Announcements

- VM on the website!
- Speedometer!

- Use anonymous feedback link
- HW0, having fun?
- Use discussion boards!
- Check if office hours work for you, let us know if they don't.
- Make sure you are subscribed to the mailing lists.
  - If you enrolled recently, you might not be on it

Roadmap

C:

```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
    c.getMPG();
```

Assembly language:

```
get_mpg:
    pushq   %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret
```

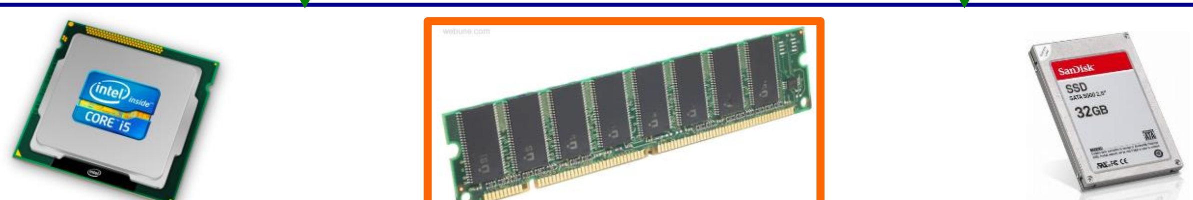
Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

OS:

Windows 8 Mac

Computer system:

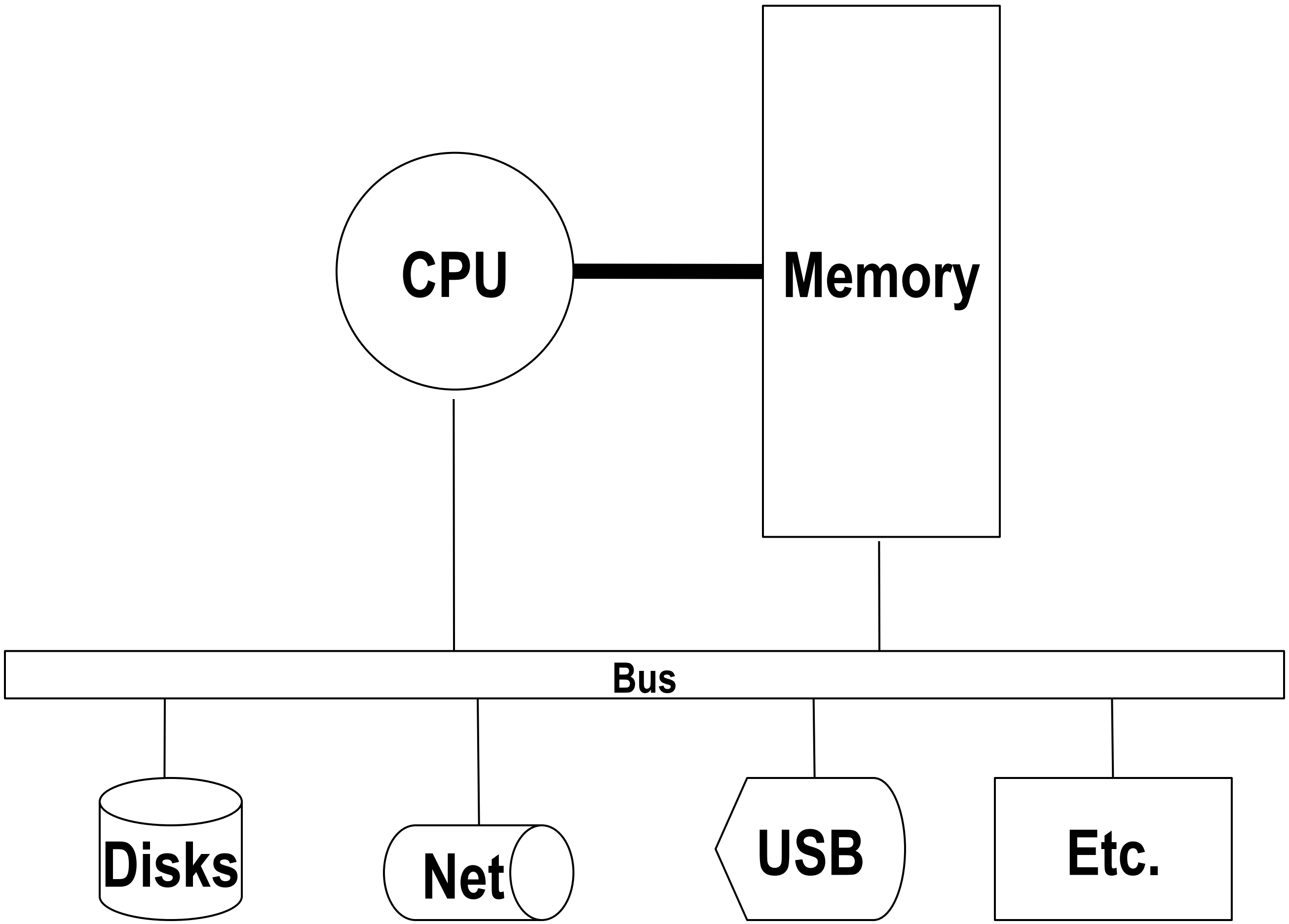


Data & addressing
Integers & floats
Machine code & C
x86 assembly programming
Procedures & stacks
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C

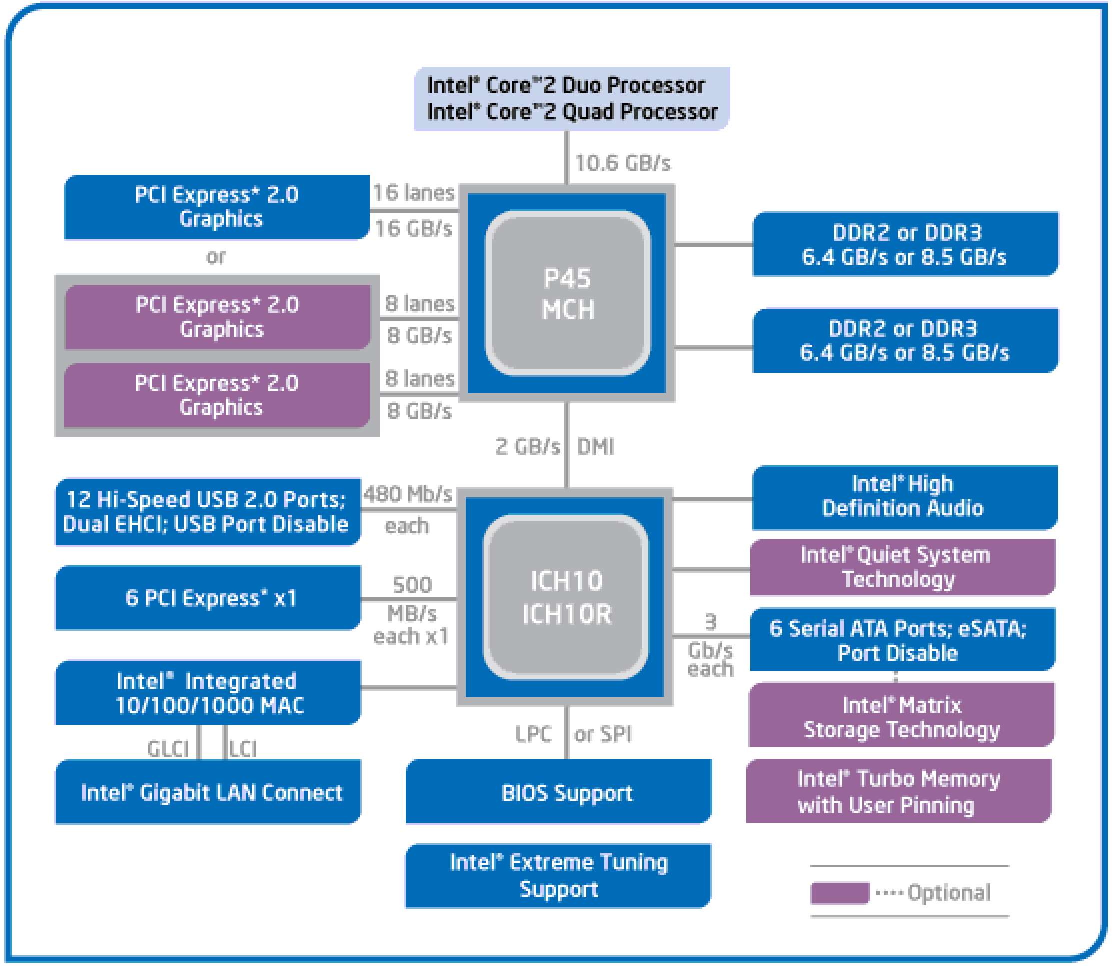
Today's (and Friday's) topics

- Representing information as bits and bytes
- Organizing and addressing data in memory
- Manipulating data in memory using C
- Boolean algebra and bit-level manipulations

Hardware: Logical View

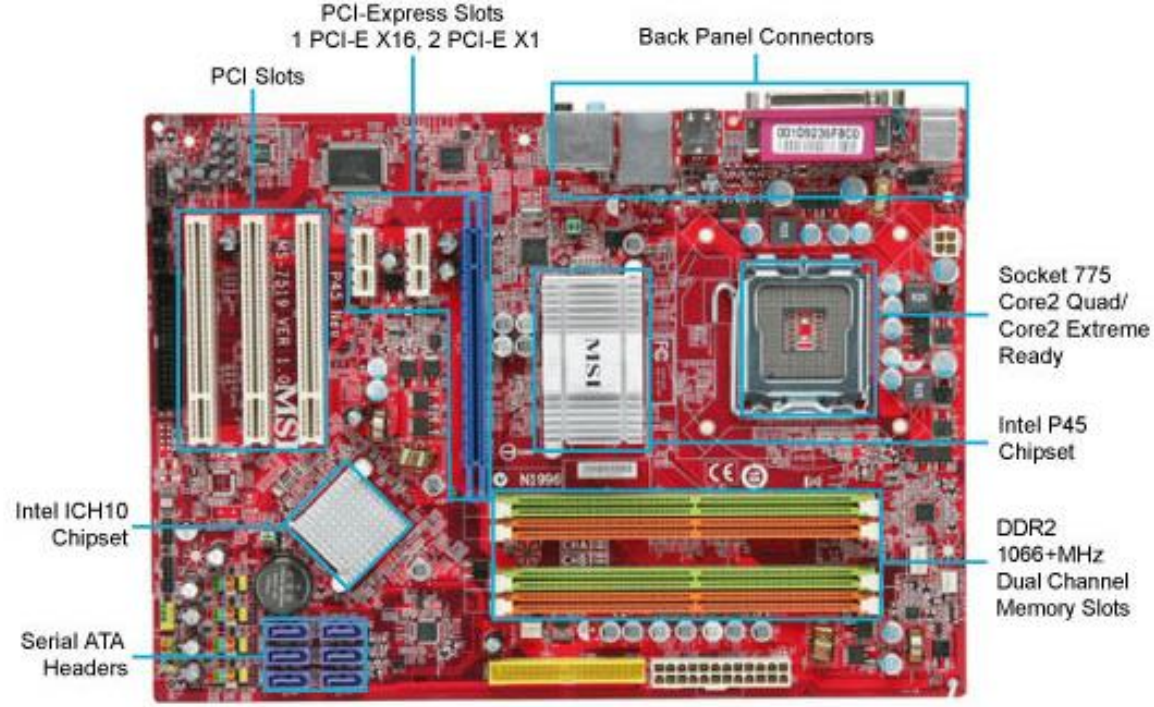


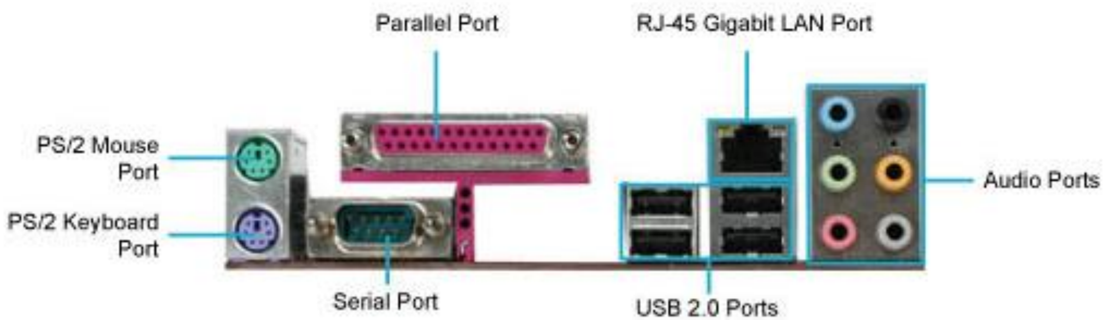
Hardware: Semi-Logical View



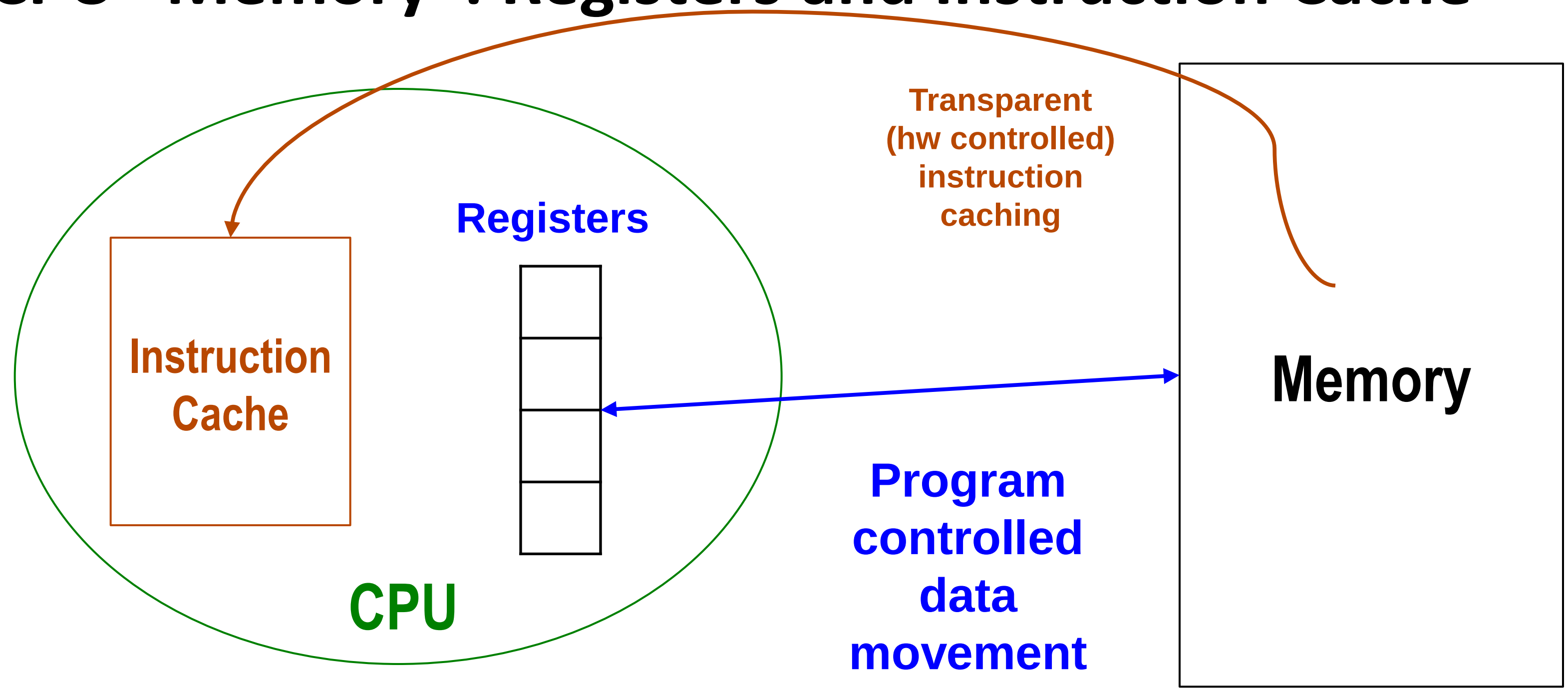
Intel® P45 Express Chipset Block Diagram

Hardware: Physical View





CPU “Memory”: Registers and Instruction Cache



- **There are a fixed number of <u>registers</u> on the CPU**
  - Registers hold data
- **There is an <u>I-cache</u> on the CPU holding recently fetched instructions**
  - If you execute a loop that fits in the cache, the CPU goes to memory for those instructions only once, then executes out of its cache
- ***This slide is just an introduction. We'll see a more full explanation later in the course.***

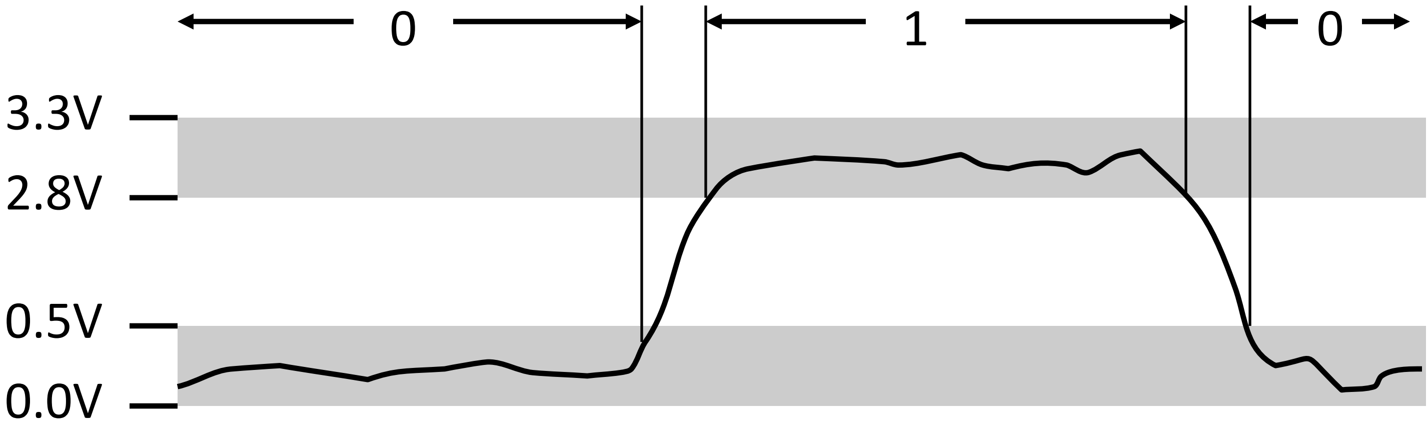
Performance: It's Not Just CPU Speed

- **Data and instructions reside in memory**
  - To execute an instruction, it must be fetched into the CPU
  - Next, the data the instruction operates on must be fetched into the CPU
- **CPU ⍰ Memory bandwidth can limit performance**
  - Improving performance 1: hardware improvements to increase memory bandwidth (e.g., DDR → DDR2 → DDR3)
  - Improving performance 2: move less data into/out of the CPU
    - Put some “memory” in the CPU chip itself (this is “cache” memory)

Binary Representations

- **Base 2 number representation**
  - Represent 351_{10} as 0000000101011111_2 or 101011111_2

- **Electronic implementation**
  - Easy to store with bi-stable elements
  - Reliably transmitted on noisy and inaccurate wires



Encoding Byte Values

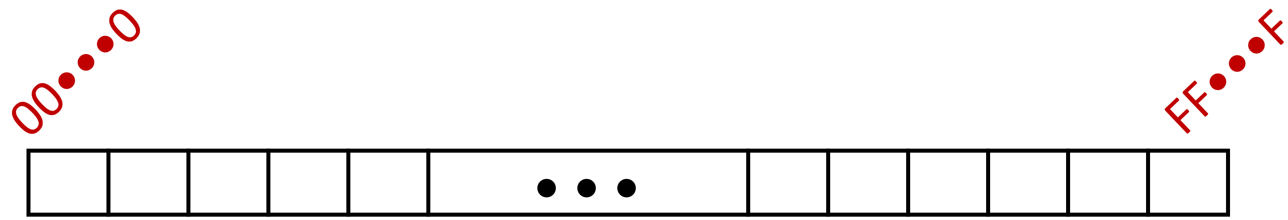
- **Binary** **00000000_2 -- 11111111_2**
  - Byte = 8 bits (binary digits)
- **Decimal** **0_{10} -- 255_{10}**
- **Hexadecimal** **00_{16} -- FF_{16}**
  - Byte = 2 hexadecimal (hex) or base 16 digits
  - Base-16 number representation
  - Use characters ‘0’ to ‘9’ and ‘A’ to ‘F’
  - Write $FA1D37B_{16}$ in C
    - as `0xFA1D37B` or `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

What is memory, really?

- How do we find data in memory?

Byte-Oriented Memory Organization



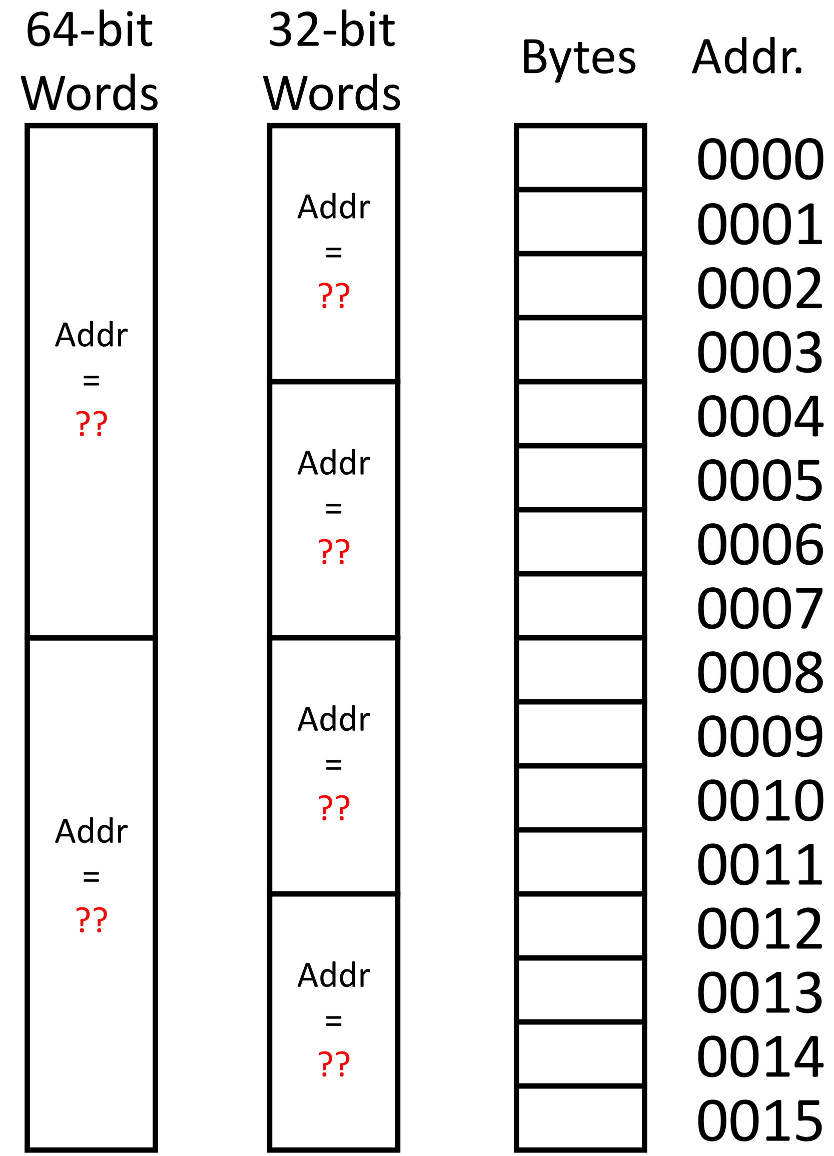
- **Programs refer to addresses**
  - Conceptually, a very large array of bytes, each with an *address* (index)
  - Operating system provides an address space private to each "process"
    - Process = program being executed + its data + its "state"
    - Program can modify its own data, but not that of others
    - Clobbering code or "state" often leads to crashes (or security holes)
- **Compiler + run-time system control memory allocation**
  - Where different program objects should be stored
  - All allocation within a single address space

Machine Words

- **Machine has a "word size"**
  - Nominal size of integer-valued data
    - *Including addresses*
  - Until recently, most machines used 32 bit (4 byte) words
    - Limits addresses to 4GB
    - Became too small for memory-intensive applications
  - Most current x86 systems use 64 bit (8 byte) words
    - Potential address space: $2^{64} \approx 1.8 \times 10^{19}$ bytes (18 EB – exabytes)
  - Machines support multiple data formats
    - Fractions or multiples of word size
    - Always a power-of-2 number of bytes: 1, 2, 4, 8, …

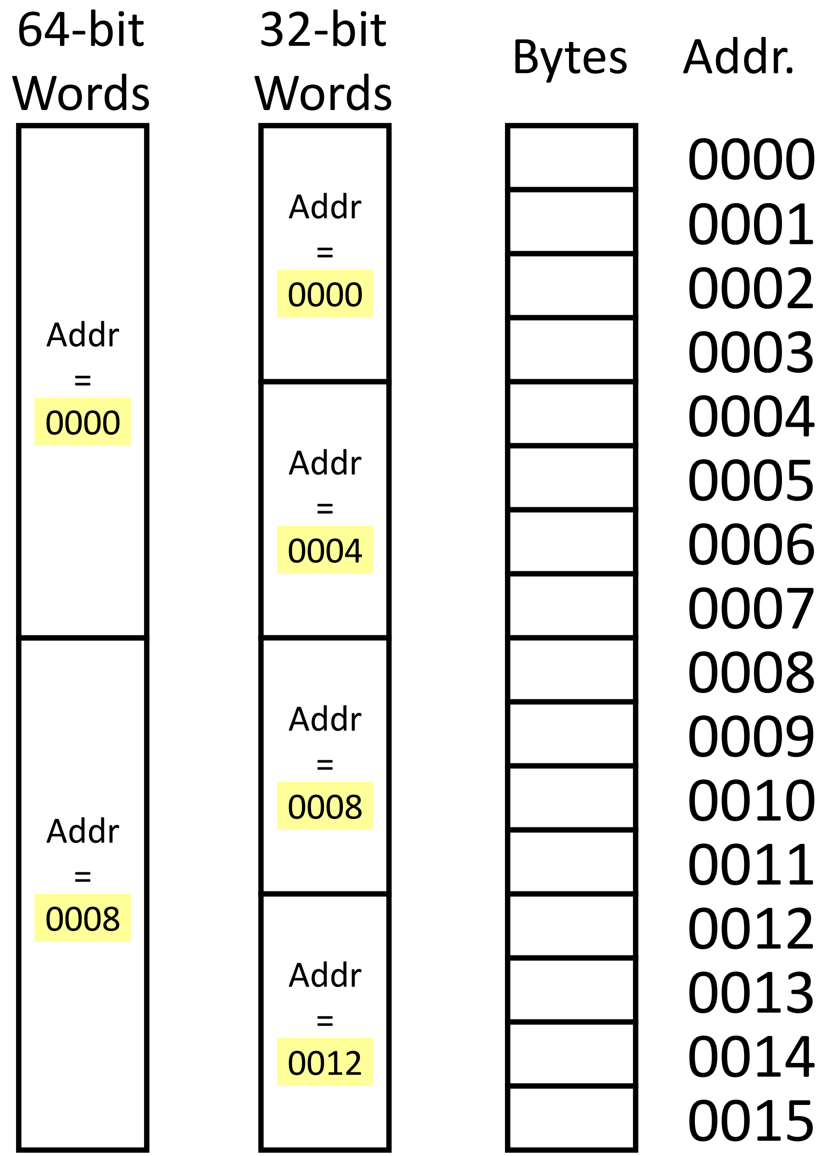
Word-Oriented Memory Organization

- **Addresses specify locations of bytes in memory**
  - Address of first byte in word
  - Addresses of successive words differ by 4 (32-bit) or 8 (64-bit)
  - Address of word 0, 1, .. 10?



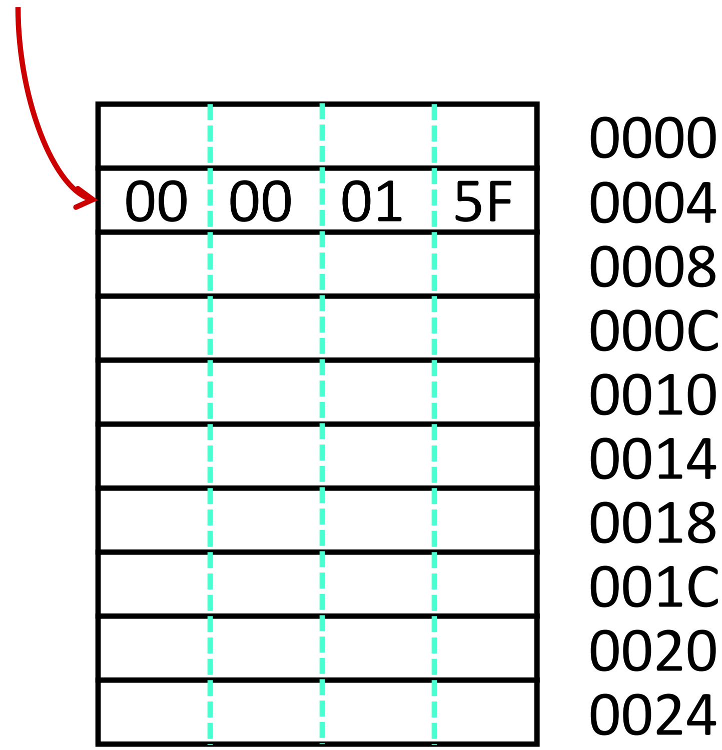
Word-Oriented Memory Organization

- **Addresses specify locations of bytes in memory**
  - Address of first byte in word
  - Addresses of successive words differ by 4 (32-bit) or 8 (64-bit)
  - Address of word 0, 1, .. 10?



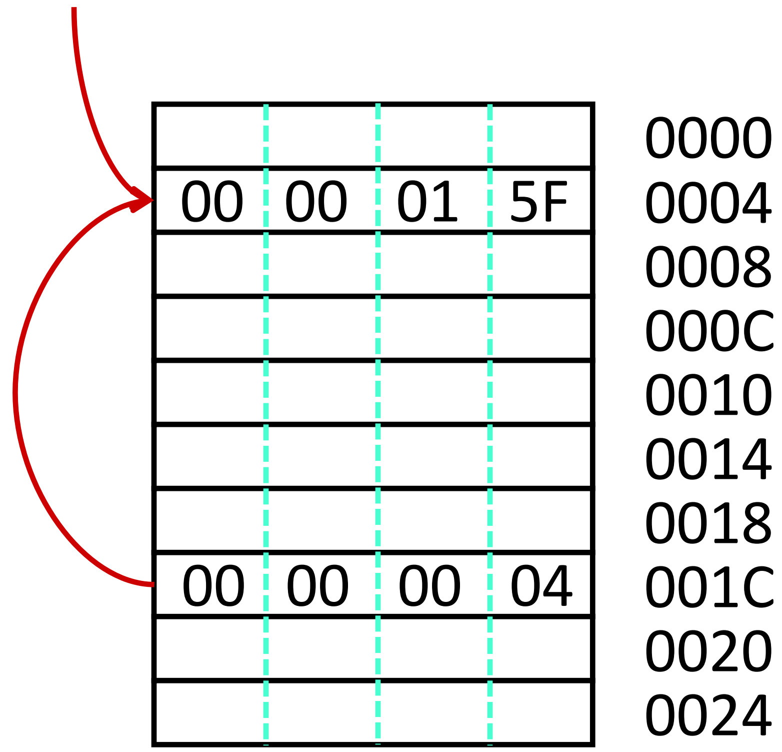
Addresses and Pointers

- Address is a *location* in memory
- Pointer is a data object
  that *contains an address*
- Address 0004
  stores the value 351 (or $15F_{16}$)



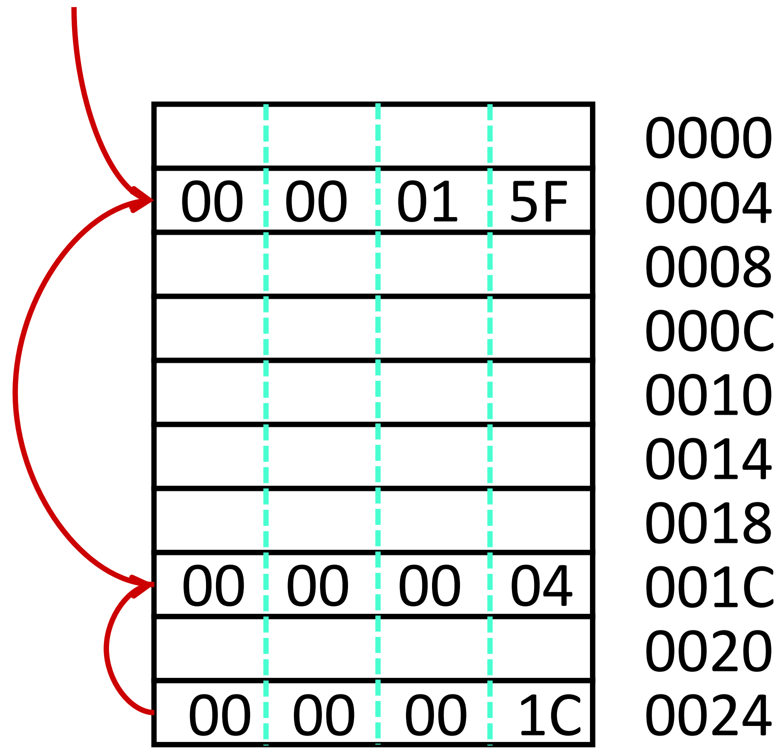
Addresses and Pointers

- Address is a *location* in memory
- Pointer is a data object
  that *contains an address*
- Address 0004
  stores the value 351 (or $15F_{16}$)
- Pointer to address 0004
  stored at address 001C



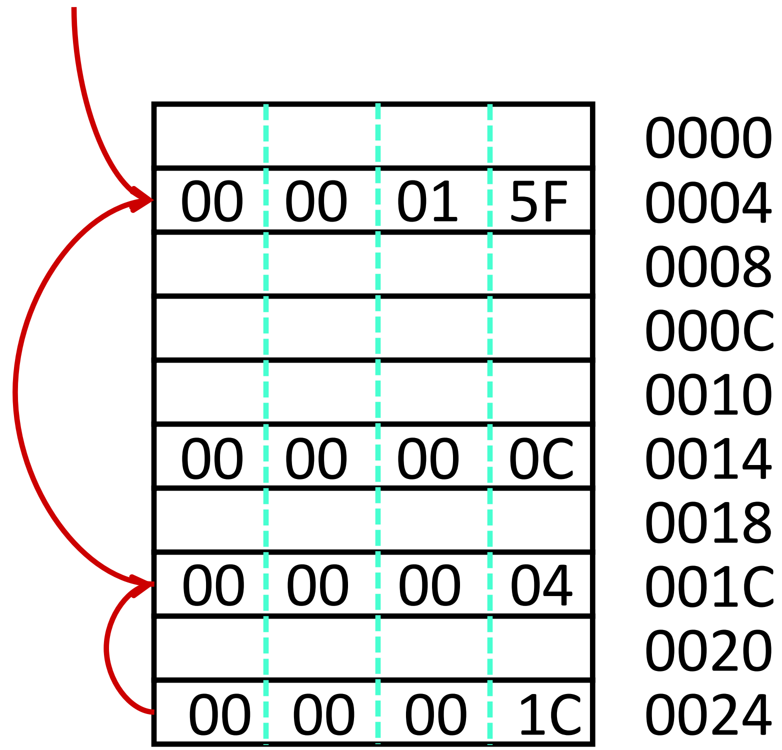
Addresses and Pointers

- Address is a *location* in memory
- Pointer is a data object
  that *contains an address*
- Address 0004
  stores the value 351 (or $15F_{16}$)
- Pointer to address 0004
  stored at address 001C
- Pointer to a pointer
  in 0024



Addresses and Pointers

- **Address is a *location* in memory**
- **Pointer is a data object that *contains an address***
- **Address 0004 stores the value 351 (or $15F_{16}$)**
- **Pointer to address 0004 stored at address 001C**
- **Pointer to a pointer in 0024**
- **Address 0014 stores the value 12**
  - Is it a pointer?



Data Representations

- **Sizes of objects (in bytes)**

Java Data Type	C Data Type	Typical 32-bit	x86-64
boolean	*bool*	1	1
byte	char	1	1
char		2	2
short	short int	2	2
int	int	4	4
float	float	4	4
	long int	4	8
double	double	8	8
long	long long	8	8
	long double	8	16
(reference)	pointer *	4	8

Byte Ordering

- **How should bytes within multi-byte word be ordered in memory?**
  - Peanut butter or chocolate first?
- **Say you want to store the 4-byte word 0xaabbccdd**
  - What order will the bytes be stored?

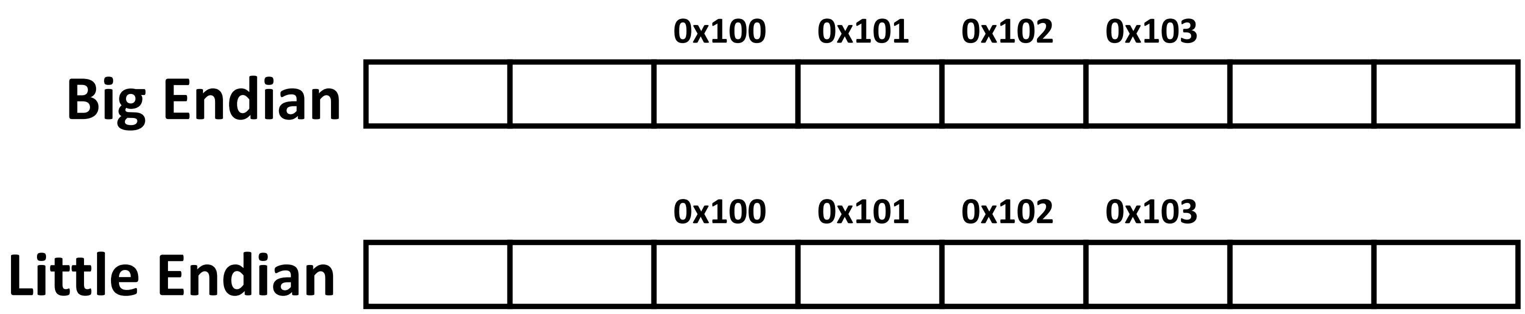
Byte Ordering

- **How should bytes within multi-byte word be ordered in memory?**
  - Peanut butter or chocolate first?
- **Say you want to store 0xaabbccdd**
  - What order will the bytes be stored?

- **Conventions!**
  - Big-endian, Little-endian
  - Based on Guliver stories, tribes cut eggs on different sides (big, little)

Byte Ordering Example

- **Big-Endian** (PPC, Internet)
  - Least significant byte has highest address
- **Little-Endian** (x86)
  - Least significant byte has lowest address
- **Example**
  - Variable has 4-byte representation `0x01234567`
  - Address of variable is `0x100`



Byte Ordering Example

- **Big-Endian** (PPC, Internet)
  - Least significant byte has highest address
- **Little-Endian** (x86)
  - Least significant byte has lowest address
- **Example**
  - Variable has 4-byte representation `0x01234567`
  - Address of variable is `0x100`

			0x100	0x101	0x102	0x103		
Big Endian			**01**	**23**	**45**	**67**		

			0x100	0x101	0x102	0x103		
Little Endian								

Byte Ordering Example

- **Big-Endian** (PPC, Sun, Internet)
  - Least significant byte has highest address
- **Little-Endian** (x86)
  - Least significant byte has lowest address
- **Example**
  - Variable has 4-byte representation `0x01234567`
  - Address of variable is `0x100`

			0x100	0x101	0x102	0x103		
Big Endian			01	23	45	67		

			0x100	0x101	0x102	0x103		
Little Endian			67	45	23	01		

Representing Integers

- `int A = 12345;`
- `int B = -12345;`
- `long int C = 12345;`

Decimal:	12345
Binary:	0011 0000 0011 1001
Hex:	3 0 3 9

IA32, x86-64 A

39
30
00
00

Sun A

00
00
30
39

IA32, x86-64 B

C7
CF
FF
FF

Sun B

FF
FF
CF
C7

IA32 C

39
30
00
00

X86-64 C

39
30
00
00
00
00
00
00

Sun C

00
00
30
39

Two's complement representation for negative integers (covered later)

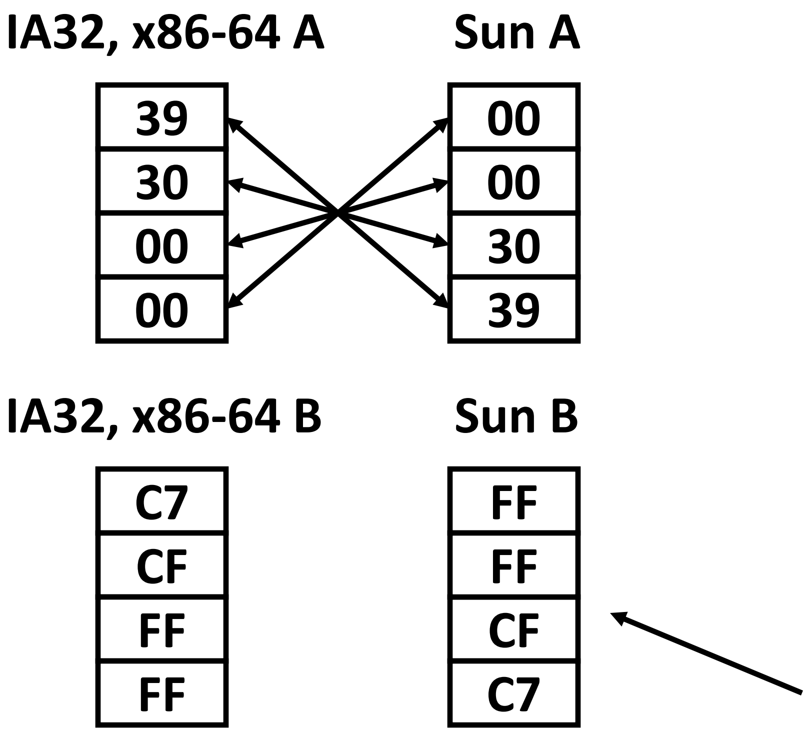
Representing Integers

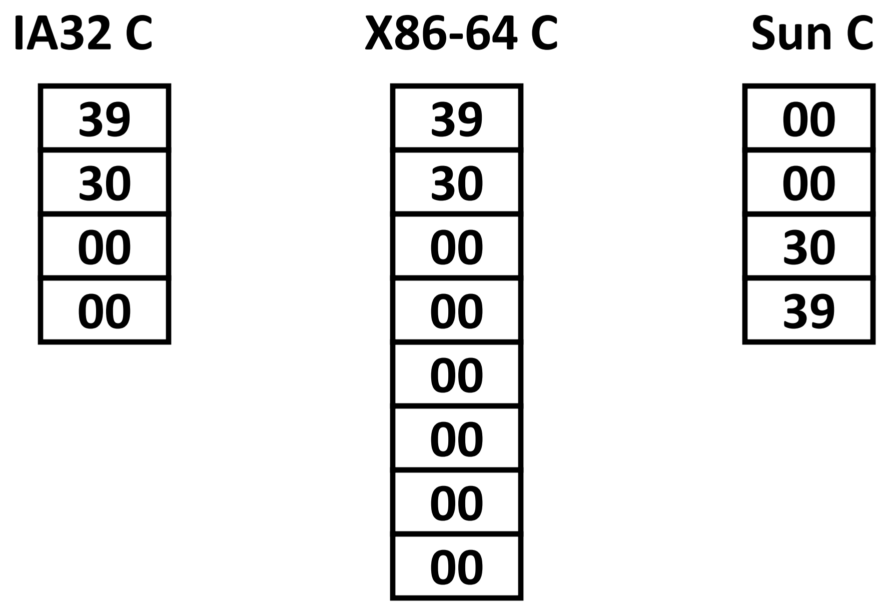
- `int A = 12345;`
- `int B = -12345;`
- `long int C = 12345;`

Decimal: 12345

Binary: 0011 0000 0011 1001

Hex: 3 0 3 9





Two's complement representation for negative integers (covered later)

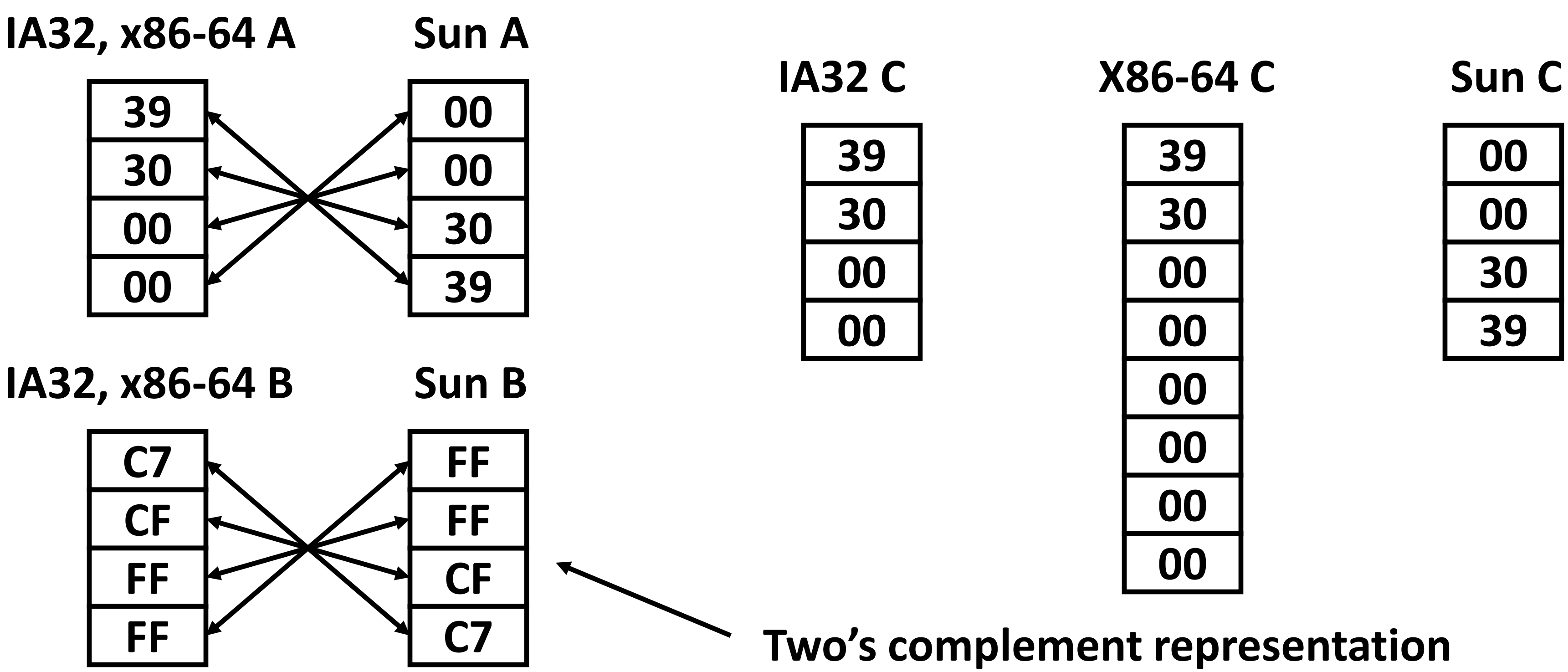
Representing Integers

- `int A = 12345;`
- `int B = -12345;`
- `long int C = 12345;`

Decimal: 12345

Binary: 0011 0000 0011 1001

Hex: 3 0 3 9



Two's complement representation for negative integers (covered later)

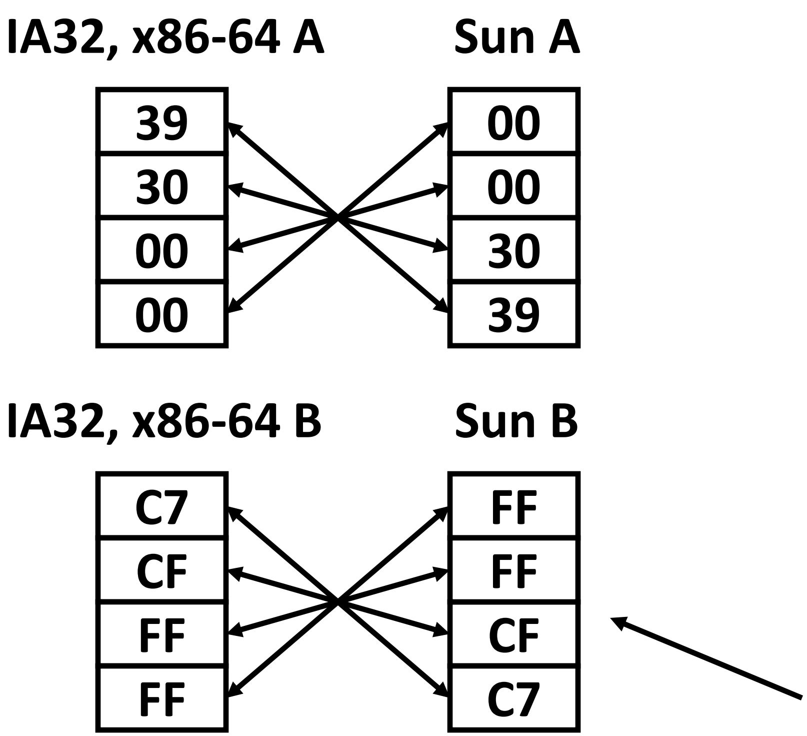
Representing Integers

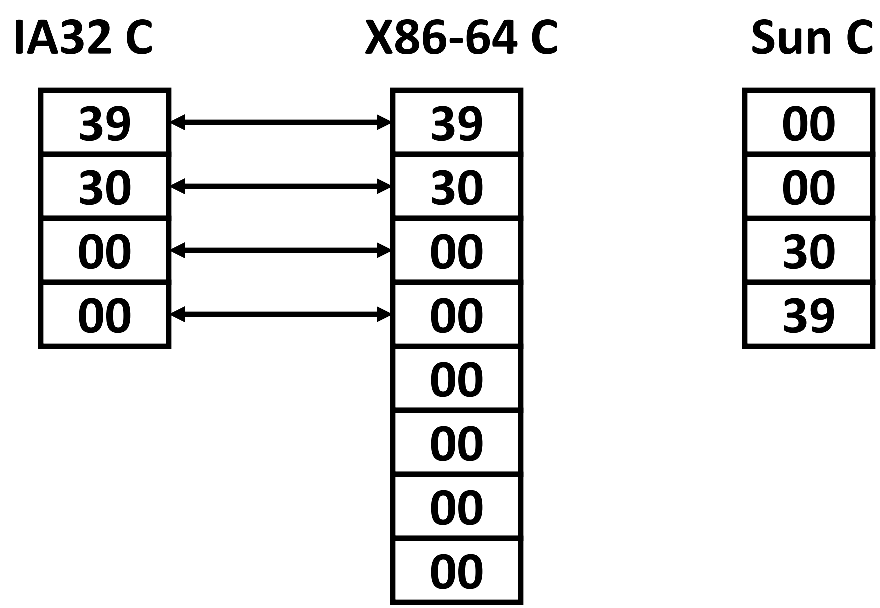
- `int A = 12345;`
- `int B = -12345;`
- `long int C = 12345;`

Decimal: 12345

Binary: 0011 0000 0011 1001

Hex: 3 0 3 9



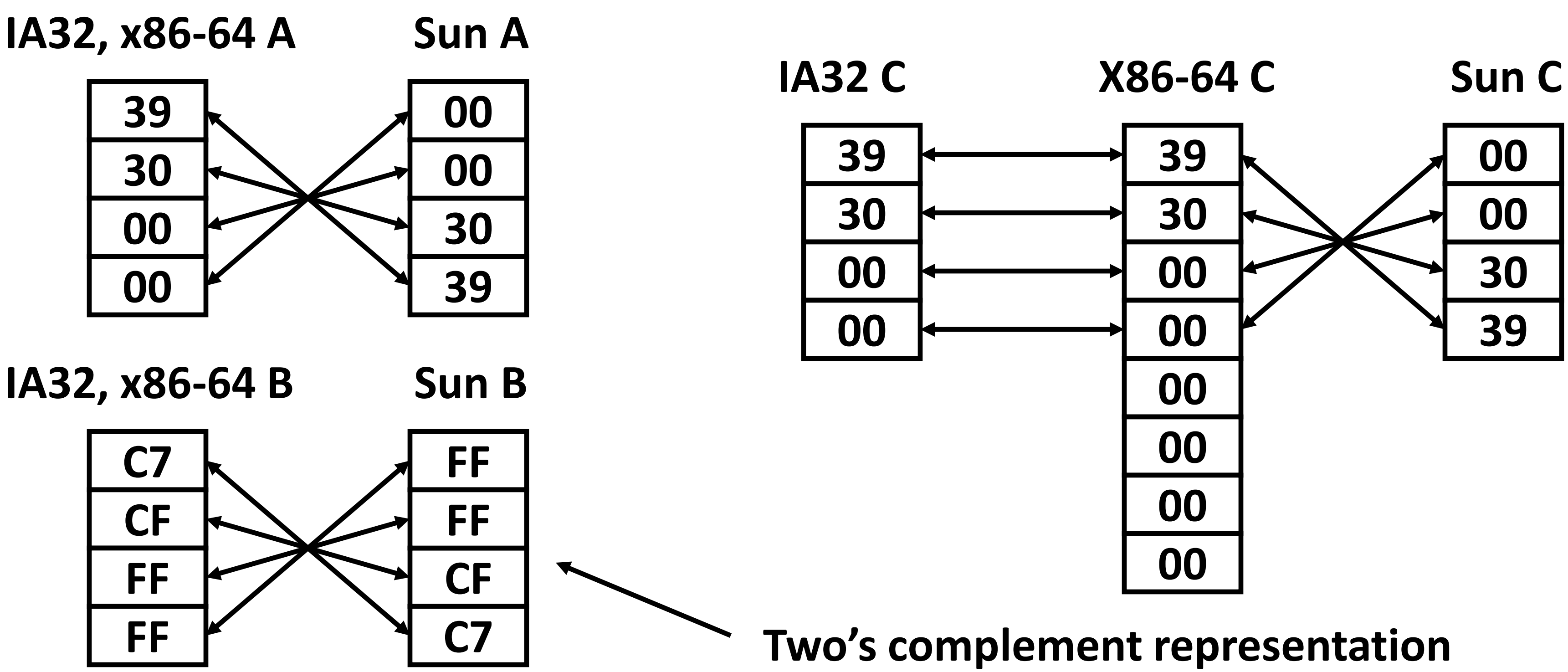


Two's complement representation for negative integers (covered later)

Representing Integers

- `int A = 12345;`
- `int B = -12345;`
- `long int C = 12345;`

Decimal:	12345
Binary:	0011 0000 0011 1001
Hex:	3 0 3 9



Two's complement representation for negative integers (covered later)

Reading Byte-Reversed Listings

- **Disassembly**
  - Text representation of binary machine code
  - Generated by program that reads the machine code
- **Example instruction in memory**
  - add value 0x12ab to register 'ebx' *(a special location in CPU's memory)*

Address	Instruction Code	Assembly Rendition
8048366:	81 c3 ab 12 00 00	add $0x12ab,%ebx

Reading Byte-Reversed Listings

- **Disassembly**
  - Text representation of binary machine code
  - Generated by program that reads the machine code
- **Example instruction in memory**
  - add value 0x12ab to register 'ebx' *(a special location in CPU's memory)*

Address	Instruction Code	Assembly Rendition
8048366:	81 c3 ab 12 00 00	add $0x12ab,%ebx

Deciphering numbers

- **Value:** **0x12ab**
- **Pad to 32 bits:** **0x000012ab**
- **Split into bytes:** **00 00 12 ab**
- **Reverse (little-endian):** **ab 12 00 00**

Addresses and Pointers in C

& = *'address of value'*
* = *'value at address'*
or 'dereference'

- **Pointer declarations use ***
  - int *ptr;   int x, y;   ptr = &x;
  - Declares a variable ptr that is a pointer to a data item that is an integer
  - Declares integer values named x and y
  - Assigns ptr to point to the address where x is stored
- **To use the value pointed to by a pointer we use *dereference***
  - If ptr = &x: then y = *ptr + 1 is the same as y = x + 1
  - If ptr = &y: then y = *ptr + 1 is the same as y = y + 1
  - *ptr is the value stored at the location to which the pointer ptr is pointing
  - What is *(&x) equivalent to?
- **We can do arithmetic on pointers**
  - ptr = ptr + 1;   *// really adds 4: type of ptr is int*, and an int uses 4 bytes!*
  - Changes the value of the pointer so that it now points to the next data item in memory (that may be y, or it may not – this is dangerous!)

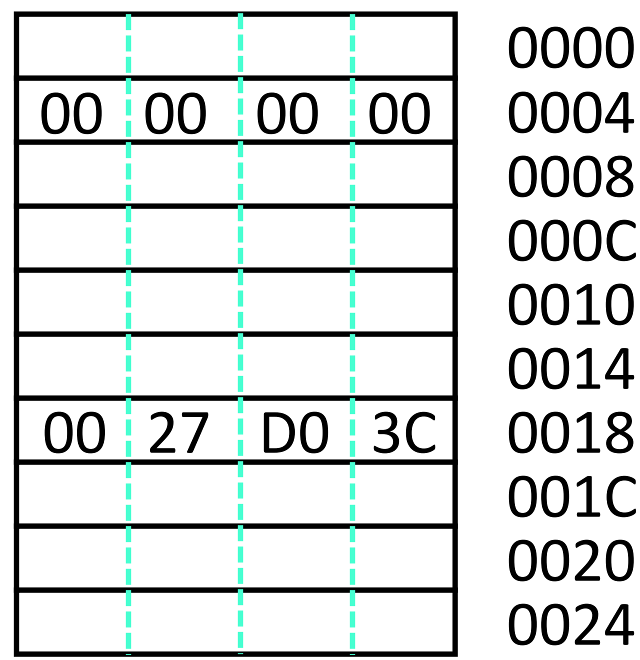
Assignment in C

- **Left-hand-side = right-hand-side**
  - LHS must evaluate to a memory *location* (a variable)
  - RHS must evaluate to a *value* (could be an address!)
- **E.g., x at location 0x04, y at 0x18**
  - x originally 0x0, y originally 0x3CD02700

				0000
00	00	00	00	0004
				0008
				000C
				0010
				0014
00	27	D0	3C	0018
				001C
				0020
				0024

Assignment in C

- **Left-hand-side = right-hand-side**
  - LHS must evaluate to a memory *location* (a variable)
  - RHS must evaluate to a *value* (could be an address!)
- **E.g., x at location 0x04, y at 0x18**
  - x originally 0x0, y originally 0x3CD02700
  - int x, y;
x = y + 3; //get value at y, add 3, put it in x



Assignment in C

- **Left-hand-side = right-hand-side**
  - LHS must evaluate to a memory *location* (a variable)
  - RHS must evaluate to a *value* (could be an address!)
- **E.g., x at location 0x04, y at 0x18**
  - x originally 0x0, y originally 0x3CD02700
  - int x, y;
    x = y + 3; //get value at y, add 3, put it in x

				0000
03	27	D0	3C	0004
				0008
				000C
				0010
				0014
00	27	D0	3C	0018
				001C
				0020
				0024

Assignment in C

- **Left-hand-side = right-hand-side**
  - LHS must evaluate to a memory *location* (a variable)
  - RHS must evaluate to a *value* (could be an address!)
- **E.g., x at location 0x04, y at 0x18**
  - x originally 0x0, y originally 0x3CD02700
  - int *x; int y;
    x = &y + 3; // get address of y, add ??

				0000
00	00	00	00	0004
				0008
				000C
				0010
				0014
00	27	D0	3C	0018
				001C
				0020
				0024

Assignment in C

- **Left-hand-side = right-hand-side**
  - LHS must evaluate to a memory *location* (a variable)
  - RHS must evaluate to a *value* (could be an address!)
- **E.g., x at location 0x04, y at 0x18**
  - x originally 0x0, y originally 0x3CD02700
  - int *x; int y;
    x = &y + 3; // get address of y, add 12
    // 0x0018 + 0x000C = 0x0024

				0000
24	00	00	00	0004
				0008
				000C
				0010
				0014
00	27	D0	3C	0018
				001C
				0020
				0024

Assignment in C

- **Left-hand-side = right-hand-side**
  - LHS must evaluate to a memory *location* (a variable)
  - RHS must evaluate to a *value* (could be an address!)
- **E.g., x at location 0x04, y at 0x18**
  - x originally 0x0, y originally 0x3CD02700
  - int *x; int y;
    x = &y + 3; // get address of y, add 12
    // 0x0018 + 0x000C = 0x0024

    *x = y; // value of y copied to
    // location to which x points

				0000
24	00	00	00	0004
				0008
				000C
				0010
				0014
00	27	D0	3C	0018
				001C
				0020
				0024

Assignment in C

- **Left-hand-side = right-hand-side**
  - LHS must evaluate to a memory *location* (a variable)
  - RHS must evaluate to a *value* (could be an address!)
- **E.g., x at location 0x04, y at 0x18**
  - x originally 0x0, y originally 0x3CD02700
  - int *x; int y;
    x = &y + 3; // get address of y, add 12
    // 0x0018 + 0x000C = 0x0024

    *x = y; // value of y copied to
    // location to which x points

				0000
24	00	00	00	0004
				0008
				000C
				0010
				0014
00	27	D0	3C	0018
				001C
				0020
00	27	D0	3C	0024

Arrays

- **Arrays represent adjacent locations in memory storing the same type of data object**
  - E.g., int big_array[128];
    allocated 512 adjacent locations in memory starting at 0x00ff0000
- **Pointers to arrays point to a certain type of object**
  - E.g., int * array_ptr;
    array_ptr = big_array;
    array_ptr = &big_array[0];
    array_ptr = &big_array[3];
    array_ptr = &big_array[0] + 3;
    array_ptr = big_array + 3;
    *array_ptr = *array_ptr + 1;
    array_ptr = &big_array[130];
  - In general: &big_array[i] is the same as (big_array + i)
    - *which implicitly computes: &bigarray[0] + i*sizeof(bigarray[0]);*

Arrays

- **Arrays represent adjacent locations in memory storing the same type of data object**
  - E.g., int big_array[128];
    allocated 512 adjacent locations in memory starting at 0x00ff0000
- **Pointers to arrays point to a certain type of object**
  - E.g., int * array_ptr;

array_ptr = big_array;	0x00ff0000
array_ptr = &big_array[0];	0x00ff0000
array_ptr = &big_array[3];	0x00ff000c
array_ptr = &big_array[0] + 3;	0x00ff000c *(adds 3 * size of int)*
array_ptr = big_array + 3;	0x00ff000c *(adds 3 * size of int)*
*array_ptr = *array_ptr + 1;	0x00ff000c *(but big_array[3] is incremented)*
array_ptr = &big_array[130];	0x00ff0208 *(out of bounds, C doesn't check)*

  - In general: &big_array[i] is the same as (big_array + i)
    - *which implicitly computes: &bigarray[0] + i*sizeof(bigarray[0]);*

Representing strings

- **A C-style string is represented by an array of bytes.**
  - Elements are one-byte ASCII codes for each character.
  - A 0 value marks the end of the array.

32	space	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	”	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	’	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	I	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	del

Null-terminated Strings

- For example, "Harry Potter" can be stored as a 13-byte array.

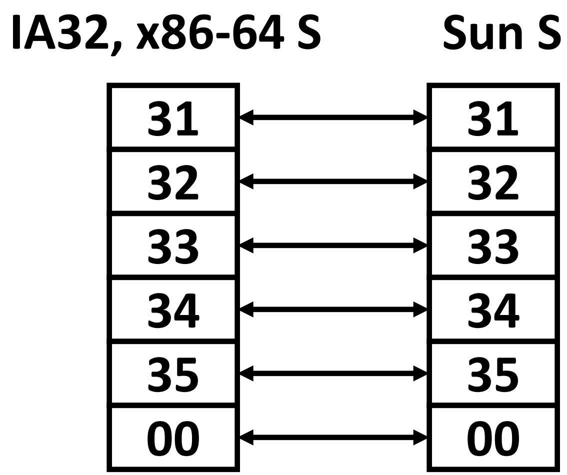
72	97	114	114	121	32	80	111	116	116	101	114	0
H	a	r	r	y		P	o	t	t	e	r	\0

- Why do we put a 0, or null zero, at the end of the string?
  - Note the special symbol: `string[12] = '\0';`

- How do we compute the string length?

Compatibility

```
char S[6] = "12345";
```



- **Byte ordering (endianness) is not an issue for standard C strings (char arrays)**
- **Unicode characters – up to 4 bytes/character**
  - ASCII codes still work (just add leading 0 bits) but can support the many characters in all languages in the world
  - Java and C have libraries for Unicode (Java commonly uses 2 bytes/char)

Examining Data Representations

- **Code to print byte representation of data**
  - Any data type can be treated as a *byte array* by casting it to `char`

```
typedef char byte;  //size of char == 1 byte

void show_bytes(byte *start, int len) {
  int i;
  for (i = 0; i < len; i++)
    printf("%p\t0x%.2x\n", start+i, *(start+i));
  printf("\n");
}
```

```
void show_int (int x) {
  show_bytes( (byte *) &x, sizeof(int));
}
```

printf directives:

%p	**Print pointer**
\t	**Tab**
%x	**Print value as hex**
\n	**New line**

show_bytes Execution Example

```
int a = 12345; // represented as 0x00003039
printf("int a = 12345;\n");
show_int(a);   // show_bytes((pointer) &a, sizeof(int));
```

Result (Linux):

```
int a = 12345;
0x11ffffcb8    0x39
0x11ffffcb9    0x30
0x11ffffcba    0x00
0x11ffffcbb    0x00
```

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 is 1 and B is 1
  - OR: A|B = 1 when either A is 1 or B is 1
  - XOR: A^B = 1 when either A is 1 or B is 1, but not both
  - NOT: ~A = 1 when A is 0 and vice-versa
  - DeMorgan’s Law: ~(A | B) = ~A & ~B

&	0	1
0	0	0
1	0	1

$\vert$	0	1
0	0	1
1	1	1

^	0	1
0	0	1
1	1	0

~	
0	1
1	0

General Boolean Algebras

- **Operate on bit vectors**
  - Operations applied bitwise

01101001	01101001	01101001	
& 01010101	| 01010101	^ 01010101	~ 01010101

- **All of the properties of Boolean algebra apply**

01010101
^ 01010101

- **How does this relate to set operations?**

Representing & Manipulating Sets

- **Representation**
  - Width *w* bit vector represents subsets of {0, …, *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 &, |, ^, ~ are 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)**
  - ~0x41 --> 0xBE

    ~01000001_2 --> 10111110_2
  - ~0x00 --> 0xFF

    ~00000000_2 --> 11111111_2
  - 0x69 & 0x55 --> 0x41

    01101001_2 & 01010101_2 --> 01000001_2
  - 0x69 | 0x55 --> 0x7D

    01101001_2 | 01010101_2 --> 01111101_2

Contrast: Logic Operations in C

- **Contrast to logical operators**
  - `&&`, `||`, `!`
    - View 0 as "False"
    - Anything nonzero as "True"
    - Always return 0 or 1
    - Early termination
- **Examples (char data type)**
  - `!0x41  -->  0x00`
  - `!0x00  -->  0x01`
  - `!!0x41 -->  0x01`

  - `0x69 && 0x55  -->  0x01`
  - `0x69 || 0x55  -->  0x01`
  - `p && *p++` (avoids null pointer access, null pointer = 0x00000000)