CSE 351
Section 2

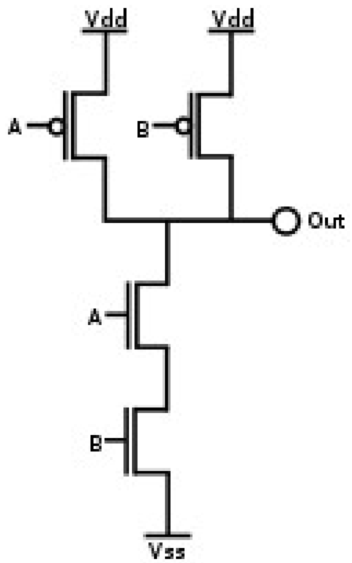
1/12/12

Agenda

- Review memory and data representation
  - NAND Gate
  - Binary/Decimal/Hex
  - Memory Organization and Pointers
  - Endianness

NAND Gate

- Output is always high (1) except when both inputs are high
  - That is, the opposite of an AND
- How does this circuit work?



Truth Table

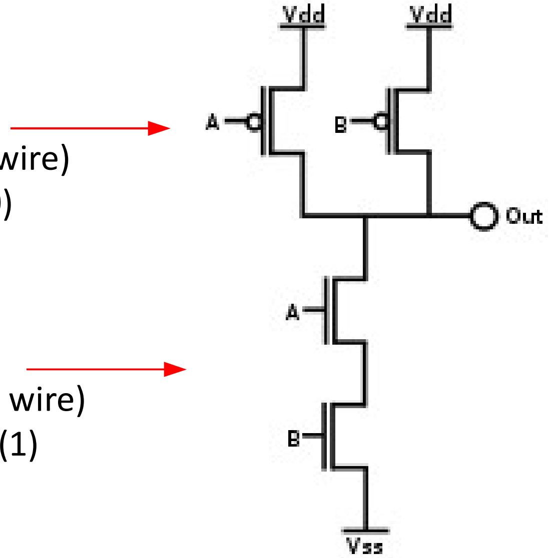
A	B	Out
0	0	1
0	1	1
1	0	1
1	1	0

NAND Gate

- Output is always high (1) except when both inputs are high
  - That is, the opposite of an AND
- How does this circuit work?

PNP Transistors
“On” (behave like a wire)
when input is **low** (0)

NPN Transistors
“On” (behave like a wire)
when input is **high** (1)

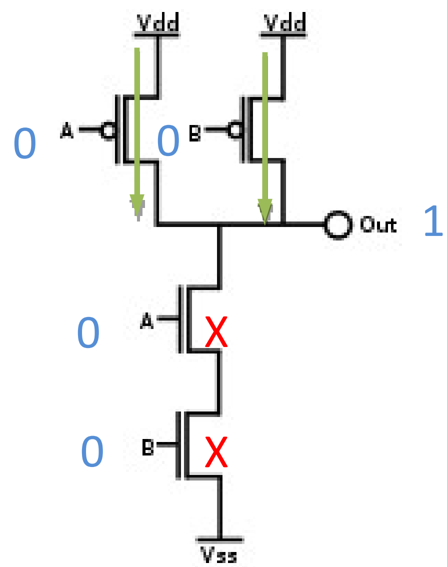


Truth Table

A	B	Out
0	0	1
0	1	1
1	0	1
1	1	0

NAND Gate

- Output is always high (1) except when both inputs are high
  - That is, the opposite of an AND
- How does this circuit work?

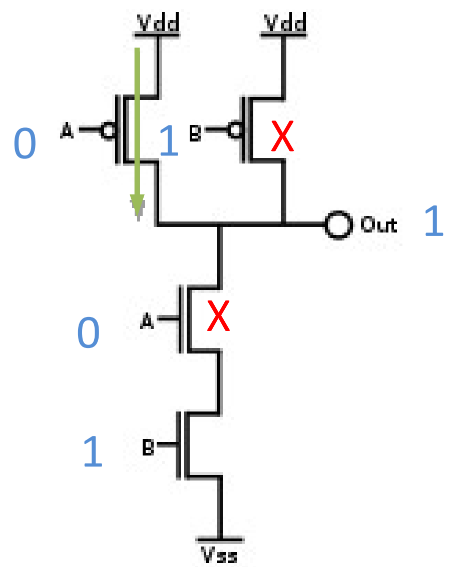


Truth Table

A	B	Out
0	0	1
0	1	1
1	0	1
1	1	0

NAND Gate

- Output is always high (1) except when both inputs are high
  - That is, the opposite of an AND
- How does this circuit work?

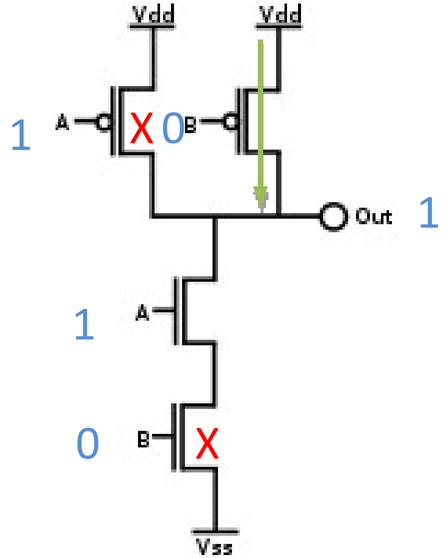


Truth Table

A	B	Out
0	0	1
0	1	1
1	0	1
1	1	0

NAND Gate

- Output is always high (1) except when both inputs are high
  - That is, the opposite of an AND
- How does this circuit work?

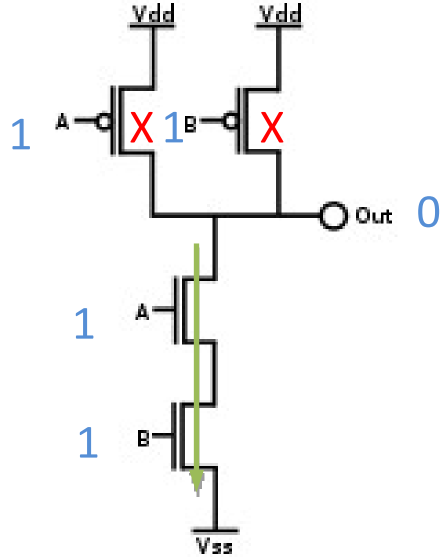


Truth Table

A	B	Out
0	0	1
0	1	1
1	0	1
1	1	0

NAND Gate

- Output is always high (1) except when both inputs are high
  - That is, the opposite of an AND
- How does this circuit work?



Truth Table

A	B	Out
0	0	1
0	1	1
1	0	1
1	1	0

Number Formats

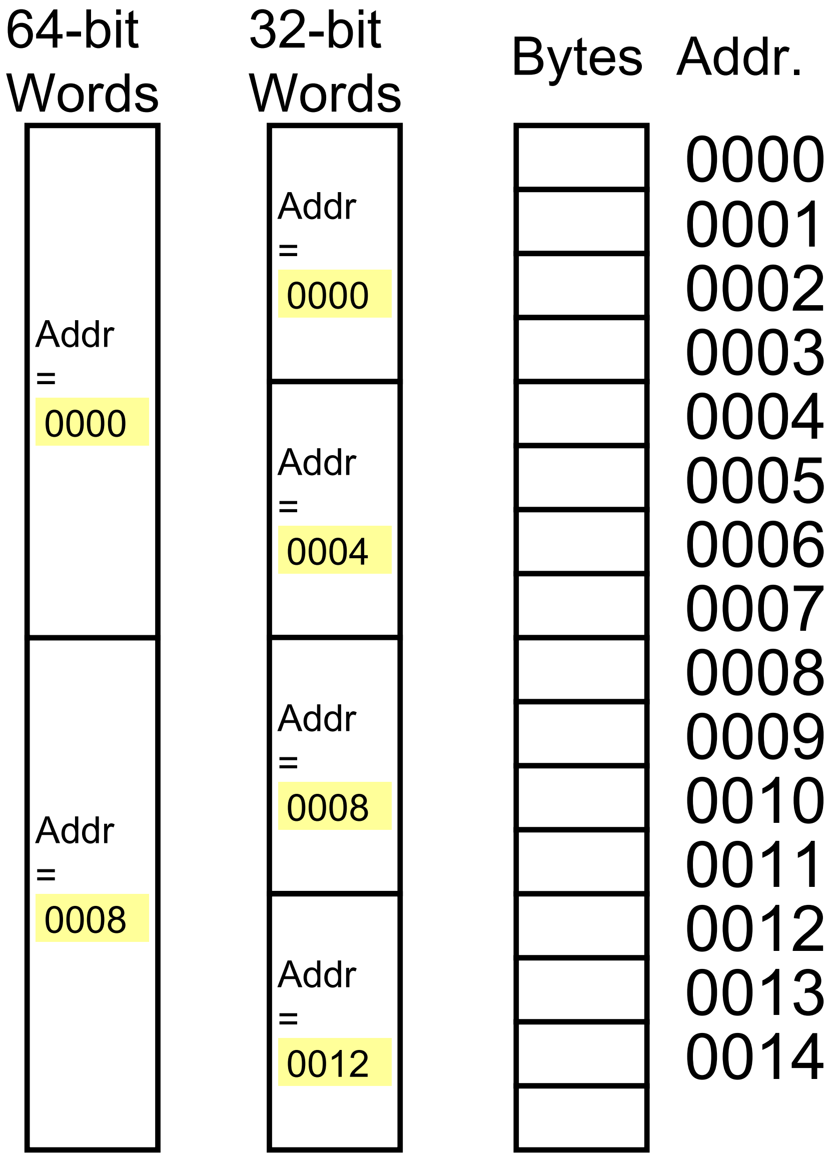
- Three bases programmers normally work in
  - Base 2: Binary
  - Base 10: Decimal
  - Base 16: Hexadecimal
- What do they mean?
  - Each digit is a representation of the base raised to a power
  - Decimal: $246_{10} = 2*10^2 + 4*10^1 + 6*10^0$
  - Binary: $11110110_2 = 1*2^7 + 1*2^6 + 1*2^5 + 1*2^4 + 0*2^3 + 1*2^2 + 1*2^1 + 0*2^0 = 246_{10}$
  - Hex: $F6_{16} = 0xF6 = 15*16^1 + 6*16^0 = 246_{10}$
- Easy way to convert between Binary and Hex
  1. Divide binary number into chunks of 4
  2. Convert each chunk of 4 binary digits into a hex number

     e.g. $11110110_2 = 1111\ \ 0110 = F\ \ 6 = F6_{16}$

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

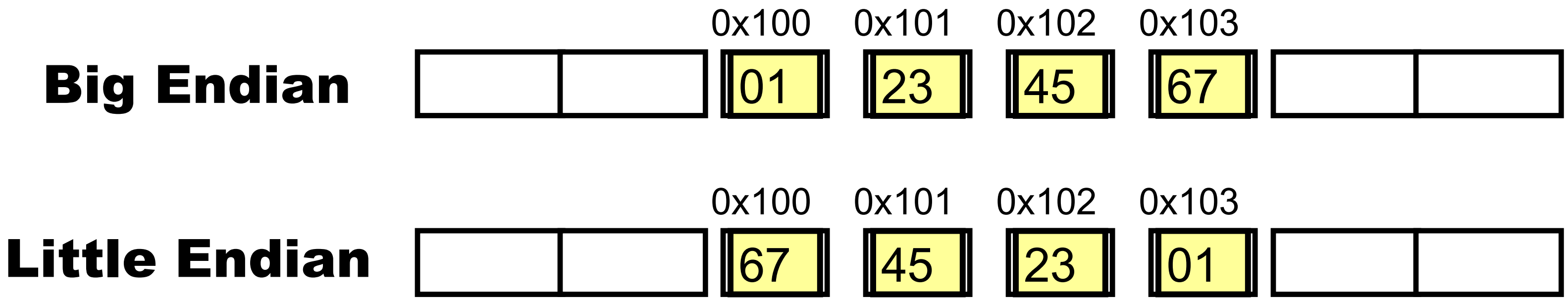
Memory Organization

- Each memory address references a particular byte in memory
- A 32-bit value (such as an int) is 4-bytes long. Therefore, it takes up 4 memory addresses. However, to reference this value, you look at the memory address of the first byte.
  - E.g. If address 0004 holds an int, addresses 0004, 0005, 0006, 0007 hold that int.



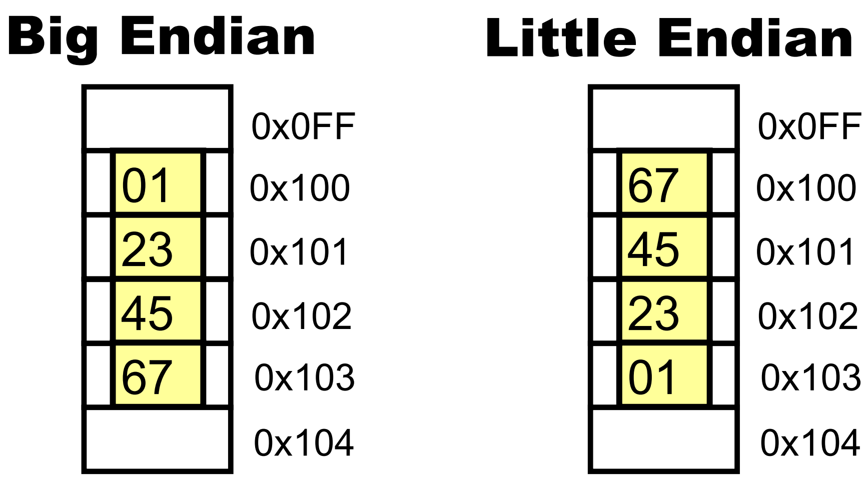
Byte Ordering Example

- 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

- Another way to visualize it



- **Little** Endian is **Least** significant byte first
- **Big** Endian is **Most** significant byte first

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
39	00
30	00
00	30
00	39

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

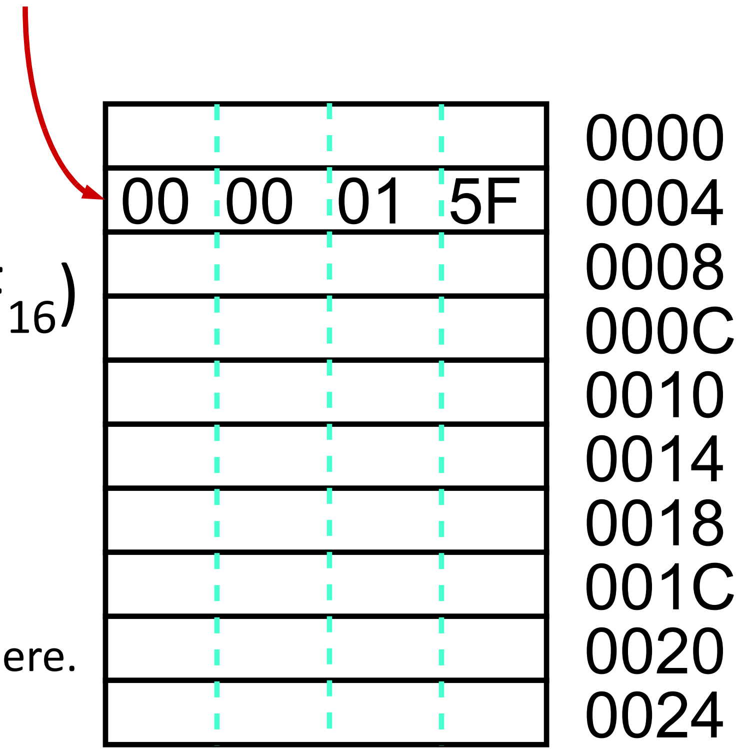
IA32 C	X86-64 C	Sun C
39	39	00
30	30	00
00	00	30
00	00	39
	00	
	00	
	00	
	00	

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

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 $15F_{16}$)
- In C:

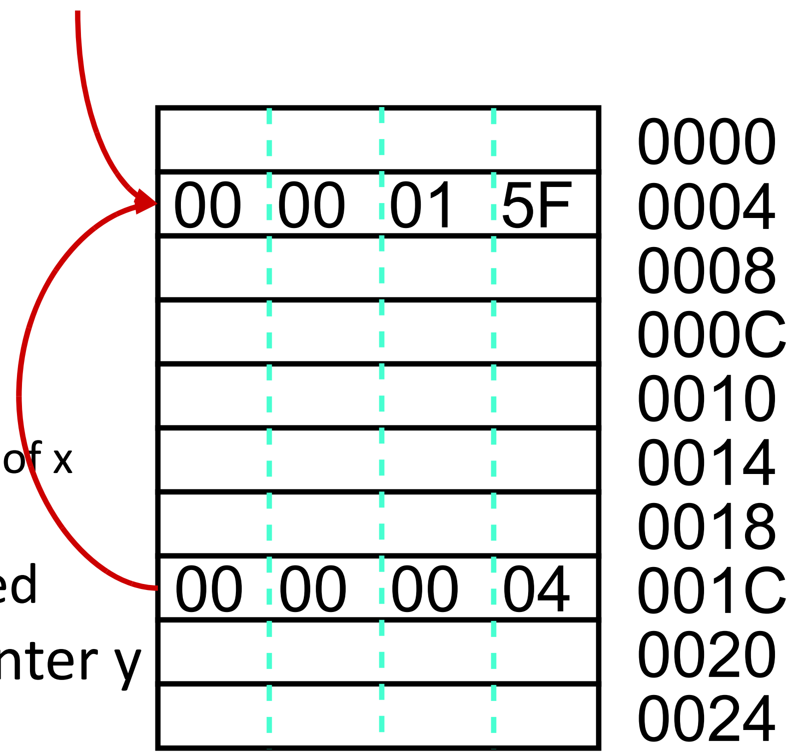
```
int x = 351;
//The compiler chooses to store x
//at address 0004. Could really be anywhere.
```



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
- C:

```
int x = 351;
int* y = &x;  //Pointer y is the address of x
```



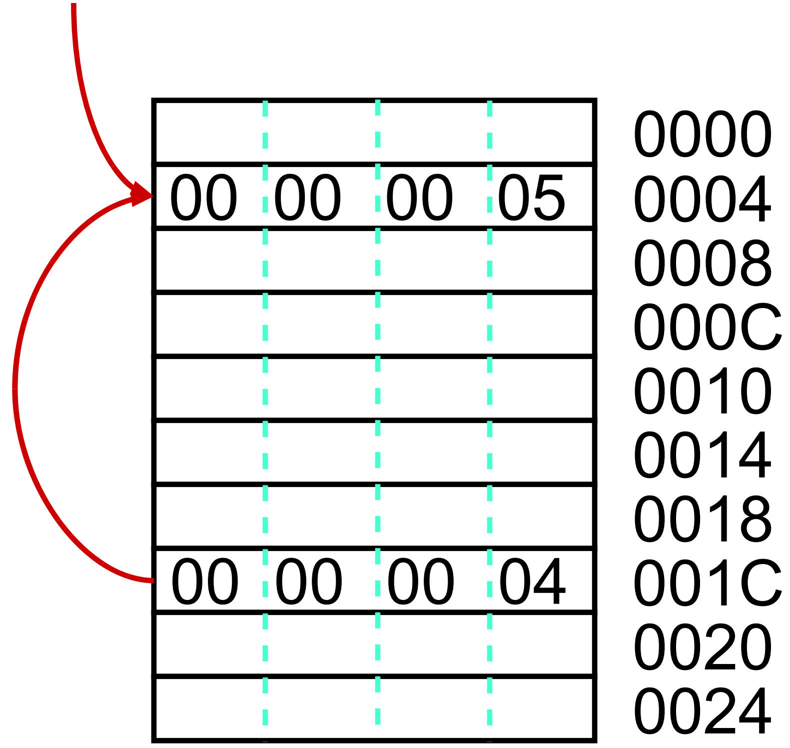
- That is, y **points to** where x is located
- Compiler chooses to put the pointer y
  at address 001C

Addresses and Pointers

- Update the value of x by using the pointer
- C:

```
int x = 351;
int* y = &x;
*y = 5;
```

- Read as "the **value of** the variable stored at the address in y gets 5". This is the same as doing "x=5"

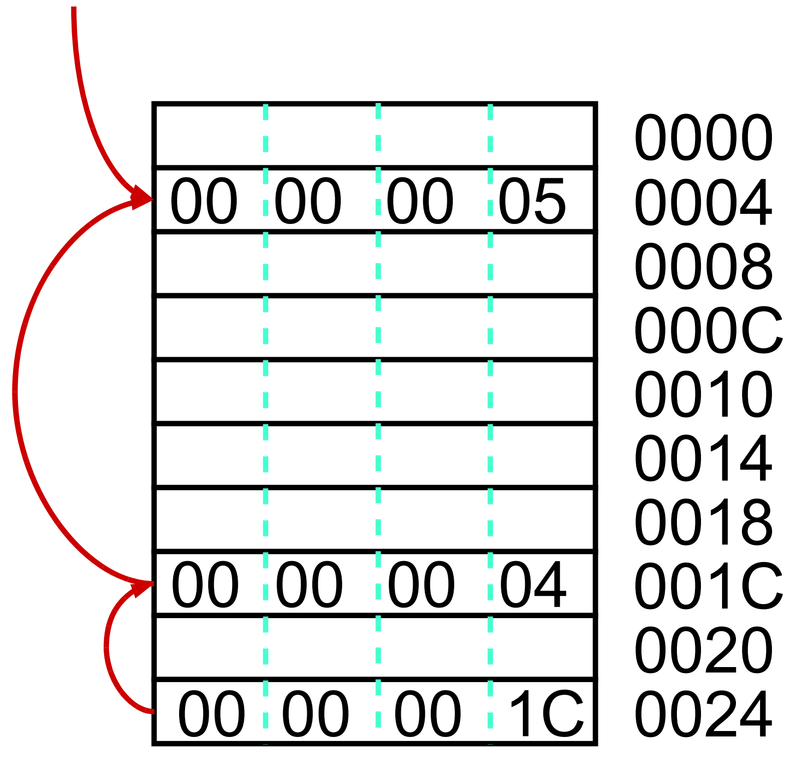


Addresses and Pointers

- Pointer to a pointer
  in 0024
- C:

```
int x = 351;
int* y = &x;
*y = 5;
int** z = &y;
```

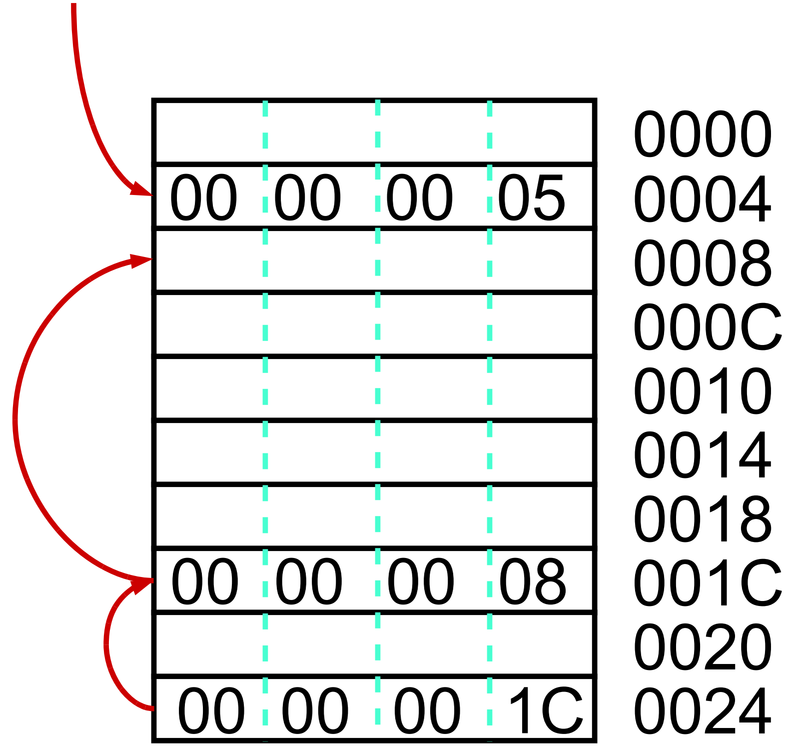
- Pointer z is stored at address 0024
  by the compiler.
- z points to y, and y points to x.
- Could do "**z" to get
  the value of x.



Addresses and Pointers

- What happens when you do y = y + 1?
- C:

```
int x = 351;
int* y = &x;
*y = 5;
int** z = &y;
y = y + 1;
```



- y gets the previous address of x plus 4 bytes (size of an int).
- y no longer points to x

HW 0

- http://www.cs.washington.edu/education/courses/cse351/12wi/homework-0.html

Questions? What were your results?