Lecture 5: Unicode & bitwise operations

CMPE 120: Computer Organization & Architecture

Olivia Weng

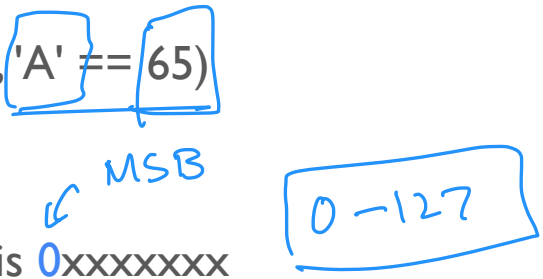
What about non-English characters?

What about non-English characters?

- Thousands more characters used in languages around the world
- ASCII does not define:
  - Spanish: é
  - Chinese: 中
  - Emoji: 🐫
- char datatype of 1 byte only encodes 256 possible bit patterns
- Challenge: Millions of lines of code written that assumed 1 byte ASCII chars

UTF-8: Unicode encoding

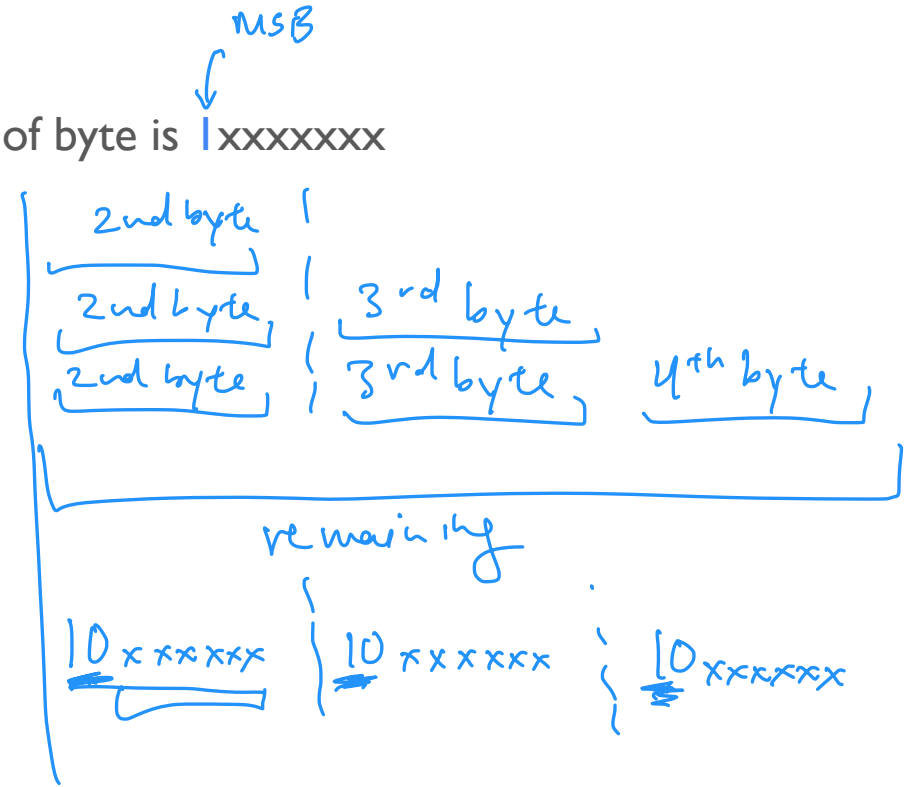
- Use more bits to encode more characters!
- Code point: an integer representing a character (e.g., 'A' == 65)



- Normal ASCII code point: Highest order bit of byte is 0xxxxxxx
  - UTF-8 is backwards compatible with ASCII!
- Multi-byte code point: Highest order bit of byte is 1xxxxxxx

How many bytes do you need?

- Multi-byte code point: Highest order bit of byte is 1xxxxxxx
- Bit flags indicate code point length
  - 110xxxxx = 2 bytes
  - 1110xxxx = 3 bytes
  - 11110xxx = 4 bytes
- Bytes after the first byte start with
  - 10xxxxxx



Code point construction

San José

2-byte code point

- é = 11000011 10101001
  - bit flags : only encodes metadata (i.e., only provides information about)
  - the code point = 000011 101001 = 233

this is a byte following the first byte

tells us this is 2 bytes long

233 ⇒ é

Demo

```
char str[] = "José";

for (int i = 0; str[i] != '\0'; i++) {
    printf("0x%x\n", (unsigned char) str[i]);
}
```

How many bytes do you need?

- Multi-byte code point: Highest order bit of byte is 1xxxxxxx
- Bit flags indicate code point length
  - 110xxxxx = 2 bytes
  - 1110xxxx = 3 bytes
  - 11110xxx = 4 bytes
- Bytes after the first byte start with
  - 10xxxxxx

- How do we check for these specific bit flags to know how many bytes we have?
  - Need a way of inspecting individual bits!

Bit operations

- Special mathematical operations in C for manipulating bits
  - &: AND
  - |: OR
  - ~: NOT
  - ^: XOR
  - >>: shift right
  - <<: shift left

AND: &

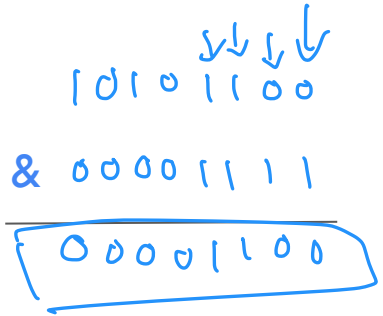
a && b = ?

&&

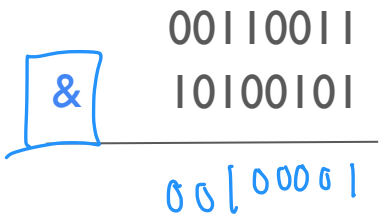
- AND each bit together

Truth table

input_a	input_b	Result
0	0	0
0	1	0
1	0	0
1	1	1



What is the result?

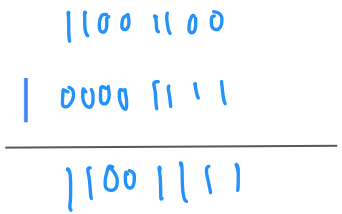


OR: |

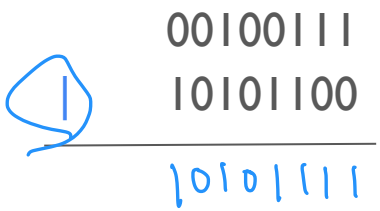
- OR each bit together

Truth table

input_a	input_b	Result
0	0	0
0	1	1
1	0	1
1	1	1



What is the result?



NOT: ~

1 operand

- NOT each bit together

Truth table

input_a	Result
0	1
1	0

~ 1111 0000

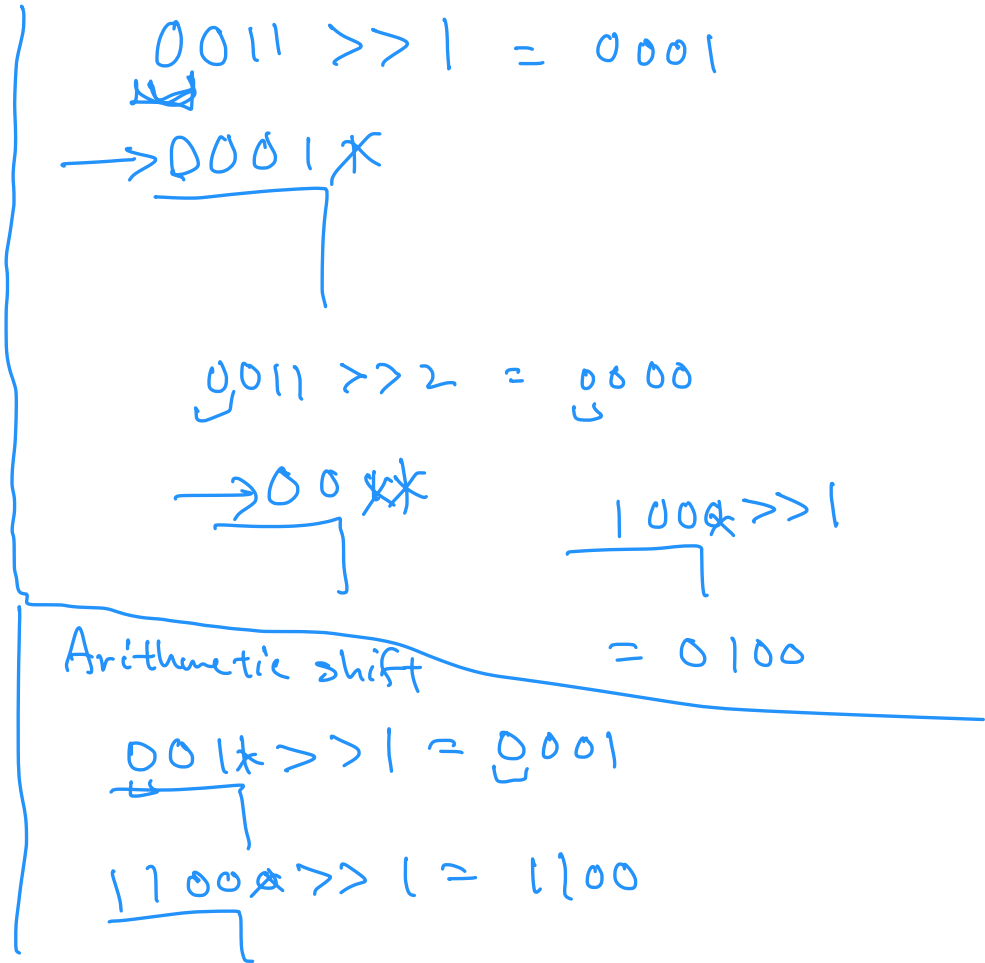
0000 1111

Shift right: >>

- Shift the bits "off a cliff" to the right
  - Logical shift
    - Zero extension
    - Used to shift unsigned values
  - Arithmetic shift:
    - Sign extension
    - Used to shift signed values

MSB: 0 → extend w/ 0-bit

1 → extend w/ 1-bit



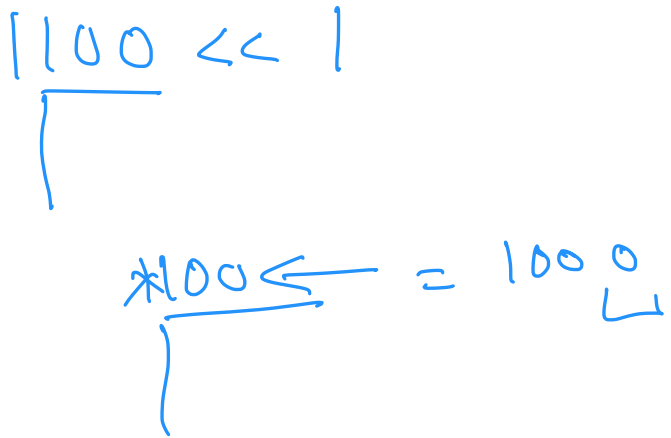
4-bit signed

What is the result in binary and decimal?

- 1010 >> 3 = 1111 (-1)
  - -6 ÷ 2^3 = -1
  - 1010 >> 1 = 1101 (-3) ÷ 2
  - 1101 >> 1 = 1110 (-2) ÷ 2
- 0100 >> 1 = 0010 (2)
  - 4 ÷ 2
- 1011 >> 4 = 1111 (-1)
  - (-5) ÷ 2^4 = -1
- 1101 >> 2 = 1111 (-1)
  - (-2) ÷ 2^2 = -1

Shift left: <<

- Shift the bits "off a cliff" to the left



4-bit unsigned

What is the result in binary and decimal?

- 0010 << 2 = 1000 (8)

  (2) → 2 × (2×2) = 8

- 0001 << 3 = 1000 (8)

  1 × (2^3) = 8

- 1000 << 4 = 0000 (0)

  (8) → 8 × (2^4)

- 0110 << 2 = 1000

Demo: practice with bitwise operators

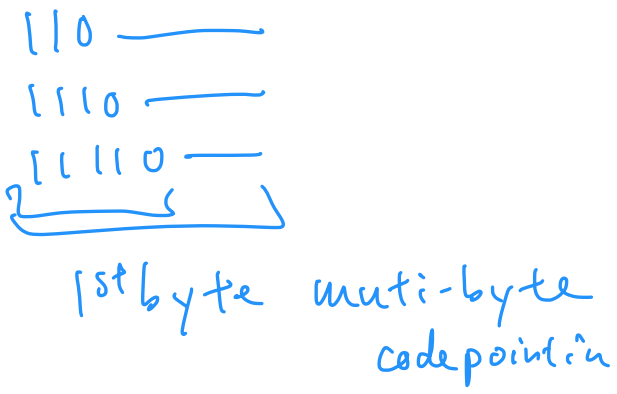
& | >> << ~

```
int bitwise_is_even(int8_t num);
```

return 1 if num is even else 0

```
int count_1_bits(int32_t num);
```

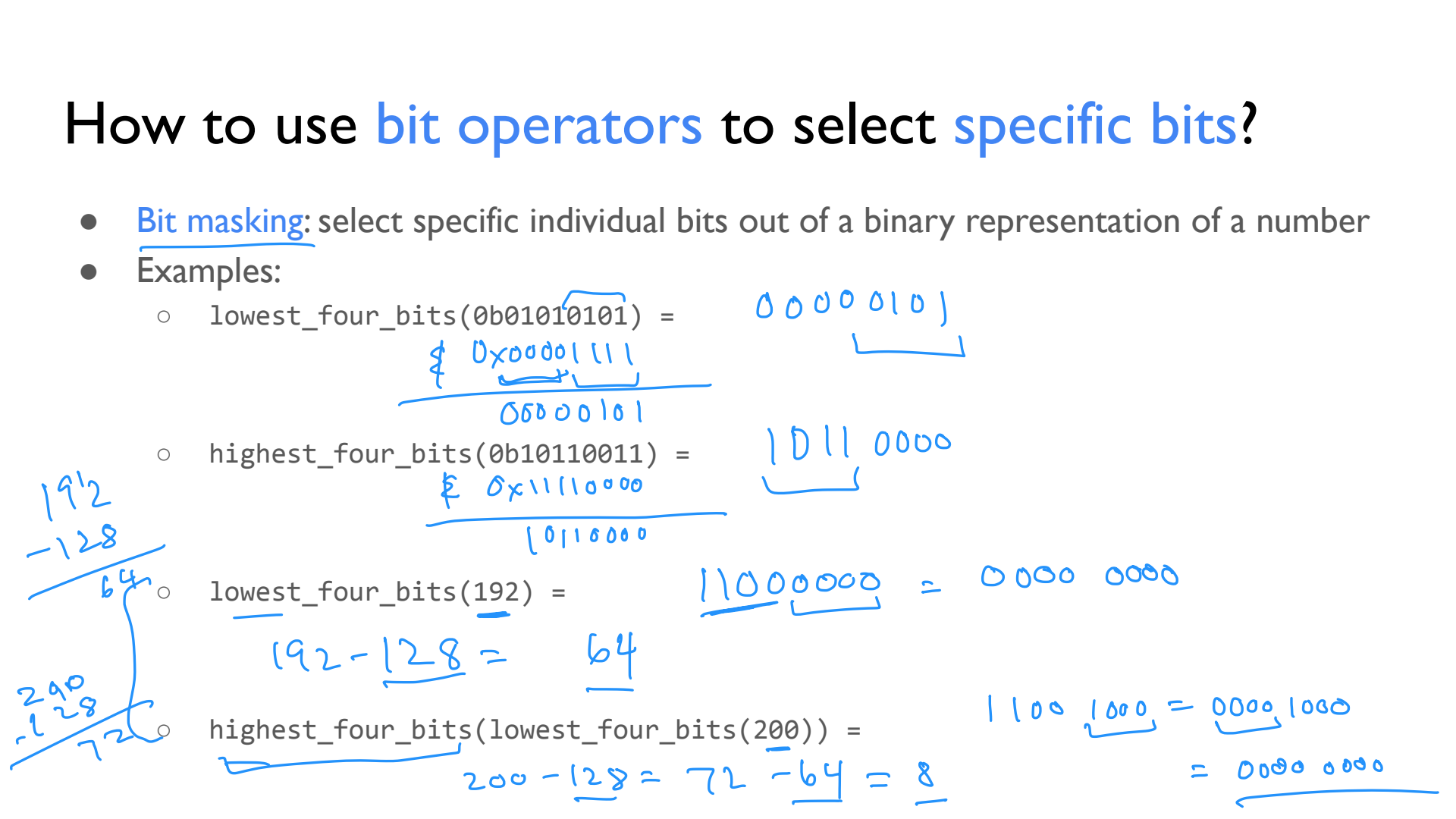
count += 1;



How to use bit operators to select specific bits?

How to use bit operators to select specific bits?

- Bit masking: select specific individual bits out of a binary representation of a number
- Examples:
  - lowest_four_bits(0b01010101) =
  - highest_four_bits(0b10110011) =
  - lowest_four_bits(192) =
  - highest_four_bits(lowest_four_bits(200)) =



How to implement masking?

```
char lowest_four_bits(char c) {
    return c & 0b00001111;
}
```

bitmask

```
char highest_four_bits(char c) {
    return c & 0b11110000;
}
```

bitmask

Why can't we use the bitwise | operator?

The bitwise & operator selects specific bit positions based on the pattern of 1's that the variable is &'d with.

& truth table	Result
0 & 0	0
0 & 1	0
1 & 0	0
1 & 1	1

How do we check the codepoint bit flags?

- Bit masking!
- `uint8_t codepoint_size(char string[])`
  - `// return the number of bytes this codepoint is, e.g., 1, 2, 3, or 4`

Demo

- `int32_t code_point_2(char c1, char c2); // return codepoint`
  - é
- `uint32_t utf8_strlen(char string[]);`

```
// Given a char[] representing a UTF-8 encoded string,
// return the number of UTF-8 codepoints (characters) in the string.
// The input will always be valid UTF-8 and will not exceed 2048 bytes.
```