * 50000 \rightarrow ??$

-352,516,352

■ Example 2: Is $(x + y) + z = x + (y + z)$?

- Unsigned & Signed Int's: Yes!

- Float's:

- $(1e20 + -1e20) + 3.14 \rightarrow 3.14$
- $1e20 + (-1e20 + 3.14) \rightarrow ??$

Memory referencing bug example

```
double fun(int i)
{
    volatile double d[1] = {3.14};
    volatile long int a[2];
    a[i] = 1073741824; /* Possibly out of bounds */
    return d[0];
}
```

fun(0) →	3.14
fun(1) →	3.14
fun(2) →	3.1399998664856
fun(3) →	2.00000061035156
fun(4) →	3.14, then segmentation fault

■ Result is architecture specific

Memory system performance example

```
void copyij(int src[2048][2048],  
           int dst[2048][2048])  
{  
    int i,j;  
    for (i = 0; i < 2048; i++)  
        for (j = 0; j < 2048; j++)  
            dst[i][j] = src[i][j];  
}
```

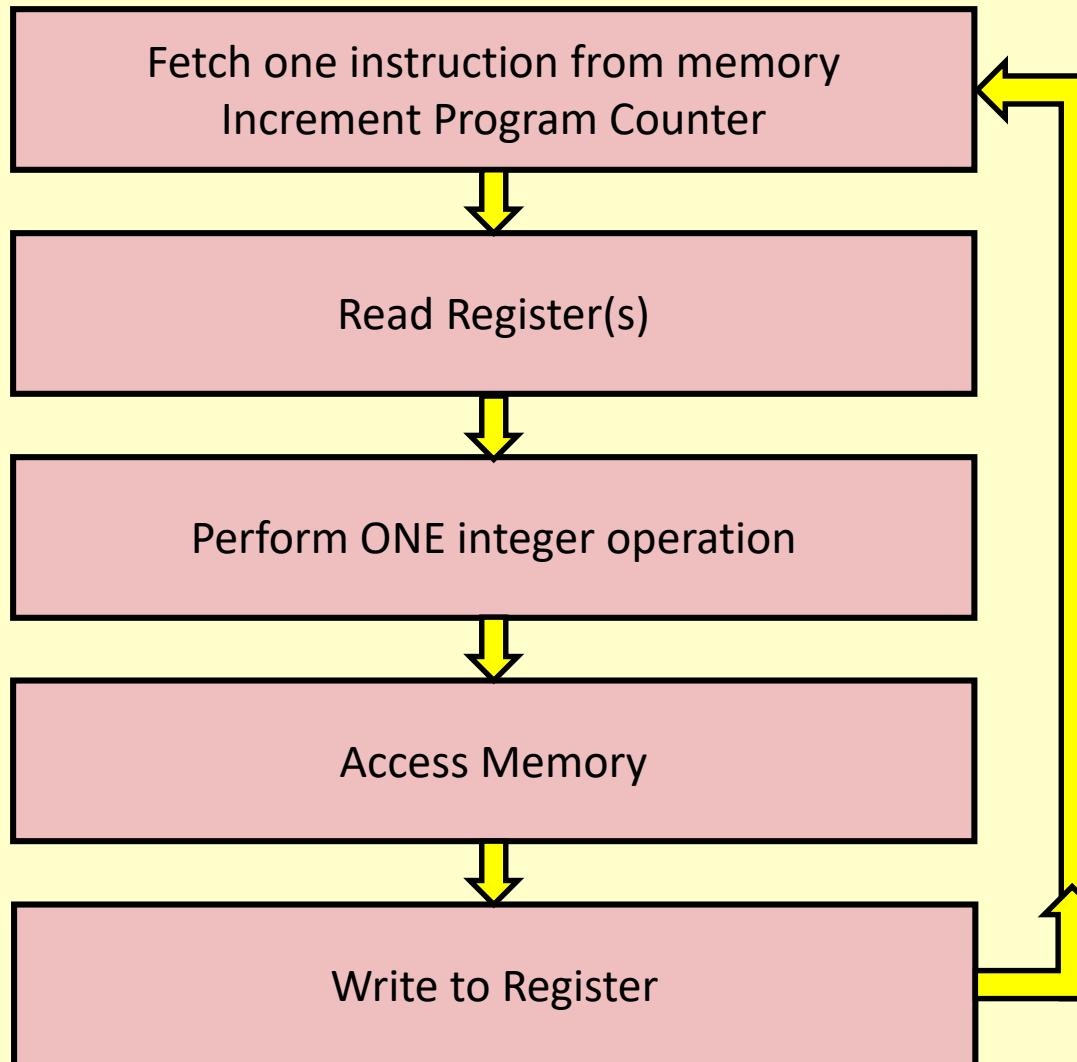


```
void copyji(int src[2048][2048],  
           int dst[2048][2048])  
{  
    int i,j;  
    for (j = 0; j < 2048; j++)  
        for (i = 0; i < 2048; i++)  
            dst[i][j] = src[i][j];  
}
```

21 times slower
(Pentium 4)

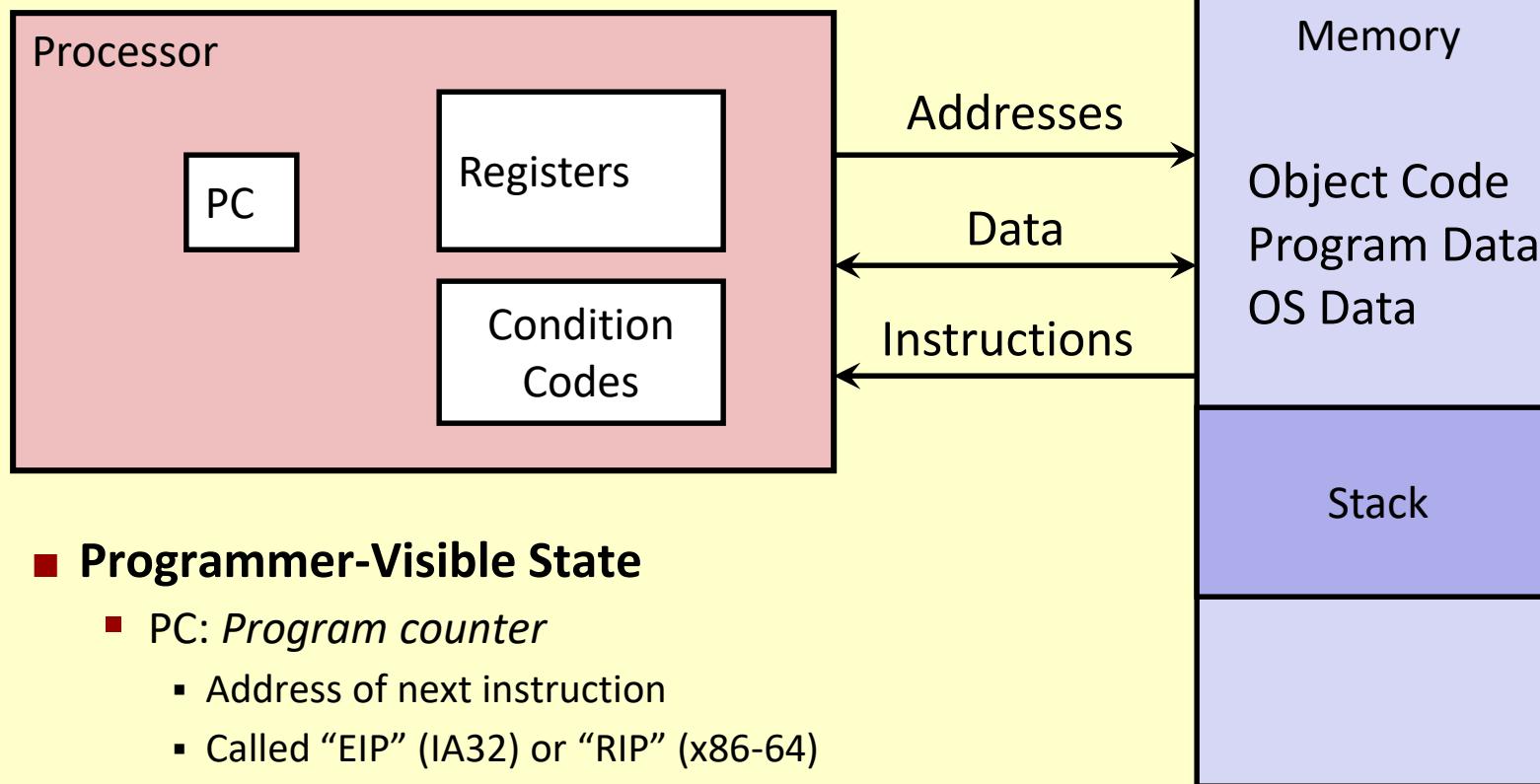
- Hierarchical memory organization
- Performance depends on access patterns
 - Including how step through multi-dimensional array

Execution Model for Modern Computers



Assembly Programmer's View

A carefully crafted illusion!



■ Programmer-Visible State

- PC: *Program counter*
 - Address of next instruction
 - Called “EIP” (IA32) or “RIP” (x86-64)
- Register file
 - Heavily used program data
- Condition codes
 - Store status information about most recent arithmetic operation
 - Used for conditional branching

■ Memory

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

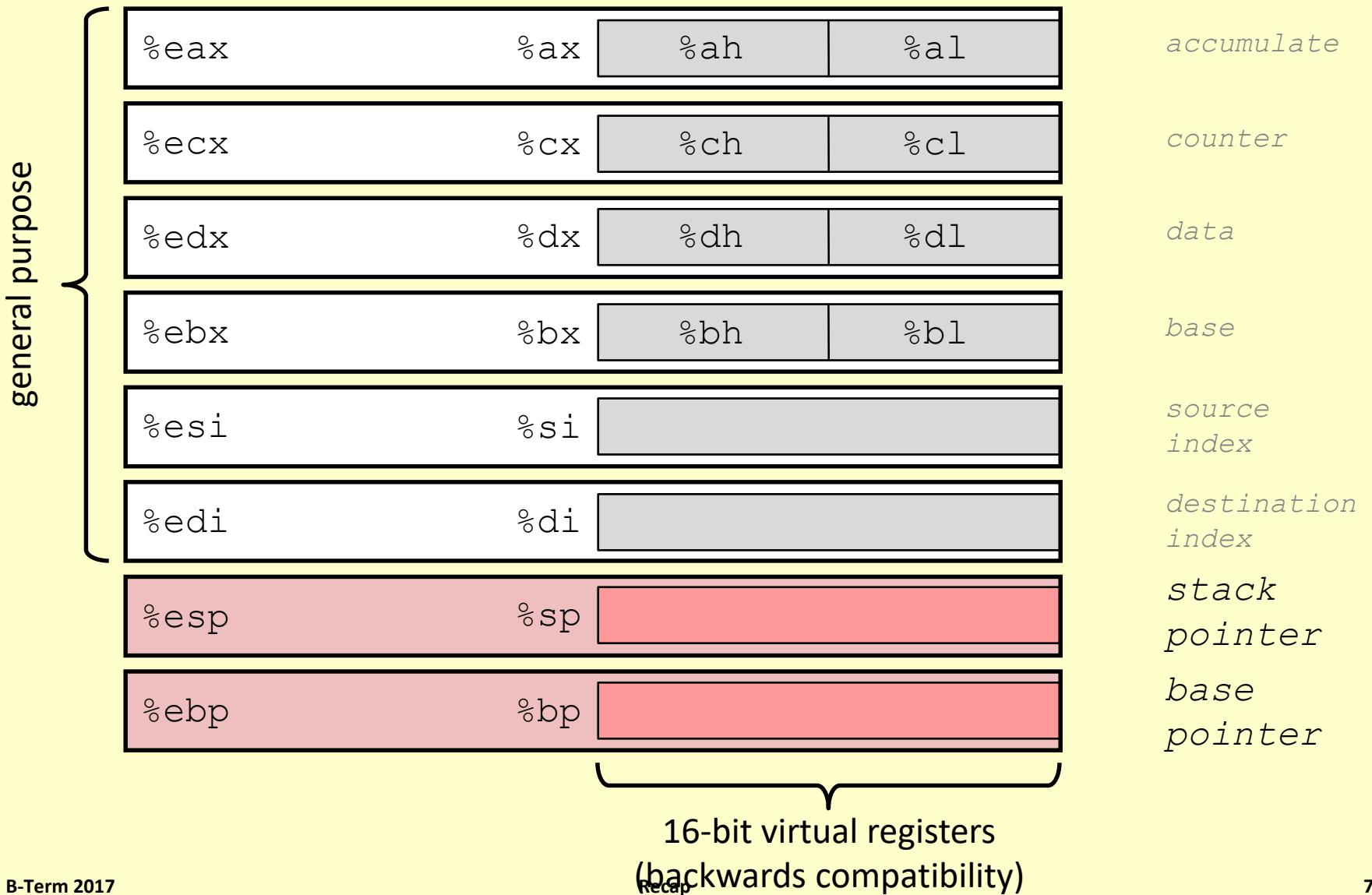
x86-64 Integer Registers

%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

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

Integer Registers (IA32)



32-bit code for swap

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

swap:

pushl %ebp	}	Set Up
movl %esp, %ebp		
pushl %ebx		
movl 8(%ebp), %edx	}	Body
movl 12(%ebp), %ecx		
movl (%edx), %ebx		
movl (%ecx), %eax		
movl %eax, (%edx)		
movl %ebx, (%ecx)		
popl %ebx	}	Finish
popl %ebp		
ret		

64-bit code for swap

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

swap:

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

} Set Up
{} Body
{} Finish

■ Operands passed in registers (why useful?)

- First (**xp**) in **%rdi**, second (**yp**) in **%rsi**
- 64-bit pointers

■ No stack operations required

■ 32-bit data

- Data held in registers **%eax** and **%edx**
- **movl** operation



“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;  
}
```

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

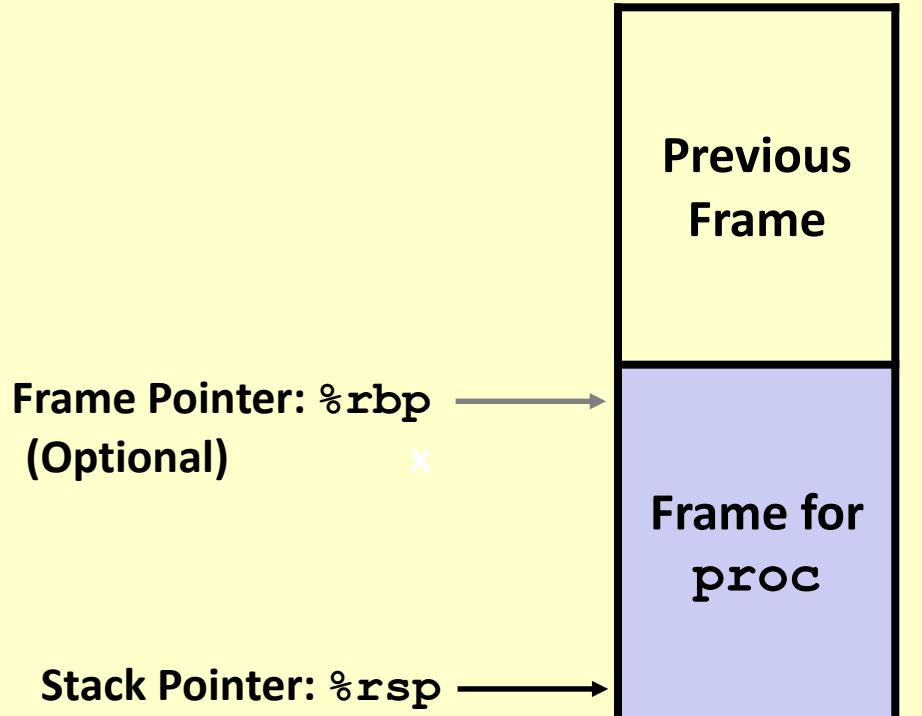
- **Procedure return: `ret`**

- Pop address from stack
 - Jump to address

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

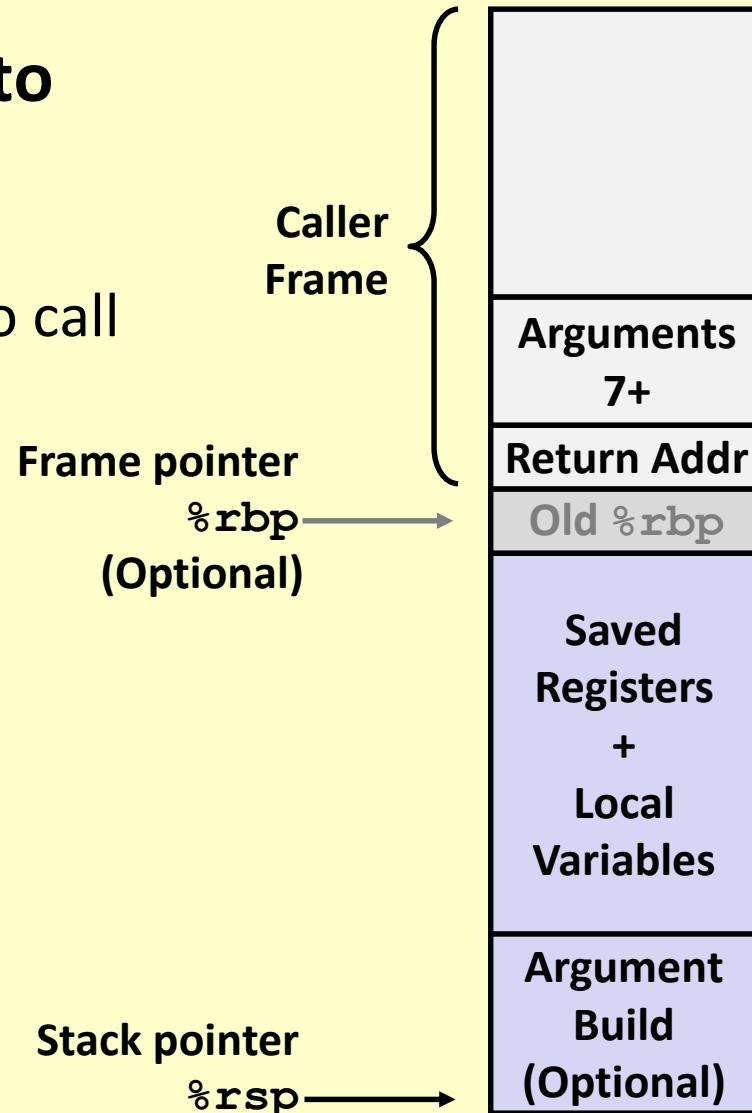
Stack “Top”

See also: Fig 3.25

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

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()`

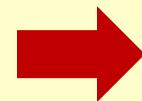
■ Data

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

■ Text / Shared Libraries

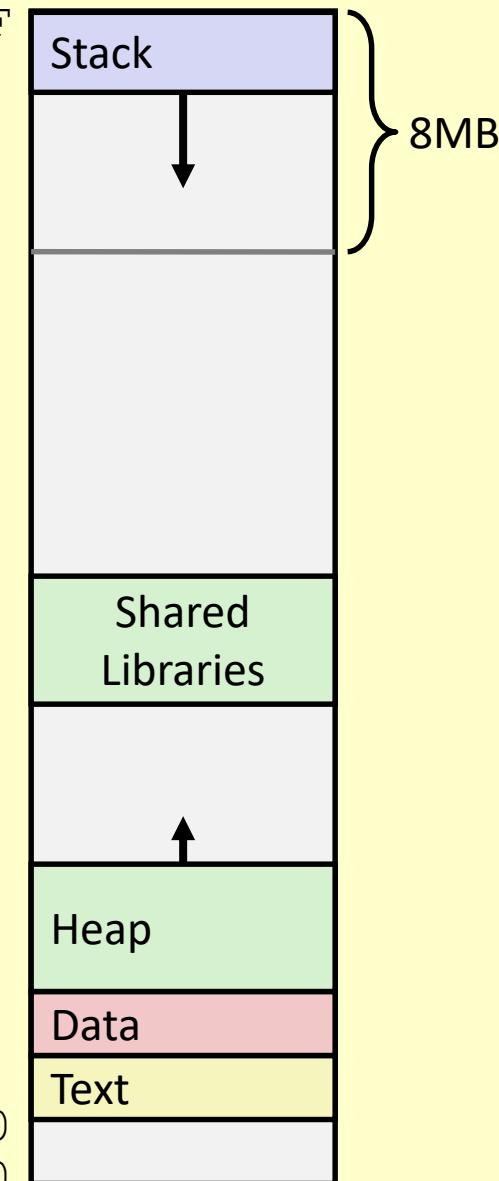
- Executable machine instructions
- Read-only

Hex Address



400000
000000

Recap



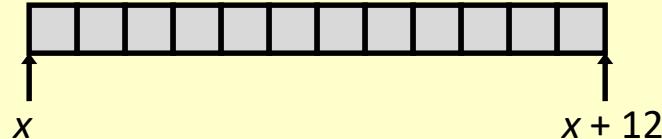
Array Allocation

■ Basic Principle

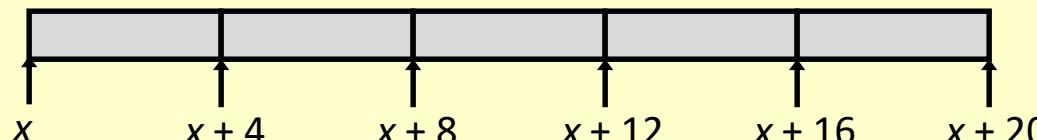
$T \ A[L] ;$

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

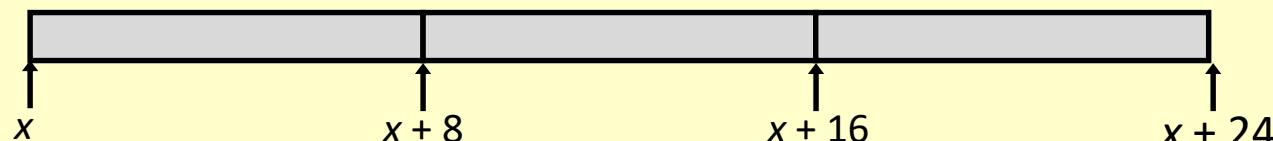
`char string[12];`



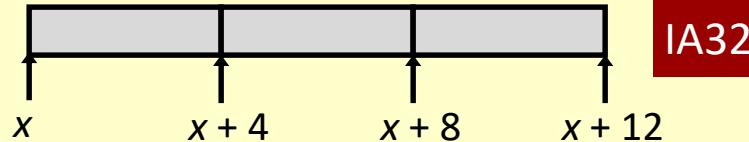
`int val[5];`



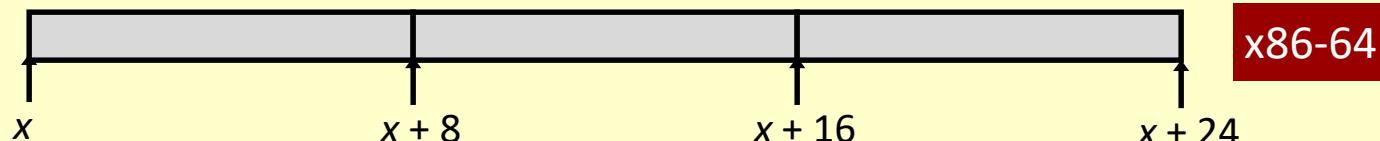
`double a[3];`



`char *p[3];`



`char *q[3];`



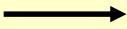
Reduction in Strength

- Replace costly operation with simpler one
- Shift, add instead of multiply or divide

$16*x \rightarrow x << 4$

- Utility machine dependent
 - Depends on cost of multiply or divide instruction
 - On Intel Nehalem, integer multiply requires 3 CPU cycles
- Recognize sequence of products

```
for (i = 0; i < n; i++) {  
    int ni = n*i;  
    for (j = 0; j < n; j++)  
        a[ni + j] = b[j];  
}
```



```
int ni = 0;  
for (i = 0; i < n; i++) {  
    for (j = 0; j < n; j++)  
        a[ni + j] = b[j];  
    ni += n;  
}
```

Share Common Subexpressions

- Reuse portions of expressions
- GCC will do this with -O1

```
/* Sum neighbors of i,j */
up = val[(i-1)*n + j];
down = val[(i+1)*n + j];
left = val[i*n      + j-1];
right = val[i*n     + j+1];
sum = up + down + left + right;
```

3 multiplications: $i \cdot n$, $(i-1) \cdot n$, $(i+1) \cdot n$

```
long inj = i*n + j;
up = val[inj - n];
down = val[inj + n];
left = val[inj - 1];
right = val[inj + 1];
sum = up + down + left + right;
```

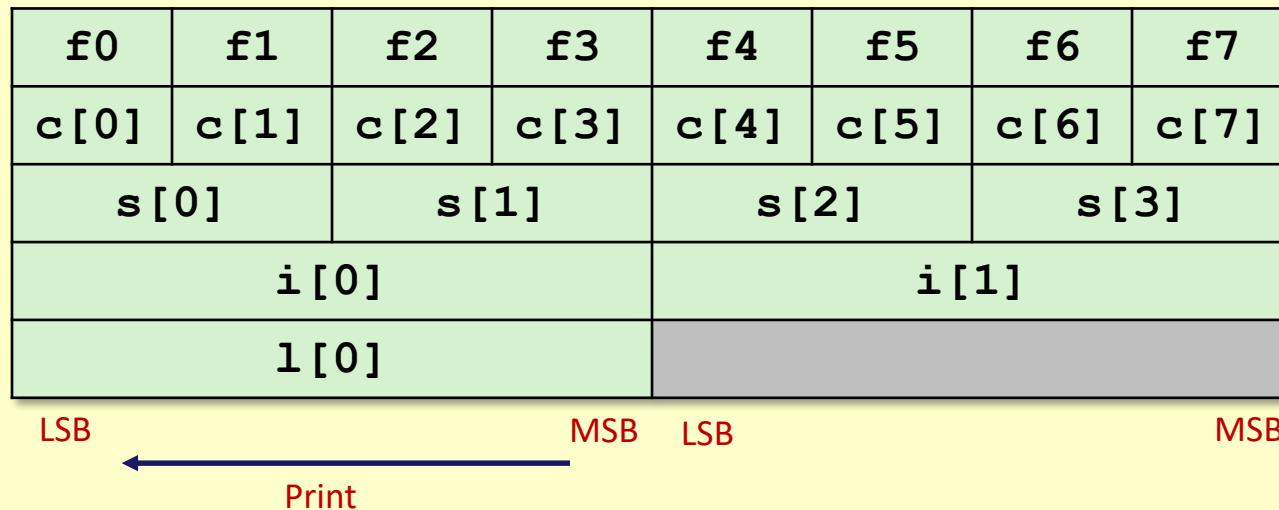
1 multiplication: $i \cdot n$

```
leaq    1(%rsi), %rax   # i+1
leaq    -1(%rsi), %r8    # i-1
imulq   %rcx, %rsi      # i*n
imulq   %rcx, %rax      # (i+1)*n
imulq   %rcx, %r8      # (i-1)*n
addq    %rdx, %rsi      # i*n+j
addq    %rdx, %rax      # (i+1)*n+j
addq    %rdx, %r8      # (i-1)*n+j
```

```
imulq   %rcx, %rsi      # i*n
addq    %rdx, %rsi      # i*n+j
movq    %rsi, %rax      # i*n+j
subq    %rcx, %rax      # i*n+j-n
leaq    (%rsi,%rcx), %rcx # i*n+j+n
```

Byte Ordering on IA32

Little Endian

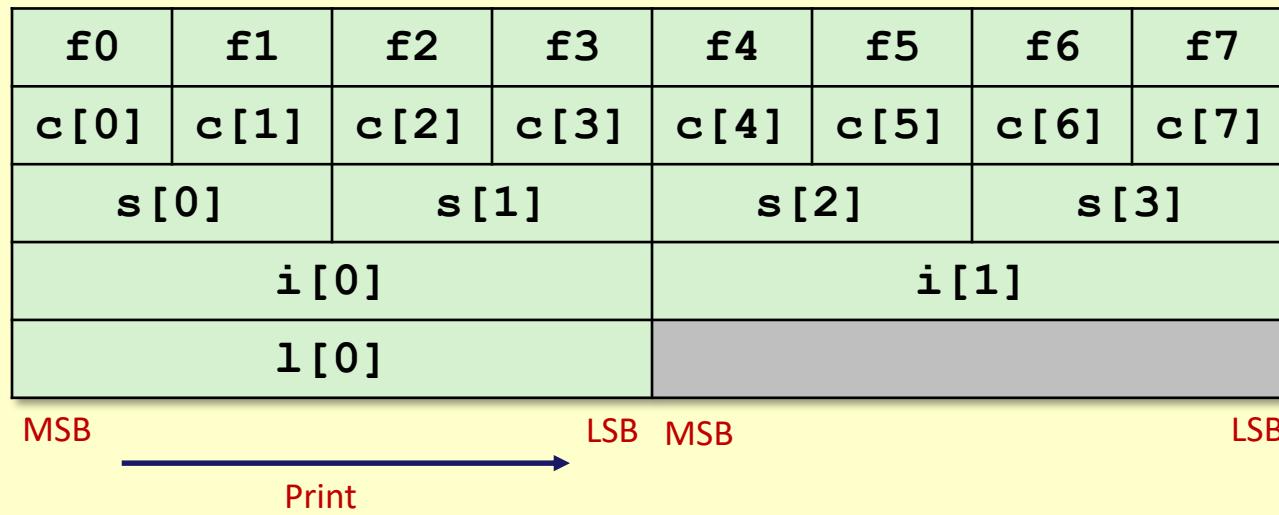


Output:

```
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

Big Endian



Output on SPARC/IBM, etc.:

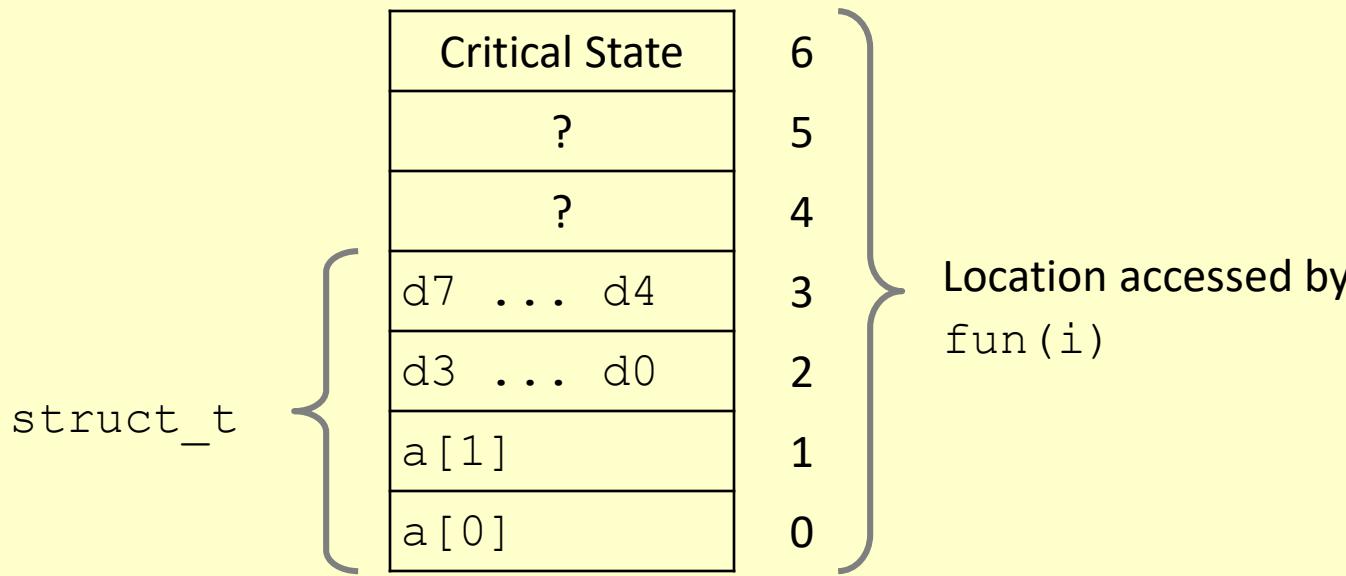
Characters	0-7	==	[0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7]
Shorts	0-3	==	[0xf0f1, 0xf2f3, 0xf4f5, 0xf6f7]
Ints	0-1	==	[0xf0f1f2f3, 0xf4f5f6f7]
Long	0	==	[0xf0f1f2f3]

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

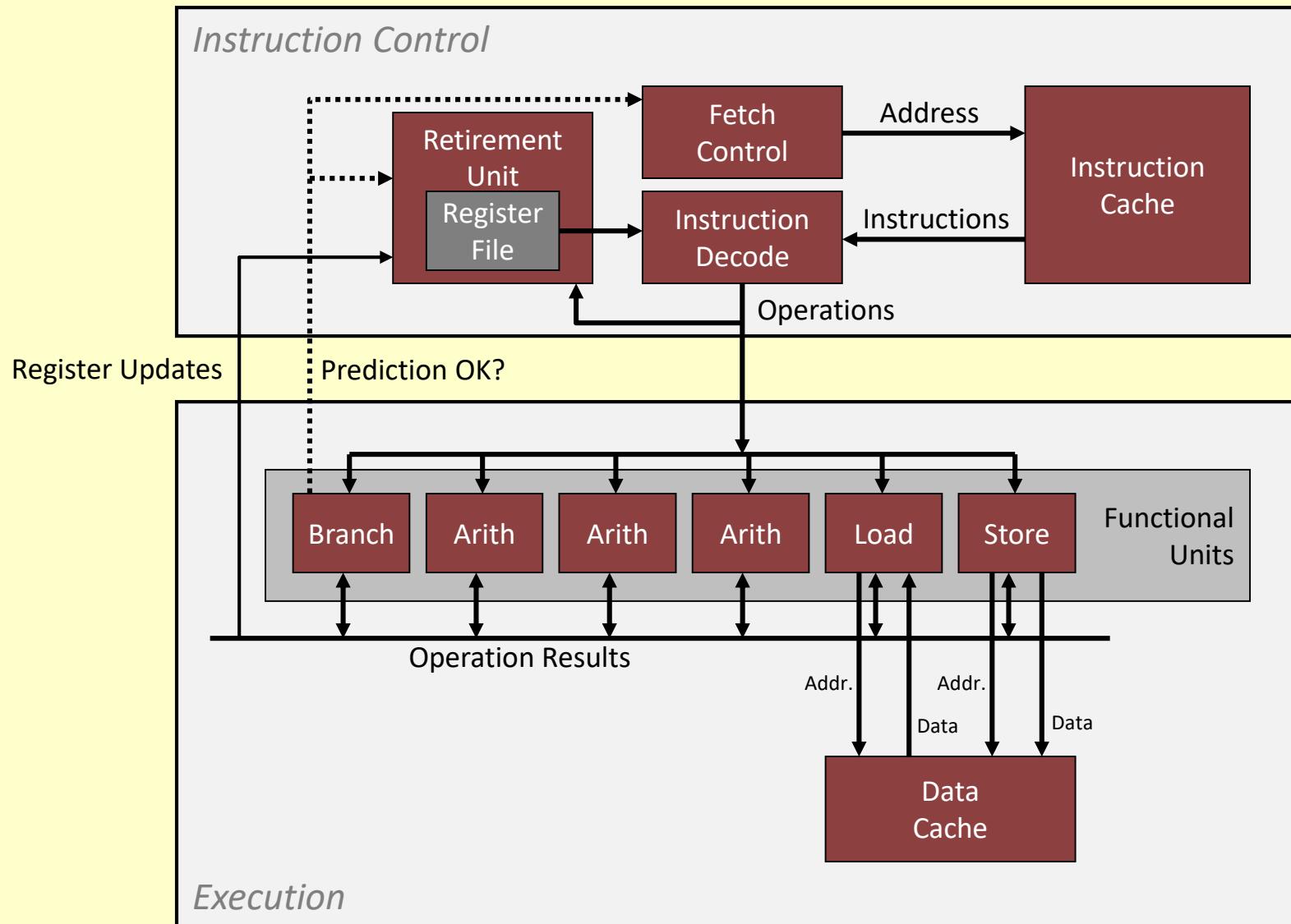
Explanation:



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

Modern CPU Design



2015 State of the Art

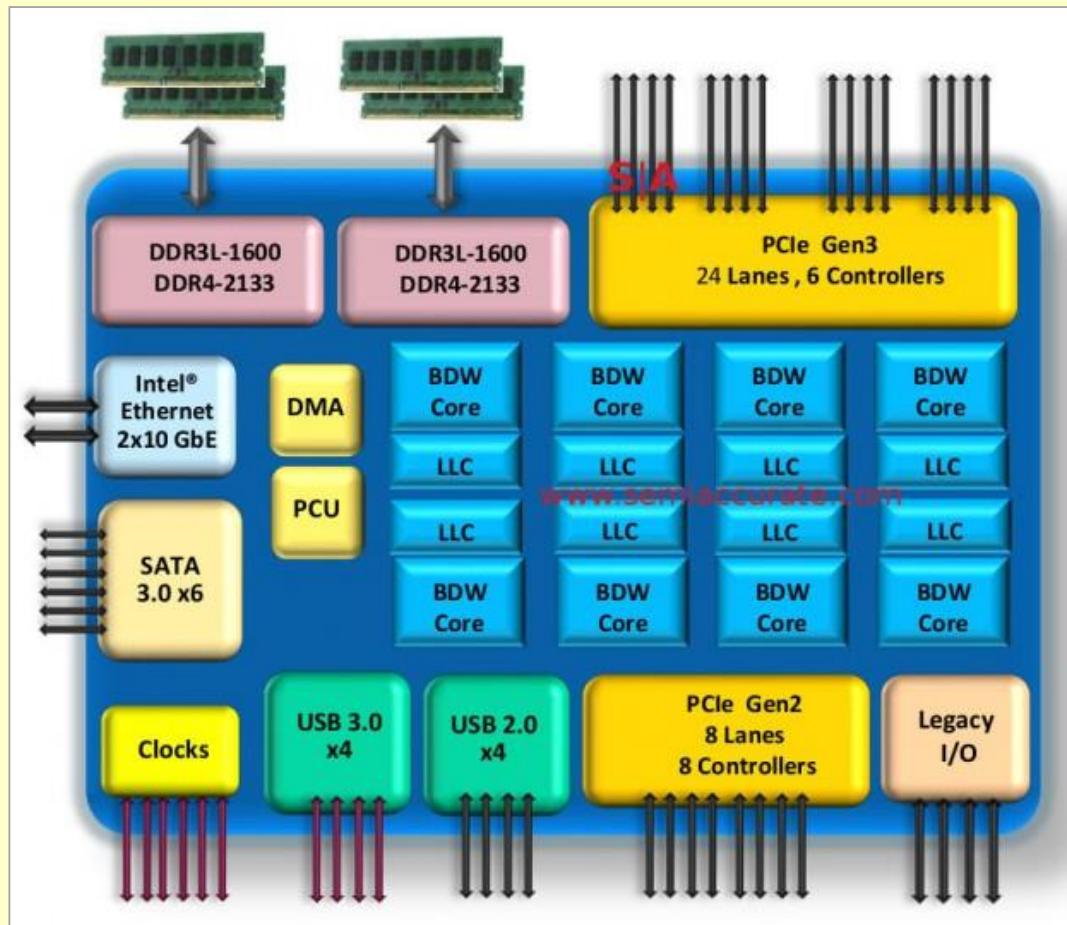
- Core i7 Broadwell 2015

■ Desktop Model

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

■ Server Model

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



The Memory Mountain

Aggressive prefetching

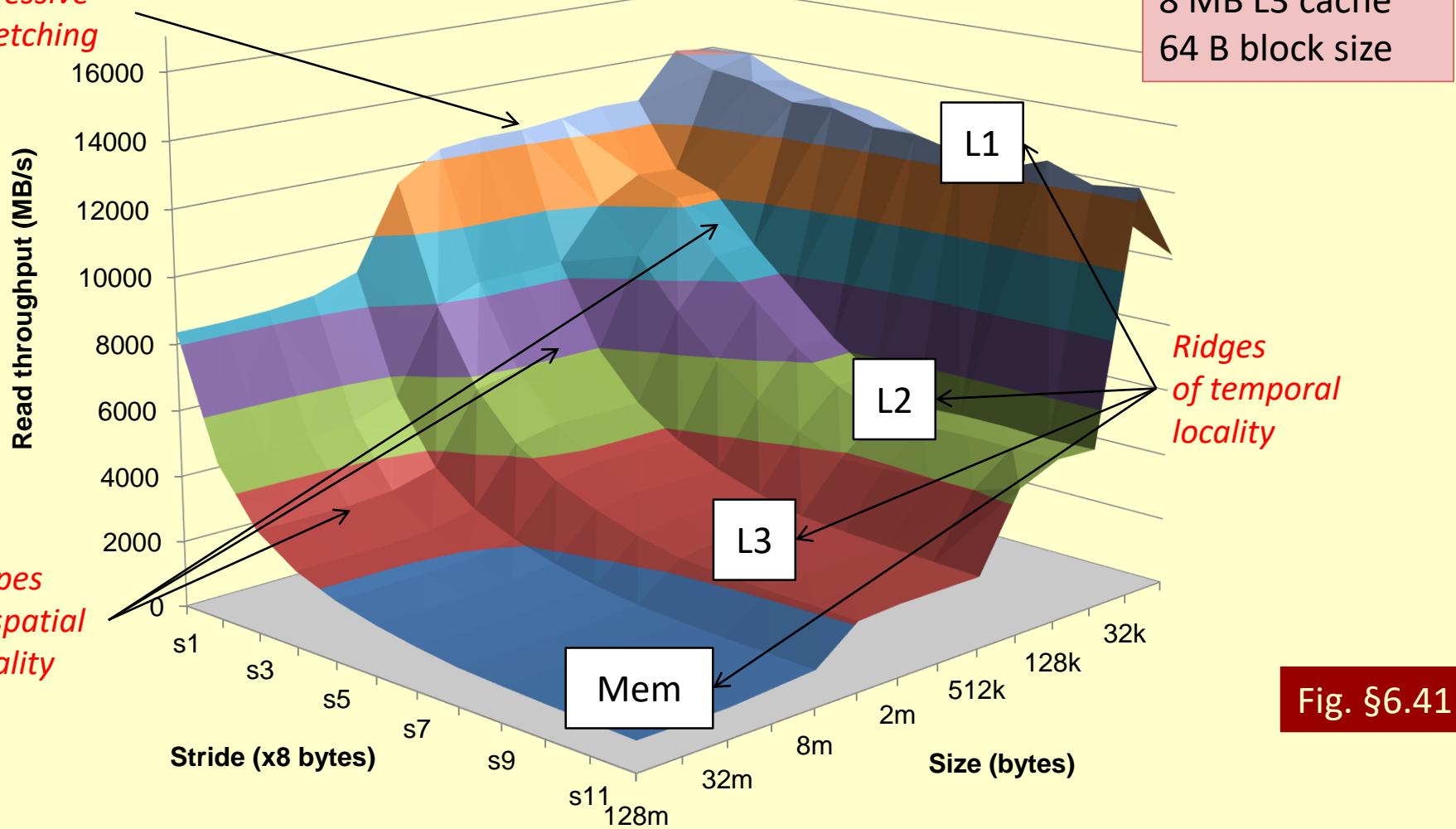


Fig. §6.41

Much more to computers ...

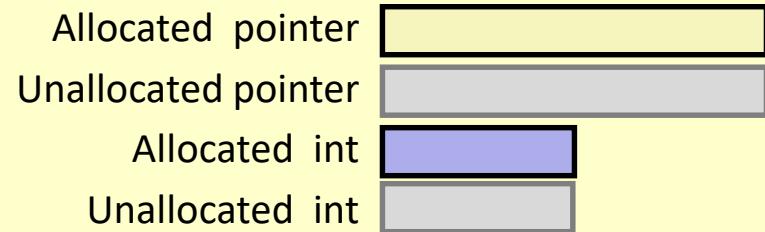
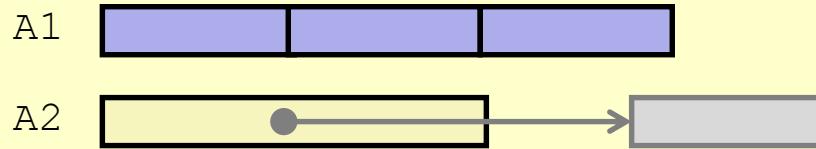
- ... than you ever expected
- Can make an entire career out of them ...
- ... or simply buy them and use them!

**Thank you for your interest and
attention**

Questions?

Understanding Pointers & Arrays #1

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



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

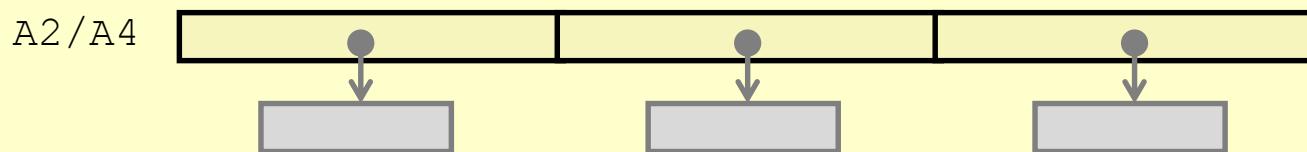
Understanding Pointers & Arrays #2

Decl	An			*An			**An		
	Cmp	Bad	Size	Cmp	Bad	Size	Cmp	Bad	Size
int A1[3]									
int *A2[3]									
int (*A3)[3]									
int (*A4[3])									

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

Understanding Pointers & Arrays #2

Decl	An			*An			**An		
	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



Allocated pointer



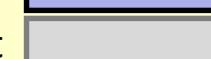
Unallocated pointer



Allocated int



Unallocated int

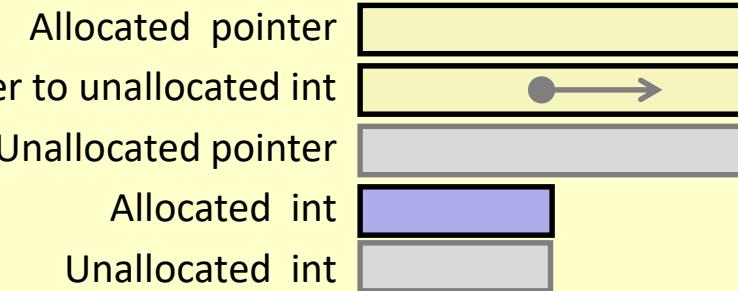


Understanding Pointers & Arrays #3

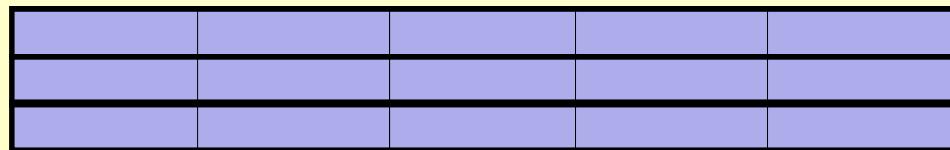
Decl	A_n			*A_n			**A_n		
	Cm p	Bad	Size	Cm p	Bad	Size	Cm p	Bad	Size
<code>int A1[3][5]</code>									
<code>int *A2[3][5]</code>									
<code>int (*A3)[3][5]</code>									
<code>int *(A4[3][5])</code>									
<code>int (*A5[3])[5]</code>									

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

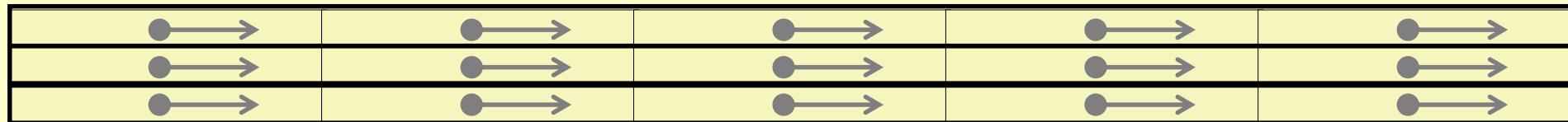
Decl	***A_n		
	Cm p	Bad	Size
<code>int A1[3][5]</code>			
<code>int *A2[3][5]</code>			
<code>int (*A3)[3][5]</code>			
<code>int *(A4[3][5])</code>			
<code>int (*A5[3])[5]</code>			



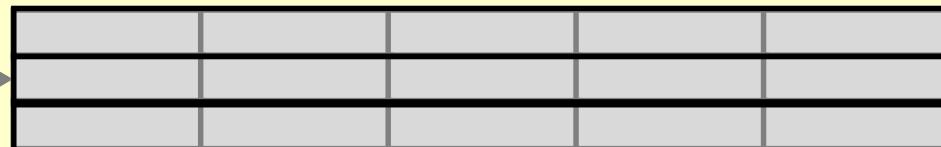
A1



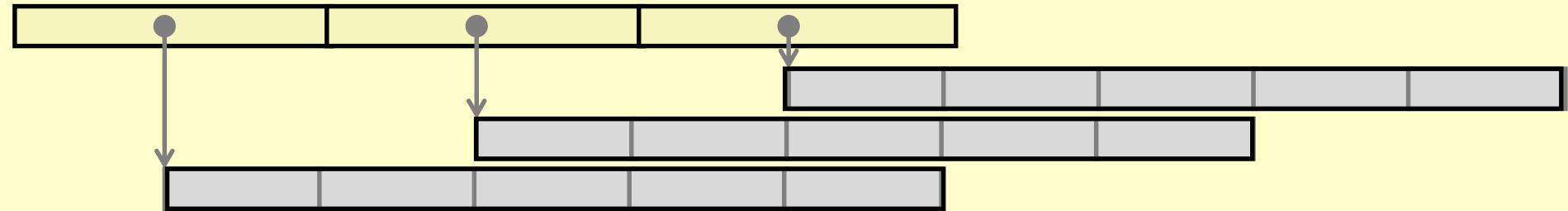
A2/A4



A3



A5



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

Questions?