

A Crash Course on Data Compression

1.1 Warmup

Giulio Ermanno Pibiri

ISTI-CNR, giulio.ermanno.pibiri@isti.cnr.it



@giulio_pibiri



@jermmp

Overview

- Warmup 1: Bit-packing, read/write binary data
- Warmup 2: Run-length encoding

Bit-Packing

Option 1. Use built-in data types.

Option 2. Bit-pack the quantities.

```
struct small_record {  
    uint8_t weight;  
    uint8_t height;  
    uint8_t day;  
    uint8_t month;  
    uint16_t year;  
};
```

← assume 256 Kg are enough, so we need 8 bits

← assume 256 cm are enough, so we need 8 bits

← at most 31 days, so we need $\lceil \log_2 31 \rceil = 5$ bits

← 12 months, so we need $\lceil \log_2 12 \rceil = 4$ bits

← assume we encode years till 4096 BC, so we use 12 bits

Bit-Packing

- Option 1.** Use built-in data types.
Option 2. Bit-pack the quantities.

```
struct small_record {  
    uint8_t weight;  
    uint8_t height;  
    uint8_t day;  
    uint8_t month;  
    uint16_t year;  
};
```

← assume 256 Kg are enough, so we need 8 bits

← assume 256 cm are enough, so we need 8 bits

← at most 31 days, so we need $\lceil \log_2 31 \rceil = 5$ bits

← 12 months, so we need $\lceil \log_2 12 \rceil = 4$ bits

← assume we encode years till 4096 BC, so we use 12 bits

Example: weight = 75 Kg, height = 175 cm, date = 13/07/1990.

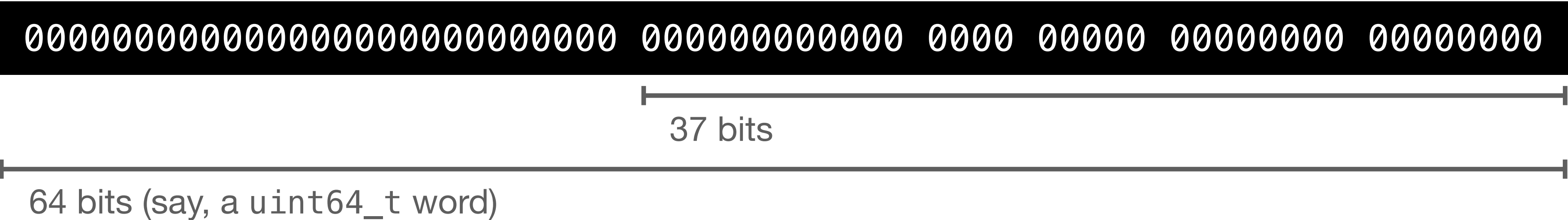
Bit-Packing

- Option 1. Use built-in data types.
- Option 2. Bit-pack the quantities.

```
struct small_record {  
    uint8_t weight;  
    uint8_t height;  
    uint8_t day;  
    uint8_t month;  
    uint16_t year;  
};
```

← assume 256 Kg are enough, so we need 8 bits
← assume 256 cm are enough, so we need 8 bits
← at most 31 days, so we need $\lceil \log_2 31 \rceil = 5$ bits
← 12 months, so we need $\lceil \log_2 12 \rceil = 4$ bits
← assume we encode years till 4096 BC, so we use 12 bits

Example: weight = 75 Kg, height = 175 cm, date = 13/07/1990.



Bit-Packing

- Option 1. Use built-in data types.
- Option 2. Bit-pack the quantities.

```
struct small_record {  
    uint8_t weight;  
    uint8_t height;  
    uint8_t day;  
    uint8_t month;  
    uint16_t year;  
};
```

← assume 256 Kg are enough, so we need 8 bits

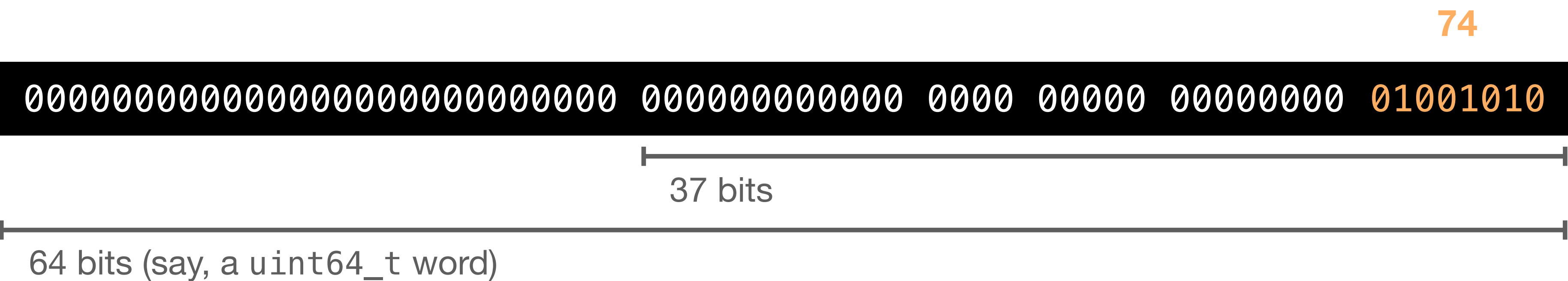
← assume 256 cm are enough, so we need 8 bits

← at most 31 days, so we need $\lceil \log_2 31 \rceil = 5$ bits

← 12 months, so we need $\lceil \log_2 12 \rceil = 4$ bits

← assume we encode years till 4096 BC, so we use 12 bits

Example: weight = 75 Kg, height = 175 cm, date = 13/07/1990.



Bit-Packing

Option 1. Use built-in data types.

Option 2. Bit-pack the quantities.

```
struct small_record {  
    uint8_t weight;  ←  
    uint8_t height; ←  
    uint8_t day;     ←  
    uint8_t month;   ←  
    uint16_t year;   ←  
};
```

assume 256 Kg are enough, so we need 8 bits

assume 256 cm are enough, so we need 8 bits

at most 31 days, so we need $\lceil \log_2 31 \rceil = 5$ bits

12 months, so we need $\lceil \log_2 12 \rceil = 4$ bits

assume we encode years till 4096 BC, so we use 12 bits

Example: weight = 75 Kg, height = 175 cm, date = 13/07/1990.

[illegible]

37 bits

64 bits (say, a `uint64_t` word)

Bit-Packing

- In general: given a list of n integers, we can identify the largest one, say max , and encode the list using $n \lceil \log_2(max + 1) \rceil$ bits.
- Or perhaps: we already know that each integer is less than a given quantity U — the *universe* of representation — and we encode the list in $n \lceil \log_2 U \rceil$ bits. (Tight if U is close to max .)

Bit-Packing

- **Q.** But how do we actually read/write the bits?
- **A.** Using bit-wise operators + some logic.

C/C++ Bit-wise operators:
& (and), | (or), ^ (xor), << (left shift), >> (right shift).

x	y	x & y
0	0	0
0	1	0
1	0	0
1	1	1

x	y	x y
0	0	0
0	1	1
1	0	1
1	1	1

x	y	x ^ y
0	0	0
0	1	1
1	0	1
1	1	0

Bit-wise Operations

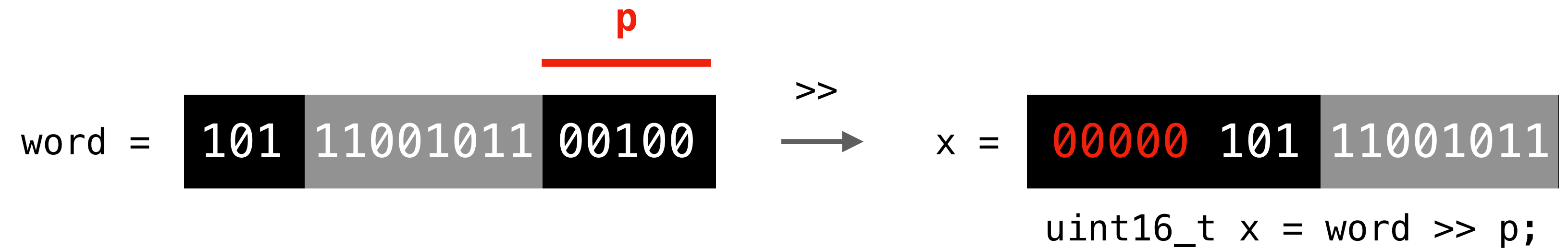
Shifting.

AND (masking).

OR (adding).

Bit-wise Operations

Shifting.

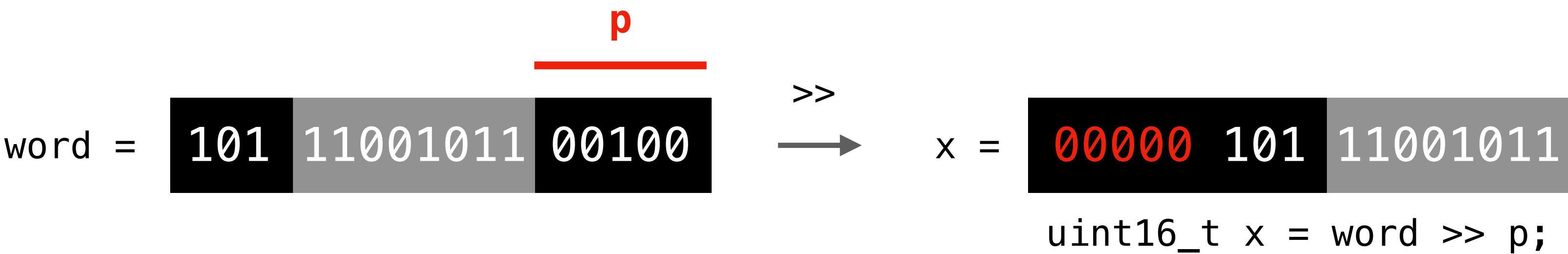


AND (masking).

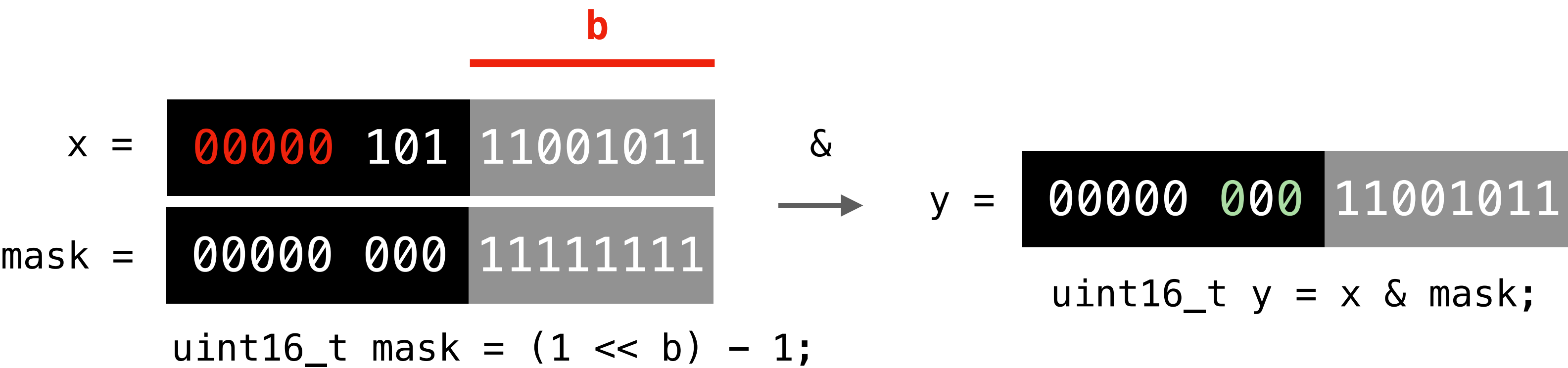
OR (adding).

Bit-wise Operations

Shifting.



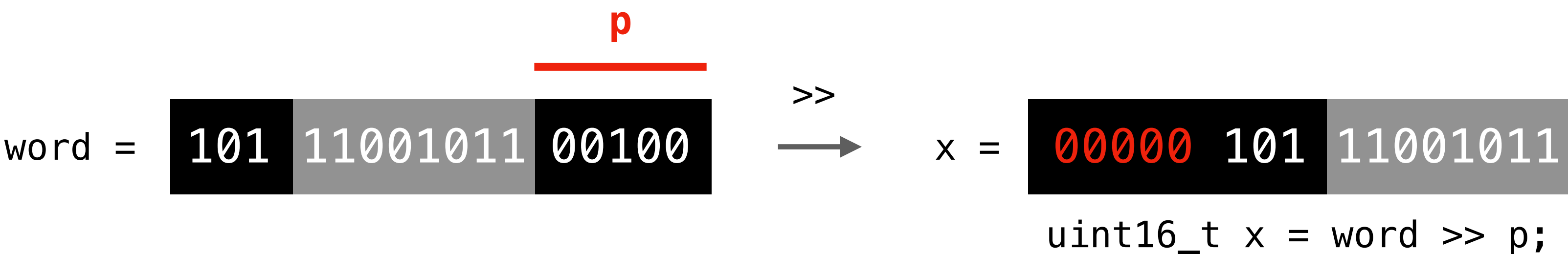
AND (masking).



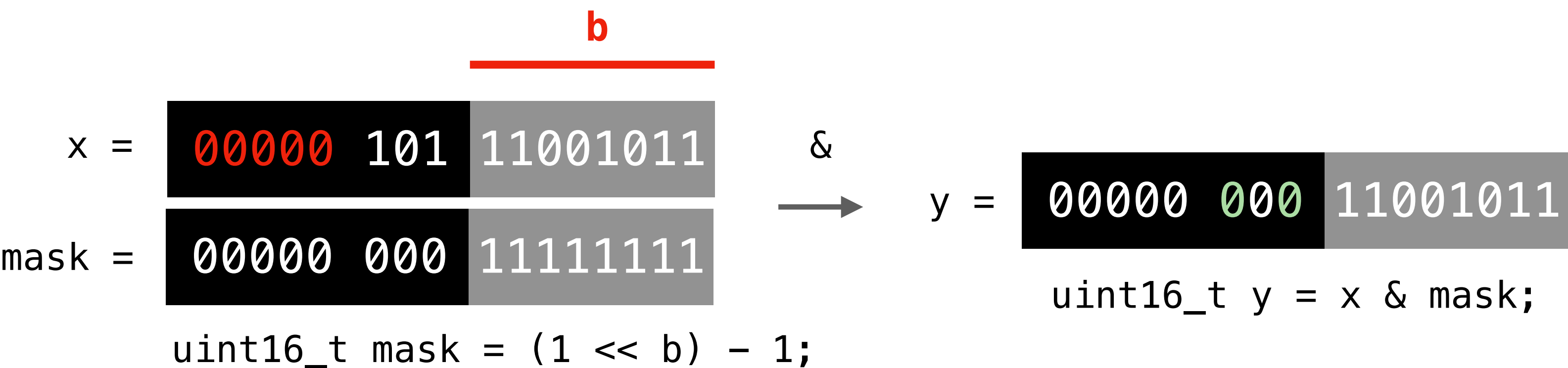
OR (adding).

Bit-wise Operations

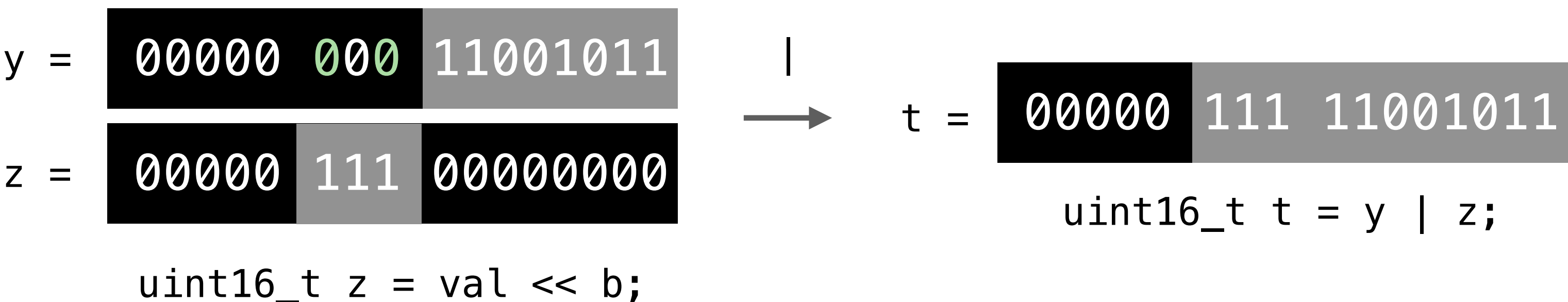
Shifting.



AND (masking).



OR (adding).



Read/Write Binary Data

- **Q.** What if you have more than 64 bits of data to read/write?

For example, you want to store 40 bit-packed records, and so you would need $37 \cdot 40 = 1480$ bits.

Read/Write Binary Data

- **Q.** What if you have more than 64 bits of data to read/write?

For example, you want to store 40 bit-packed records, and so you would need $37 \cdot 40 = 1480$ bits.

- **A.** Allocate a `std::vector<uint64_t>`.

If the vector has size n , then it holds $n \cdot 64$ bits.

So for 1480 bits we are going to need a vector of $\lceil 1480/64 \rceil = 24$ words.

- Armed with these basic tools (bit-wise operations and `std::vector<uint64_t>`), we can read/write *any* amount of bits.

Read/Write Binary Data

```
struct bit_vector_builder {
    bit_vector_builder() : m_num_bits(0) {}

    (...)

    void build(bit_vector& bv) {
        std::swap(m_num_bits, bv.m_num_bits);
        m_bits.swap(bv.m_bits);
    }

    void append_bits(uint64_t x, uint64_t len) {
        if (len == 0) return;
        uint64_t pos_in_word = m_num_bits % 64;
        m_num_bits += len;
        if (pos_in_word == 0) {
            m_bits.push_back(x);
        } else {
            *m_cur_word |= x << pos_in_word;
            if (len > 64 - pos_in_word) {
                m_bits.push_back(x >> (64 - pos_in_word));
            }
        }
        m_cur_word = &m_bits.back();
    }

private:
    uint64_t m_num_bits;
    std::vector<uint64_t> m_bits;
    uint64_t* m_cur_word;
};
```

build the bit_vector
by copying the bits

appends the len **least
significant** bits of x to the
current end of the bit-vector

keep track of the number of written bits

write

split the writes if necessary

backed data

```
struct bit_vector {
    bit_vector() : m_num_bits(0) {}

    (...)

    uint64_t get_bits(uint64_t pos, uint64_t len) const {
        if (len == 0) return 0;
        uint64_t block = pos / 64;
        uint64_t shift = pos % 64;
        uint64_t mask = -(len == 64) | ((1ULL << len) - 1);
        if (shift + len <= 64) return m_bits[block] >> shift & mask;
        return (m_bits[block] >> shift) |
            (m_bits[block + 1] << (64 - shift) & mask);
    }

    friend struct bit_vector_builder;

private:
    uint64_t m_num_bits;
    std::vector<uint64_t> m_bits;
};
```

read len bits starting from
position pos and return them

read

split the reads
if necessary

bit_vector_builder is granted
access to private members

Example

```
#include <iostream>
#include <vector>
```

some useful library includes

```
#include "bit_vector.hpp"
```

include the "bit_vector.hpp" file that contains the implementation of the classes bit_vector and bit_vector_builder

```
/* Definition of struct record. */
(...)
```

```
int main() {
    constexpr uint64_t n = 10000;
```

create an object a std::vector of records and reserve space for n records

```
    std::vector<record> records;
    records.reserve(n);
```

```
    /* Initialize the vector of records. */
    (...)
```

create an object bit_vector_builder and reserve space for $37n$ bits

```
    bit_vector_builder builder;
    builder.reserve(37 * n);
```

for each record: bit-pack its quantities

```
    for (auto r : records) {
        builder.append_bits(r.weight, 8);
        builder.append_bits(r.height, 8);
        builder.append_bits(r.day, 5);
        builder.append_bits(r.month, 4);
        builder.append_bits(r.year, 12);
    }
```

create a bit_vector object and "steal" (swap) the bits from the bit_vector_builder

```
    bit_vector packed_records;
    builder.build(packed_records);
```

```
    return 0;
```

```
}
```

Compile with:
g++ -std=c++11 packed_records.cpp -o packed_records

Run with:
./packed_records

Run-Length Encoding

- Observation: sometimes data features *long runs* of equal symbols.

Example 1:

```
111.00000000000000000000000000000000.1111111111111111.00000000000000.111111111111111111111111111111.000000000  
000.1.00000000000000000000000000000000.111111111.0000000000000000.1111111.00000000000000000000000000000000000000000  
000000000000
```

Example 2: this slide! If you model a pixel with a bit — colored pixel = 1; white pixel = 0 — then most bits are 0.

- Encode the *lengths* of the runs.

Example 1 — continued:

$[3, 26, 18, 13, 27, 12, 1, 25, 10, 15, 8, 45] \rightarrow (12 \cdot 8) = 96$ bits vs. 203 bits (assuming a run is shorter than 256)

Example 2 — more general:

aaaaaaa.bbbb.aaaaaaaaaaaaaaaaa.cccccccccc.aaaaaaa \rightarrow (a,7) (b,4) (a,16) (c,12) (a,9)