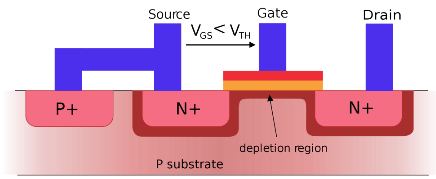
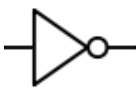
Today's topics

- Just enough EE to be dangerous
- Memory and its bits, bytes, and integers
- Representing information as bits
- Bit-level manipulations
  - Boolean algebra
  - Boolean algebra in C

Just enough EE to be dangerous

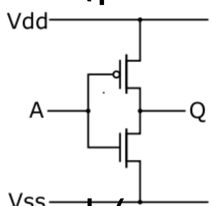


A transistor is a switch
CMOS has 2 types NPN and PNP



Vdd (positive)

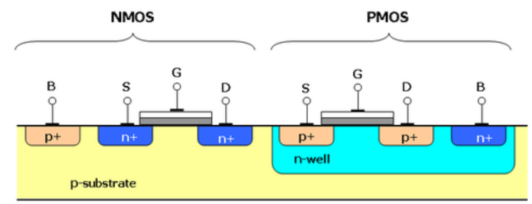
input

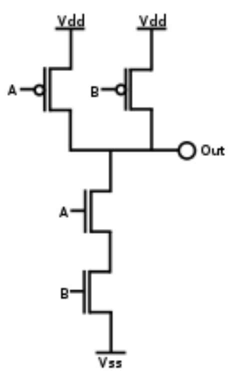


output

Ground (negative)

An inverter

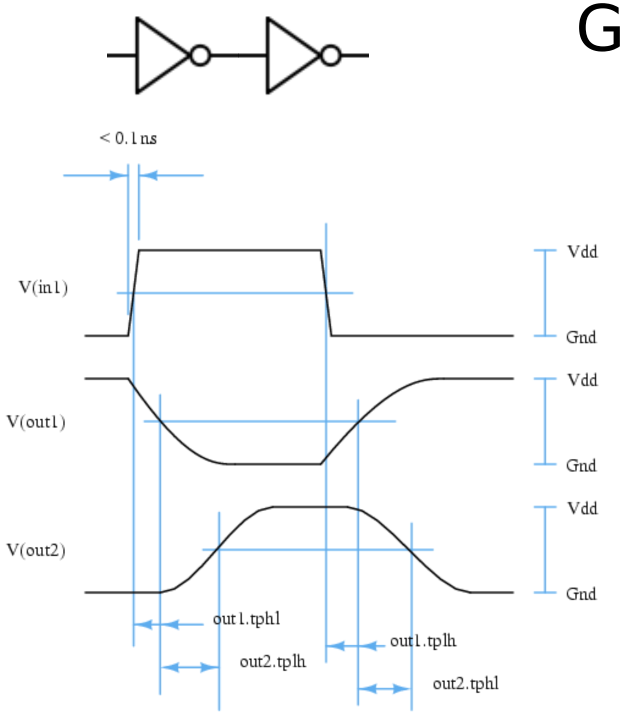




A NAND gate.

Theoretically NAND gates are “universal” and all other combinatorial circuits can be synthesized from them. However, real systems (except the Cray from way back when), do not take this approach because it is more efficient to specialize the circuit to the logic.

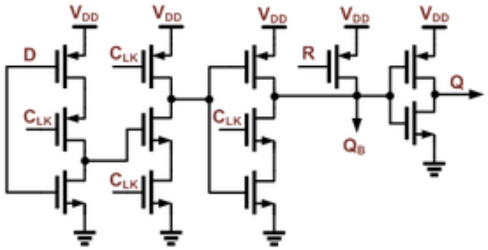
Just enough EE to be dangerous

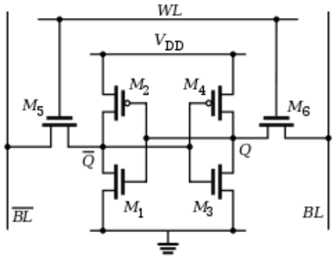


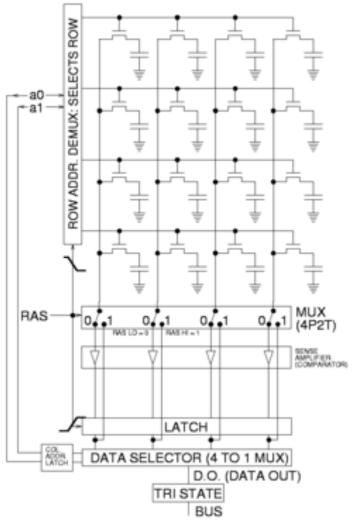
Gates are not infinitely fast

Note too that “binary” isn’t perfect in the real world, we call this non-ideal

What is memory, really?







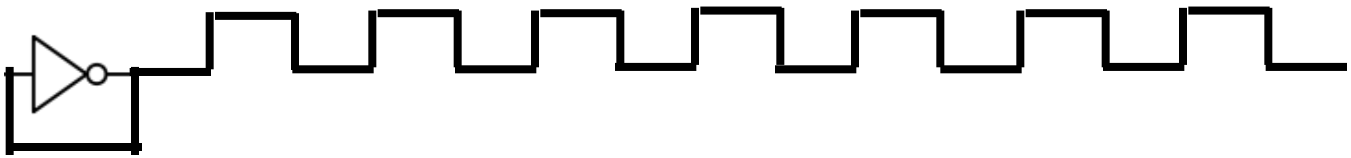
Flip-flop	SRAM cell	DRAM
Used for registers	Used for caches	Used for main memory
+Very fast -Very large	+Fast -Large	+Small -Slow

Review

- Combinatorial circuits are just bigger badder and uncut combinations of NPN and PNP transistors
  - Not infinitely fast
  - Not ideal
- Memory is either built from the same NPN and PNP transistors and/or exploits capacitance
  - Range of trade-offs in speed and density

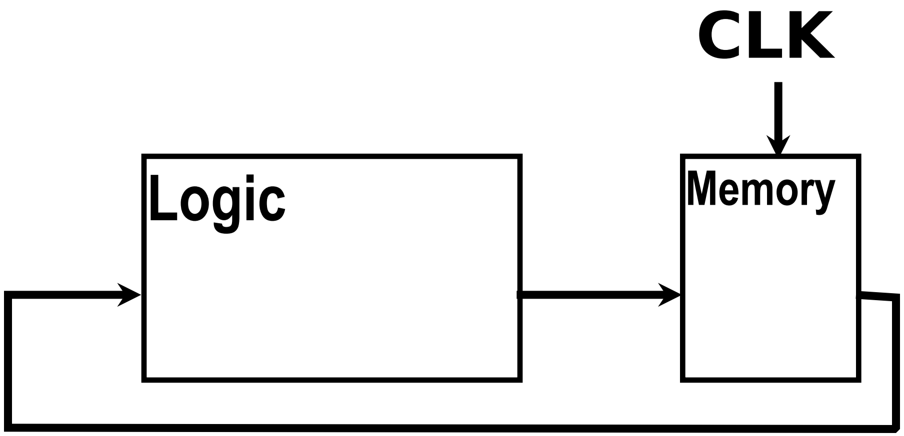
- NEED one more important thing to make a computer: a clock

Clock



(not done this way exactly, but is a sufficient model)

We use a **clock** to **control when** state (memory) is modified. We **time** the clock to be slow enough that combinatorial logic circuit outputs are **stable**.



Computer!

Real computers distribute state and logic all over the place and run on multiple clocks, but an entire computing system **can be built** as described above.

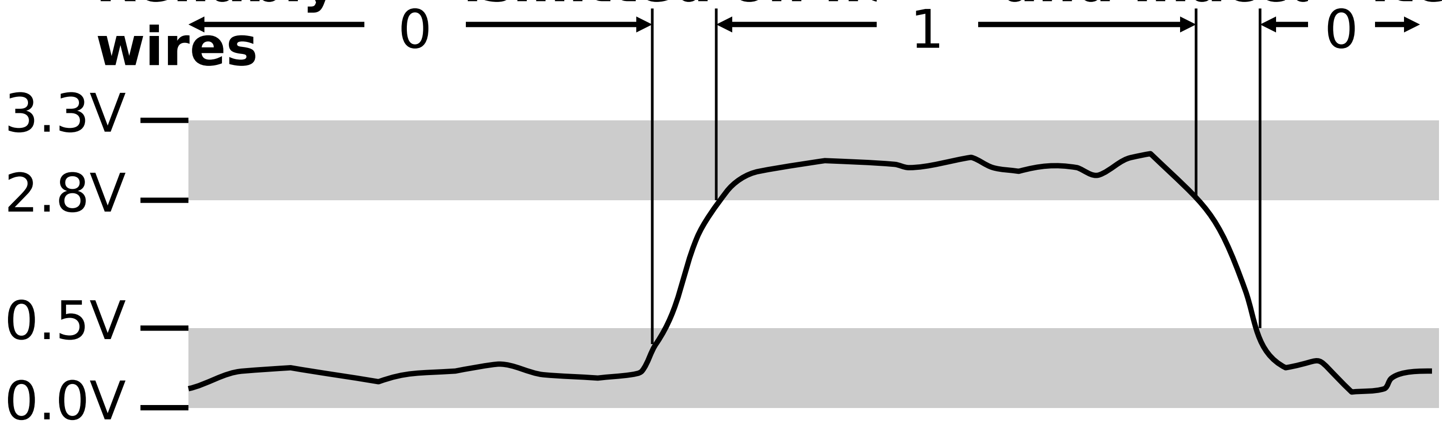
Review

- Computers need 3 things to work:
  - The ability to transform input to output (combinatorial circuits)
  - The ability to store state (memory)
  - The ability to precisely control when state evolves (timing)
- Building a computer from component pieces is not terribly difficult. A basic MIPS processor can be designed in ~ 100 hours for someone that has never done digital design before.
- Computers can be really really small (how small?)
- The processor in your laptop has O(100M) or so logic transistors and O(1000M) memory

- A programmers view of Memory

Binary Representations

- Base 2 number representation
  - Represent 351_{10} as 0000000101011111_2 or 101011111_2
- Why Binary?
  - Electronic implementation
    - Easy to store with bi-stable elements
    - Reliably transmitted on noisy and inaccurate wires

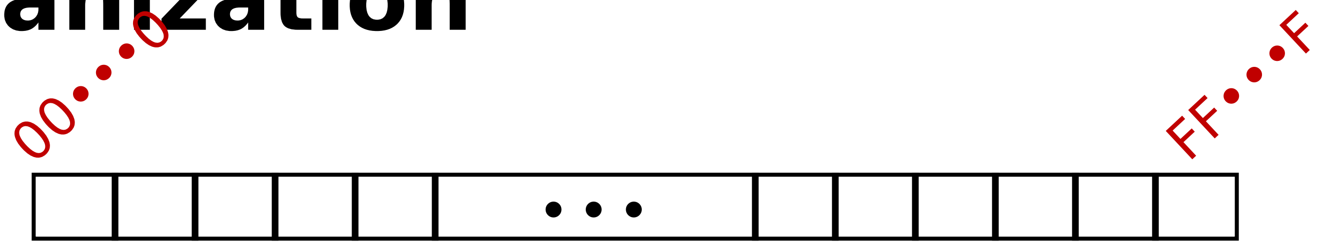


Encoding Byte Values

- **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`
- **Programmers use hex (16), decimal (10), and sometimes octal (8).**
- **Machines use binary for storage, and internal wires; often something else for external communications**

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



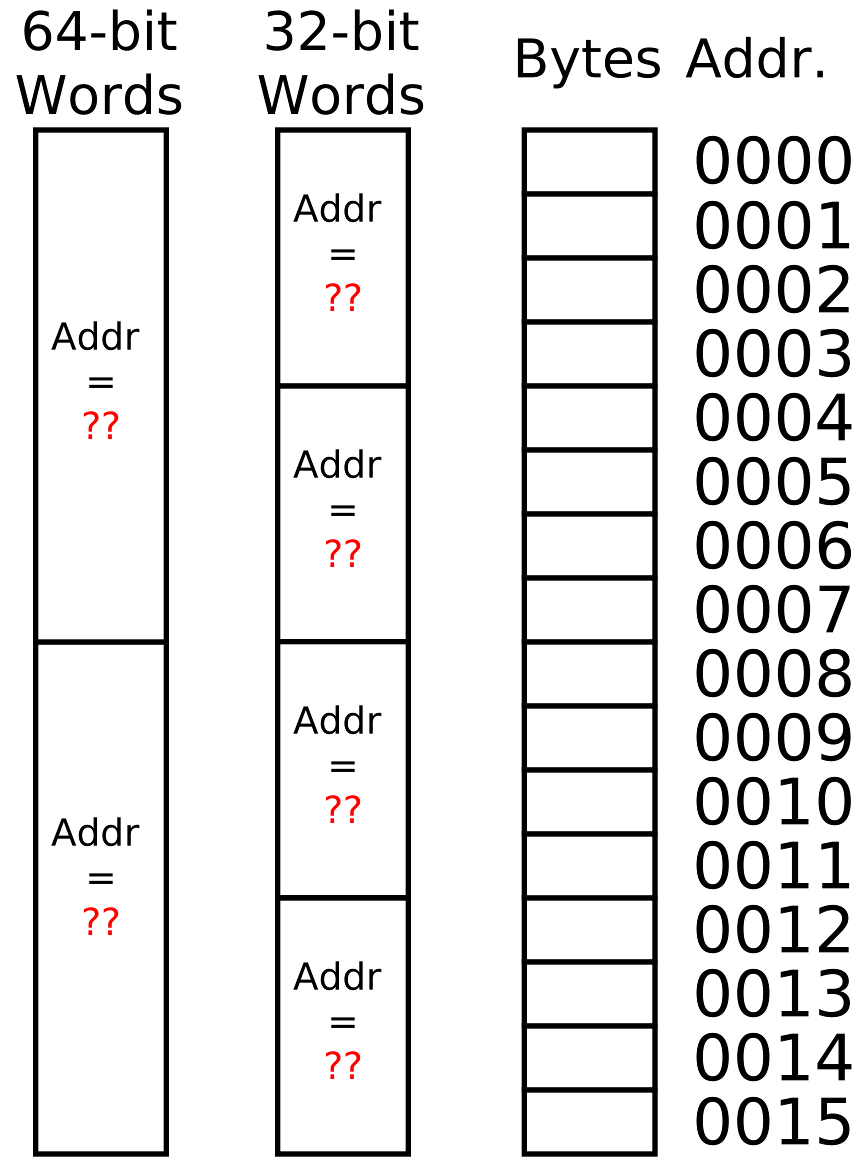
- **Programs refer to addresses**
  - **Conceptually, a very large array of bytes**
  - **System provides an address space private to each "process"**
    - **Process = program being executed + its data + its "state"**
    - **Program can clobber its own data, but not that of others**
    - **Clobbering code or "state" often leads to crashes (or security holes)**
- **Compiler + run-time system control memory allocation**

Machine Words

- **Machines have a “word size”**
  - **Used to be the nominal size of integer-valued data**
    - **Danger Will Robinson: { int p = (int) &a; }**
  - **Now only reliably the size of an address**
    - **{ void *p = &a; }**
  - **For a very long time (and likely still now) most machines are 32 bits**
    - **Limits addresses to 4GB**
    - **Becoming too small for memory-intensive applications**
  - **Eventually the world will switch to 64 bits**
    - **Potential address space □ 1.8 X 10^{19} bytes (to put that in perspective, however, there are ~ 10^{80} atoms in the universe)**
  - **Trivia: Word size does not have to always equal physical memory size. For example, x86 (real mode) in the 80’s was 16 bit word size, but 20 bits of physical address space. x86-64 is 64 bit addresses, but only 48 bits of physical address**

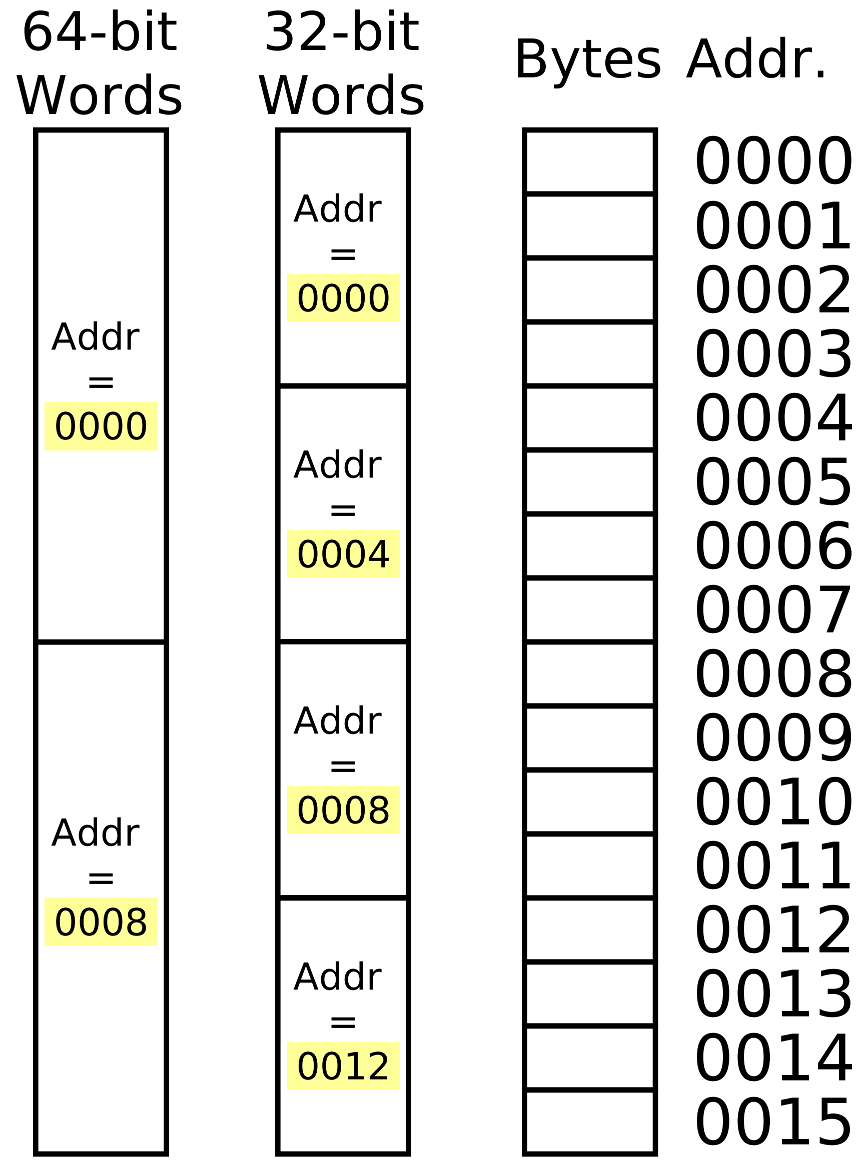
Word-Oriented Memory Organization

- 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?



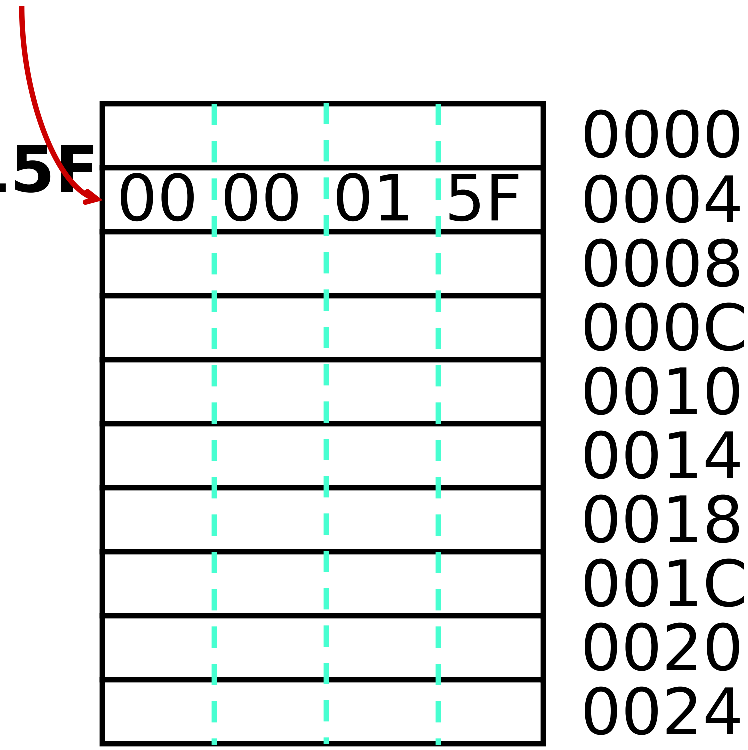
Word-Oriented Memory Organization

- **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?**
- **Alignment**
  - **Generally speaking it is a good idea to align items on their "natural" width.
i.e., (&a) % sizeof(a) = 0**
  - **This tends to aid with performance**
  - **This is required for correct multithreaded code!**
  - **This is often not true when talking to hardware devices!**



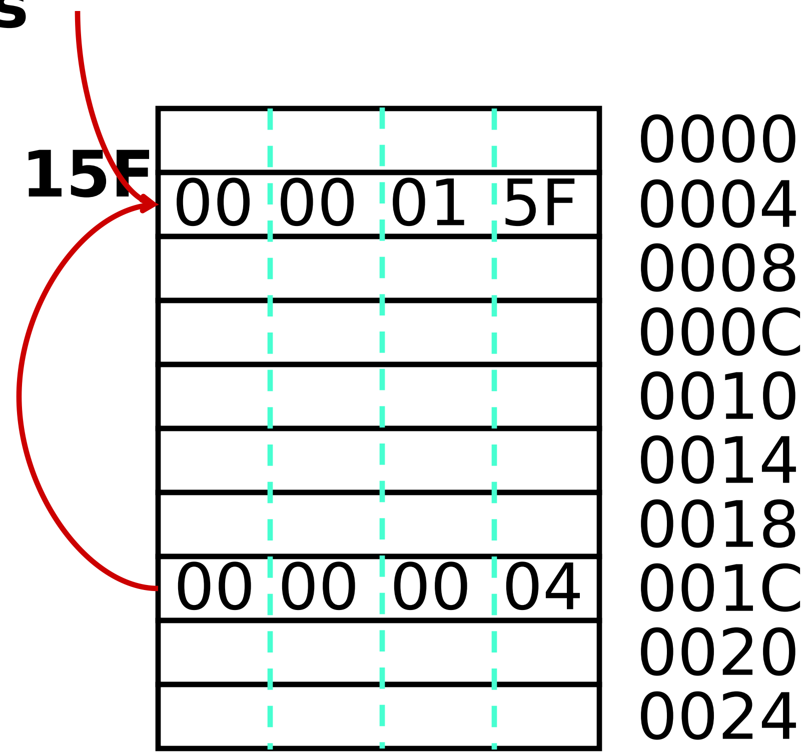
Addresses and Pointers

- **Address is a location in memory**
- **Pointer is a data object that contains an address**
- **Address 0004 stores the value 351 (or 15F**



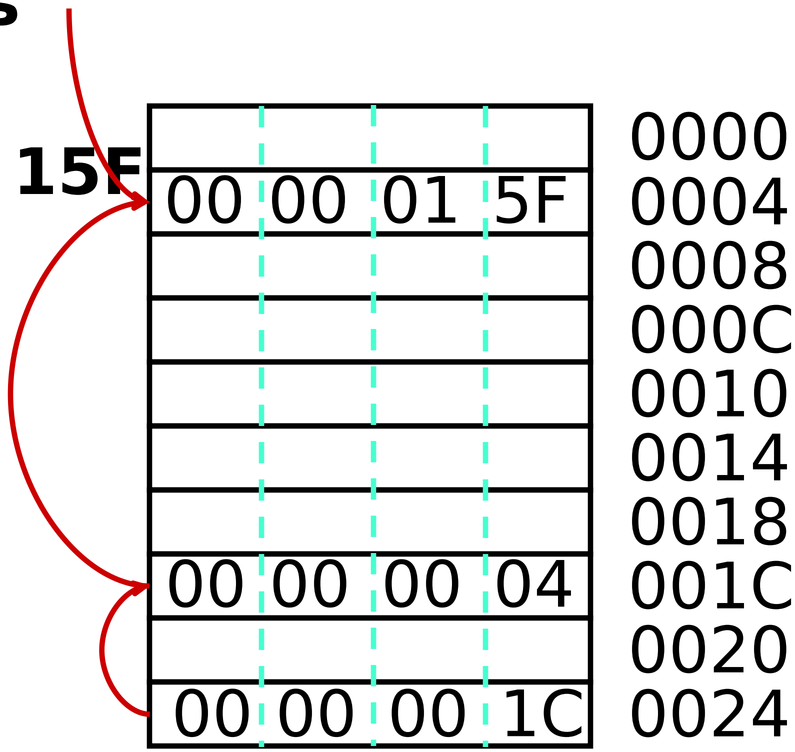
Addresses and Pointers

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



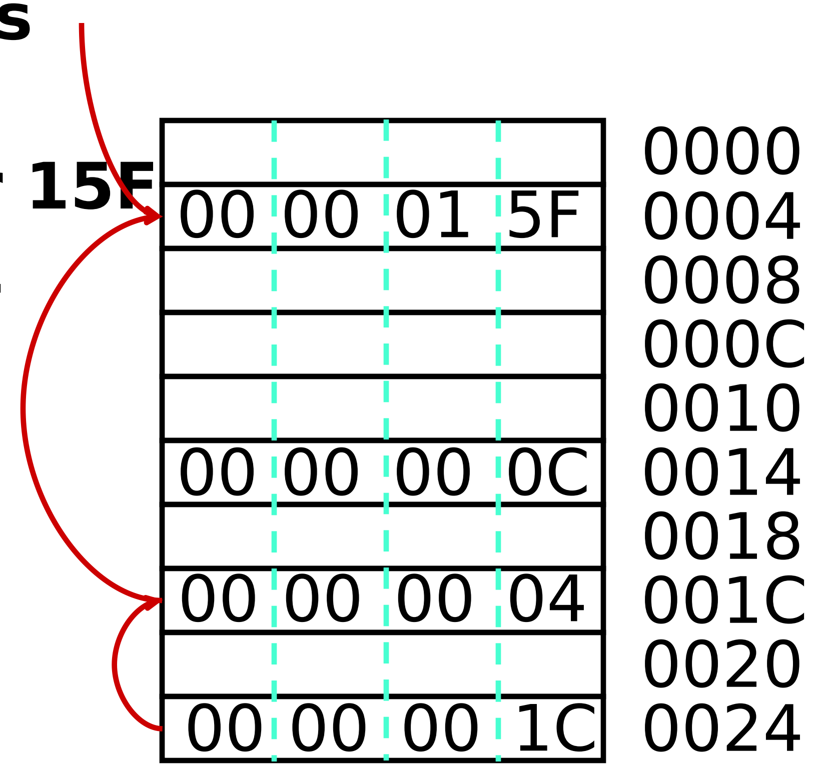
Addresses and Pointers

- **Address is a location in memory**
- **Pointer is a data object that contains an address**
- **Address 0004 stores the value 351 (or 15F**
- **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

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

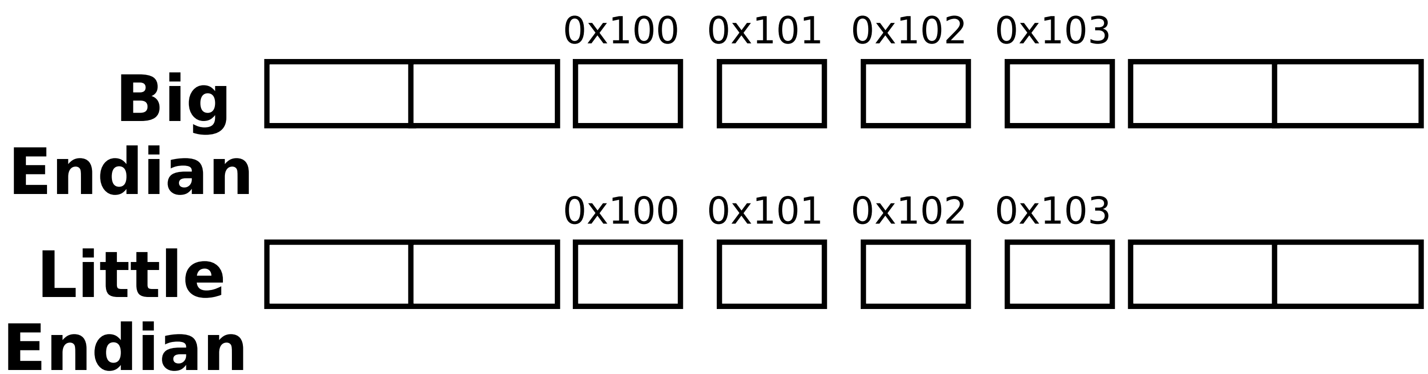
- How should bytes within multi-byte word be ordered in memory?
- Say you want to store 0xaabbccdd
  - What order will the bytes be stored?

Byte Ordering

- How should bytes within multi-byte word be ordered in memory?
- Say you want to store 0xaabbccdd
  - What order will the bytes be stored?
- Conventions!
  - Big-endian, Little-endian, and the rare Big-Bad Little Endian
  - Based on Guliver stories, tribes cut eggs on different sides (big, little)

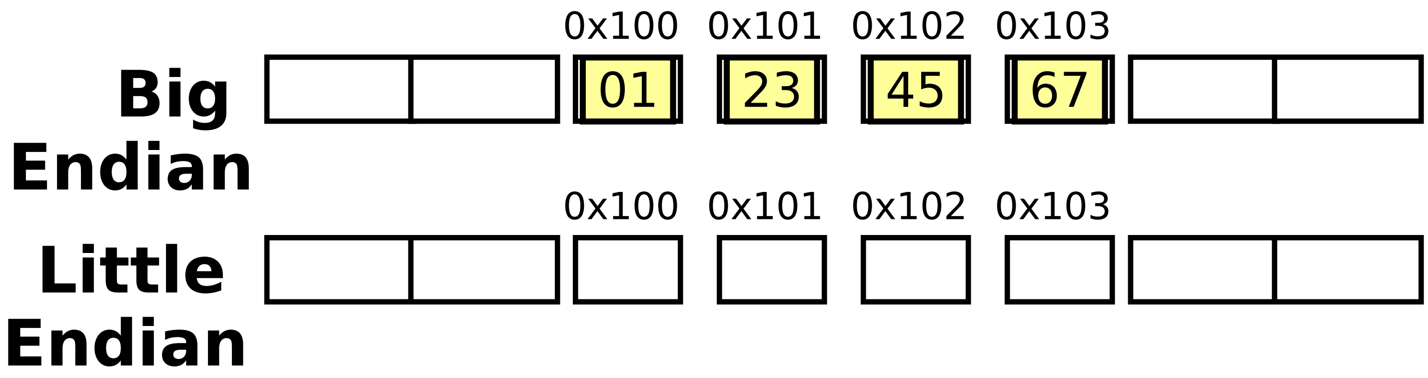
Byte Ordering Example

- **Big-Endian** (PPC, Sparc, 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`



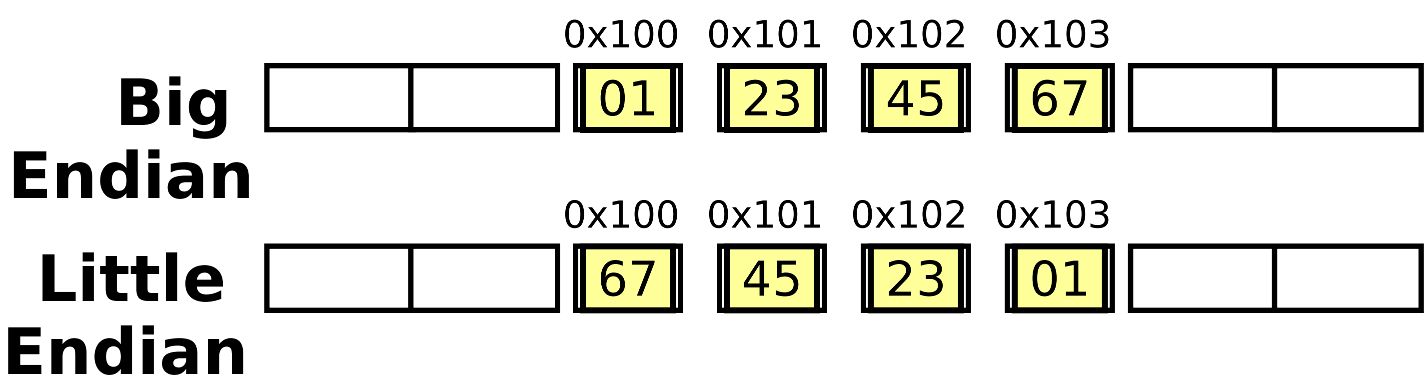
Byte Ordering Example

- **Big-Endian** (PPC, Sparc, 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, Sparc, 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 - why care?

- **Most** of the time you don't
- The most common place you will care is when you write code to interface over the network.
  - **By historical convention, most protocols on the internet use Big Endian.**
  - **You most likely own / write code on an x86(-64) machine which is little endian**
  - **This means you will end up doing a lot of byte swapping in protocol code.**
    - **And yes, when an x86 machine talks to another x86 machine on the internet, each is converting numbers to big endian before sending them on the wire, and after receiving them :-(**
- The second most common place you care is when you are reading / writing "standard" file formats.

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**

& = 'address of value'
* = 'value at address'
or 'de-reference'

*(&x) is equivalent to
....

Addresses and Pointers in C

- **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**
- **We can do arithmetic on pointers**
  - **ptr = ptr + 1; // really adds 4 (because an integer 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, may not – dangerous!)**
- **To use the value pointed to by a pointer we use de-reference**
  - **y = *ptr + 1; is the same as y = x + 1;**
  - **But, 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**

Arrays

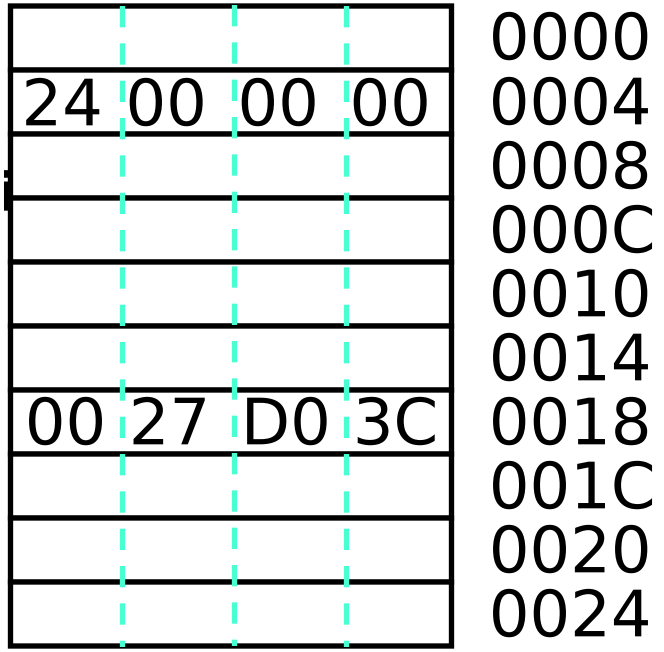
- Arrays represent adjacent locations in memory storing the same type of data object
  - E.g., int big_array[128];
    allocated 512 adjacent locations in memory starting at 0x00ff0000
- Pointers to arrays point to a certain type of object
  - E.g., int * array_ptr;
    array_ptr = big_array;
    array_ptr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

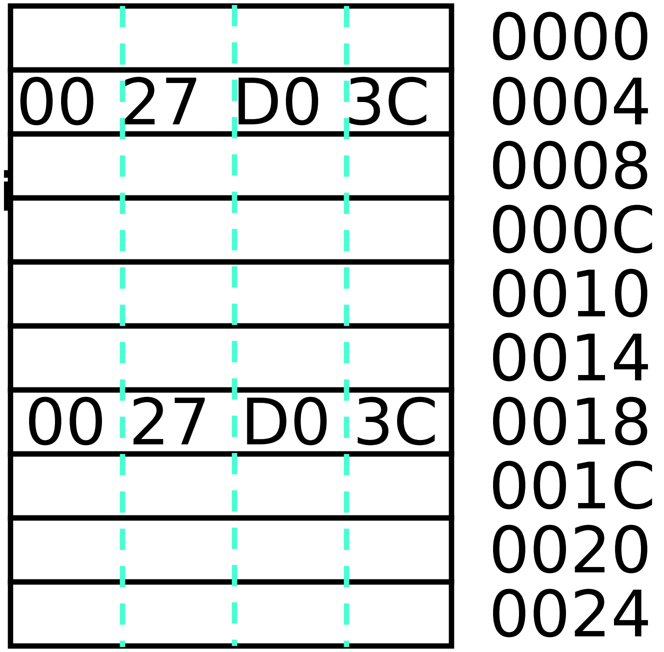
General rules for C (assignments)

- **Left-hand-side = right-hand-side**
  - **LHS must evaluate to a memory LOCATION**
  - **RHS must evaluate to a VALUE (could be an address)**
- **E.g., x at location 0x04, y at**
  - **int x, y;**
    x = y; // get value at y and put i



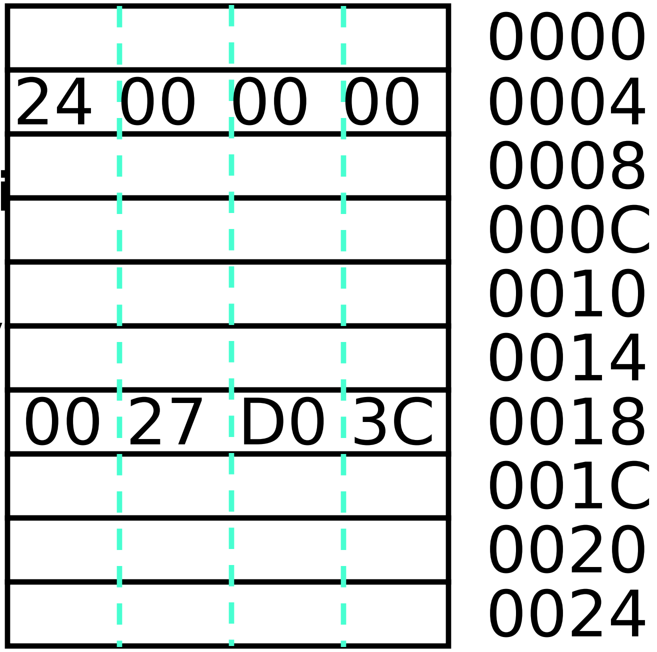
General rules for C (assignments)

- **Left-hand-side = right-hand-side**
  - **LHS must evaluate to a memory LOCATION**
  - **RHS must evaluate to a VALUE (could be an address)**
- **E.g., x at location 0x04, y at**
  - **int x, y;**
    x = y; // get value at y and put i



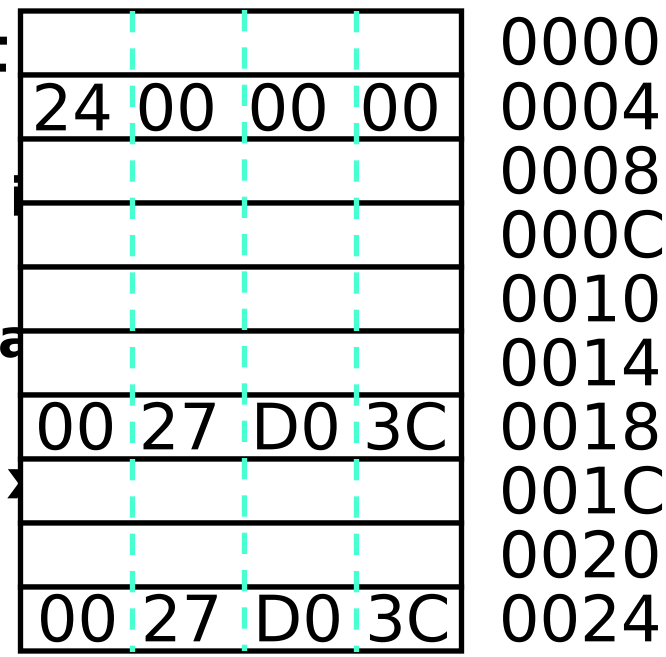
General rules for C (assignments)

- **Left-hand-side = right-hand-side**
  - **LHS must evaluate to a memory LOCATION**
  - **RHS must evaluate to a VALUE (could be an address)**
- **E.g., x at location 0x04, y at**
  - **int x, y;**
    x = y; // get value at y and put i
  - **int * x; int y;**
    x = &y + 12; // get address of y



General rules for C (assignments)

- **Left-hand-side = right-hand-side**
  - **LHS must evaluate to a memory LOCATION**
  - **RHS must evaluate to a VALUE (could be an address)**
- **E.g., x at location 0x04, y at**
  - **int x, y;**
    x = y; // get value at y and put i
  - **int * x; int y;**
    x = &y + 3; // get address of y a
  - **int * x; int y;**
    ***x = y; // value of y to location x**



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("0x%p\t0x%.2x\n", start+i, start[i]);
  printf("\n");
}
```

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

Some printf directives:
%p: Print pointer
%x: Print hexadecimal
"\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
```

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 ASun A

IA32, x86-64 A	Sun A
39	00
30	00
00	30
00	39

IA32, x86-64 BSun B

IA32, x86-64 B	Sun B
C7	FF
CF	FF
FF	CF
FF	C7

IA32 C	X86-64 C	Sun C
39	39	00
30	30	00
00	00	30
00	00	39
	00	
	00	
	00	
	00	

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

IA32, x86-64 ASun A

IA32, x86-64 A	Sun A
39	00
30	00
00	30
00	39

IA32, x86-64 BSun B

IA32, x86-64 B	Sun B
C7	FF
CF	FF
FF	CF
FF	C7

IA32 C	X86-64 C	Sun C
39	39	00
30	30	00
00	00	30
00	00	39
	00	
	00	
	00	
	00	

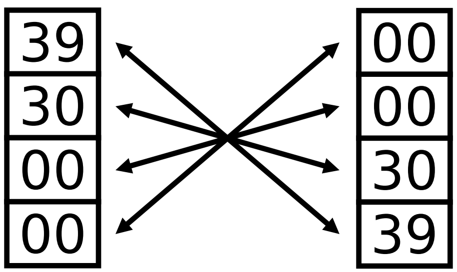
Two's complement representation for negative integers (covered later)

Representing Integers

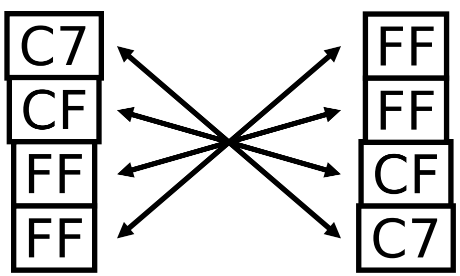
- `int A = 12345;`
- `int B = -12345;`
- `long int C = 12345;`

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

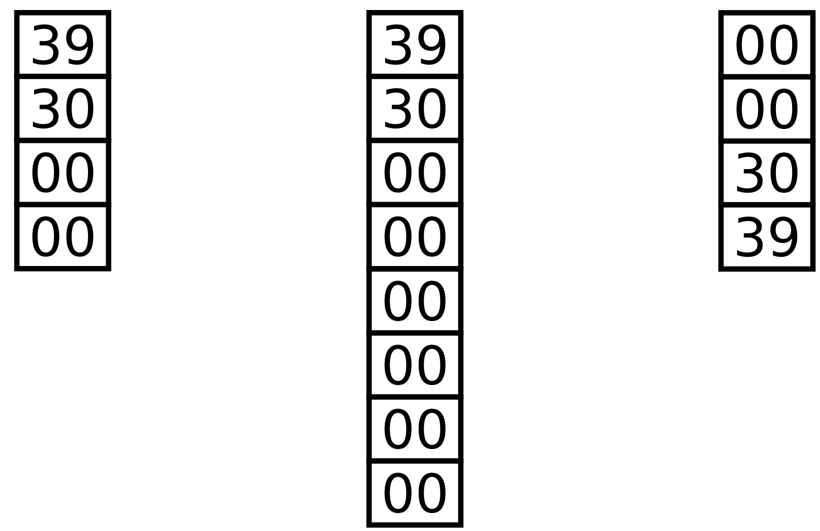
IA32, x86-64 A **Sun A**



IA32, x86-64 B **Sun B**



IA32 C **X86-64 C** **Sun C**



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

Representing Integers

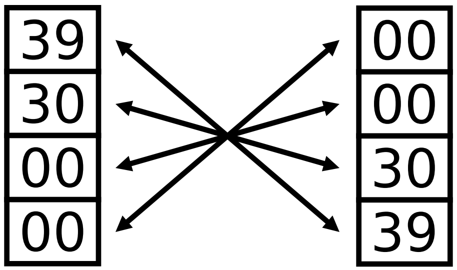
- `int A = 12345;`
- `int B = -12345;`
- `long int C = 12345;`

Decimal: 12345

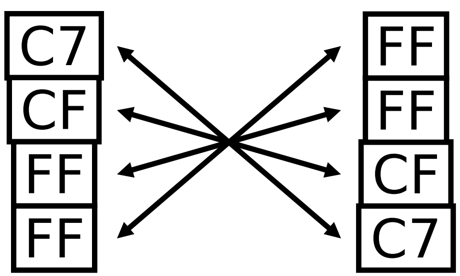
Binary: 0011 0000 0011 1001

Hex: 3 0 3 9

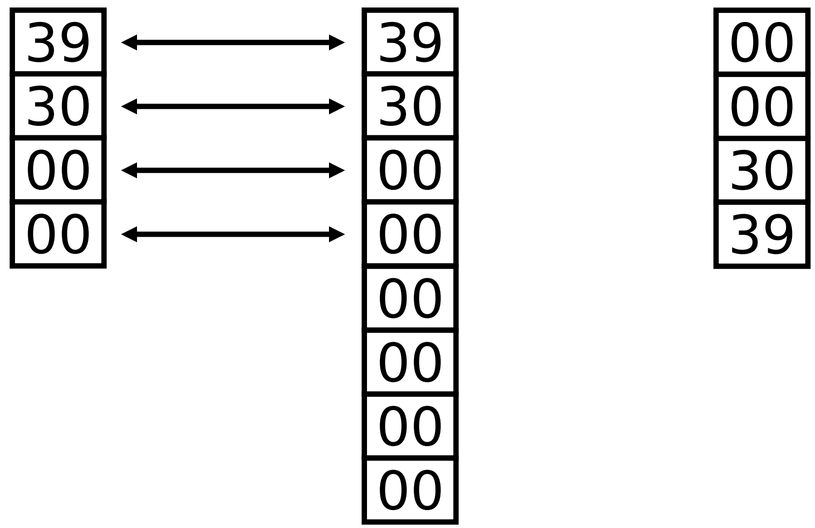
IA32, x86-64 ASun A



IA32, x86-64 BSun B



IA32 C **X86-64 C** **Sun C**



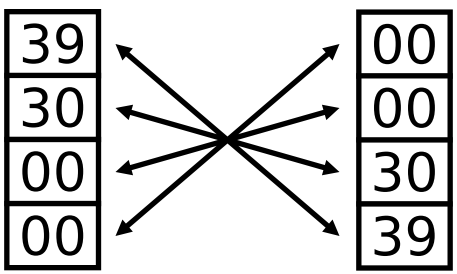
Two's complement representation for negative integers (covered later)

Representing Integers

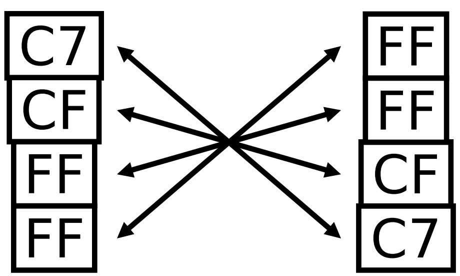
- `int A = 12345;`
- `int B = -12345;`
- `long int C = 12345;`

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

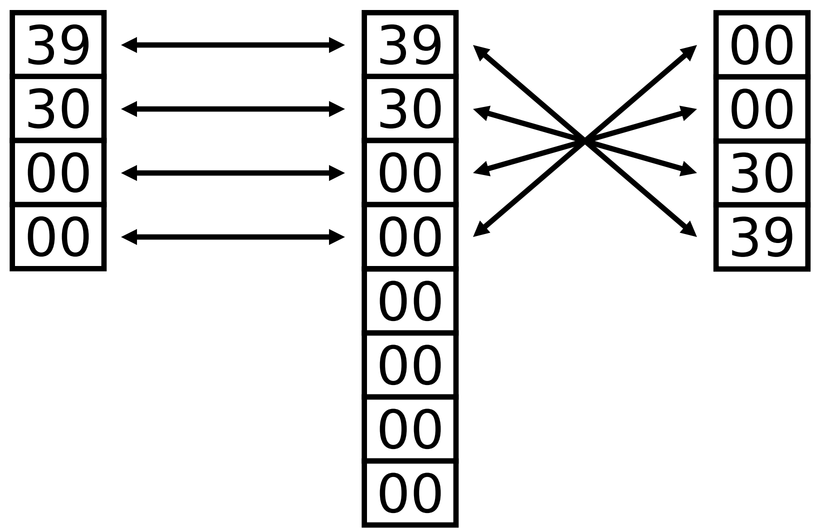
IA32, x86-64 ASun A



IA32, x86-64 BSun B



IA32 C **X86-64 C** **Sun C**



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

Representing Pointers

- `int B = -12345;`
- `int *P = &B;`

Sun P	IA32 P	x86-64 P
EF	D4	0C
FF	F8	89
FB	FF	EC
2C	BF	FF
		FF
		7F
		00
		00

Depending on the operating system, architecture, type and “location” of a variable, where it is stored varies. Generally speaking there are 4 “areas” of memory in a process: small statics, large statics, heap, and stack. **But each library can have it’s own statics as well.**

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

ASCII now rules the computing landscape, but it was not always so. UNICODE is also displacing it, but slowly

Null-terminated Strings

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

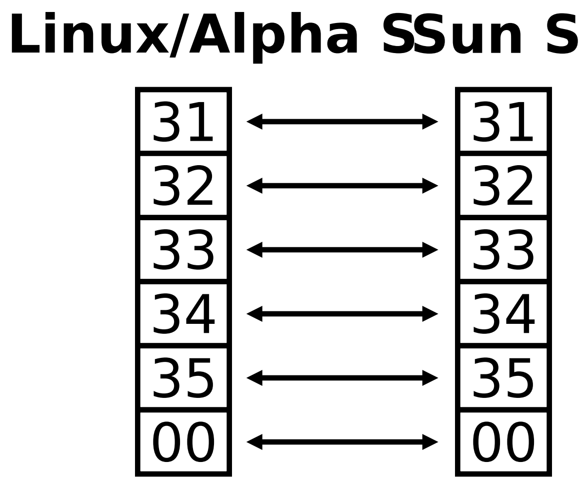
72	97	114	114	121	32	80	111	116	116	101	114	0
H	a	r	r	y		P	o	t	t	e	r	\0

- Why do we put a a 0, or null, at the end of the string?

- Computing string length?

Compatibility

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



- **Byte ordering not an issue**
- **Unicode characters – up to 4 bytes/character**
  - **ASCII codes still work (leading 0 bit) 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)**

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

|	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 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 <-- Danger Will Robinson: Not always the same size 0 or 1. Depends on compiler...**
    - **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**)