CPSC 213: Introduction to Computer Systems

Unit 1a: Numbers and Memory

All slides adapted from materials by Mike Feeley, Jonatan Schroeder, Robert Xiao, and Jordon Johnson

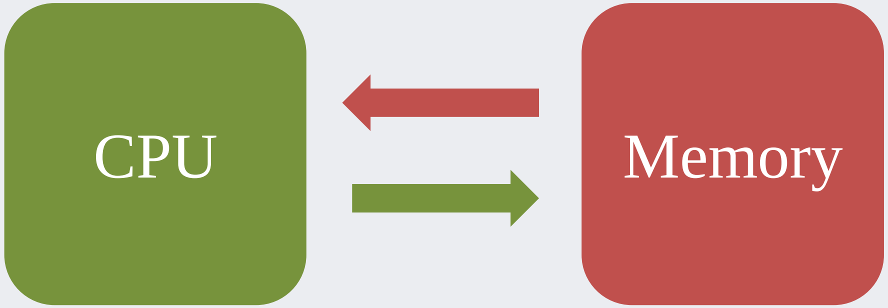
Overview

- Reading
  - Companion: 2.2.2
  - Textbook: 2.1 – 2.3

- Learning objectives:
  - know the number of bits in a byte and the number of bytes in a short, long, and quad
  - 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 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

A simple computing machine

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



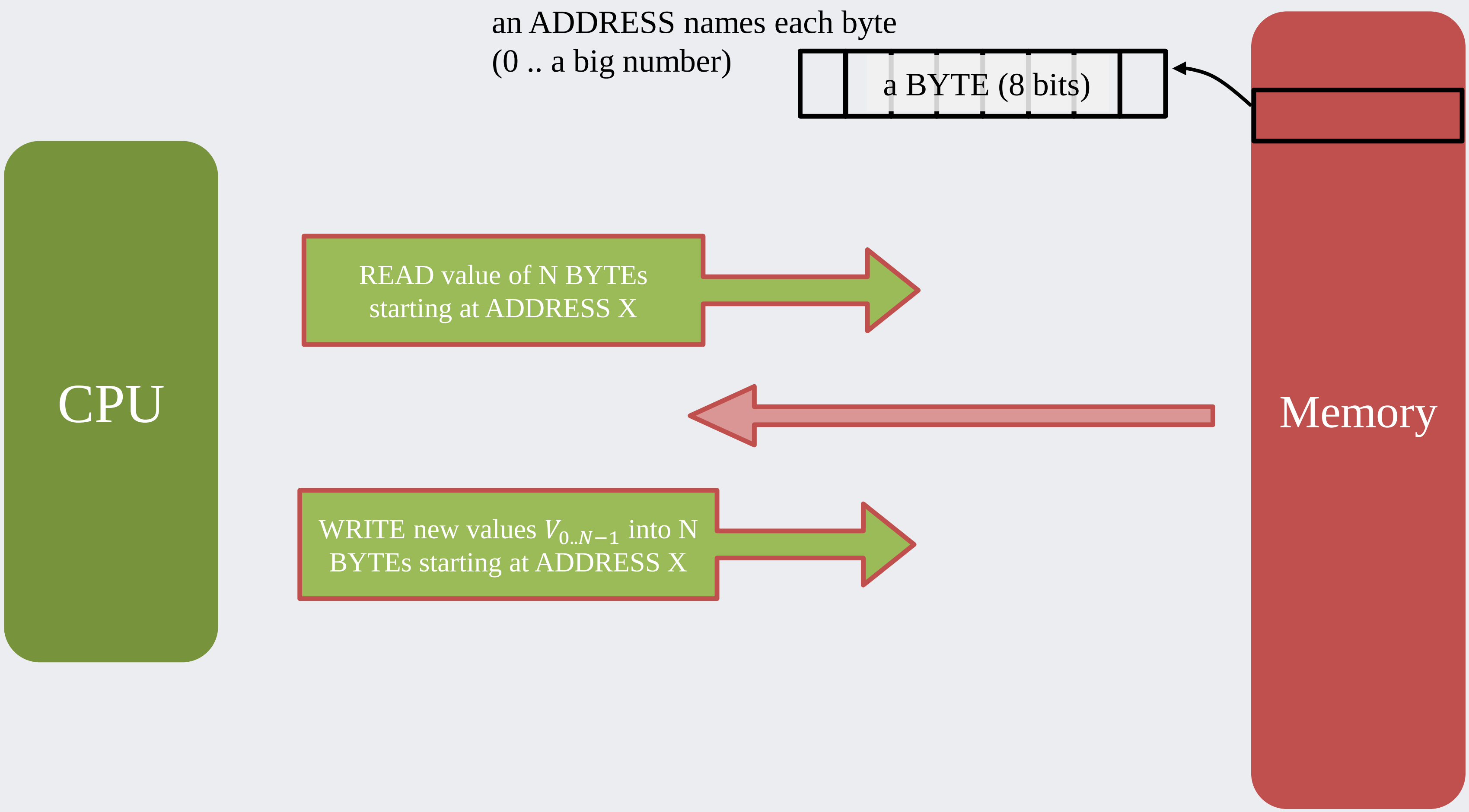
- CPU
  - reads instruction and data from memory
  - performs specified computation and writes result back to memory

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

Memory

Abstract view

Memory is a big bag of BYTEs



Memory

- Naming
  - unit of addressing is byte (8 bits)
  - every byte of memory has an unique address
  - some machines have 32-bit memory addresses, some have 64
    - our machine for this class will use 32-bit addresses

- Access
  - many things are too big to fit in a single byte
    - e.g. unsigned numbers > 255, signed numbers < -128 or > 127, most instructions, etc.
  - CPU accesses memory in contiguous, power-of-two-sized chunks of bytes
  - address of a chunk is address of its first byte

Memory

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

Numbers and representation

01101101001011110100110010100111 → ?

- Sometimes we are interested in the integer value of a chunk of bytes
  - base-10, decimal, is commonly used to represent this number (our "normal" number system)
  - we need to convert from binary to decimal to get this value

- Sometimes we are more interested in the bits themselves
  - In such cases the decimal value isn't particularly important
  - For example, consider memory addresses
    - big numbers that name power-of-two-sized things
    - we do not usually care what the base-10 value of an address is
    - we can use a power-of-two sized way to identify addresses

Numbers and representation

Identification using powers of 2

- We might use base-2, binary
  - a small 256-byte memory has addresses 0_2 to 11111111_2
  - may be represented as $\mathbf{0b}11111111$
  - becomes tedious and hard to read as addresses get larger

- Once upon a time we used base-8, octal
  - 64KB memory addresses go up to $1111111111111111_2 = 177777_8$
  - may be represented as $\mathbf{0}177777$
  - also got tedious as address sizes increased

- Now we use base-16, hexadecimal
  - 4GB memory addresses up to $11111111111111111111111111111111_2 = \text{ffffffff}_{16}$
  - we can write this as **0x**ffffffff

Binary ↔ hexadecimal

01101010010101010000111010100011

- 4 bits in a hex "digit" (hexit)

0110 1010 0101 0101 0000 1110 1010 0011

- Consider one hexit at a time

6 a 5 5 0 e a 3

0x6a550ea3

- A byte (8 bits) is just 2 hexits ($2^8 = 16^2$):

0x6a550ea3 → 0x6a 0x55 0x0e 0xa3

32 bits (4 bytes) 0x35D

pad with 0

bin	hex
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

iClicker 1a.1

Which of the following statements is true?

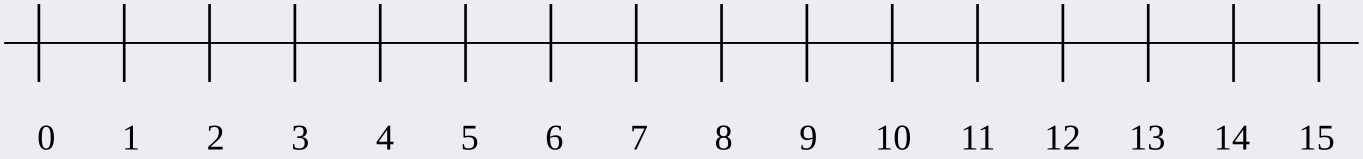
A. The Java constants `16` and `0x10` are exactly the same integer
B. The Java constants `16` and `0x10` are different integers
C. Neither of the statements above is always true
D. I don't know
E. I'm just choosing E for participation credit

Hexadecimal operations

$$\begin{array}{r} 18_{10} \\ -\,10_{10} \\ \hline 08_{10} \end{array}$$

- We use hexadecimal for memory addresses
  - base-10 value is unimportant
- Addition/subtraction in hex
  - can convert both to decimal, add/subtract, and convert back to hex (tedious)
  - can calculate in hex directly
  - alternative for subtraction: use addition with two's complement

$$\begin{array}{r} \overset{0}{\cancel{1}}\hphantom{0} \\ 0\text{x}\cancel{1}2 \\ -\,0\text{x}0\text{A} \\ \hline 0\text{x}08 \end{array}$$



- Carry for addition, and borrow for subtraction work the same way as with decimal, on a number line with 16 values decimal

0 1 2 3 4 5 6 7 8 9

$$\begin{array}{r} 5 \\ +\ 6 \\ \hline 11 \end{array}$$

10^2 10^1 10^0

$$\begin{array}{r} \overset{2}{\cancel{3}}3 \\ -\ 17 \\ \hline 16 \end{array}$$

hex

0 1 2 3 4 5 6 7 8 9 A B C D E F

$$\begin{array}{r} 9_{16} \\ +\ 9_{16} \\ \hline 12_{16} \end{array}$$

16^2 16^1 16^0

$$\begin{array}{r} 33_{16} \\ -\ 17_{16} \\ \hline 1C_{16} \end{array}$$

iClicker 1a.2

- Object A is at address `0x10d4`, and object B is at `0x1110`. They are stored contiguously in memory (i.e. they are adjacent to each other without any gap). How big is A?

A. 16 bytes

B. 48 bytes

C. 60 bytes

D. 80 bytes

E. Not enough information to tell

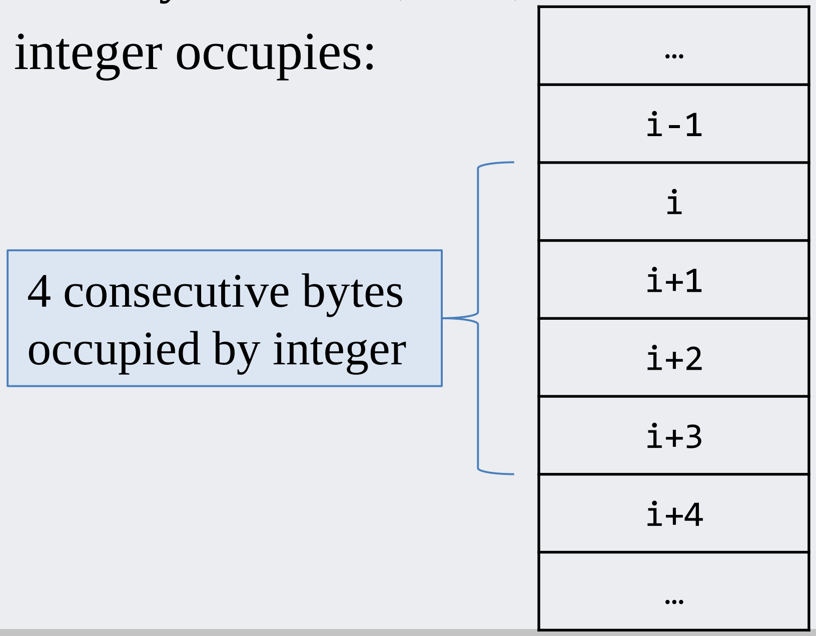
$$\begin{array}{r} 0x\,1110 \\ -\;0x\,10d4 \\ \hline 0x\,003C \end{array}$$

3×16^1 $\quad 12 \times 16^0$

$48 + 12 = 60$ bytes

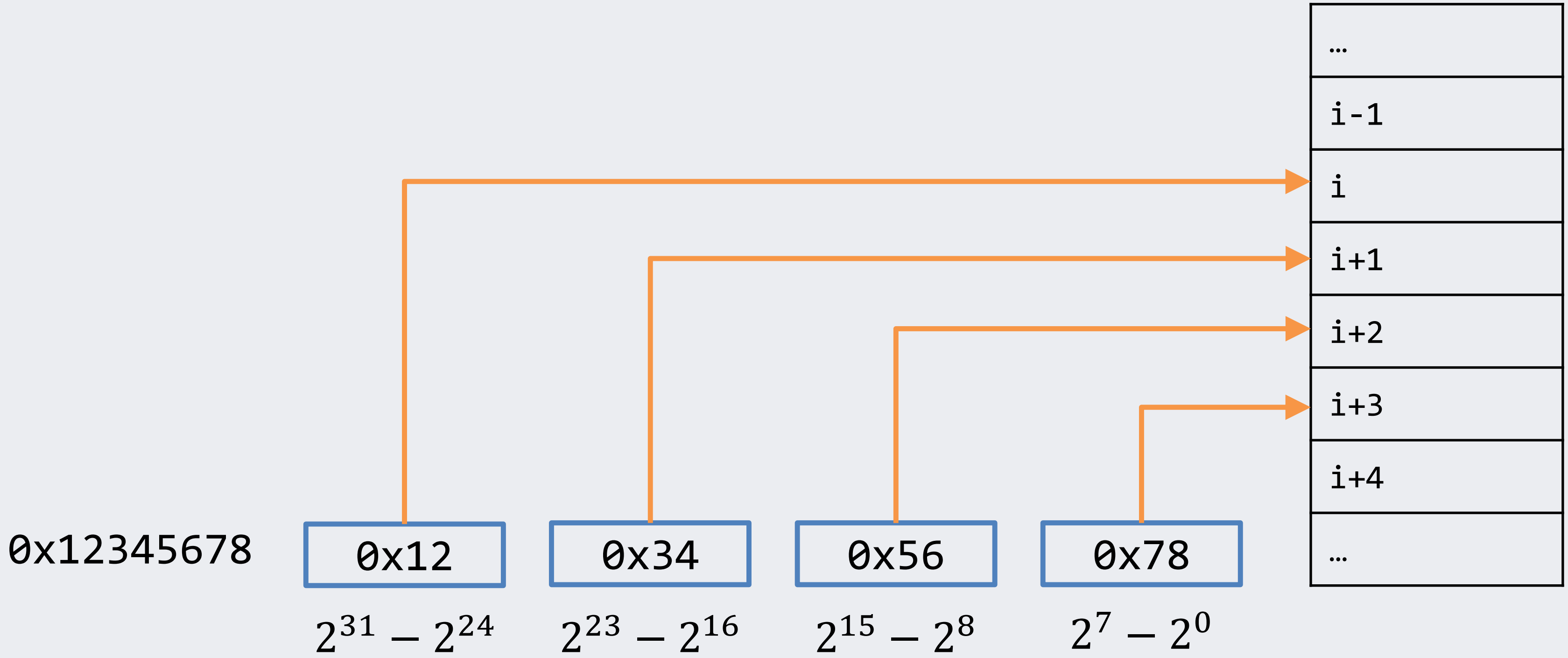
Making integers from bytes

- Each memory address holds 1 byte (8 bits)
- First architectural decision:
  - assembling multiple bytes of memory into integers
- Consider a 32-bit integer (`int` in Java or C)
  - this must occupy 4 bytes in memory
  - If memory address is `i`, then we also need the bytes at `i+1`, `i+2`, and `i+3`
  - Example: if address is `0x2014`, then the integer occupies:
    - `0x2014, 0x2015, 0x2016, 0x2017`
  - What do each of these bytes represent?



Big-endian

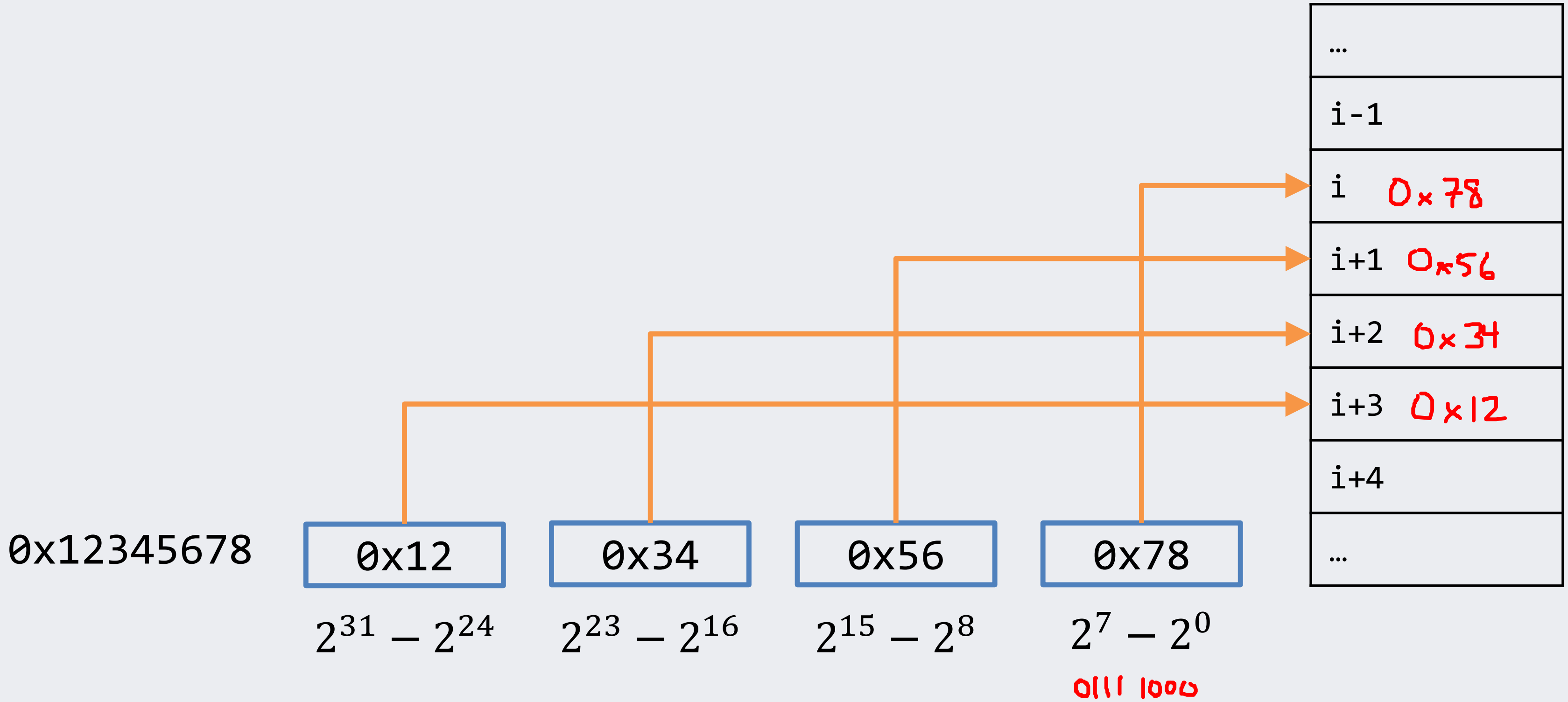
- Integer needs to be chopped into 4 bytes and stored in some order
- We can start at the "big end"
  - The first byte stored is the most significant byte
  - Used in old IBM servers, network connections



1_{10} 0x 00 00 00 01

Little-endian

- Integer needs to be chopped into 4 bytes and stored in some order
- We can start at the "little end"
  - The first byte stored is the least significant byte
  - Used in most Intel-based systems



iClicker 1a.3

aaa

- What is the Little-endian 4-byte integer value at address `0x4`?

A. `0xc1406b37`
B. `0x1c04b673`
C. `0x73b6041c`
D. `0x376b40c1`
E. `0x739a8732`

Address	Value
0x0:	0xfe
0x1:	0x32
0x2:	0x87
0x3:	0x9a
0x4:	0x73
0x5:	0xb6
0x6:	0x04
0x7:	0x1c

Least significant

most significant

0 x 1c 04 b6 73

most sig.

least sig.

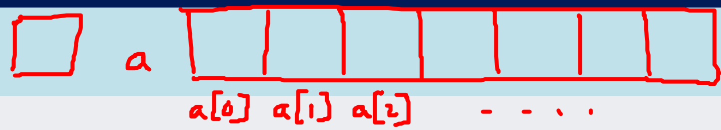
Can we just put data anywhere?

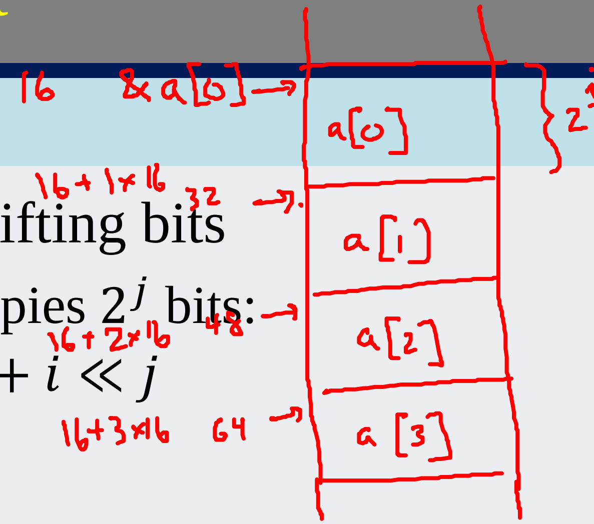
- We could place a 4-byte integer at (almost) any address

- However, forcing addresses to be aligned is better for hardware
  - Address whose numeric value is a multiple of the object size
    - It depends on the object; it gets slightly more complicated with arrays and structs
- Aligned addresses are better – smaller things fit inside larger things
  - Two `short`s fit inside an `int`, etc.
  - This is significant for arrays

Some CPUs don't support misaligned addresses!

Power-of-two alignment





- Address computation in arrays is achieved by shifting bits
  - e.g. accessing element i, where each element occupies 2^j bits:
    $\&a[i] == \&a[0] + i \cdot \left(size == 2^j\right) == \&a[0] + i \ll j$

- Memory implementation detail (simplified)
  - memory is actually organized internally into larger chunks called *blocks*
  - suppose a block is 16 bytes
  - then every memory access, internally requires accessing one of these blocks
  - you will see in 313 that this relates to memory caches (cool!)

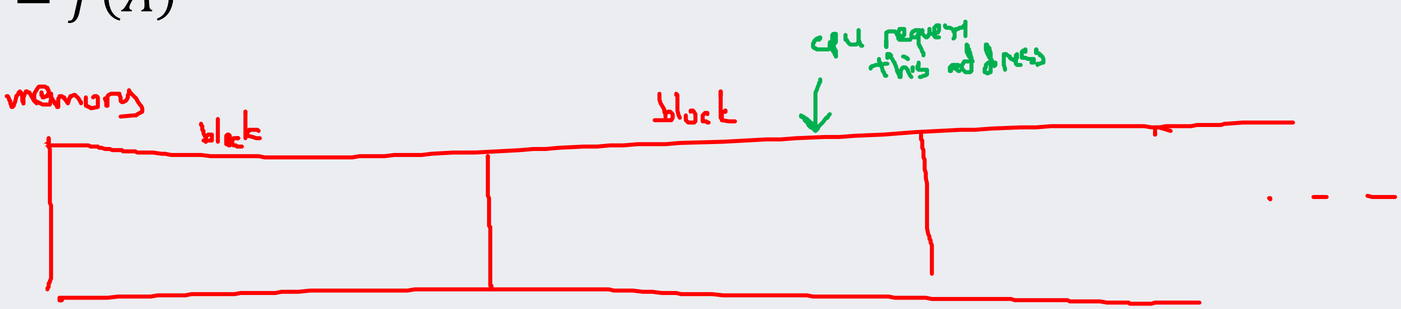
Alignment and memory access

- Anyway… a CPU memory access looks like this:
  - Read/write N bytes starting at address A

- The memory (with 16-byte blocks) converts this to:
  - Read/write N bytes starting at the O^{th} 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

- The block number and offset are obtained from the calculation:

$$(B, O) = f(A)$$



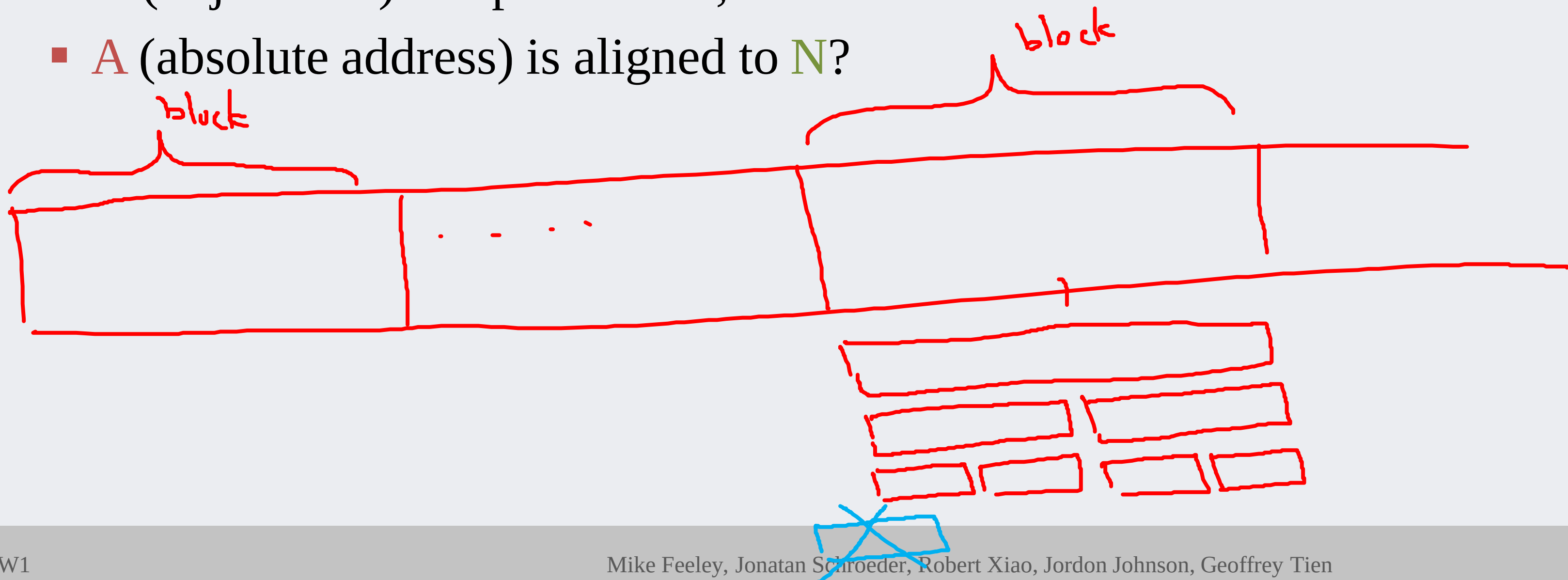
Alignment and memory access

Blocks and offsets

(from previous slide):

- convert a request of N bytes starting at address A, into
- N bytes from the O^{th} byte of block B

- How is this simplified if:
  - block size (in bytes) is a power of 2, and
  - N (object size) is a power of 2, and
  - A (absolute address) is aligned to N?



- Which of the following statement(s) is/are true?

A. The address 6 (110_2) is aligned for addressing a `short` 2 bytes

B. The address 6 (110_2) is aligned for addressing an `int` (i.e. 4 bytes)

C. The address 20 (10100_2) is aligned for addressing an `int`

D. The address 20 (10100_2) is aligned for addressing a `long` (i.e. 8 bytes)

E. More than one of the above are true

binary: _ _ 0 divisible by 2

_ 0 0 " " 4

0 0 0 " " 8

iClicker 1a.5

- Which of the following statement(s) is/are true?

A. The address `0x14` is aligned for addressing a `short`, but not an `int`
B. The address `0x14` is aligned for addressing a `short` and `int`, but not a `long`
C. The address `0x14` is aligned for addressing `short`, `int`, and `long`
D. None of the above
E. P-A-R-T-I-C-I-P-A-T-I-O-N!

$0001\ 0100_2$ aligns with short & int

Shifting bits

- Shifting multiplies or divides by a power of 2
  - shifting left by b bits is the same as multiplying by 2^b
  - shifting right by b bits is the same as dividing by 2^b

$0x6 \ll 1 == 110_2 \ll 1$
$== 1100_2$
$== 0xC$

```
0x6 >> 1 == 110₂ >> 1
         == 11₂
         == 0x3
```

- What about negative numbers?
  - Recall that negative numbers are represented as two's complement
  - –6 == 0xfa == 11111010_2
  - –6 / 2 == –3, but 11111010_2 shifted right is 01111101_2 == 125, not –3

Shifting bits

Java vs C $94_{10} >> 1 == 47_{10}$

$0101\ 1110_2 >> 1 == 0010\ 1111_2$

- Java has two kinds of right shifts
  - SIGNED shift ">>", keeps the most significant bit the same
    - e.g. –6 >> 1 == $1111\ 1010_2$ >> 1 == $1111\ 1101_2$ == –3
  - UNSIGNED shift ">>>", shifts and sets most significant bit to zero
    - e.g. –6 >>> 1 == $1111\ 1010_2$ >>> 1 == $0111\ 1101_2$ == 0x7d

$47_{10} / 4 == 47_{10} >> 2 == 0010\ 1111_2 >> 2 == 0000\ 1011_2 == 11_{10}$

- Java: you choose the type of shift by using >> or >>>
- C: compiler chooses based on variable's declared type
  - `unsigned int u;    int s;`

`unsigned` keyword creates an unsigned value

$55_{10} << 1 == 0011\ 0111_2 << 1 == 0110\ 1110_2 == 110_{10}$

What about left shifts?

$110_{10} << 1 == 0110\ 1110_2 << 1 == 1101\ 1100_2 ==$

8 bits 2's complement [-128, 127]

Extending integers

b: 1111 1010₂

- Extending is: i ← 1111 1010₂
  - increasing the number of bits (or 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: 0xfffffffa, -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
  - can be forced by masking using logical (bitwise) AND operator

b: 1111 1010
Sign-extended: 1111 ...... 1111 1010
mask : 0000 ...... 1111 1111
AND : 0000 ...... 1111 1010

```
int u = b & 0xff;
out.printf("u: 0x%x %d\n", u, u);
```

```
u: 0xfa 250
```

Truncating integers

- We can also use fewer bits to represent a value

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

i : 0x ffff ff $\boxed{\text{fa}}$

b : 0xfa

  - what could go wrong?
    - if `i` is 256, what is `b`? What if `i` is 128?

i: 0x 0000 01 $\boxed{00}$ b: 0x00

i : 0x 0000 00 $\boxed{80}$

b : 0x80 -128

- Java warns you if you truncate implicitly
  - To hide warning, cast explicitly

```
byte b = (byte) i;
```

iClicker 1a.6

- What is the value of `i` after this Java statement executes?

4 bytes

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

1 byte

sign-extended, then shifted.

A. `0x8b`

B. `0x0000008b`

C. `0x008b0000`

D. `0xff8b0000`

E. None of these

F. I don't know

0x 0000 008b

0x8b

0xffff ff8b

0xff8b 0000

iClicker 1a.7

- What is the value of `i` after this Java statement executes?

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

& 0x00ff0000

00 8b 0000

x ^ T ≡ x

x ^ F ≡ F

8 b

1000 1011 ← x

& 1111 1111 ← T

1000 1011

A. `0xffff0000`

B. `0xff8b0000`

C. `0x008b0000`

D. None of these

E. I don't know