CPSC 213: Introduction to Computer Systems

Unit 1a: Numbers and Memory

Jordon Johnson

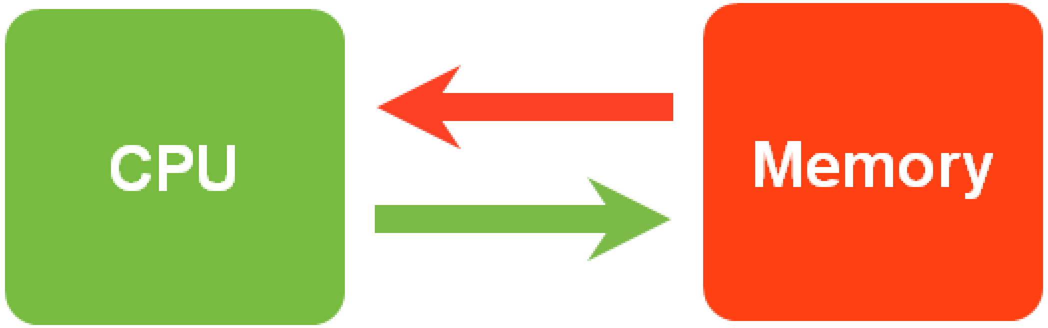
Adapted from slides by Jonatan Schroeder, Mike Feeley, and Robert Xiao

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, add/subtract hexadecimal numbers and convert small numbers between binary and decimal

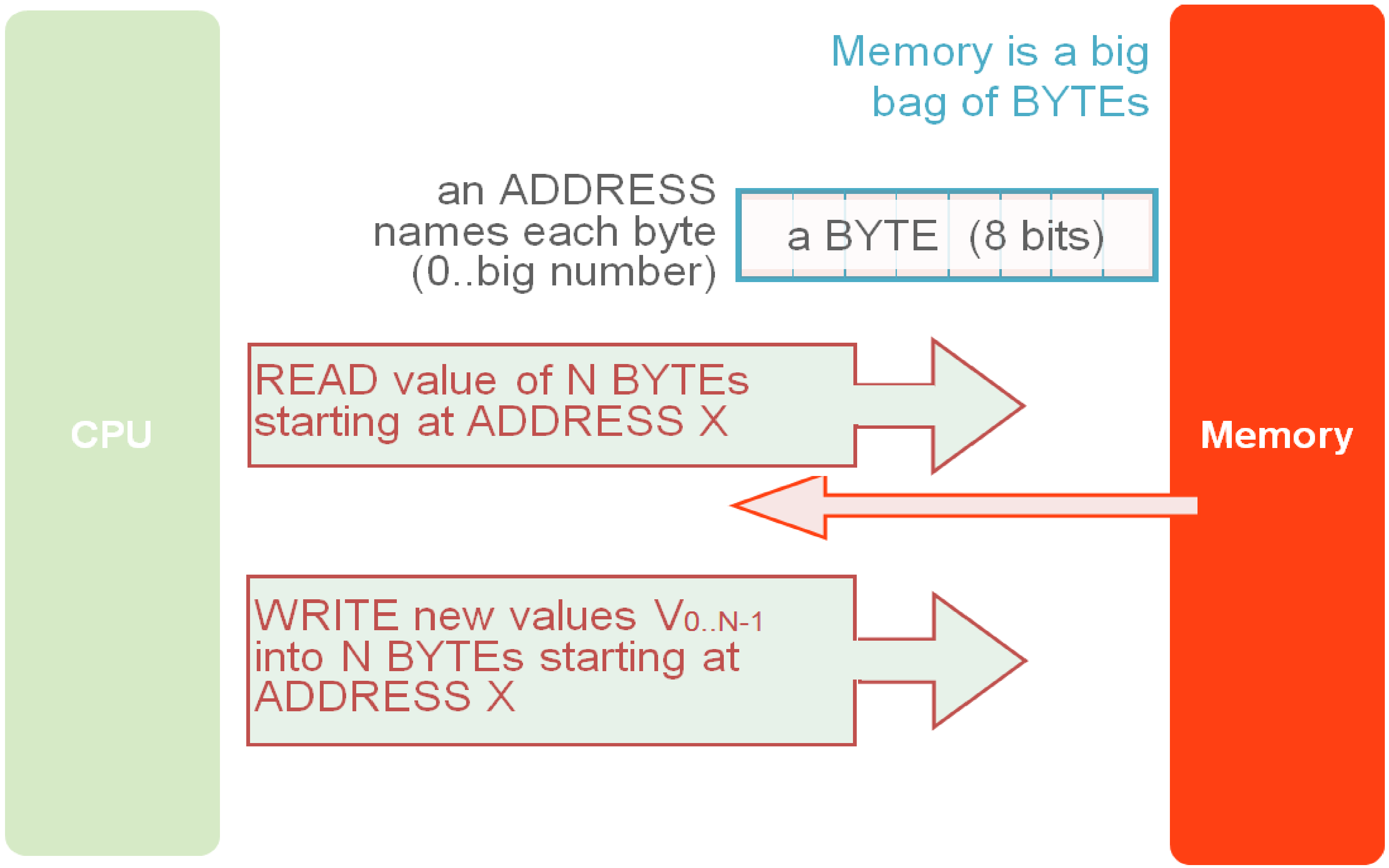
A Simple Computing Machine

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



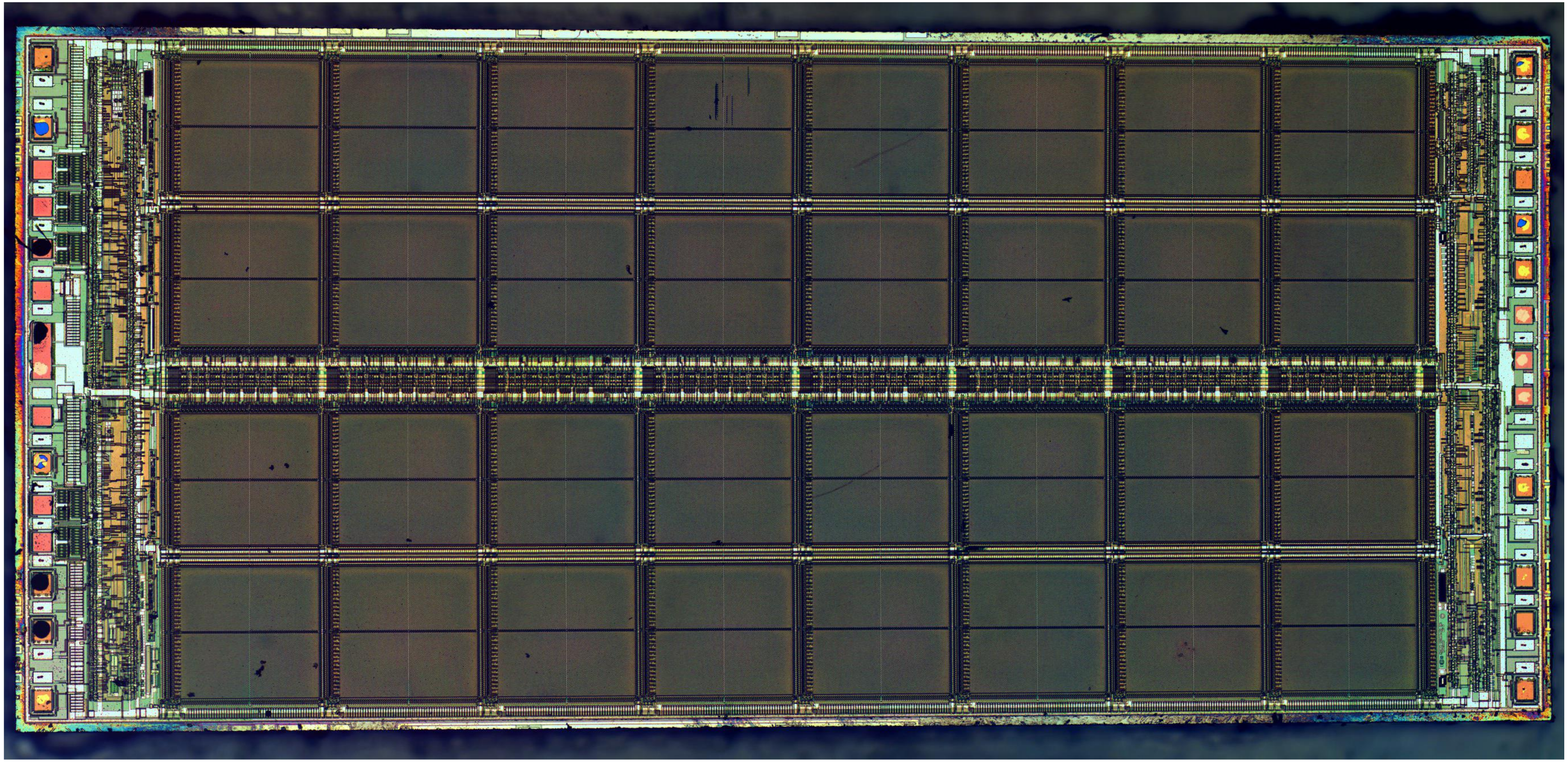
- **CPU**
  - reads instruction and data from memory
  - performs specified computation and writes result back to memory
- **Example**
  - C = A + B
  - memory stores: add instruction, and variables A, B and C
  - CPU reads instruction and values of A and B, adds values, writes result to C

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
WRITE new values $V_{0..N-1}$
into N BYTEs starting at
ADDRESS X
Memory

Memory

This is an **implementation** of memory; we will focus on the **abstraction**



Micron MT4C1024 128KB RAM, zeptobars.org

Memory

- **Naming**
  - unit of addressing is a **byte** (8 bits)
  - every byte of memory has a **unique address**
  - some machines have 32-bit addresses, some have 64-bit addresses
    - We will (usually) assume our machine uses 32-bit addresses, and that all addresses are valid
- **Access**
  - many 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
  - 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

- 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 **bits** themselves
  - In such cases the decimal value isn't particularly important
  - For example, consider **memory addresses**
    - 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

Numbers and Representation

- We might use base-2, **binary**
  - a small 256-byte memory has addresses 0_2 to 11111111_2
  - may be represented as ***0b***11111111
  - becomes tedious and hard to read as addresses get larger
- Once we used base-8, **octal**
  - 64-KB memory addresses go up to $1111111111111111_2 = 177777_8$
  - may be represented as ***0o***177777
  - gets tedious and hard to read too
- Now we use base-16, **hexadecimal**
  - 4-GB memory addresses go up to $37777777777_8 = \mathrm{ffffffff}_{16}$
  - if you don't have subscripts, $\mathrm{ffffffff}_{16}$ is written as ***0x***ffffffff

Binary ⇔ Hexadecimal

01101010010101010000111010100011

- 4 bits in a hex "digit", a hexit (or "nibble")

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

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

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. 42 is the answer to the ultimate question of life, the universe, and everything

Hexadecimal Operations

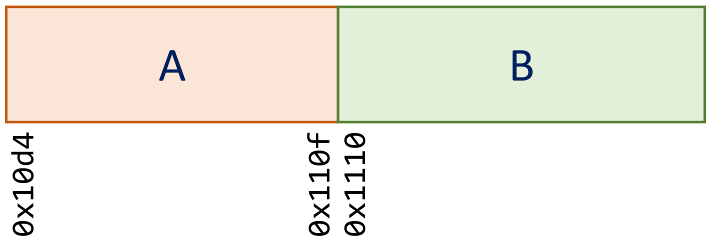
- We use hexadecimal for addresses
  - We don’t really care what their base-10 value is
- Addition/subtraction in hex
  - You could convert both to decimal, but that might be too tedious
  - You can calculate in hex directly
  - Alternative for subtraction: use addition with two’s complement
- Remember:
  - carry when result is 0x10 == 16_{10} or more
  - hexits A..F convert to their decimal value

iClicker 1a.2

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

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 (in bytes)?



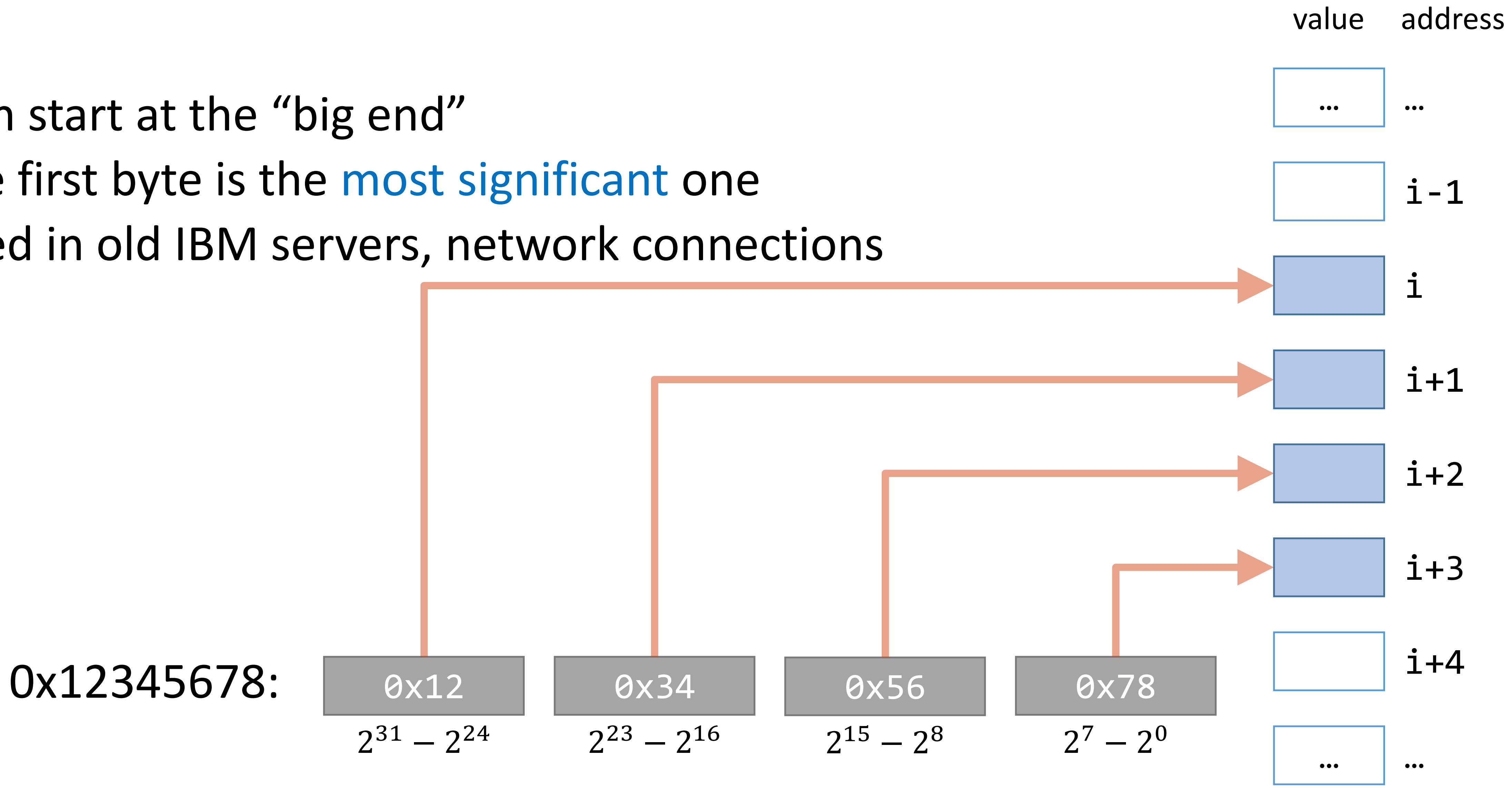
Making Integers from Bytes

- First architectural design decision:
  - assembling memory bytes into integers
- Consider a 32-bit integer (`int` in Java or C)
  - It uses 4 bytes
  - If memory address is **i**, then we also need bytes at **i+1**, **i+2**, **i+3**
  - Example: if address is **0x2014**, then integer is in **0x2014**, **0x2015**, **0x2016**, **0x2017**
- What do each of these bytes represent?

value	address
…	…
	`i-1`
	`i`
	`i+1`
	`i+2`
	`i+3`
	`i+4`
…	…

Big-Endian

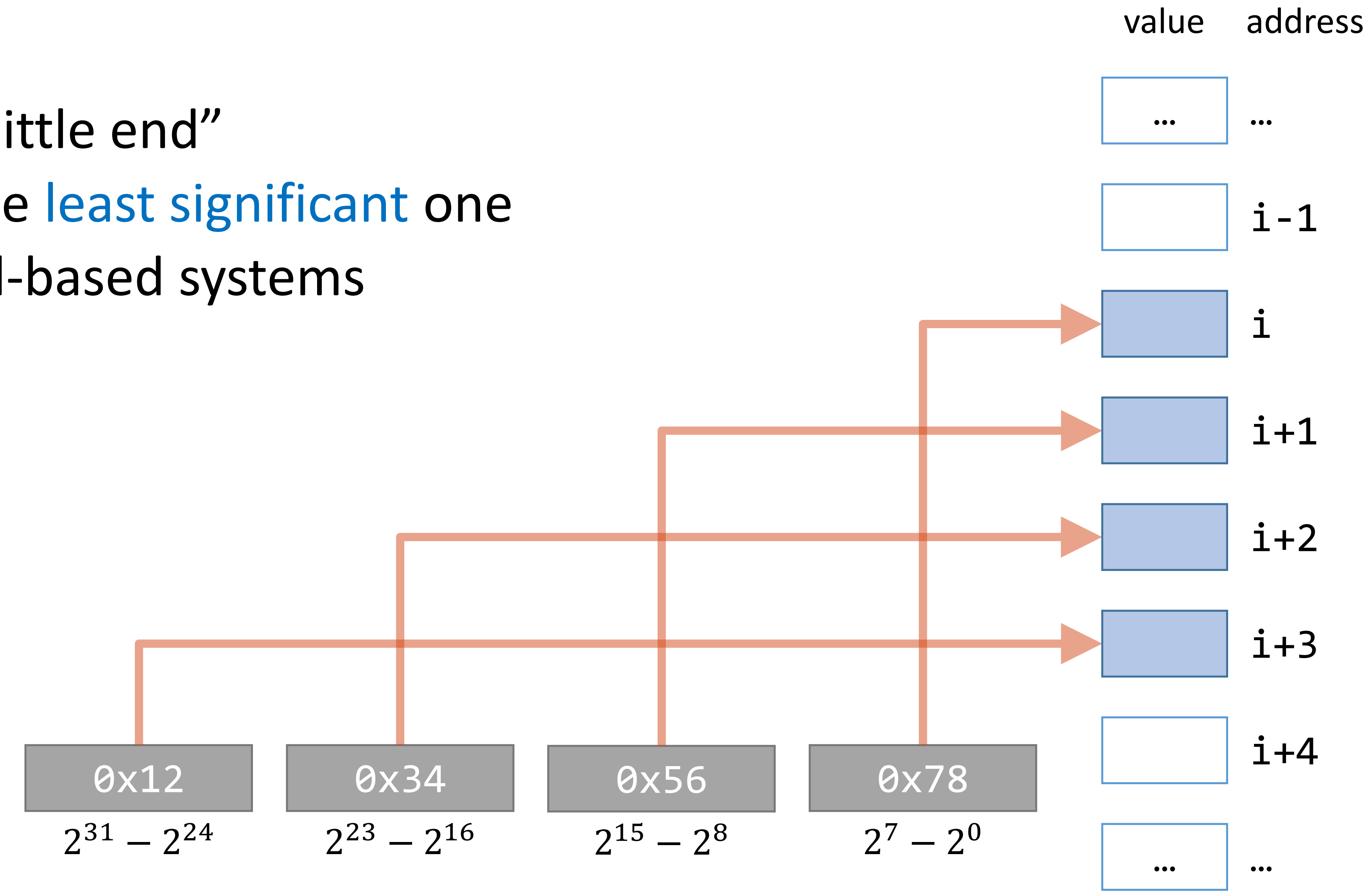
- We can start at the "big end"
  - The first byte is the most significant one
  - Used in old IBM servers, network connections



Little-Endian

- We can start at the “little end”
  - The first byte is the least significant one
  - Used in most Intel-based systems

0x12345678:



iClicker 1a.3

The memory of some machines stores Big Endian integers.

A. True
B. False

iClicker 1a.4

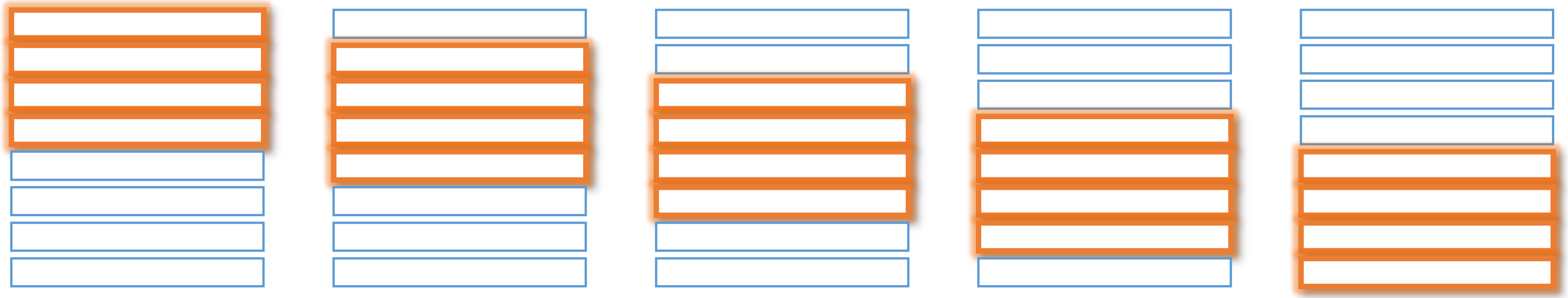
What is the Little-Endian 4-byte integer value at address 0x4?

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

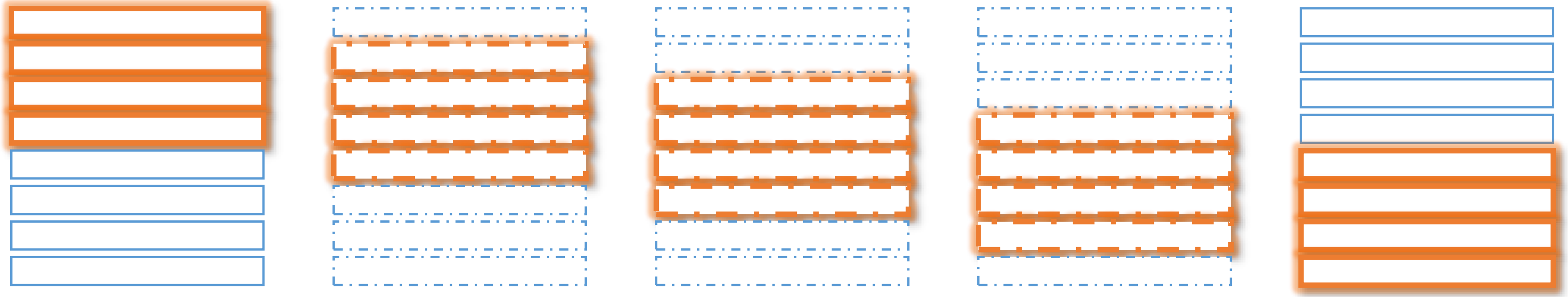
Address	Value
0x0:	0xfe
0x1:	0x32
0x2:	0x87
0x3:	0x9a
0x4:	0x73
0x5:	0xb6
0x6:	0x04
0x7:	0x1c

Should We Put Data Just Anywhere?

- We could place a 4-byte integer at any address



- However, requiring addresses to be **aligned** is better for hardware



Address Alignment

- What is an "aligned" address?
  - 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** **(spoilers!)**
- Aligned addresses are better:
  - You can fit **two shorts** inside an **int**, etc.
  - This is significant in **arrays** as well
  - Some CPUs don't support misaligned addresses
    - Intel: misaligned access is supported but slower

Address Alignment

- CPU implementation encourages alignment
  - Memory is organized internally into chunks (**blocks**)
  - Every **memory access** requires accessing a **whole block**
    - *Details: see **CPSC 313***
- CPU memory access looks like:
  - Read/write N bytes starting at address A
- This is translated by memory to:
  - Block B contains addresses $X..(X + blocksize - 1)$
  - $X \leq A \leq X + blocksize - 1$

    O for offset, not zero
    - So A is at some offset (let's call it O) from the beginning of block B
  - Read/write N bytes starting at O^{th} byte of block B $(A = X + O)$

Address Alignment

- *(From last slide):*
  - Block B contains addresses $X..(X + blocksize - 1)$
  - $X \leq A \leq X + blocksize - 1$
  - Read/write N bytes starting at O^{th} byte of block B $(A = X + O)$
- How is this **simplified** if:
  - $blocksize$ is a power of 2,
  - N is a power of 2,
  - A is aligned to N?

iClicker 1a.5

Which of the following statements is/are true?

A. the address $6_{10} = 110_2$ is aligned for a short (2 byte) integer
B. the address $6_{10} = 110_2$ is aligned for a 4 byte integer
C. the address $20_{10} = 10100_2$ is aligned for a long (8 byte) integer
D. Two or more of the above
E. None of the above

iClicker 1a.6

Which of the following statements is (are) true?

A. The address 0x14 is aligned for addressing a 2-byte integer, but not a 4-byte integer
B. The address 0x14 is aligned for addressing a 2-byte or a 4-byte integer, but not an 8-byte integer
C. The address 0x14 is aligned for addressing a 2-byte, 4-byte, or 8-byte integer
D. None of the above
E. If I say the answer is 42, maybe I’ll get the point anyway

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

Changing Data Types: Extending

- Extending an integer: increasing the number of bytes used to store it

```
byte b = -6; // stored as 11111010
int i = b; // stored as 11111111111111111111111111111010
```

- Signed extension: used in **signed** data types
  - Everything is signed in Java
  - Copy first bit (sign) to extended bits
- Zero extension: used in **unsigned** data types (C)
  - Set all extended bits to zero
  - How to do it in Java? ***(spoilers!)***

Changing Data Types: Truncating

- Truncating an integer: reducing the number of bytes used to store it

  `int i = -6; // stored as 11111111111111111111111111111010`

  `byte b = i; // stored as` ~~111111111111111111111111~~11111010

- What could **go wrong**?
  - What value would b get if i were 256? or 128?
- Java warns you if you truncate implicitly
  - To avoid warning, **cast** explicitly:

    `byte b = (byte) i;`

Bit Operations in C/Java

- a << b: shift all bits in a to the left b times, fill remaining right bits with zero
- a >> b: shift all bits in a to the right b times
  - C: if a is unsigned, zero-extends, otherwise sign-extends
  - Java: operator >> sign-extends, operator >>> zero-extends

- a & b: AND applied to corresponding bits in a and b
- a | b: OR applied to corresponding bits in a and b
- a ^ b: XOR applied to corresponding bits in a and b
- ~a: inverts every bit of a

Making Use of Bit Operations

- Shifting multiplies/divides by power of 2
  - “a<<b” is equivalent to $a \times 2^b$
  - “a>>b” is equivalent to $^{a}/_{2^b}$
- Example: 22 in binary is 00010110
  - 11 is 00001011 (22 shifted right once, $^{22}/_{2^1}$)
  - 44 is 00101100 (22 shifted left once, 22×2^1)
  - 88 is 01011000 (22 shifted left twice, 22×2^2)
- Works for negative numbers too, if using sign-extended shift
  - -22 is 11101010
  - -11 is 11110101 (-22 shifted right once, $^{-22}/_{2^1}$)
  - -44 is 11010100 (-22 shifted left once, -22×2^1)
  - -88 is 10101000 (-22 shifted left twice, -22×2^2)

Making Use of Bit Operations

- Let's use our example from before:

```
byte b = -6; // stored as 11111010
int i = b; // stored as 1111…1111 11111010
```

- How can we get it to zero-extend instead of sign-extend?

- **Answer:** using **bit operations**:

```
// 0xFF in bits is: 0000…000011111111
int i = b & 0xFF; // stored as 0000…000011111010
```

- Note that **the result is different**: 250 instead of -6
  - That's because zero-extension is used for unsigned integers

iClicker 1a.7

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

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

A. `0xffff0000`
B. `0x0000008b`
C. `0x008b0000`
D. `0xff8b0000`
E. None of the above

iClicker 1a.8

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

```
int i = 0x0000008b << 16;
```

A. `0x008b`
B. `0x0000008b`
C. `0x008b0000`
D. `0xff8b0000`
E. None of the above

iClicker 1a.9

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

```
int i = 0x8b << 16;
```

A. `0x8b`
B. `0x0000008b`
C. `0x008b0000`
D. `0xff8b0000`
E. None of the above

iClicker 1a.10

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 the above