CPSC 213

Introduction to Computer Systems

Winter Session 2016, Term 2

Unit 1a — Jan 6, 8 and 11

Memory and Numbers

Unit 1a Overview

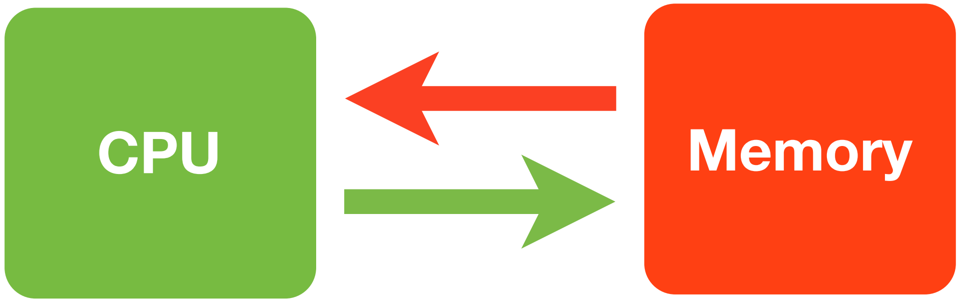
Reading

- 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*** and ***C***
  - CPU reads instruction and values of A and B, adds values and writes result to C

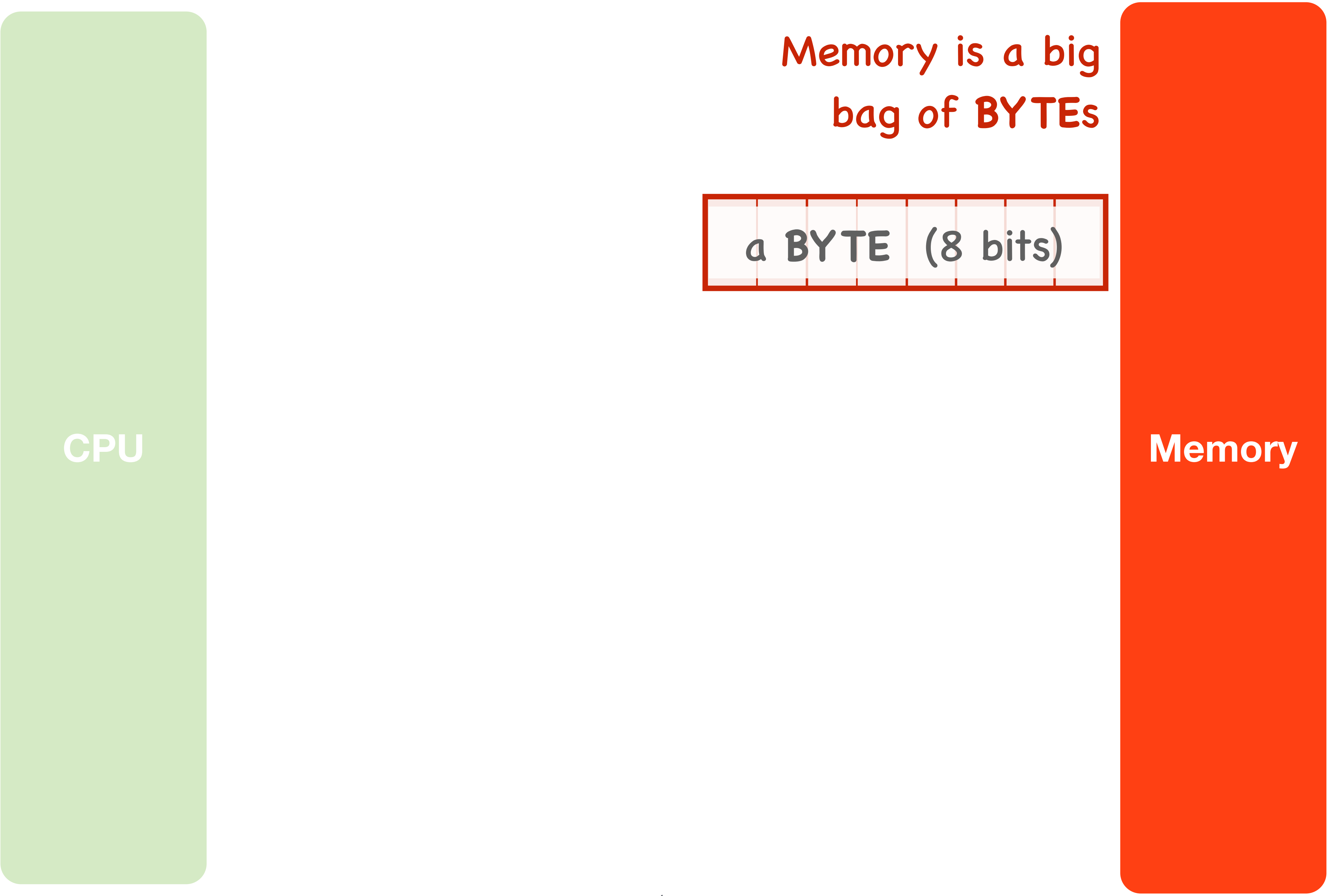
Memory

CPU

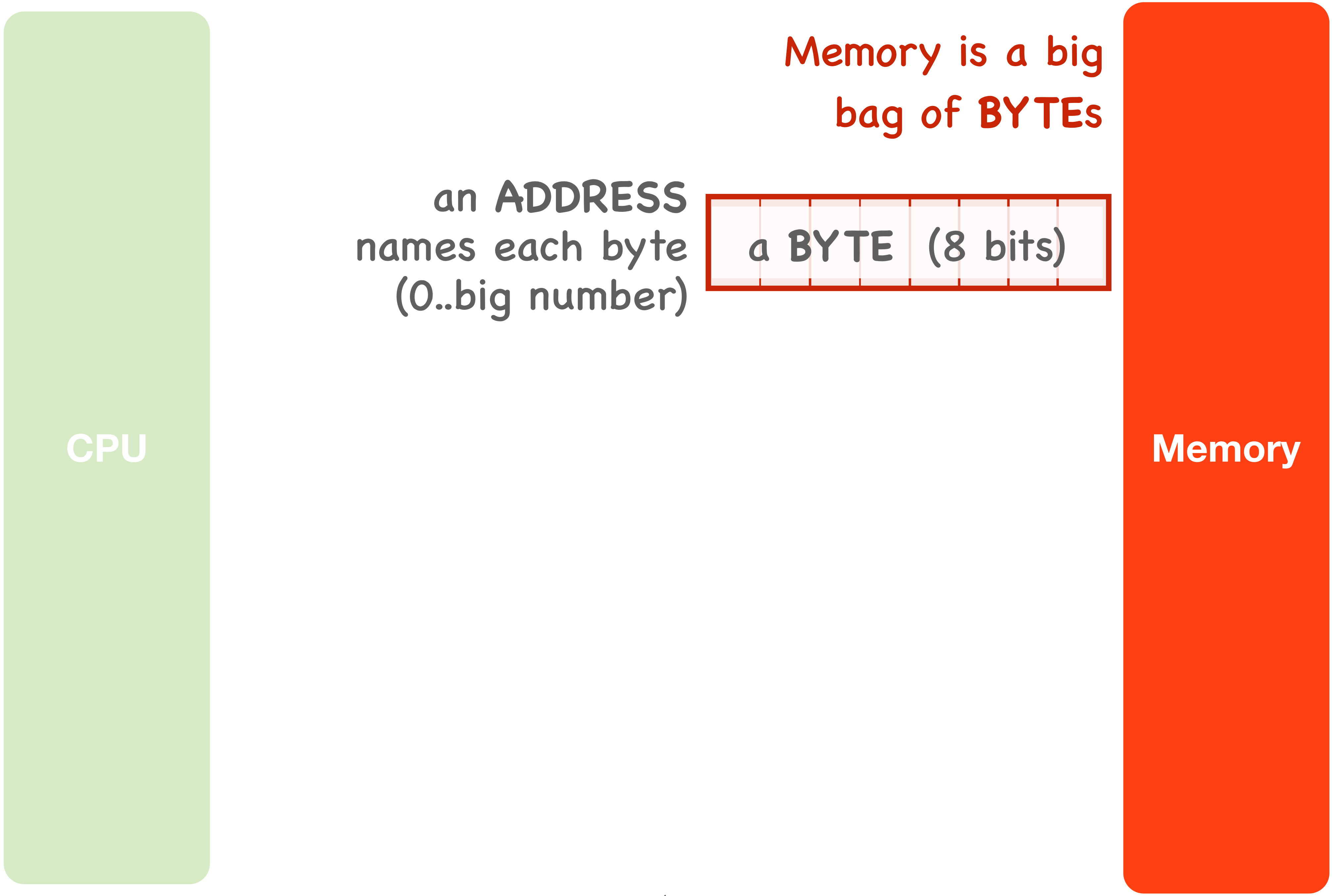
Memory is a big
bag of **BYTEs**

Memory

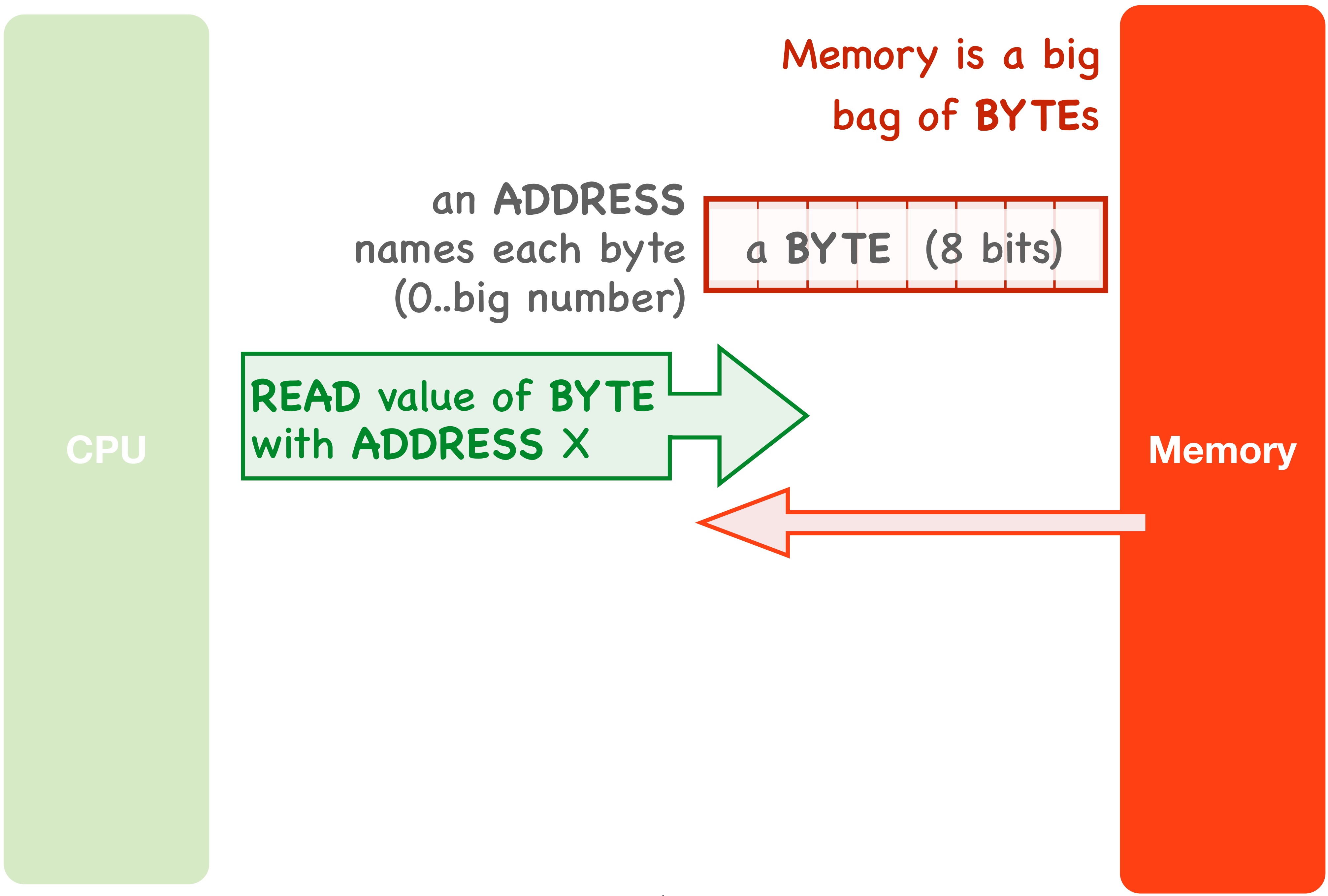
Memory



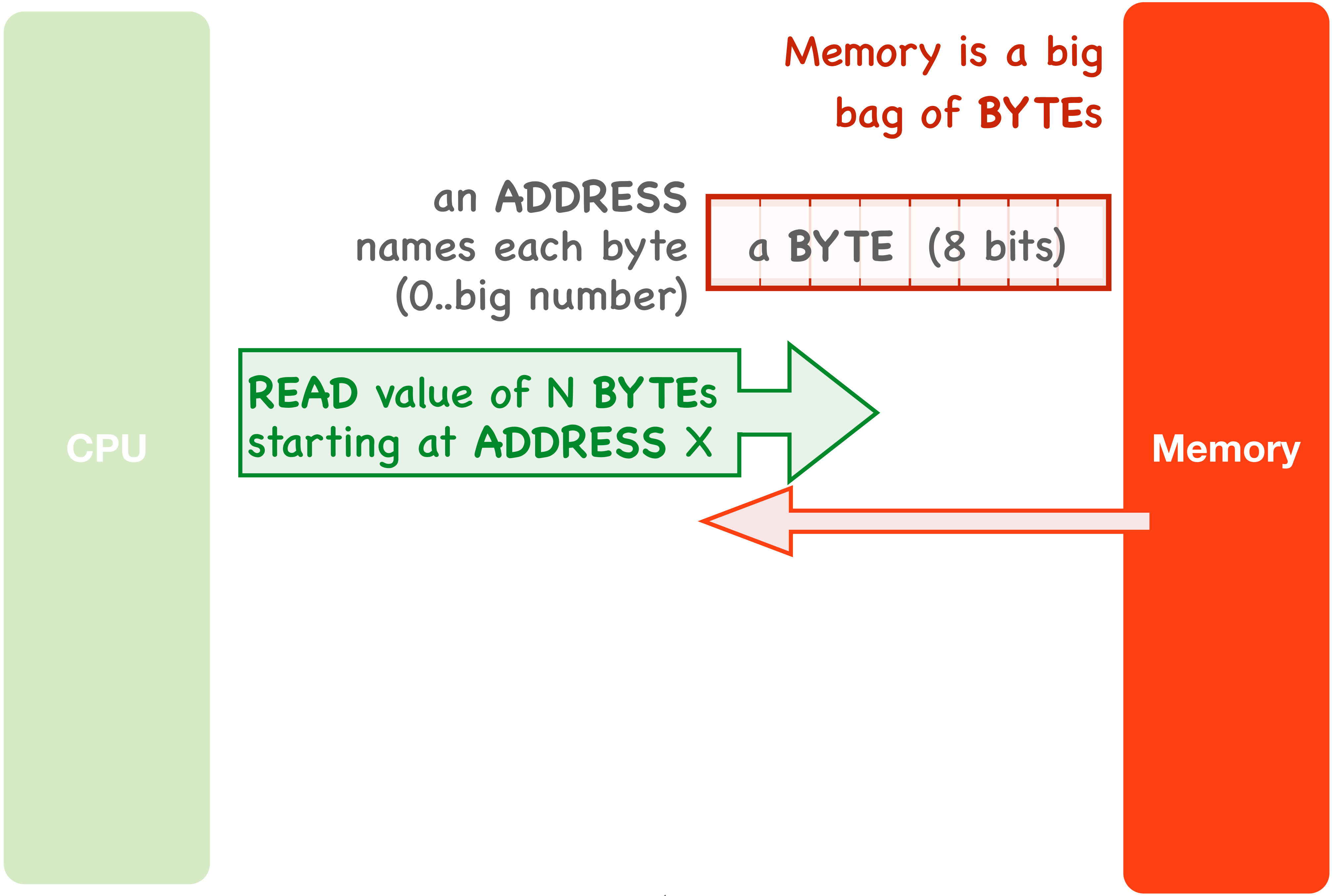
Memory


CPU
Memory is a big
bag of BYTEs
an ADDRESS
names each byte
(0..big number)
a BYTE (8 bits)
Memory

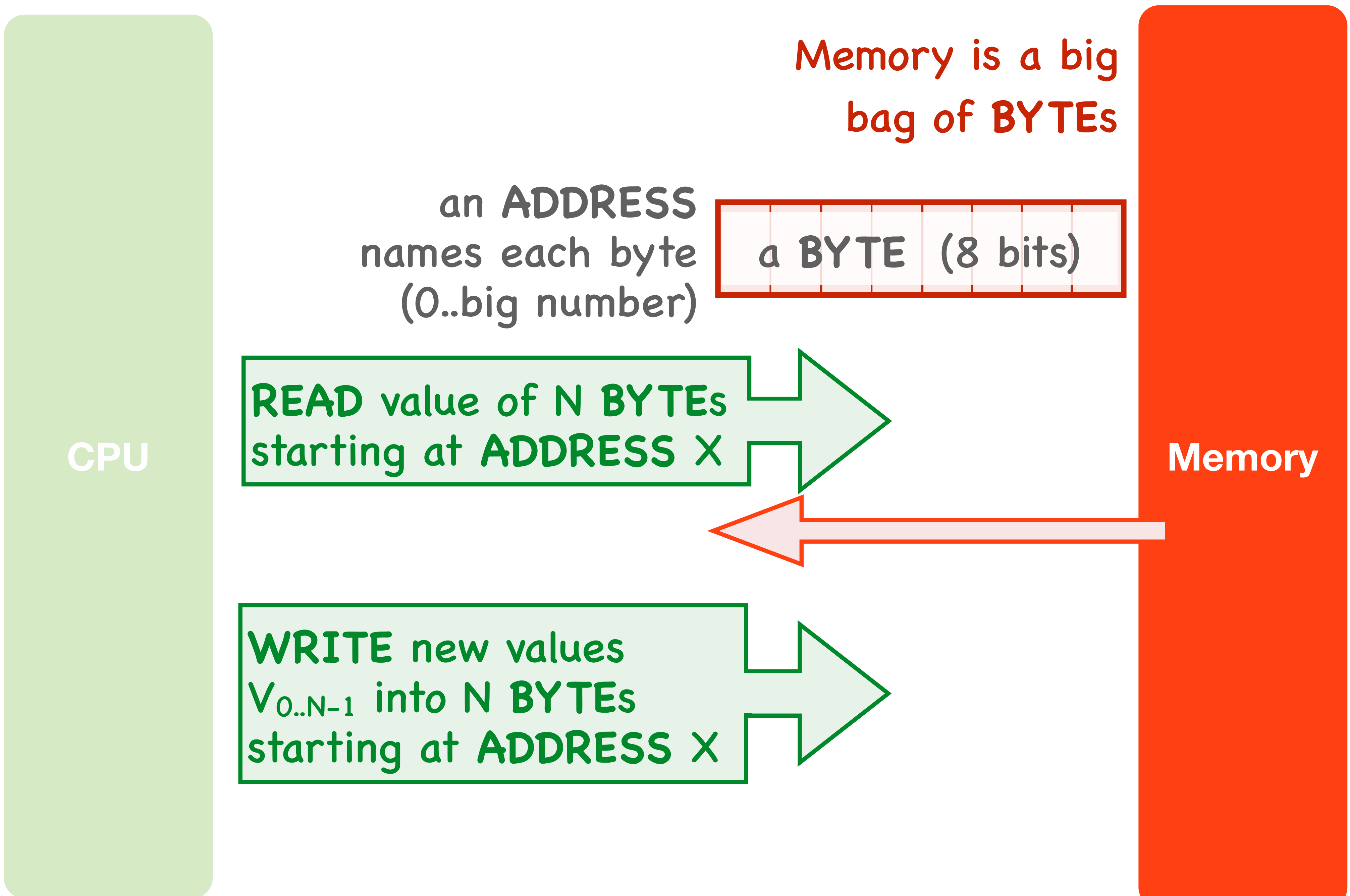
Memory


CPU
Memory is a big
bag of BYTEs
an ADDRESS
names each byte
(0..big number)
a BYTE (8 bits)
READ value of BYTE
with ADDRESS X
Memory

Memory


CPU
Memory is a big
bag of BYTEs
an ADDRESS
names each byte
(0..big number)
a BYTE (8 bits)
READ value of N BYTEs
starting at ADDRESS X
Memory

Memory



CPU
Memory
Memory is a big bag of BYTEs
an ADDRESS names each byte (0..big number)
a BYTE (8 bits)
READ value of N BYTEs starting at ADDRESS X
WRITE new values $V_{0..N-1}$ into N BYTEs starting at ADDRESS X

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

- address of a chunk is address of first byte

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

$\boxed{01001011110100110010100111} \rightarrow$?

First, Numbers and Humans

- Sometimes we are interested in integer value of chunk of bytes
  - **base 10 is natural for this**

First, Numbers and Humans

01001011110100110010100111 → ?

- 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

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

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

First, Numbers and Humans

`01001011110100110010100111` → ?

- 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, $\mathbf{11111111_2}$ **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, $\mathbf{177777_8}$ **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 $37777777777_8 = \mathtt{ffffffff}_{16}$
  - if you don't have subscripts, $\mathbf{\mathtt{ffffffff}_{16}}$ **is written as `0xffffffff`**

Binary <=> Hex is Easy

01101010010101010000111010100011

How many bits in a hex "digit", a **hexit**?

Binary <=> Hex is Easy

01101010010101010000111010100011

How many bits in a hex "digit", a **hexit**?

0110 1010 0101 0101 0000 1110 1010 0011

Binary <=> Hex is Easy

01101010010101010000111010100011

How many bits in a hex "digit", a **hexit**?

0110 1010 0101 0101 0000 1110 1010 0011

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

Binary <=> Hex is Easy

01101010010101010000111010100011

How many bits in a hex "digit", a **hexit**?

0110 1010 0101 0101 0000 1110 1010 0011

Consider ONE hexit 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

Binary <=> Hex is Easy

01101010010101010000111010100011

How many bits in a hex "digit", a **hexit**?

0110 1010 0101 0101 0000 1110 1010 0011

Consider ONE hexit 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

Binary <=> Hex is Easy

01101010010101010000111010100011

How many bits in a hex “digit”, a **hexit**?

0110 1010 0101 0101 0000 1110 1010 0011

Consider ONE hexit 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 hexits)

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
- or you can use two's compliment addition
  - negate a number of 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

Converting Hex to Decimal

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

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

```
                       1                  1
 0x2000            0x1f00             0x1f00
-0x1ff0           -0x1ff0            -0x1ff0
-------           -------            -------
      0                 0               0010
```

- or you can use two's compliment addition
  - negate a number of 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

Converting Hex to Decimal

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

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

```
                        1              1
 0x2000           0x1f00         0x1f00
-0x1ff0          -0x1ff0        -0x1ff0
-------          -------        -------
      0                0           0010
```

- or you can use two's compliment addition
  - negate a number of 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

```
!0x1ff0 == 0xffffe00f
-0x1ff0 == !0x1ff0 + 1
        == 0xffffe010
```

Converting Hex to Decimal

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

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

```
                      1               1
  0x2000          0x1f00          0x1f00
 -0x1ff0    →    -0x1ff0    →    -0x1ff0
 -------         -------         -------
       0               0            0010
```

  - or you can use two's compliment addition
    - negate a number of 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

```
!0x1ff0 == 0xffffe00f
-0x1ff0 == !0x1ff0 + 1
        == 0xffffe010

 0x2000 - 0x1ff0 == 0x2000 + 0xffffe010
                 == 0x10
```

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

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

```
              1              1
 0x2000     0x1f00         0x1f00
-0x1ff0 →  -0x1ff0  →     -0x1ff0
-------     -------        -------
      0           0          0010
```

- or you can use two's compliment addition
  - negate a number of 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

```
!0x1ff0 == 0xffffe00f
-0x1ff0 == !0x1ff0 + 1
        == 0xffffe010

0x2000 – 0x1ff0 == 0x2000 + 0xffffe010
                == 0x10
```

Converting Hex to Decimal

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

```
0x10 == 1*16 + 0
     == 16
```

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

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

i	i + 1	i + 2	i + 3	
2^{31} to 2^{24}	2^{23} to 2^{16}	2^{15} to 2^{8}	2^{7} to 2^{0}	Integer bits

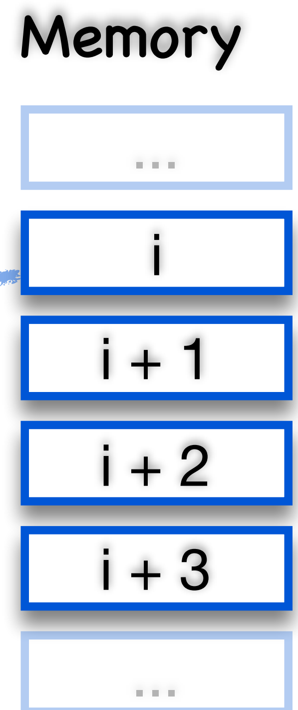
  - or we could start at the LITTLE END (Intel)

i + 3	i + 2	i + 1	i	
2^{31} to 2^{24}	2^{23} to 2^{16}	2^{15} to 2^{8}	2^{7} to 2^{0}	Integer bits

Memory

Memory
...
i
i + 1
i + 2
i + 3
...

Making Integers from Bytes



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

i	i + 1	i + 2	i + 3	
2^{31} to 2^{24}	2^{23} to 2^{16}	2^{15} to 2^{8}	2^{7} to 2^{0}	Integer bits

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

i + 3	i + 2	i + 1	i	
2^{31} to 2^{24}	2^{23} to 2^{16}	2^{15} to 2^{8}	2^{7} to 2^{0}	Integer bits

Making Integers from Bytes

Memory

...
i
i + 1
i + 2
i + 3
...

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

i	i + 1	i + 2	i + 3	
2^{31} to 2^{24}	2^{23} to 2^{16}	2^{15} to 2^{8}	2^{7} to 2^{0}	Integer bits

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

i + 3	i + 2	i + 1	i	
2^{31} to 2^{24}	2^{23} to 2^{16}	2^{15} to 2^{8}	2^{7} to 2^{0}	Integer bits

Making Integers from Bytes

Memory

...
i
i + 1
i + 2
i + 3
...

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

i	i + 1	i + 2	i + 3	
2^{31} to 2^{24}	2^{23} to 2^{16}	2^{15} to 2^{8}	2^{7} to 2^{0}	**Integer bits**

We will use Big Endian

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

i + 3	i + 2	i + 1	i	
2^{31} to 2^{24}	2^{23} to 2^{16}	2^{15} to 2^{8}	2^{7} to 2^{0}	**Integer bits**

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);
  …
```

- 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

In Lab: Endianness.java

```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.

address 0

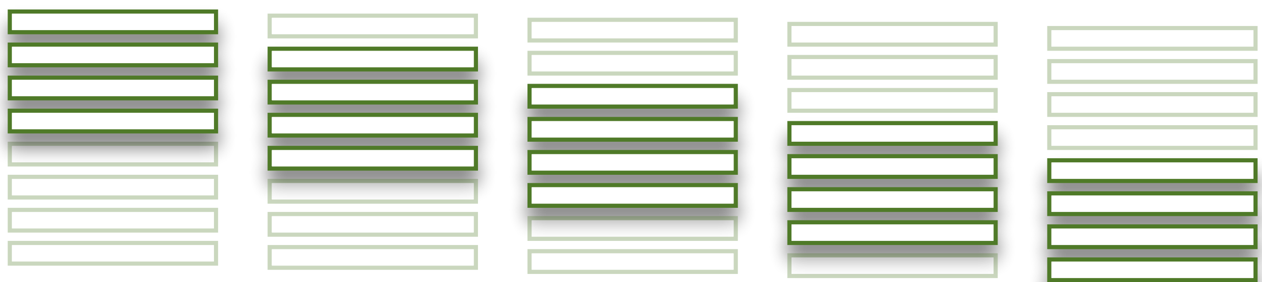
address 3

- 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

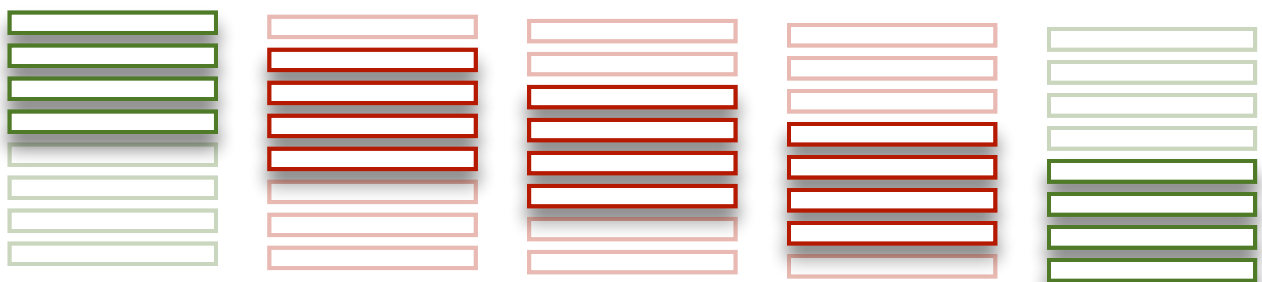
Some addresses are better than others

Address Alignment

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



- 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

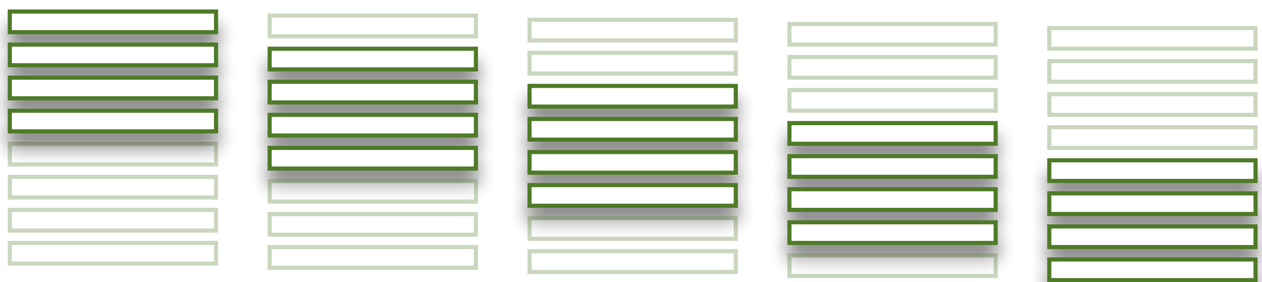
e.g., a word contains exactly two complete shorts

- 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**

Some addresses are better than others

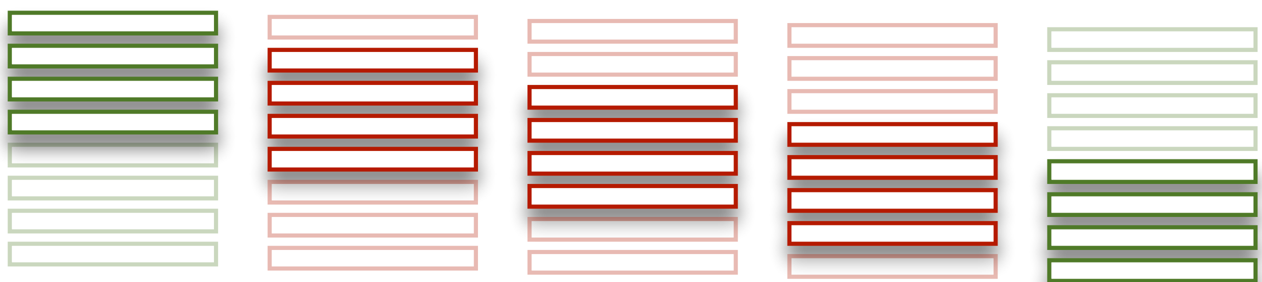
Address Alignment

- 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 chucks 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**$^{\text{th}}$ 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
  - 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 mod N == 0)?

Question 1a.4: Alignment

- Which of the following statement (s) is (are) true?
  - A. the address 6 (110_2) is aligned for addressing a *short*
  - 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)

Shifting Bits

Shifting Bits

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

$$
\begin{aligned}
\texttt{0x6 >> 1} &== 110_2 \texttt{ >> 1} \\
&== 11_2 \\
&== \texttt{0x3}
\end{aligned}
$$

Shifting Bits

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

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

- 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 `01111101`$_2$ == 125, not -3

Shifting Bits

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

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

- 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 01111101_2 == 125, not -3
- There are two kinds of right shifts
  - SIGNED "**>>**" shifts, but keeps high-order bit (the sign bit) the same
    - 6- >> 1 == 11111101_2 == -3 (i.e., 11111101_2 + 00000011_2 == 0)
  - UNSIGNED "**>>>**", shifts and sets high-order bit to 0
    - -6 >>> 1 == 01111101_2 … `0xfa >>> 1 == 0x7d`
  - In Java you choose. In C the compiler chooses.
    - C as both signed and unsigned integer data types and no ">>>". Java has only signed.

Extending an Integer

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?

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: 0xfffffffa -6
```

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: 0xfffffffa -6
```

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

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: 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
- you can force zero-extension using logical, bit-wise AND (e.g., in Java):

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

```
u: 0xfa 250
```

Truncating an Integer

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?

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