

# CPSC 213

## Introduction to Computer Systems

Summer Session 2019, Term 2

*Unit 1a – July 2*

***Memory and Numbers***

# Unit 1a Overview

## ▶ Reading

- Companion: 2.2.2
- Textbook: 2.1 - 2.3

## ▶ Learning Objectives ... you should be able to ...

- describe the relationship between bits, bytes, shorts, longs and quads
  - using the names for these things in Java, C, and assembly
- determine whether an address is aligned to a given size
- translate between integers and values stored in memory for both big- and little-endian machines
- evaluate and write Java expressions using bitwise operators (&, |, <<, >>, and >>>)
- determine when sign extension is unwanted and how to eliminate it in Java
- evaluate and write C expressions that include type casting and the addressing operators (& and \*)
- translate integer values by hand (no calculator) between binary and hexadecimal, subtract hexadecimal numbers and convert small numbers between binary and decimal

# Question 0.1

```
class A {  
    static int s = 0;  
    int i;  
    A () {  
        s = s + 1;  
        i = s;  
    }  
}
```

```
class Main {  
    static A a0, a1;  
  
    public static void main(String[] args) {  
        a0 = new A();  
        a1 = new A();  
        a0 = a1;  
        a1.i = a1.i + 1;  
        out.printf ("a0.i=%d a1.i=%d\n", a0.i, a1.i);  
    }  
}
```

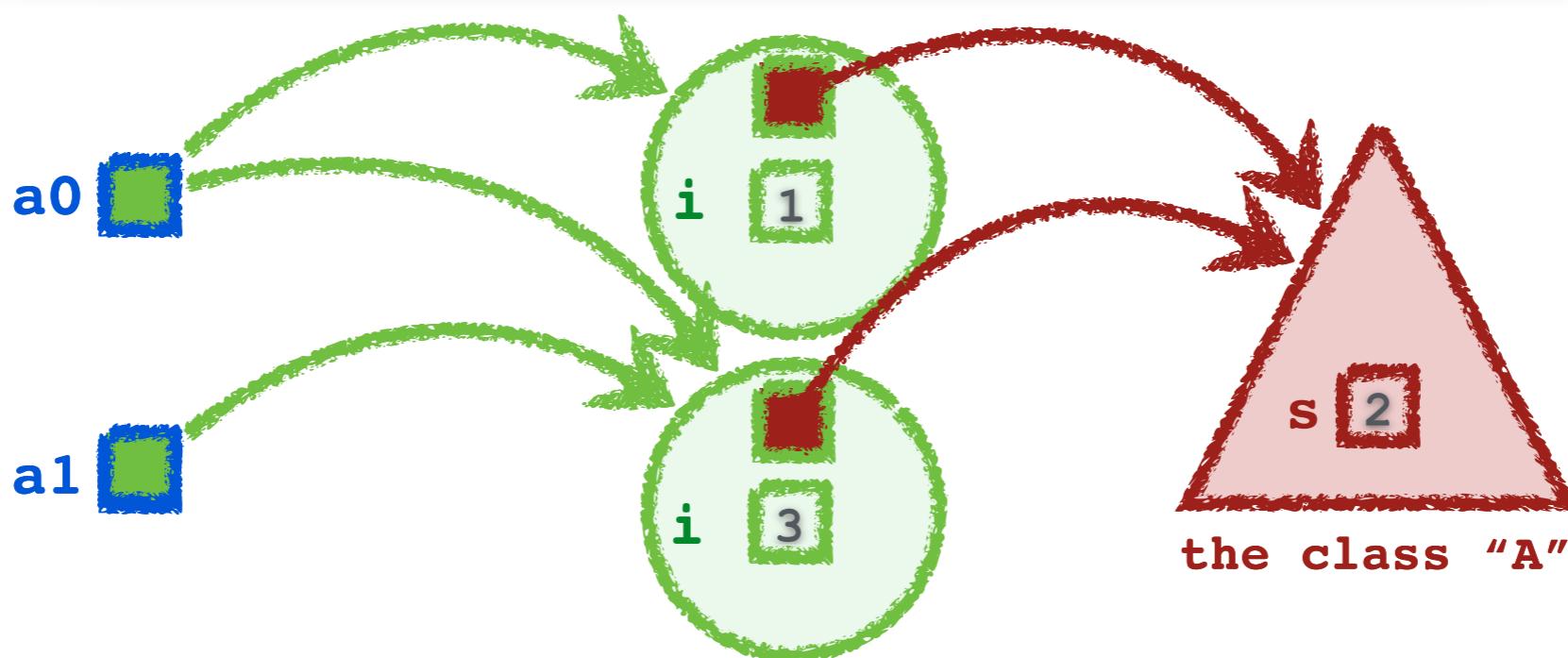
► What prints when **main()** executes?

- (A) a0.i=0 a1.i=2
- (B) a0.i=1 a1.i=3
- (C) a0.i=2 a1.i=3
- (D) a0.i=3 a1.i=3
- (E) none of the above
- (F) I don't know

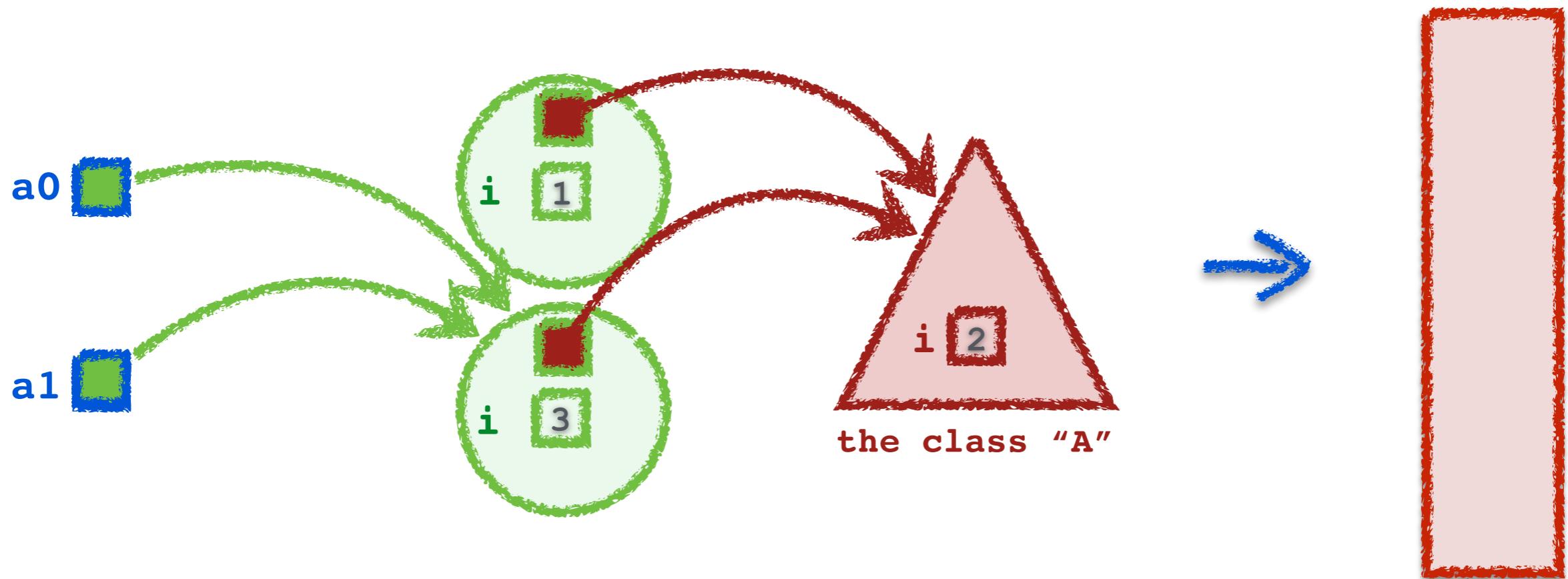
# Draw a Picture

```
class A {  
    static int s = 0;  
    int i;  
    A () {  
        s = s + 1;  
        i = s;  
    }  
}
```

```
class Main {  
    static A a0, a1;  
  
    public static void main(String[] args) {  
        a0 = new A();  
        a1 = new A();  
        a0 = a1;  
        a1.i = a1.i + 1;  
        out.printf ("a0.i=%d a1.i=%d\n", a0.i, a1.i);  
    }  
}
```



# In the Machine



- ▶ All state is stored in a single, main memory
  - each byte has a unique numeric address
- ▶ Variable
  - name for memory location that has an *address*, a *size* and a *value*
  - some are allocated *statically* (compiler) and some *dynamically* (execution)
- ▶ Pointer (aka Reference)
  - variable whose value is a memory addresses

# Question 0.2

```
class B {  
    int val;  
    B (int val) {this.val = val;}  
  
    int add (B x, B y) {  
        this.val = this.val + x.val;  
        this.val = this.val + y.val;  
        return this.val;  
    }  
  
    public static void main (String[] args) {  
        B a = new B (4);  
        B b = a;  
        B c = a;  
        out.println (a.add (b, c));  
    }  
}
```

► What prints when **main()** executes?

- (A) 4
- (B) 8
- (C) 12

- (D) 16
- (E) none of the above
- (F) I don't know

# Draw a Picture

```
class B {  
    int val;  
    B (int val) {this.val = val;}  
  
    int add (B x, B y) {  
        this.val = this.val + x.val;  
        this.val = this.val + y.val;  
        return this.val;  
    }  
  
    public static void main (String[] args) {  
        B i = new B (4);  
        B j = i;  
        B k = i;  
        out.println (i.add (j, k));  
    }  
}
```



# In the Machine



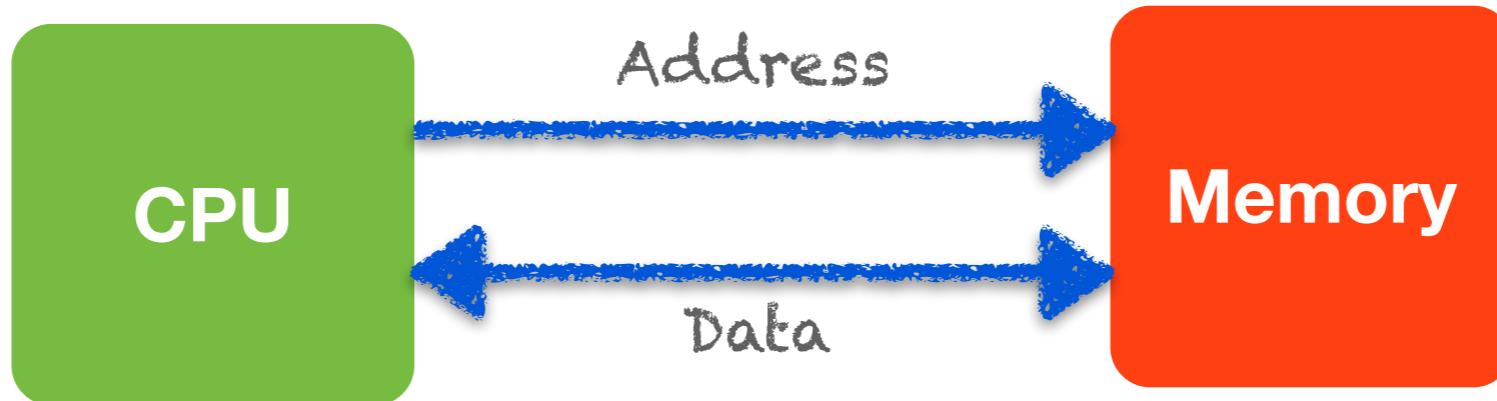
## ▶ There is just one object

- allocated storage space in memory when “new” executes
- object is “named” by its memory address — a number

## ▶ Pointer variables

- stored in memory too
- their value is the memory address of the object
- every variables stores the same number

# A Simple Computing Machine



## ▶ Memory

- stores data encoded as bits
- program instructions and state (variables, objects, etc.)

## ▶ CPU

- reads instruction and data from memory
- performs specified computation and writes result back to memory

## ▶ Example

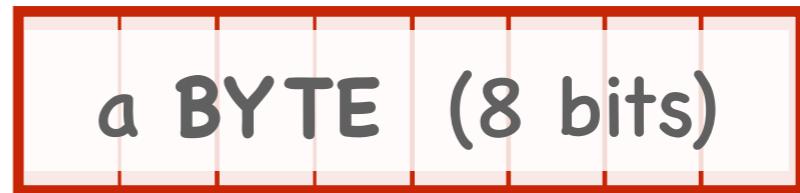
- $C = A + B$
- memory stores: ***add*** instruction, and variables ***A***, ***B*** and ***C***
- CPU reads instruction and values of *A* and *B*, adds values and writes result to *C*

# Memory

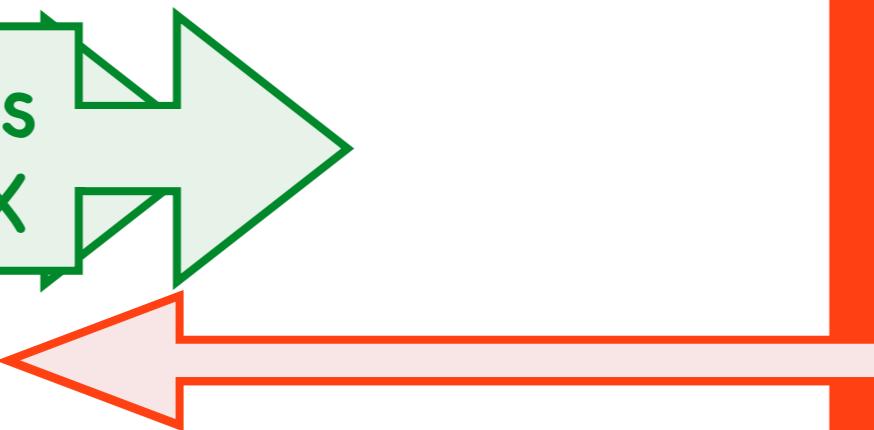
CPU

Memory is a big  
bag of BYTES

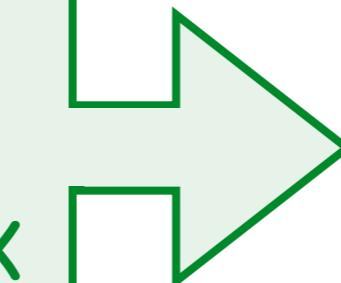
an ADDRESS  
names each byte  
(0..big number)



READ value of N BYTES  
starting at ADDRESS X

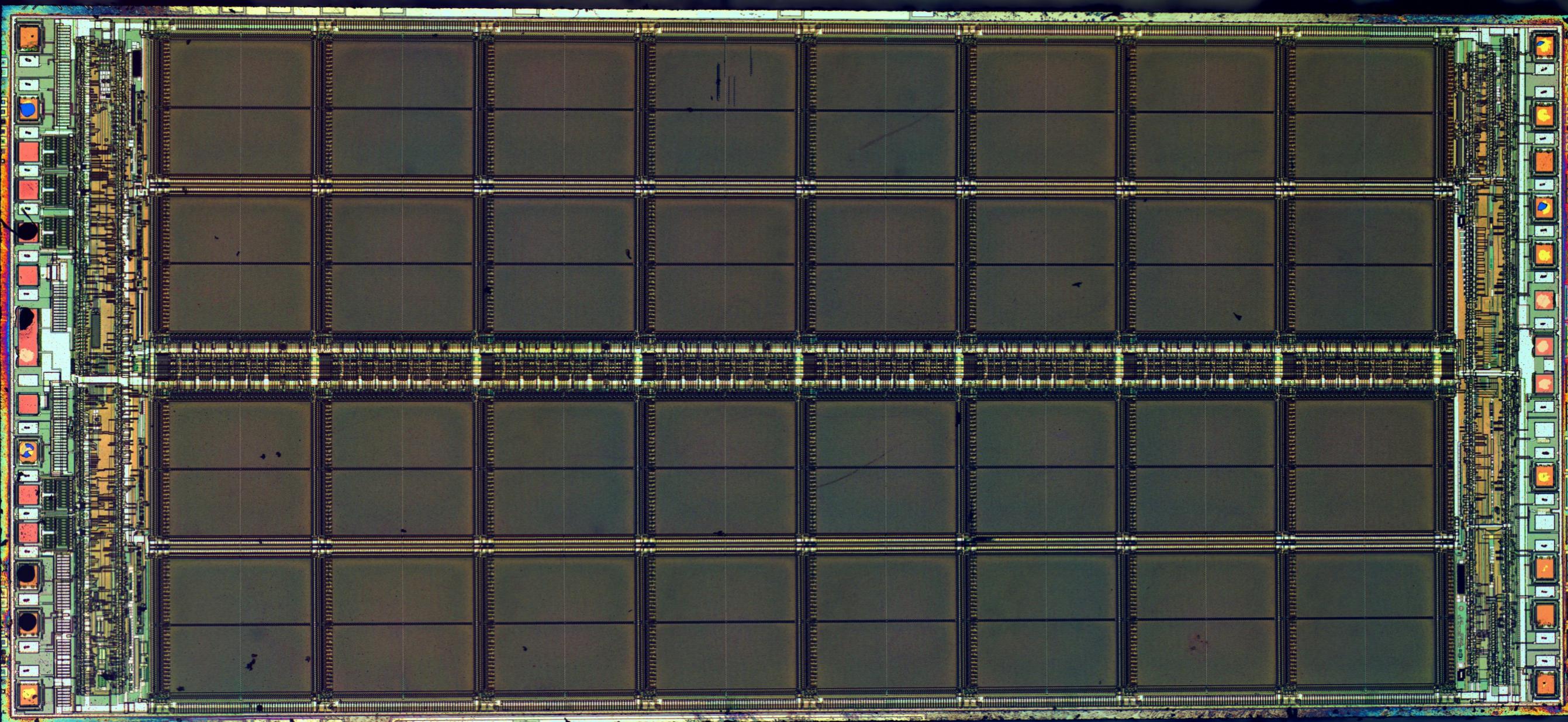


WRITE new values  
 $v_{0..N-1}$  into N BYTES  
starting at ADDRESS X



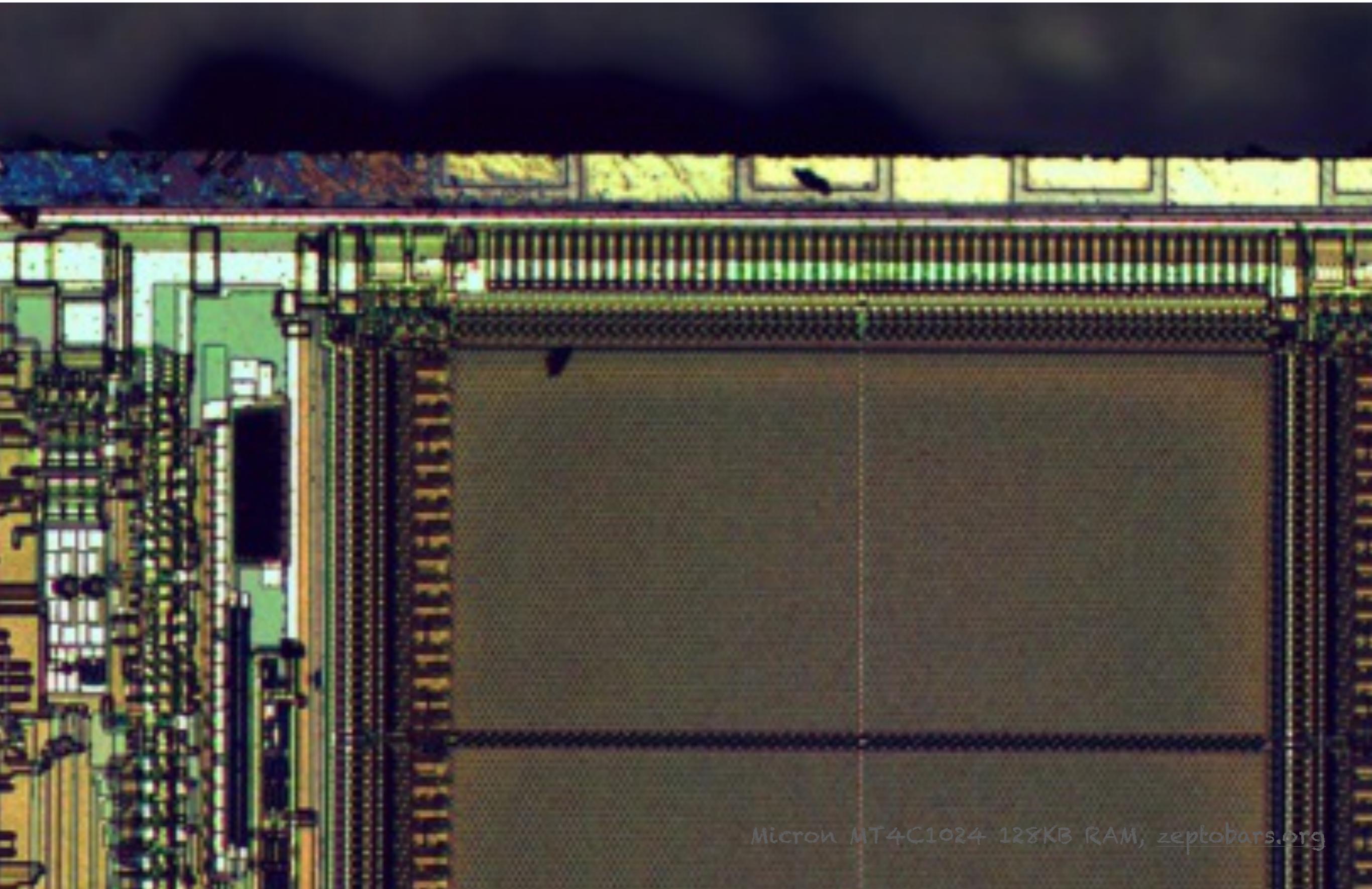
Memory

# Memory



Micron MT4C1024 128KB RAM, [zeptobars.org](http://zeptobars.org)

# Memory



Micron MT4C1024 128KB RAM, [zeptobars.org](http://zeptobars.org)

# Memory Summary

Memory

## ▶ Naming

- unit of addressing is a byte (8 bits)
- every byte of memory has a unique address
- some machines have 32-bit memory addresses, some have 64
  - our machine will have 32

## ▶ Access

- lots of things are too big to fit in a single byte
  - unsigned numbers > 255, signed numbers < -128 or > 127, most instructions, etc.
- CPU accesses memory in contiguous, power-of-two-size chunks of bytes

Integer Data Types by Size

# bytes	# bits	C	Java	Asm
1	8	char	byte	b byte
2	16	short	short	w word
4	32	int	int	l long
8	64	long	long	q quad

We will use only 32-bit integers

- address of a chunk is address of first byte

# First, Numbers and Humans



- ▶ Sometimes we are interested in integer value of chunk of bytes
  - base 10 is natural for this
- ▶ Sometimes we are more interested in bits themselves
  - memory addresses are big numbers that name power-of-two size things
  - we do not usually care what the base-10 value of an address is
  - we'd like a power-of-two sized way to name addresses
- ▶ We might use base-2, binary
  - a small 256-byte memory has addresses  $0_2$  to  $11111111_2$
  - if you don't have subscripts,  $11111111_2$  is written as **0b11111111**
  - but, as addresses get bigger, this is tedious
- ▶ Once we used base-8, octal
  - 64-KB memory addresses go up to  $1111111111111111_2 = 177777_8$
  - if you don't have subscripts,  $177777_8$  is written as **0177777**
  - but, as addresses got bigger, this got tedious too
- ▶ Now we use base-16, hexadecimal
  - 4-GB memory addresses go up to  $3777777777_8 = \text{ffffffffff}_{16}$
  - if you don't have subscripts,  $\text{ffffffffff}_{16}$  is written as **0xffffffff**

# Binary $\leftrightarrow$ Hex is Easy

01101010010101010000111010100011

# How many bits in a hex “digit”, a nibble?

0110 1010 0101 0101 0000 1110 1010 0011

Consider ONE nibble at a time:  $8 \times i_4 + 4 \times i_3 + 2 \times i_2 + 1 \times i_1$

6 a 5 5 0 e a 3

0x6a550ea3

Its easy to see the value of each byte (i.e., two nibbles)

0x6a 0x55 0x0e 0xa3

# Question 1a.1: Hexadecimal Notation

► Which of these statements is true

- A. The Java constants 16 and 0x10 are exactly the same integer
- B. 16 and 0x10 are different integers
- C. Neither
- D. I don't know

# Subtracting Hex Numbers

## ► We use Hex for addresses

- while we don't really care what their base-10 value is
- we sometimes compute the size of things by subtracting two addresses
  - and that will then be in decimal

## ► Subtracting in Hex

$$0x2000 - 0x1ff0 = ?$$

- you could convert both numbers to decimal, but that might be too hard
- you can subtract in hex
  - carry is  $0x10 == 16$
  - to subtract a..f digits convert to their decimal value

A diagram illustrating the subtraction of hex numbers. It shows three rows of hex digits: 0x2000, -0x1ff0, and the result 0010. Above the first row is a green arrow pointing right. Above the second row is another green arrow pointing right. Above the third row is a green arrow pointing right. The first row has a green underline under the 0. The second row has a green underline under the 0. The third row has a green underline under the 0. The first row has a green underline under the 0. The second row has a green underline under the 0. The third row has a green underline under the 0.

$$\begin{array}{r} 0x2000 \\ -0x1ff0 \\ \hline 0 \end{array} \quad \begin{array}{r} 0x1f00 \\ -0x1ff0 \\ \hline 0 \end{array} \quad \begin{array}{r} 0x1f00 \\ -0x1ff0 \\ \hline 0010 \end{array}$$

- or you can use two's compliment addition
  - negate a number by complimenting and incrementing it
    - $-x == !x + 1$
  - complimenting in hex is easy
    - swap 1's and 0's
    - need a quick switch to/from binary

$$\begin{aligned} !0x1ff0 &== 0xffffe00f \\ -0x1ff0 &== !0x1ff0 + 1 \\ &== 0xffffe010 \end{aligned}$$

$$\begin{aligned} 0x2000 - 0x1ff0 &== 0x2000 + 0xffffe010 \\ &== 0x10 \end{aligned}$$

## ► Converting Hex to Decimal

- not too bad for small numbers ... tedious for large ones
- $0xijkl = i*16^3 + j*16^2 + k*16^1 + l*16^0$

$$\begin{aligned} 0x10 &== 1*16 + 0 \\ &== 16 \end{aligned}$$

# Question 1a.2: Subtracting in Hex

- ▶ Object A is at address  $0x10d4$  and object B at  $0x1110$ . They are stored *contiguously* in memory (i.e., they are adjacent to each other). How big is A?
  - A. 16 bytes
  - B. 48 bytes
  - C. 60 bytes
  - D. 80 bytes
  - E. You can't tell for sure from the information given
  - F. I need a calculator
  - G. I don't know

# Making Integers from Bytes

Memory

## ► Our first architectural decision

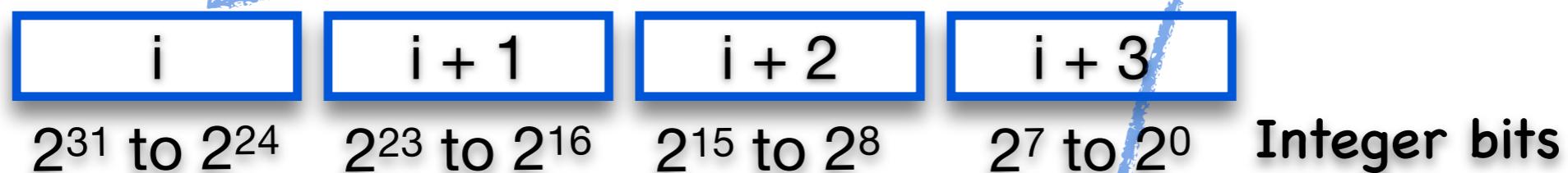
- assembling memory bytes into integers

## ► Consider 4-byte memory word and 32-bit integer

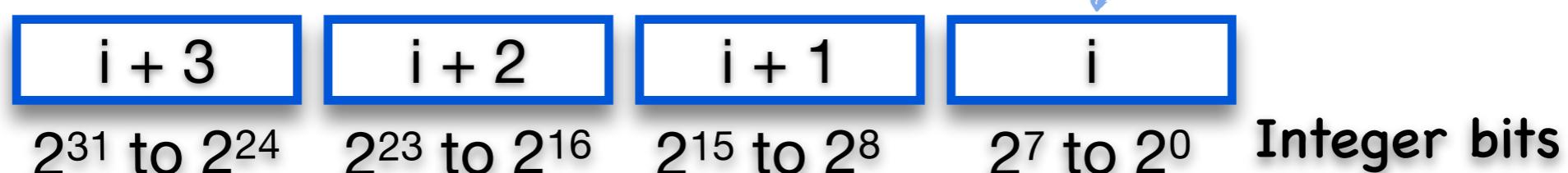
- it has memory addresses  $i$ ,  $i+1$ ,  $i+2$ , and  $i+3$
- we'll just say its "**at address  $i$  and is 4 bytes long**"
- e.g., the word at address 4 is in bytes 4, 5, 6 and 7.

## ► Big or LittleEndian

- we could start addressing at the BIG END of the number



- or we could start at the LITTLE END (Intel)



# Question 1a.3: Endianness

► What is the Little-Endian integer value at address 4 below?

- A. 0x1c04b673
- B. 0xc1406b37
- C. 0x73b6041c
- D. 0x376b40c1
- E. none of these
- F. I don't know

Memory	
Addr	value
0x0:	0xfe
0x1:	0x32
0x2:	0x87
0x3:	0x9a
0x4:	0x73
0x5:	0xb6
0x6:	0x04
0x7:	0x1c

# In Lab: Endianness.java

```
public static void main (String[] args) {  
    Byte mem[] = new Byte[4];  
    try {  
        for (int i=0; i<4; i++)  
            mem [i] = Integer.valueOf (args[i], 16) .byteValue();  
    } catch (Exception e) {  
    }  
  
    int bi = bigEndianValue (mem);  
    int li = littleEndianValue (mem);  
    ...  
}
```

Load 4 byte values provided on  
“command line” into memory in  
sequence.

## ▶ Complete this program

- implement bigEndianValue and littleEndianValue

## ▶ Run in for various byte sequences

- four **command-line** arguments are for consecutive byte values, in hex

- for example typing the following at UNIX shell command line

- java Endianness 0 0 0 1

should print big-endian value of 1

- java Endianness 1 0 0 0

should print little-endian value of 1

- java Endianness ff ff ff ff

should print value of -1 for both

# Some addresses are better than others

## ▶ Address Alignment

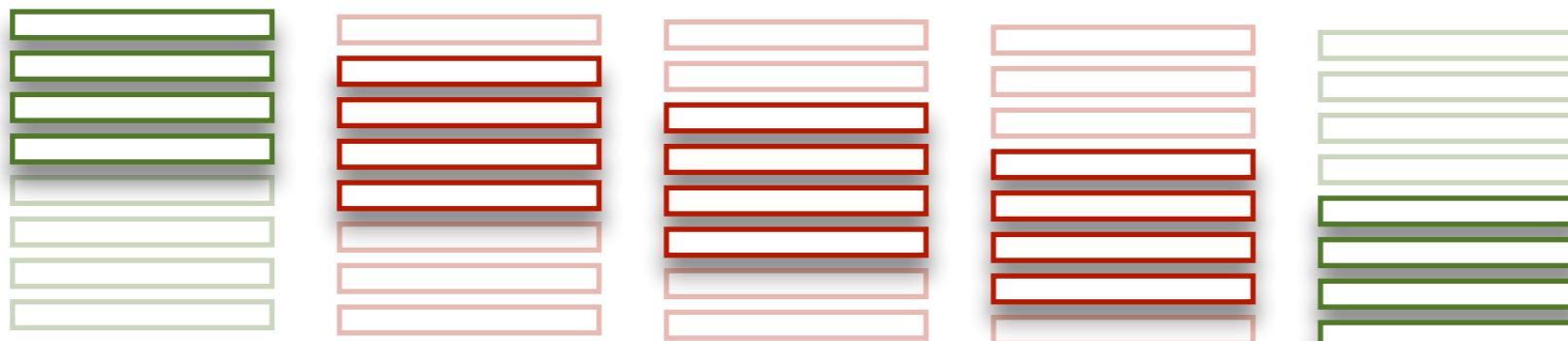
- we could allow any number to address a 4-byte integer, e.g.



- \* disallowed on many architectures
- \* allowed on Intel, but usually slower example ....

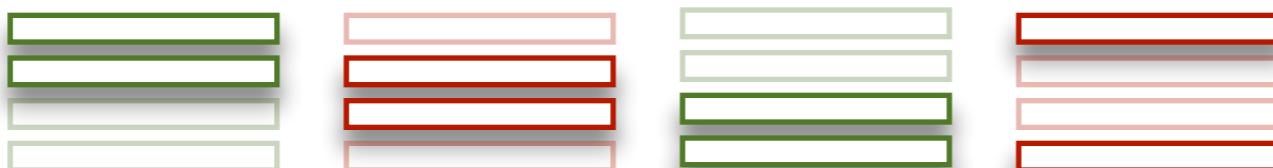
- but, better for hardware is to require addresses be aligned

- an address is aligned to size  $x$  if the address mod  $x$  is zero



## ▶ Alignment to power-of-two size

- smaller things always fit completely inside of bigger things



e.g., a word contains exactly two complete shorts

- address computations are achieved by shifting bits; e.g., array-element address from index  $\&a[i] == \&a[0] + i * (s == 2^j) == \&a[0] + i << j$

# Advantages of Power-of-Two Alignment

## ▶ Memory Implementation Detail (*simplified*)

- memory is actually organized internally into larger chunks called *blocks*
- lets say a block is 16 bytes
- every memory access, internally requires accessing one of these blocks
- you'll see in 313 that this relates to memory caches

## ▶ Anyway ...

- a CPU memory access looks like this
  - Read/Write **N bytes** starting at **address A**
- the memory converts this to
  - R/W **N bytes** starting at **O<sup>th</sup> byte of block B** (**O** is the *block offset* and **B** is the *block number*)
  - blocks are numbered, such that block 0 contains addresses 0 .. 15
- do the calculation
  - $(B, O) = f(A)$
- how is this simplified IF
  - N is a power of 2 and
  - A is aligned (i.e.,  $A \bmod N == 0$ )?

# Question 1a.4: Alignment

- ▶ Which of the following statement (s) is (are) true?
  - A. the address 0x6 ( $110_2$ ) is aligned for addressing a *short*
  - B. the address 0x6 ( $110_2$ ) is aligned for addressing an *int* (i.e., 4-bytes)
  - C. the address 0x14 ( $10100_2$ ) is aligned for addressing an *int*
  - D. the address 0x14 ( $10100_2$ ) is aligned for addressing a *long* (i.e., 8-bytes)

# Shifting Bits

## ► Shifting multiplies or divides by power of 2



## ▶ But, what about negative numbers

- recall that negative numbers are represented in ***two's compliment*** form
  - $-6 == 0xfa == 11111010_2$  (i.e.,  $11111010_2 + 00000110_2 == 0$ )
  - $-6 / 2 == -3$ , but  $11111010_2$  shifted right is  $0111101_2 == 125$ , not -3

- ▶ There are two kinds of right shifts

- SIGNED “**>>**” shifts, but keeps high-order bit (the sign bit) the same
    - $-6 \gg 1 == 11111101_2 == -3$  (i.e.,  $11111101_2 + 00000011_2 == 0$ )
  - UNSIGNED “**>>>**”, shifts and sets high-order bit to 0
    - $-6 \ggg 1 == 01111101_2 \dots 0xfa \ggg 1 == 0x7d$
  - In Java you choose. In C the compiler chooses.
    - C has both signed and unsigned integer data types and no “**>>>**”. Java has only signed.

# Extending an Integer

## ► Extending is

- when you increase the number of bytes used to store an integer

```
byte b = -6;  
int i = b;  
out.printf ("b: 0x%x %d, i: 0x%x %d\n", b, b, i, i);
```

- what prints? b: 0xfa -6, i: 0xfffffffffa -6

## ► Signed Extension

- used with signed numbers (everything in Java is signed)
- copies sign bit into upper, empty bits of the extended number

## ► Zero Extension

- used with unsigned numbers (e.g., in C)
- sets upper, empty bits to 0
- you can force zero-extension by **MASKING** using logical, bit-wise AND:

```
int u = b & 0xff;
```

```
out.printf ("u: 0x%x %d\n", u, u);
```

u: 0xfa 250

# Truncating an Integer

## ► You can also go the other way

- more bits to fewer bits

```
int i = -6;  
byte b = i;  
out.printf ("b: 0x%x %d, i: 0x%x %d\n", b, b, i, i);
```

- what could go wrong?

- If i is 256, what is b? What if i is 128?

## ► Java warns you

- if you truncate an integer without an explicit cast as above
  - “Possible Loss of Precision”
- you get rid of the warning by explicitly casting
  - the cast has no effect on the value of b
  - it just tells the compiler that you know what your doing ... obviously, be sure you do

```
int i = -6;  
byte b = (byte) i;  
out.printf ("b: 0x%x %d, i: 0x%x %d\n", b, b, i, i);
```

# Questions 1a.5 and 1a.6: Shift and Mask

1. What is the value of i after this Java statement executes?

```
int i = ((byte) 0x8b) << 16;
```

- A. 0x8b
- B. 0x0000008b
- C. 0x008b0000
- D. 0xff8b0000
- E. None of these
- F. I don't know

2. What is the value of i after this Java statement executes?

```
i = 0xff8b0000 & 0x00ff0000;
```

- A. 0xffff0000
- B. 0xff8b0000
- C. 0x008b0000
- D. None of these
- E. I don't know