

Fusion Trees

Part One

Recap from Last Time

Ordered Dictionaries

- An **ordered dictionary** maintains a set S drawn from an ordered universe \mathcal{U} and supports these operations:
 - **lookup**(x), which returns whether $x \in S$;
 - **insert**(x), which adds x to S ;
 - **delete**(x), which removes x from S ;
 - **max**() / **min**(), which return the maximum or minimum element of S ;
 - **successor**(x), which returns the smallest element of S greater than x ; and
 - **predecessor**(x), which returns the largest element of S smaller than x .
- Reasoning by analogy:
Ordered Dictionary : BST :: Queue : Linked List

Our Machine Model

- We will assume we're working on a machine where memory is segmented into w -bit words.
- Although on any one fixed machine w is a constant, in general, don't assume this is the case. 32-bit was the norm until fairly recently, and before that 16-bit was standard.
- We'll assume that the C integer operators work in constant time, and will not assume we have access to operators beyond them.

+ - * / % << >> & | ^ = <=

- This restriction is somewhat arbitrary and has been revisited / questioned / refined by future work in this area. Come talk to me after class if you're curious about this!

Integer Ordered Dictionaries

- Suppose that $\mathcal{U} = [U] = \{0, 1, \dots, U - 1\}$.
- The ***y-Fast Trie*** is an ordered dictionary structure for the set $[U]$ where all operations run in expected, amortized time $O(\log \log U)$.
 - Note that when $n = \omega(\log U)$, this is exponentially better than a binary search tree!
- Space usage is $\Theta(n)$, where n is the number of elements in the trie.

New Stuff!

A Key Technique: ***Word-Level Parallelism***

Word-Level Parallelism

- On a standard computer, arithmetic and logical operations on a machine word take time $O(1)$.
- We can perform certain classes of operations (addition, shifts, etc.) on $\Theta(w)$ bits in time $O(1)$.
 - Think of this as a weak form of parallel computation, where we can work over multiple bits in parallel with a limited set of operations.
- With some creativity, we can harness these primitives to build operations that run in time $O(1)$ but work on $\omega(1)$ objects.
- Let's see a quick example...

Word-Level Parallelism

a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8
1101110	0101110	1111000	1001101	0101111	0001101	1110111	1100001
b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8
0011010	1000101	0010100	0100000	1010000	0100010	1000100	0001000

Word-Level Parallelism

a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8

1101110 **0101110** **1111000** **1001101** **0101111** **0001101** **1110111** **1100001**

b_1 b_2 b_3 b_4 b_5 b_6 b_7 b_8

0011010 **1000101** **0010100** **0100000** **1010000** **0100010** **1000100** **0001000**

Word-Level Parallelism

a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8

01101110 00101110 01111000 01001101 00101111 00001101 01110111 01100001

b_1 b_2 b_3 b_4 b_5 b_6 b_7 b_8

00011010 01000101 00010100 00100000 01010000 00100010 01000100 00001000

Word-Level Parallelism

01101110 00101110 01111000 01001101 00101111 00001101 01110111 01100001
+ 00011010 01000101 00010100 00100000 01010000 00100010 01000100 00001000

10001000 01110011 10001100 01101101 01111111 00101111 10111011 01101001

We've performed eight logical additions with a single add instruction!

The Landscape

- Preprocessing/runtime tradeoffs:
“Yes, we have to do a lot of work, but it’s a one-time cost and everything is cheaper after that.”
- Randomization:
“We might have to do a lot of work, but it’s unlikely that we’ll do so.”
- Amortization:
“Yes, we have to do a lot of work every once and a while, but only after a period of doing very little.”
- Word-level parallelism:
“We have to do a lot of work, but we don’t have to perform many operations to do it.”

Where We're Going

- Today is all about using word-level parallelism to speed up integer data structures.
- Today, we'll see two techniques:
 - First, the *sardine tree*, a fast ordered dictionary for extremely small integers.
 - Next, a technique for finding the *most-significant bit* of an integer in $O(1)$ machine operations.
- When we come back next time, we'll see how to adapt these techniques into the *fusion tree*, an ordered dictionary for integers that fit into a machine word.

Sardine Trees

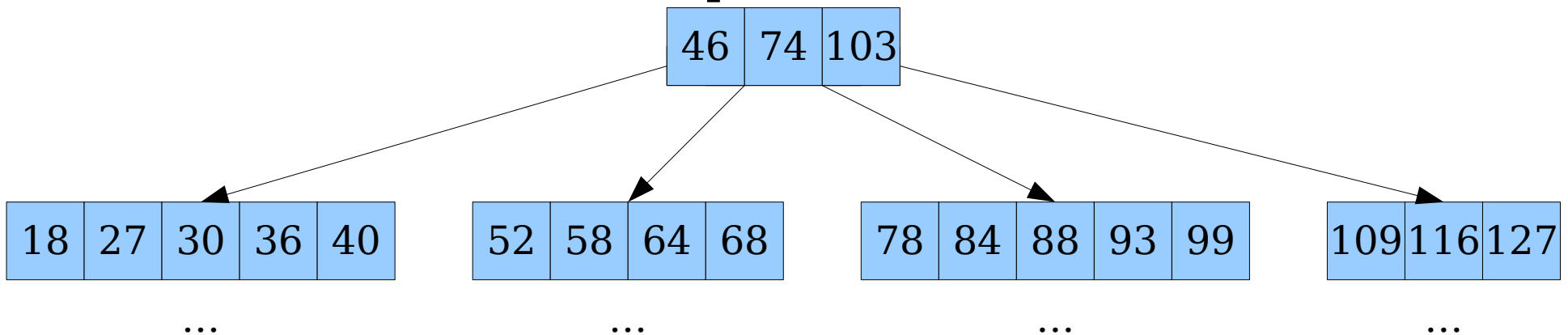
These actually aren't called sardine trees. I couldn't find a name for them anywhere and thought that this title was appropriate. Let me know if there's a more proper name to associate with them!

The Setup

- Let w denote the machine word size.
- Imagine you want to store a collection of s -bit integers, where s is small compared to w .
 - For example, storing 7-bit integers on a 64-bit machine would have $s = 7$ and $w = 64$.
- Can we build an ordered dictionary that takes advantage of the small key size?

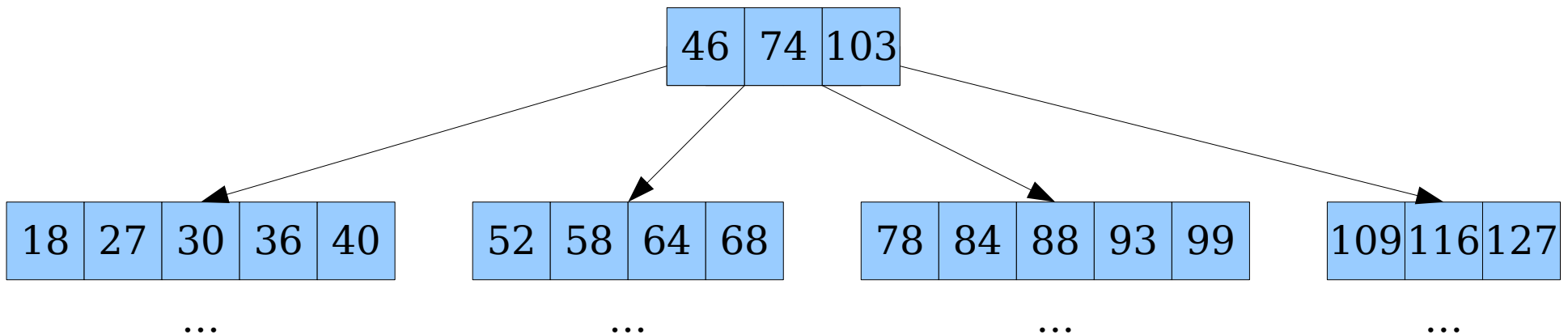
A Refresher: B-Trees

- A **B-tree** is a multiway tree with a tunable parameter b called the **order** of the tree.
- Each nodes stores $\Theta(b)$ keys. The height of the tree is $\Theta(\log_b n)$.
- Most operations (**lookup**, **insert**, **delete**, **successor**, **predecessor**, etc.) perform a top-down search of the tree, doing some amount of work per node.
- Runtime of each operation is $O(f(b) \log_b n)$, where $f(b)$ is the amount of work done per node.



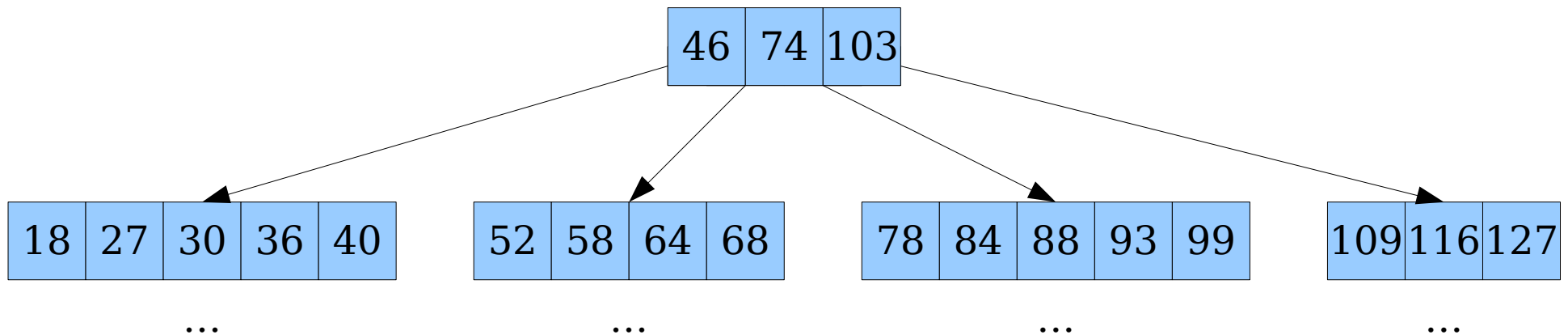
B-Tree Lookups

- When performing a *lookup* of a key k in a B-tree node, we need to determine how many keys in the node are less than or equal to k .
 - This is called the *rank* of k .
- For example, in the top node:
 $rank(40) = 0$ $rank(74) = 2$ $rank(107) = 3$



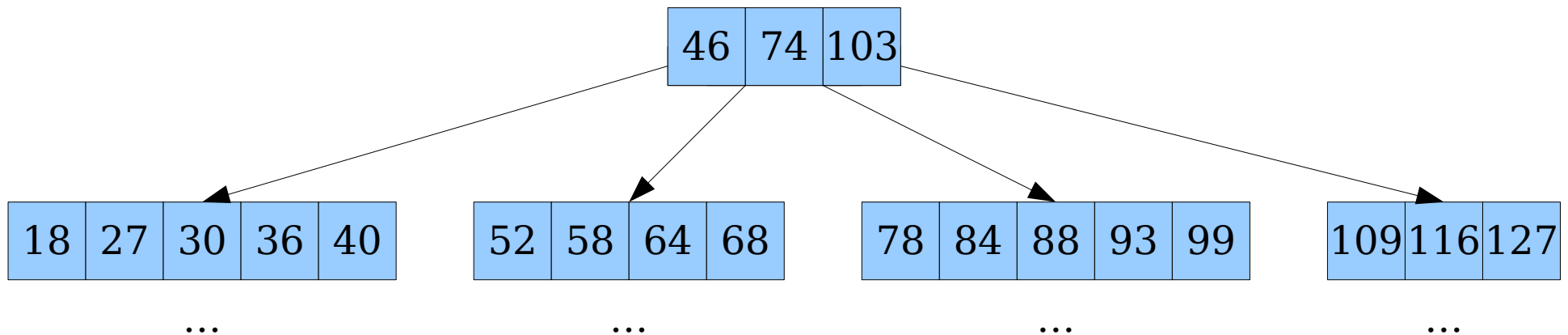
B-Tree Lookups

- Knowing *rank*(k) in a particular node tells us which key to compare against and which child to descend into.
- **Question:** How quickly can we determine *rank*(k) in a B-tree node?



B-Tree Lookups

- We can determine *rank*(k) with a linear search in each B-tree node for a total lookup cost of $O(b \cdot \log_b n)$.
- We can determine *rank*(k) with a binary search in each B-tree node for a total lookup cost of
$$O(\log_b n \cdot \log b) = O(\log n).$$
- **Claim:** If we can fit all the keys in a node into $O(1)$ machine words, we can determine *rank*(k) in time $O(1)$ for total lookup cost of $O(\log_b n)$.



How is this possible?

Warmup: Comparing Two Values

- Imagine we have two s -bit integers x and y and want to determine whether $x \geq y$.
- How might we do this?

$$\begin{array}{r} \\ \\ \\ \\ \hline \end{array}$$

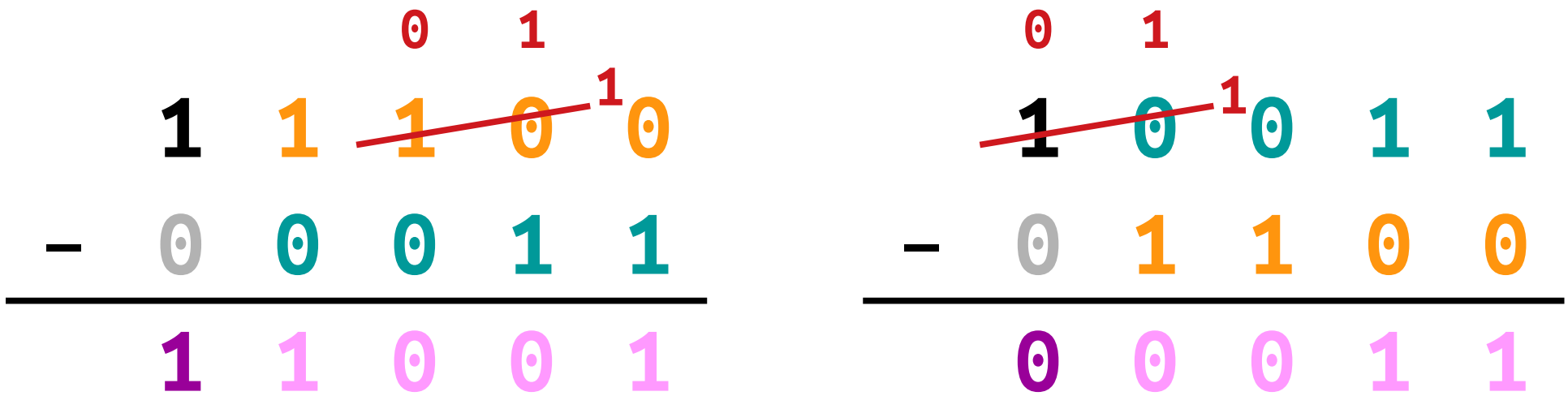
Diagram illustrating the comparison of two 5-bit integers. The top number is 11100 (with a red '1' above the second bit) and the bottom number is 00011. A red line is drawn from the top bit of the top number to the top bit of the bottom number, indicating the first bit where they differ. The result of the comparison is 11001.

$$\begin{array}{r} \\ \\ \\ \\ \hline \end{array}$$

Diagram illustrating the comparison of two 5-bit integers. The top number is 10011 (with a red '1' above the second bit) and the bottom number is 01100. A red line is drawn from the top bit of the top number to the top bit of the bottom number, indicating the first bit where they differ. The result of the comparison is 00011.

Warmup: Comparing Two Values

- Imagine we have two s -bit integers x and y and want to determine whether $x \geq y$.
- How might we do this?



This bit tells us whether the first number was as least as big as the second!

Comparing Multiple Values

- This technique can be extended to work on multiple values in parallel.
- For example, here's how we'd compare eight pairs of 7-bit numbers by doing a single 64-bit subtraction:

a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8

1101110 0101110 1111000 1001101 0101111 0001101 1110111 1100001

b_1 b_2 b_3 b_4 b_5 b_6 b_7 b_8

0011010 1000101 0010100 0100000 1010000 0100010 1000100 0001000

Comparing Multiple Values

- This technique can be extended to work on multiple values in parallel.
- For example, here's how we'd compare eight pairs of 7-bit numbers by doing a single 64-bit subtraction:

a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8

11101110 10101110 11111000 11001101 10101111 10001101 11110111 11100001

b_1 b_2 b_3 b_4 b_5 b_6 b_7 b_8

00011010 01000101 00010100 00100000 01010000 00100010 01000100 00001000

Comparing Multiple Values

- This technique can be extended to work on multiple values in parallel.
- For example, here's how we'd compare eight pairs of 7-bit numbers by doing a single 64-bit subtraction:

```
  11101110  10101110  11111000  11001101  10101111  10001101  11110111  11100001
- 00011010  01000101  00010100  00100000  01010000  00100010  01000100  00001000
-----
  11010100  01101001  11100100  10101101  01011111  01101011  10110011  11011001
```

Comparing Multiple Values

- This technique can be extended to work on multiple values in parallel.
- For example, here's how we'd compare eight pairs of 7-bit numbers by doing a single 64-bit subtraction:

```
  11101110 10101110 11111000 11001101 10101111 10001101 11110111 11100001
- 00011010 01000101 00010100 00100000 01010000 00100010 01000100 00001000
-----
  11010100 01101001 11100100 10101101 01011111 01101011 10110011 11011001
```

Comparing Multiple Values

- This technique can be extended to work on multiple values in parallel.
- For example, here's how we'd compare eight pairs of 7-bit numbers by doing a single 64-bit subtraction:

```
11101110 10101110 11111000 11001101 10101111 10001101 11110111 11100001
- 00011010 01000101 00010100 00100000 01010000 00100010 01000100 00001000
-----
11010100 01101001 11100100 10101101 01011111 01101011 10110011 11011001
```

This technique is used in practice, including [*the glibc implementation of strlen*](#). Thanks to former CS166 student Jane Lange for pointing this out!

Fundamental Primitive: *Parallel Compare*

1. Pack a list of values x_1, \dots, x_k into a machine word X , separated by 1s.
2. Pack a list of values y_1, \dots, y_k into a machine word Y , separated by 0s.
3. Compute $X - Y$. The bit preceding $x_i - y_i$ is 1 if $x_i \geq y_i$ and 0 otherwise.

Assuming the packing can be done in $O(1)$ time, this compares all the pairs in $O(1)$ machine word operations.

Back to B-Trees

- **Recall:** The whole reason we're interested in making these comparisons is so that we can find how many keys in a B-tree node are less than or equal to a query key k .
- **Idea:** Store the (s -bit) keys in the B-tree node in a single (w -bit) machine word, with zeros interspersed:

y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8
41	93	103	106	107	109	110	127

Back to B-Trees

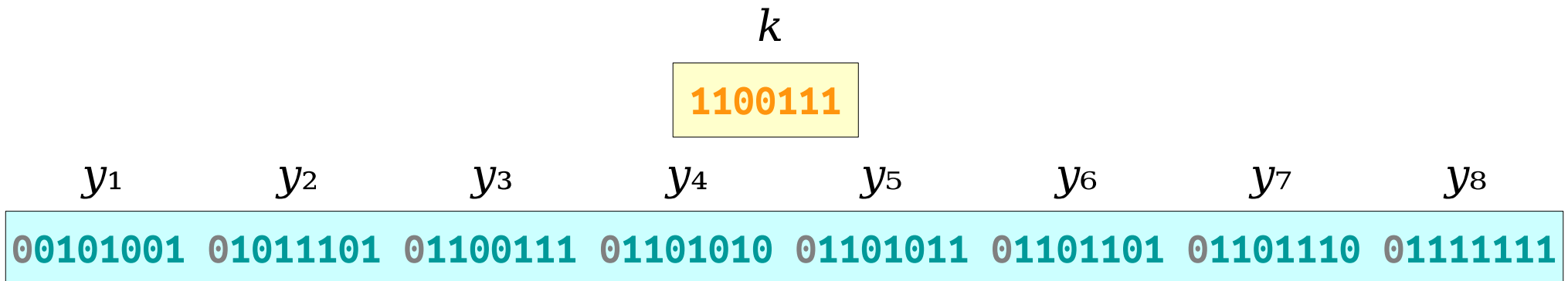
- **Recall:** The whole reason we're interested in making these comparisons is so that we can find how many keys in a B-tree node are less than or equal to a query key k .
- **Idea:** Store the (s -bit) keys in the B-tree node in a single (w -bit) machine word, with zeros interspersed:

y_1 y_2 y_3 y_4 y_5 y_6 y_7 y_8

00101001 01011101 01100111 01101010 01101011 01101101 01101110 01111111

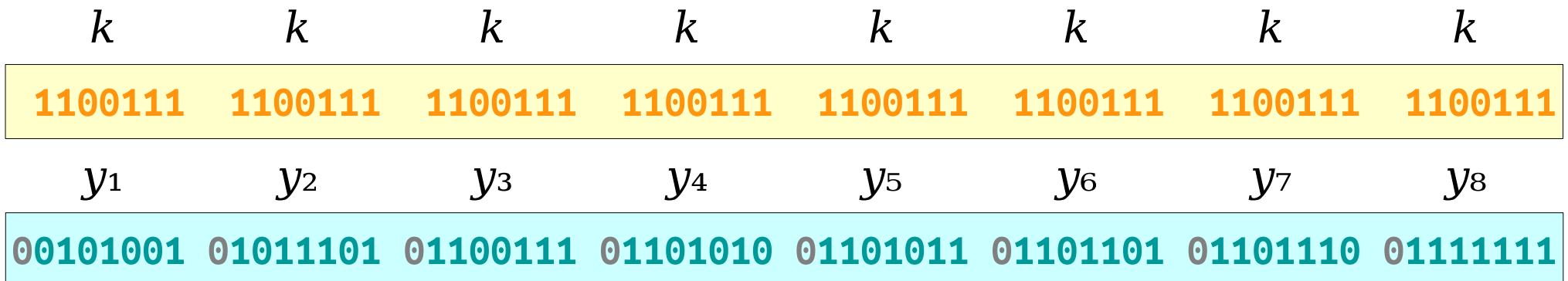
Rank in $O(1)$

- To perform a lookup for the key k , form a number by replicating k multiple times with 1s interspersed.



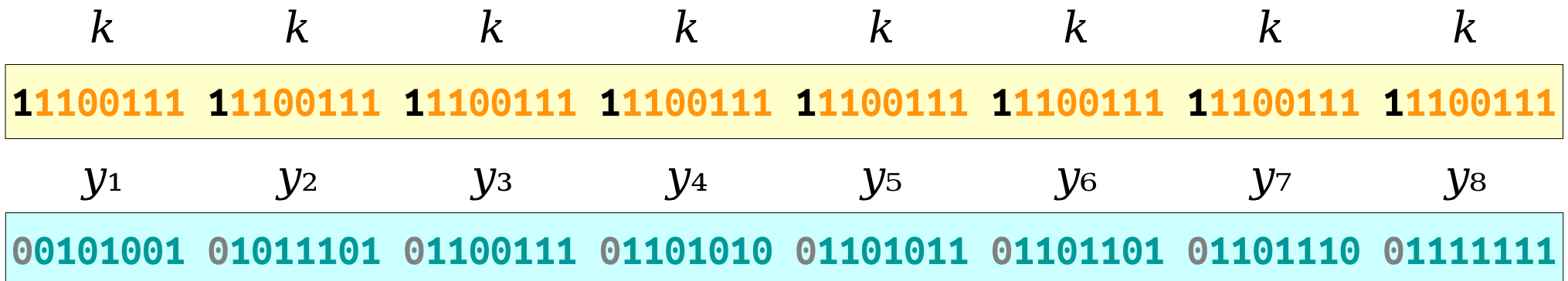
Rank in $O(1)$

- To perform a lookup for the key k , form a number by replicating k multiple times with 1s interspersed.



Rank in $O(1)$

- To perform a lookup for the key k , form a number by replicating k multiple times with 1s interspersed.
- Subtract the B-tree key number from it to do a parallel comparison.



Rank in $O(1)$

- To perform a lookup for the key k , form a number by replicating k multiple times with 1s interspersed.
- Subtract the B-tree key number from it to do a parallel comparison.

```
11100111 11100111 11100111 11100111 11100111 11100111 11100111 11100111
- 00101001 01011101 01100111 01101010 01101011 01101101 01101110 01111111
-----
10111110 10001010 10000000 01111101 01111100 01111010 01111001 01101000
```

Rank in $O(1)$

How do we do this?

- To perform a lookup for the key k , form a number by replicating k multiple times with 1s interspersed.
- Subtract the B-tree key number from it to do a parallel comparison.
- Count up how many of the sentinel bits in the resulting number are equal to 1. This is the number of keys in the node less than or equal to k .

Or this?

11100111	11100111	11100111	11100111	11100111	11100111	11100111	11100111
- 00101001	01011101	01100111	01101010	01101011	01101101	01101110	01111111
<hr/>							
10111110	10001010	10000000	01111101	01111100	01111010	01111001	01101000

Rank: **3**

Back in Base Ten

- Suppose you have a one-digit number m .
- You want to form this base-10 number:

mmm

- Is there a nice series of arithmetical operations that will produce this?
- **Answer:** Compute $m \times 111$.
- Why does this work?

$$\begin{aligned} m \times 111 &= m \times 100 + m \times 10 + m \times 1 \\ &= m00 + 0m0 + 00m \\ &= mmm. \end{aligned}$$

Back in Base Ten

- Suppose you have a one-digit number m .
- You want to form this base-10 number:

mmm

- Is there a nice series of arithmetical operations that will produce this?
- **Answer:** Compute $m \times 111$.
- Why does this work?

$$\begin{aligned} m \times 111 &= m \ll 2 + m \ll 1 + m \ll 0 \\ &= m00 + 0m0 + 00m \\ &= mmm. \end{aligned}$$

Back in Base Ten

- Suppose you have a two-digit number mn .
- You want to form this base-10 number:

$mnmnmn$

- Is there a nice series of arithmetical operations that will produce this?
- **Answer:** Compute $mn \times 10,101$.
- Why does this work?

$$\begin{aligned} mn \times 10,101 &= mn \times 10,000 + mn \times 100 + mn \times 1 \\ &= mn0000 + 00mn00 + 0000mn \\ &= mnmnmn. \end{aligned}$$

Back in Base Ten

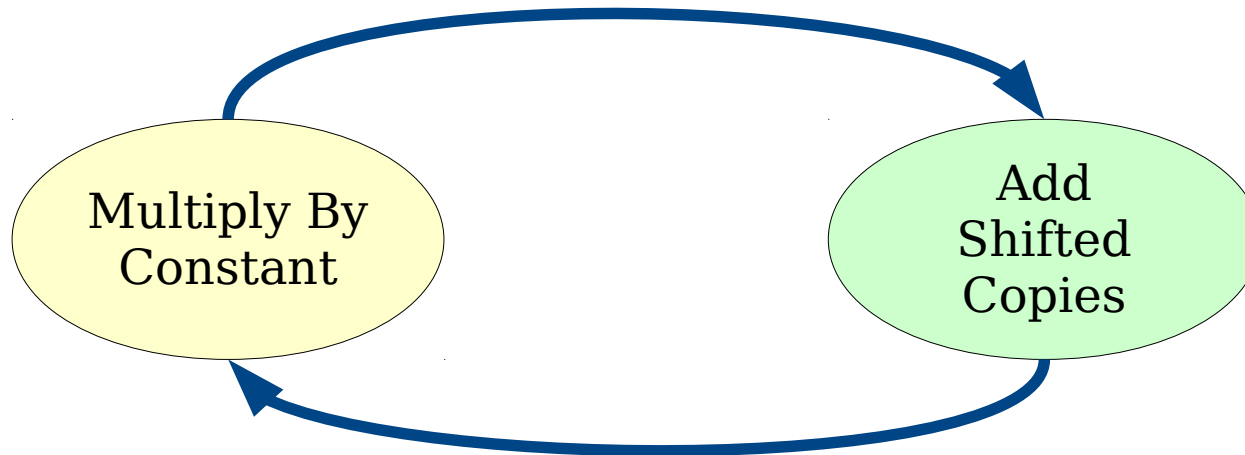
- Suppose you have a two-digit number mn .
- You want to form this base-10 number:

$mnmnmn$

- Is there a nice series of arithmetical operations that will produce this?
- **Answer:** Compute $mn \times 10,101$.
- Why does this work?

$$\begin{aligned} mn \times 10,101 &= mn \ll 4 + mn \ll 2 + mn \ll 0 \\ &= mn0000 + 00mn00 + 0000mn \\ &= mnmnmn. \end{aligned}$$

Back in Base Ten



- **Answer:** Compute $mn \times 10,101$.
- Why does this work?

$$\begin{aligned} mn \times 10,101 &= mn \ll 4 + mn \ll 2 + mn \ll 0 \\ &= mn0000 + 00mn00 + 0000mn \\ &= mnmnmn. \end{aligned}$$

Back in Base Ten

- Suppose you have a three-digit number mnp .
- You want to form this base-10 number:

$mnp000mnp0mnp$

- Is there a nice series of arithmetical operations that will produce this?
- **Answer:** Compute $mnp \times /* \text{ something } */$.

$mnp000mnp0mnp$

$$= mnp \ll 10 + mnp \ll 4 + mnp \ll 0$$

$$= mnp \times 10^{10} + mnp \times 10^4 + mnp \times 10^0$$

$$= mnp \times 10,000,010,001$$

Back in Base Ten

- Suppose you have a three-digit number mnp .
- You want to form this base-10 number:

$mnp000mnp0mnp$

- Is there a nice series of arithmetical operations that will produce this?
- **Answer:** Compute $mnp \times 10,000,010,001$.

$mnp000mnp0mnp$

$$= mnp \ll 10 + mnp \ll 4 + mnp \ll 0$$

$$= mnp \times 10^{10} + mnp \times 10^4 + mnp \times 10^0$$

$$= mnp \times 10,000,010,001$$

Computing Rank in $O(1)$

```
const uint64_t kMultiplier = 0b100000000100000000...1000000001;
```

```
uint64_t tiledK = k * kMultiplier;
```

k

1100111

y_1

y_2

y_3

y_4

y_5

y_6

y_7

y_8

00101001 01011101 01100111 01101010 01101011 01101101 01101110 01111111

Computing Rank in $O(1)$

```
const uint64_t kMultiplier = 0b100000000100000000...1000000001;
```

```
uint64_t tiledK = k * kMultiplier;
```

k k k k k k k k

1100111 1100111 1100111 1100111 1100111 1100111 1100111 1100111

y_1 y_2 y_3 y_4 y_5 y_6 y_7 y_8

00101001 01011101 01100111 01101010 01101011 01101101 01101110 01111111

Fundamental Primitive: *Parallel Tile*

1. Form a number M with a 1 bit at the end of each location to tile k .
2. Compute $M \times k$.

Assuming step (1) can be done in time $O(1)$, this produces many copies of k in time $O(1)$.

Computing Rank in $O(1)$

```
const uint64_t kMultiplier = 0b100000000100000000...1000000001;
```

```
uint64_t tiledK = k * kMultiplier;
```

k k k k k k k k

1100111 1100111 1100111 1100111 1100111 1100111 1100111 1100111

y_1 y_2 y_3 y_4 y_5 y_6 y_7 y_8

00101001 01011101 01100111 01101010 01101011 01101101 01101110 01111111

Computing Rank in $O(1)$

```
const uint64_t kMultiplier = 0b10000000010000000...100000001;  
const uint64_t kOnesMask   = 0b10000000010000000...010000000;
```

```
uint64_t tiledK      = k * kMultiplier;
```

k k k k k k k k

1100111 1100111 1100111 1100111 1100111 1100111 1100111 1100111

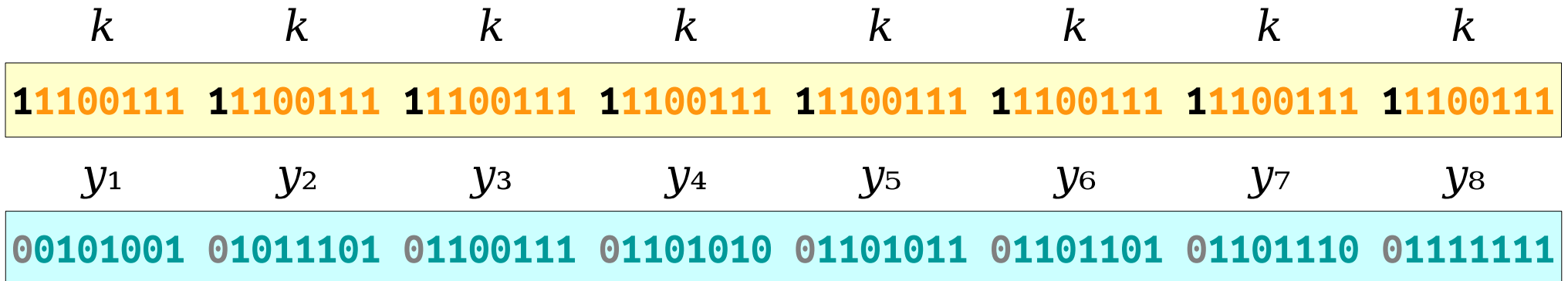
y_1 y_2 y_3 y_4 y_5 y_6 y_7 y_8

00101001 01011101 01100111 01101010 01101011 01101101 01101110 01111111

Computing Rank in $O(1)$

```
const uint64_t kMultiplier = 0b10000000010000000...1000000001;  
const uint64_t kOnesMask   = 0b10000000010000000...0100000000;
```

```
uint64_t tiledK      = (k * kMultiplier) | kOnesMask;
```



Computing Rank in $O(1)$

```
const uint64_t kMultiplier = 0b10000000010000000...100000001;  
const uint64_t kOnesMask   = 0b10000000010000000...010000000;
```

```
uint64_t tiledK      = (k * kMultiplier) | kOnesMask;
```

```
11100111 11100111 11100111 11100111 11100111 11100111 11100111 11100111  
00101001 01011101 01100111 01101010 01101011 01101101 01101110 01111111
```

Computing Rank in $O(1)$

```
const uint64_t kMultiplier = 0b10000000010000000...1000000001;  
const uint64_t kOnesMask   = 0b10000000010000000...0100000000;
```

```
uint64_t tiledK      = (k * kMultiplier) | kOnesMask;  
uint64_t comparison = tiledK - packedKeys;
```

```
11100111 11100111 11100111 11100111 11100111 11100111 11100111 11100111  
- 00101001 01011101 01100111 01101010 01101011 01101101 01101110 01111111  
-----  
10111110 10001010 10000000 01111101 01111100 01111010 01111001 01101000
```

Computing Rank in $O(1)$

```
const uint64_t kMultiplier = 0b10000000010000000...1000000001;  
const uint64_t kOnesMask   = 0b10000000010000000...0100000000;
```

```
uint64_t tiledK      = (k * kMultiplier) | kOnesMask;  
uint64_t comparison = (tiledK - packedKeys) & kOnesMask;
```

```
11100111 11100111 11100111 11100111 11100111 11100111 11100111 11100111  
- 00101001 01011101 01100111 01101010 01101011 01101101 01101110 01111111  
-----  
10000000 10000000 10000000 00000000 00000000 00000000 00000000 00000000
```

How do we count
how many of these
bits are set?

Summing Up Flags

- After performing our subtraction, we're left with a number like this one, where the highlighted bits are "interesting" to us.
- **Goal:** Add up these "interesting" values using $O(1)$ word operations.

a00000000 **b**00000000 **c**00000000 **d**00000000

An Initial Idea

- To sum up the flags, we could extract each bit individually and add the result.
- ***The catch:*** This takes time $\Theta(r)$, where r is the number of times we tiled our value.
- Can we do better?

a00000000 **b**00000000 **c**00000000 **d**00000000

A Shifty Solution

- Given this number:

a00000000 **b**00000000 **c**00000000 **d**00000000

we want to compute **a** + **b** + **c** + **d**.

- We can't efficiently isolate **a**, **b**, **c**, and **d**.
- **Claim:** We don't have to!

$$\begin{array}{r} d \\ c \\ b \\ + a \\ \hline \end{array}$$

A Shifty Solution

- Given this number:

a00000000 **b**00000000 **c**00000000 **d**00000000

we want to compute **a** + **b** + **c** + **d**.

- We can't efficiently isolate **a**, **b**, **c**, and **d**.
- **Claim:** We don't have to!

	d
	c
	b
+	a

A Shifty Solution

- Given this number:

a00000000 **b**00000000 **c**00000000 **d**00000000

we want to compute **a** + **b** + **c** + **d**.

- We can't efficiently isolate **a**, **b**, **c**, and **d**.
- Claim:** We don't have to!

$$\begin{array}{r} \mathbf{a}00000000 \mathbf{b}00000000 \mathbf{c}00000000 \mathbf{d}00000000 \text{ 00000000 00000000 00000000} \\ \mathbf{a}00000000 \mathbf{b}00000000 \mathbf{c}00000000 \mathbf{d}00000000 \text{ 00000000 00000000} \\ \mathbf{a}00000000 \mathbf{b}00000000 \mathbf{c}00000000 \mathbf{d}00000000 \text{ 00000000} \\ + \mathbf{a}00000000 \mathbf{b}00000000 \mathbf{c}00000000 \mathbf{d}00000000 \end{array}$$

A Shifty Solution

- Given this number:

a00000000 **b**00000000 **c**00000000 **d**00000000

we want to compute **a** + **b** + **c** + **d**.

- We can't efficiently isolate **a**, **b**, **c**, and **d**.
- Claim:** We don't have to!

a00000000 b00000000 c00000000 **d**00000000 00000000 00000000 00000000

a00000000 b00000000 **c**00000000 d00000000 00000000 00000000

a00000000 **b**00000000 c00000000 d00000000 00000000

+

a00000000 b00000000 c00000000 d00000000

A Shifty Solution

- Given this number:

a00000000 **b**00000000 **c**00000000 **d**00000000

we want to compute **a** + **b** + **c** + **d**.

- We can't efficiently isolate **a**, **b**, **c**, and **d**.
- Claim:** We don't have to!

a00000000 b00000000 c00000000 **d**00000000

a00000000 b00000000 **c**00000000

a00000000 **b**00000000

+

a00000000 b00000000 c00000000 d00000000

????????? ?????????? ?????????? **sum** ?????????? ?????????? ?????????? ??????????

This is a series of shifts and adds. It's equivalent to multiplying our original number by some well-chosen spreader!

Fundamental Primitive: *Parallel Add*

1. Perform a *parallel tile* with an appropriate multiplier to place all leading bits on top of one another.
2. Use a bitmask and bitshift to isolate those bits.

Assuming the multiplier for part (1) and the mask and shift for part (2) can be computed in time $O(1)$, this takes time $O(1)$.

Computing Rank in $O(1)$

```
const uint64_t kMultiplier = 0b10000000010000000...100000001;  
const uint64_t kOnesMask   = 0b10000000010000000...010000000;
```

```
uint64_t tiledK      = (k * kMultiplier) | kOnesMask;  
uint64_t comparison = (tiledK - packedKeys) & kOnesMask;
```

a0000000 b0000000 c0000000 d0000000

Computing Rank in $O(1)$

```
const uint64_t kMultiplier = 0b10000000010000000...100000001;  
const uint64_t kOnesMask   = 0b10000000010000000...010000000;
```

```
uint64_t tiledK      = (k * kMultiplier) | kOnesMask;  
uint64_t comparison = (tiledK - packedKeys) & kOnesMask;
```

```
const uint64_t kStacker = 0b100000001000000...100000001;
```

```
uint64_t rank = comparison * kStacker;
```

a0000000 b0000000 c0000000 d0000000

Computing Rank in $O(1)$

```
const uint64_t kMultiplier = 0b10000000010000000...1000000001;  
const uint64_t kOnesMask   = 0b10000000010000000...0100000000;
```

```
uint64_t tiledK      = (k * kMultiplier) | kOnesMask;  
uint64_t comparison = (tiledK - packedKeys) & kOnesMask;
```

```
const uint64_t kStacker = 0b100000001000000...100000001;
```

```
uint64_t rank = comparison * kStacker;
```

```
  a00000000 b00000000 c00000000 d00000000 00000000 00000000 00000000  
    a0000000 b0000000 c0000000 d0000000 00000000 00000000  
      a0000000 b0000000 c0000000 d0000000 00000000  
+                               a0000000 b0000000 c0000000 d0000000
```

Computing Rank in $O(1)$

```
const uint64_t kMultiplier = 0b10000000010000000...1000000001;  
const uint64_t kOnesMask   = 0b10000000010000000...0100000000;
```

```
uint64_t tiledK      = (k * kMultiplier) | kOnesMask;  
uint64_t comparison = (tiledK - packedKeys) & kOnesMask;
```

```
const uint64_t kStacker = 0b100000001000000...100000001;
```

```
uint64_t rank = comparison * kStacker;
```

a0000000 b0000000 c0000000 **d**0000000 00000000 00000000 00000000

 a0000000 b0000000 **c**0000000 d0000000 00000000 00000000

 a0000000 **b**0000000 c0000000 d0000000 00000000

+

a0000000 b0000000 c0000000 d0000000

???????? ???? sum ????????? ????????? ?????????

Computing Rank in $O(1)$

```
const uint64_t kMultiplier = 0b10000000010000000...1000000001;  
const uint64_t kOnesMask   = 0b10000000010000000...0100000000;
```

```
uint64_t tiledK      = (k * kMultiplier) | kOnesMask;  
uint64_t comparison = (tiledK - packedKeys) & kOnesMask;
```

```
const uint64_t kStacker = 0b100000001000000...100000001;  
const uint8_t  kShift   = 31;
```

```
uint64_t rank = comparison * kStacker;
```

a00000000 b00000000 c00000000 **d**00000000 00000000 00000000 00000000

a00000000 b00000000 **c**00000000 d00000000 00000000 00000000

a00000000 **b**00000000 c00000000 d00000000 00000000

+

a00000000 b00000000 c00000000 d00000000

????????? ?????????? ?????????? **sum** ?????????? ?????????? ?????????? ??????????

Computing Rank in $O(1)$

```
const uint64_t kMultiplier = 0b10000000010000000...1000000001;  
const uint64_t kOnesMask   = 0b10000000010000000...0100000000;
```

```
uint64_t tiledK      = (k * kMultiplier) | kOnesMask;  
uint64_t comparison = (tiledK - packedKeys) & kOnesMask;
```

```
const uint64_t kStacker = 0b100000010000000...100000001;  
const uint8_t  kShift   = 31;
```

```
uint64_t rank = (comparison * kStacker) >> kShift;
```

a0000000 b0000000 c0000000 **d**0000000 00000000 00000000 00000000

a0000000 b0000000 **c**0000000 d0000000 00000000 00000000

a0000000 **b**0000000 c0000000 d0000000 00000000

+

a0000000 b0000000 c0000000 d0000000

???????? ???? ?????? ?????????? ?????????? ?????????? ?????????? ?????? **sum**

Computing Rank in $O(1)$

```
const uint64_t kMultiplier = 0b10000000010000000...1000000001;  
const uint64_t kOnesMask   = 0b10000000010000000...0100000000;
```

```
uint64_t tiledK      = (k * kMultiplier) | kOnesMask;  
uint64_t comparison = (tiledK - packedKeys) & kOnesMask;
```

```
const uint64_t kStacker = 0b10000001000000...100000001;  
const uint8_t  kShift   = 31;  
const uint64_t kMask    = 0b111;
```

```
uint64_t rank = (comparison * kStacker) >> kShift;
```

a0000000 b0000000 c0000000 **d**0000000 00000000 00000000 00000000

a0000000 b0000000 **c**0000000 d0000000 00000000 00000000

a0000000 **b**0000000 c0000000 d0000000 00000000

+

a0000000 b0000000 c0000000 d0000000

???????? ???? ?????? ?????????? ?????????? ?????????? ?????????? ?????? **sum**

Computing Rank in $O(1)$

```
const uint64_t kMultiplier = 0b10000000010000000...1000000001;  
const uint64_t kOnesMask   = 0b10000000010000000...0100000000;
```

```
uint64_t tiledK      = (k * kMultiplier) | kOnesMask;  
uint64_t comparison = (tiledK - packedKeys) & kOnesMask;
```

```
const uint64_t kStacker = 0b100000010000000...100000001;  
const uint8_t  kShift   = 31;  
const uint64_t kMask    = 0b1111;
```

```
uint64_t rank = ((comparison * kStacker) >> kShift) & kMask;
```

a0000000 b0000000 c0000000 **d**0000000 00000000 00000000 00000000

a0000000 b0000000 **c**0000000 d0000000 00000000 00000000

a0000000 **b**0000000 c0000000 d0000000 00000000

+

a0000000 b0000000 c0000000 d0000000

SUM

Fundamental Primitive: *Parallel Rank*

1. Perform a *parallel tile* to create n copies of the key k , prefixed by 1's.
2. Perform a *parallel compare* of the key k against values x_1, \dots, x_n .
3. Perform a *parallel add* to sum those values into some total t .
4. Return t .

Assuming the parallel compare and parallel have their internal constants computed in advance, this runs in time $O(1)$.

The Sardine Tree

- Let w be the word size and s be some (much) smaller number of bits.
- A **sardine tree** is a B-tree of order $\Theta(w/s)$ where the keys in a node are packed into a single machine word.
 - Get it? The keys are “packed” tightly into a machine word! I’m funny.
- Each node is annotated with several values (the masks and multipliers from the preceding slide), which are updated in time $O(1)$ whenever a key is added or removed.
- Supports all ordered dictionary operations in time

$$O(\log_b n) = \mathbf{O(\log_{w/s} n)}.$$

The Scorecard

- Here's the performance breakdown for the sardine tree.
- Notice that the runtime performance is strictly better than that of a BST!
- Notice that the space usage is sublinear, since each node stores multiple keys!

The Sardine Tree

- **lookup**: $O(\log_{w/s} n)$
- **insert**: $O(\log_{w/s} n)$
- **delete**: $O(\log_{w/s} n)$
- **max**: $O(\log_{w/s} n)$
- **succ**: $O(\log_{w/s} n)$
- Space: $\Theta(n \cdot s/w)$

For Comparison

- Here's what that would look like if we used a y-fast trie instead.
- Since our keys range from 0 to $2^s - 1$ and the y-fast trie operations take time $O(\log \log U)$, each operation takes time $O(\log s)$.

The y-Fast Trie

- **lookup**: $O(\log s)$
 - **insert**: $O(\log s)^*$
 - **delete**: $O(\log s)^*$
 - **max**: $O(\log s)$
 - **succ**: $O(\log s)$
 - Space: $\Theta(n)$
- * Expected, amortized

Time-Out for Announcements!

Problem Sets

- Problem Set Five was due at 2:30PM today.
 - Using late days, you can submit it up until Thursday at 2:30PM.
- ***Congrats! You're done with the CS166 problem sets!***

Take-Home Exam

- The take-home midterm is now available online, and hardcopies are available up front.
- It's due Thursday at 2:30PM on GradeScope.
- ***Best of luck on the exam!*** Seriously, we hope you all knock this one out of the park.

Back to CS166!

Word-Level Parallelism Tricks #2:
Most-Significant Bits

Most-Significant Bits

- The ***most-significant bit*** function, denoted **$\text{msb}(n)$** , outputs the index of the highest 1 bit set in the binary representation of number n .
- Some examples:
 $\text{msb}(0110) = 2$ $\text{msb}(010100) = 4$ $\text{msb}(1111) = 3$
- Note that $\text{msb}(0)$ is undefined.
- Mathematically, $\text{msb}(n)$ is the largest value of k such that $2^k \leq n$. (*Do you see why?*)

Most-Significant Bits

- Although we didn't have this name earlier in the quarter, you've seen a place where we needed to efficiently compute $\text{msb}(n)$.
- Do you remember where?
- **Answer:** In the sparse table RMQ structure, where computing $\text{RMQ}(i, j)$ requires computing the largest number k where $2^k \leq j - i + 1$.
- That's exactly the value of $\text{msb}(j - i + 1)$!

Most-Significant Bits

- On many architectures, there's a single assembly instruction that computes $\text{msb}(n)$.
 - on x86, it's **BSR** (**b**it **s**can **r**everse).
- On others, nothing like this exists.
 - MIPS, for example.
- **Question:** How would we compute $\text{msb}(n)$ assuming we only have access to the regular C operators?

+ - * / % << >> & | ^ == <=

Computing msb

- In Problem Set 1, you (probably) computed $\text{msb}(n)$ by building a lookup table mapping each value of n to $\text{msb}(n)$.
- **The Good:** This takes time $O(1)$ to evaluate.
- **The Bad:** The preprocessing time, and space usage, is $\Theta(U)$, where U is the maximum value we'll be querying for.
- **The Ugly:** In the worst case $U = 2^w$.
- Can we do better?

Most-Significant Bits

- There's a simple $O(w)$ -time algorithm for computing $\text{msb}(n)$ that just checks all the bits until a 1 is found:

```
for (uint8_t bit = 64; bit > 0; bit--) {  
    if (n & (uint64_t(1) << (bit - 1))) {  
        return bit;  
    }  
}  
flailAndPanic();
```

- Can we do better?

Computing msb

- We can improve this runtime to $O(\log w)$ by using a binary search:
 - Check if any bits in the bottom half of the bits of n are set.
 - If so, recursively explore the upper half of n .
 - If not, recursively explore the lower half of n .
- We can test whether any bit in a range is set by ANDing with a mask of 1s and seeing if the result is nonzero:

```
11011100 10111011 11000100 11010101 11100110 11110111 11000010 00110010
^ 11111111 11111111 11111111 11111111 00000000 00000000 00000000 00000000
-----
11011100 10111011 11000100 11010101 00000000 00000000 00000000 00000000
```

- Can we do better?

Claim: For any machine word size w , there is an algorithm that uses $O(1)$ machine operations and $O(1)$ space - independently of w - and computes $\text{msb}(n)$.

This is not obvious!

How is this possible?

Not Starting from Scratch

- We're not going into this problem blind. We've seen a bunch of useful techniques so far:
 - **Parallel compare:** We can compare a bunch of small numbers in parallel in $O(1)$ machine word operations.
 - **Parallel tile:** We can take a small number and "tile" it multiple times in $O(1)$ machine word operations.
 - **Parallel add:** If we have a bunch of "flag" bits spread out evenly, we can add them all up in $O(1)$ machine word operations.
 - **Parallel rank:** We can find the rank of a small number in an array of small numbers in $O(1)$ machine word operations.
- This is an impressive array of techniques. Let's see if we can reuse or adapt them.

Not Starting from Scratch

- We're not going into this problem blind. We've seen a bunch of useful techniques so far:
 - **Parallel compare:** We can compare a bunch of small numbers in parallel in $O(1)$ machine word operations.
 - **Parallel tile:** We can take a small number and “tile” it multiple times in $O(1)$ machine word operations.
 - **Parallel add:** If we have a bunch of “flag” bits spread out evenly, we can add them all up in $O(1)$ machine word operations.
 - **Parallel rank:** We can find the rank of a small number in an array of small numbers in $O(1)$ machine word operations.
- This is an impressive array of techniques. Let's see if we can reuse or adapt them.

MSBs as Ranks

- **Recall:** $\text{msb}(n)$ is the largest value of k for which $2^k \leq n$.
- **Idea:** Imagine we have an array of all the powers of two that we can represent in a machine word. Then $\text{msb}(n)$ is the rank of n in that array!

00000000 00000000 00100000 00000010 00000000 00100000 00100001 00101111

2^0	2^1	2^2	2^3	2^4	2^5	2^6	...	2^{50}	2^{51}	2^{52}	2^{53}	2^{54}	2^{55}	2^{56}	2^{57}	2^{58}	...	2^{61}	2^{62}	2^{63}
-------	-------	-------	-------	-------	-------	-------	-----	----------	----------	----------	----------	----------	----------	----------	----------	----------	-----	----------	----------	----------

The Problem

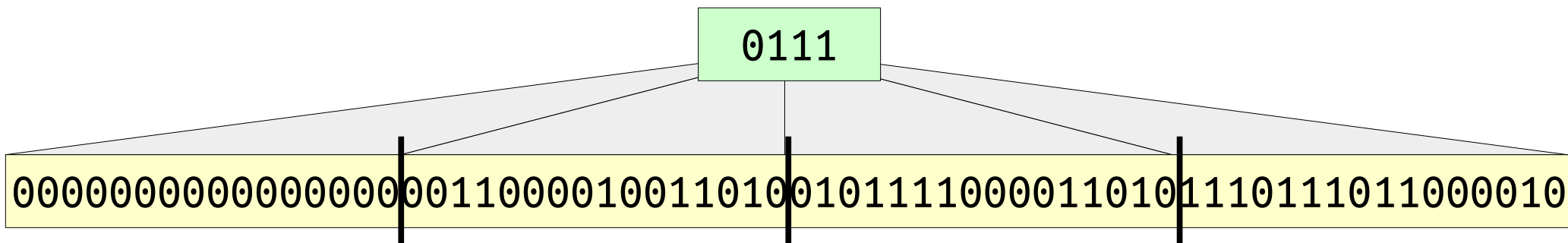
- We can compute the rank of a value in an array assuming that
 - all the array entries fit into a single machine word, and
 - the value in question is the same size as the array entries.
- Neither of these requirements hold here.
- **Question:** Can we reduce the size of our number?

00000000 00000000 00100000 00000010 00000000 00100000 00100001 00101111

2^0	2^1	2^2	2^3	2^4	2^5	2^6	...	2^{50}	2^{51}	2^{52}	2^{53}	2^{54}	2^{55}	2^{56}	2^{57}	2^{58}	...	2^{61}	2^{62}	2^{63}
-------	-------	-------	-------	-------	-------	-------	-----	----------	----------	----------	----------	----------	----------	----------	----------	----------	-----	----------	----------	----------

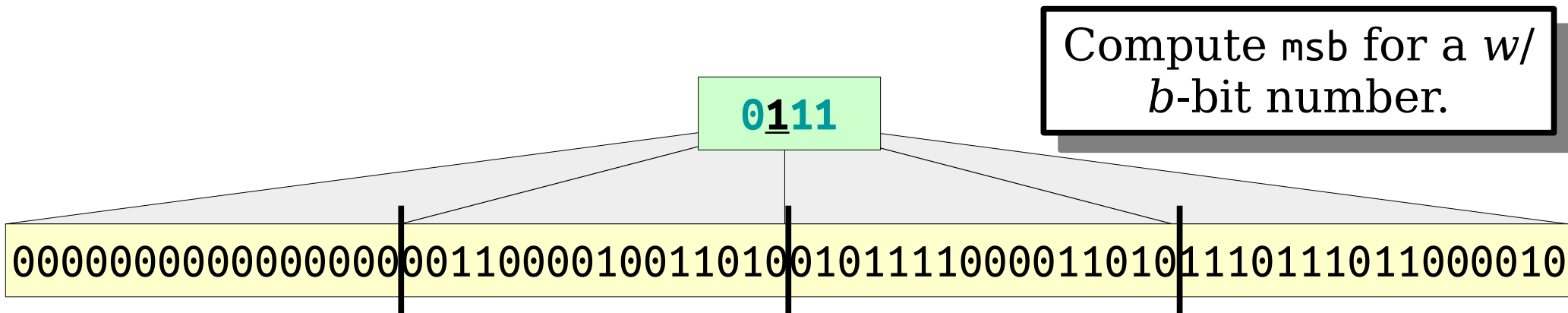
A Nice Decomposition

- Imagine we want to compute the most-significant bit of a w -bit integer.
 - In what follows, we'll pick $w = 64$, but this works for any w .
- We ultimately want to be finding the MSB of numbers with way fewer than w bits.
- **Idea:** Split w into some number of blocks of size b . Then,
 - find the index of the highest block with at least one 1 bit set, then
 - find the index of the highest bit within that block.



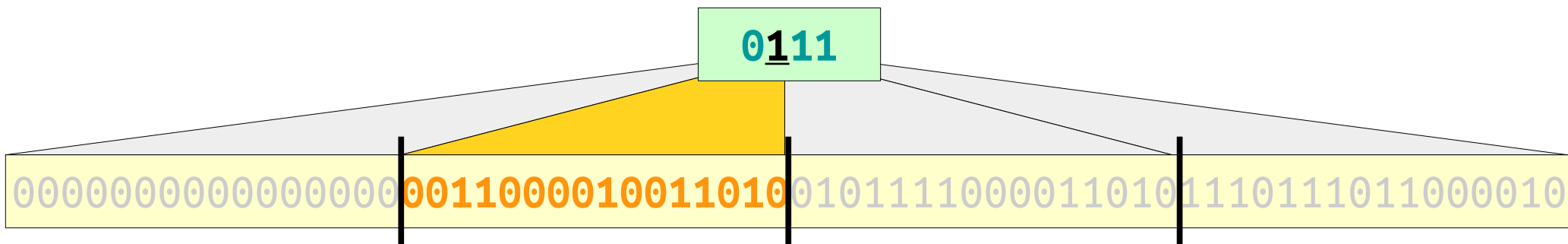
A Nice Decomposition

- Imagine we want to compute the most-significant bit of a w -bit integer.
 - In what follows, we'll pick $w = 64$, but this works for any w .
- We ultimately want to be finding the MSB of numbers with way fewer than w bits.
- **Idea:** Split w into some number of blocks of size b . Then,
 - find the index of the highest block with at least one 1 bit set, then
 - find the index of the highest bit within that block.



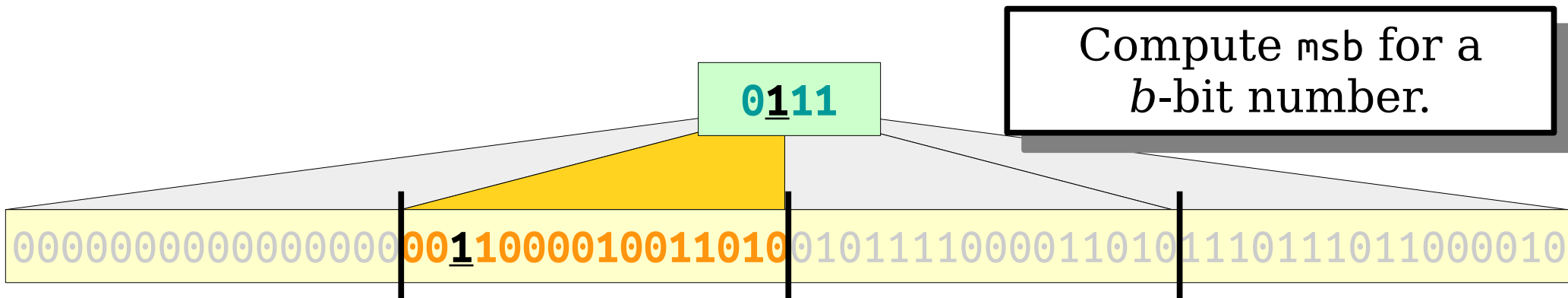
A Nice Decomposition

- Imagine we want to compute the most-significant bit of a w -bit integer.
 - In what follows, we'll pick $w = 64$, but this works for any w .
- We ultimately want to be finding the MSB of numbers with way fewer than w bits.
- **Idea:** Split w into some number of blocks of size b . Then,
 - find the index of the highest block with at least one 1 bit set, then
 - find the index of the highest bit within that block.



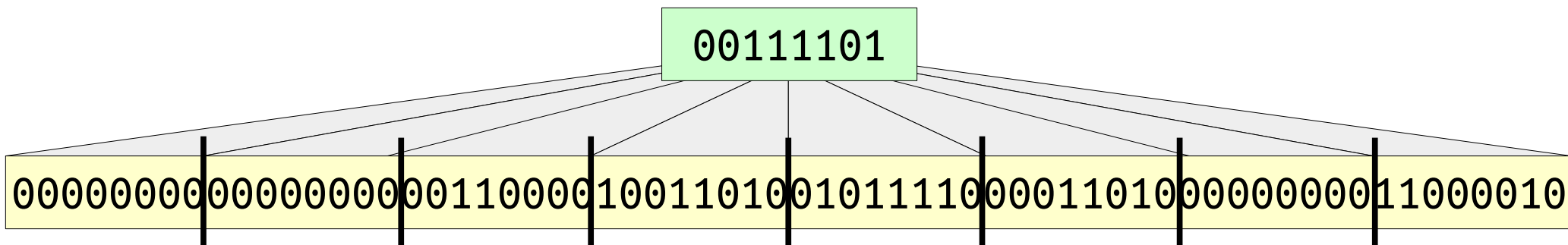
A Nice Decomposition

- Imagine we want to compute the most-significant bit of a w -bit integer.
 - In what follows, we'll pick $w = 64$, but this works for any w .
- We ultimately want to be finding the MSB of numbers with way fewer than w bits.
- **Idea:** Split w into some number of blocks of size b . Then,
 - find the index of the highest block with at least one 1 bit set, then
 - find the index of the highest bit within that block.



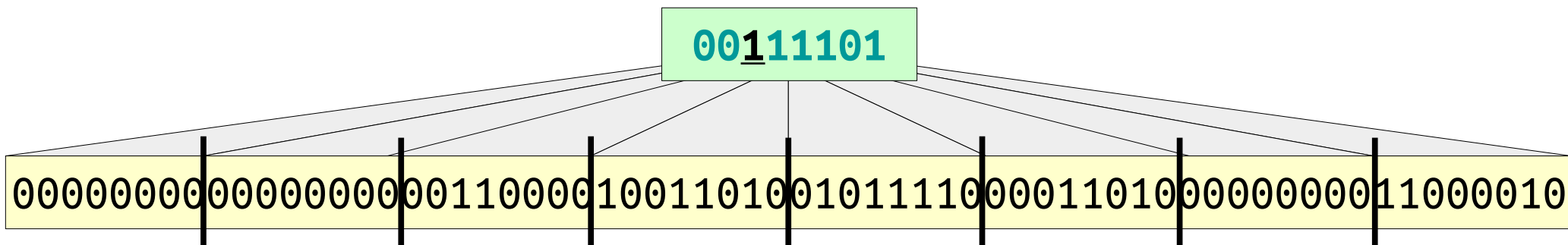
A Nice Decomposition

- We will compute the MSB for w -bit integers by solving MSB for b and w/b -bit integers.
- What choice of b minimizes $\max\{b, w/b\}$?
- **Answer:** Pick $b = w^{1/2}$.
- So now we need to see how to
 - solve $\text{msb}(n)$ for integers with $w^{1/2}$ bits, and
 - replace each block with a bit indicating whether that block contains a 1.



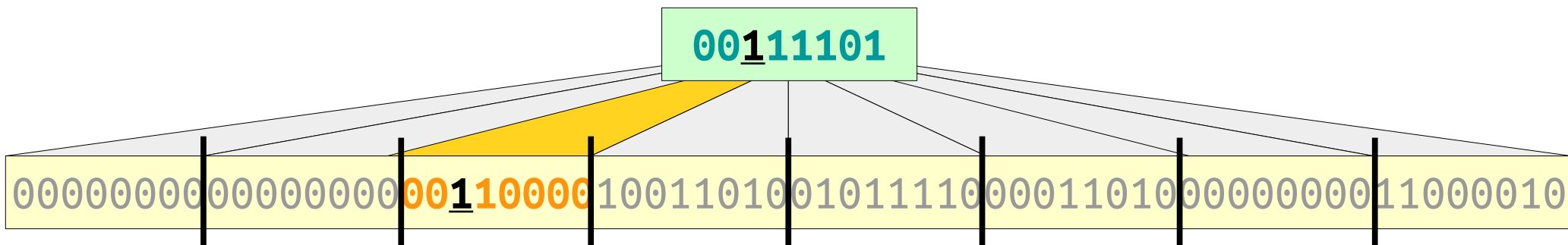
A Nice Decomposition

- We will compute the MSB for w -bit integers by solving MSB for b and w/b -bit integers.
- What choice of b minimizes $\max\{b, w/b\}$?
- **Answer:** Pick $b = w^{1/2}$.
- So now we need to see how to
 - solve $\text{msb}(n)$ for integers with $w^{1/2}$ bits, and
 - replace each block with a bit indicating whether that block contains a 1.



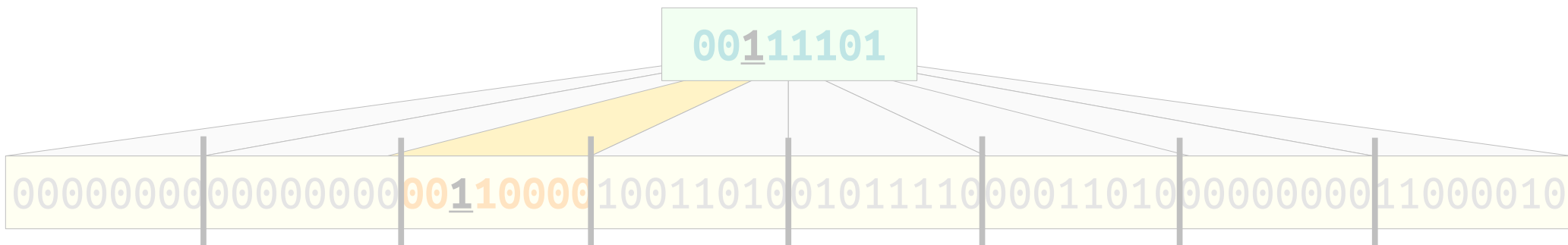
A Nice Decomposition

- We will compute the MSB for w -bit integers by solving MSB for b and w/b -bit integers.
- What choice of b minimizes $\max\{b, w/b\}$?
- **Answer:** Pick $b = w^{1/2}$.
- So now we need to see how to
 - solve $\text{msb}(n)$ for integers with $w^{1/2}$ bits, and
 - replace each block with a bit indicating whether that block contains a 1.



A Nice Decomposition

- We will compute the MSB for w -bit integers by solving MSB for b and w/b -bit integers.
- What choice of b minimizes $\max\{b, w/b\}$?
- **Answer:** Pick $b = w^{1/2}$.
- So now we need to see how to
 - solve $\text{msb}(n)$ for integers with $w^{1/2}$ bits, and
 - replace each block with a bit indicating whether that block contains a 1.



MSB for $w^{1/2}$ Bits

- **Recall:** We can compute $\text{msb}(n)$ by counting how many powers of two are less than or equal to n .
- If our numbers have size $w^{1/2}$, there are $w^{1/2}$ powers of two to compare against.
- Each of those powers of two has $w^{1/2}$ bits, so all of those powers of two can be packed into a single machine word!
- **Idea:** Use our $O(1)$ -time rank algorithm!

00000001 00000010 00000100 00001000 00010000 00100000 01000000 10000000

MSB for $w^{1/2}$ Bits

- If our numbers have size $w^{1/2}$, there are $w^{1/2}$ powers of two to compare against, each of which has $w^{1/2}$ bits.
- Our parallel comparison prepends an extra bit to each number to compare.

00000001 00000010 00000100 00001000 00010000 00100000 01000000 10000000

MSB for $w^{1/2}$ Bits

- If our numbers have size $w^{1/2}$, there are $w^{1/2}$ powers of two to compare against, each of which has $w^{1/2}$ bits.
- Our parallel comparison prepends an extra bit to each number to compare.
- That's barely – just barely – too many bits to fit into a machine word.

00000000100000001000000010000000100000001000000010000000100000001000000010000000

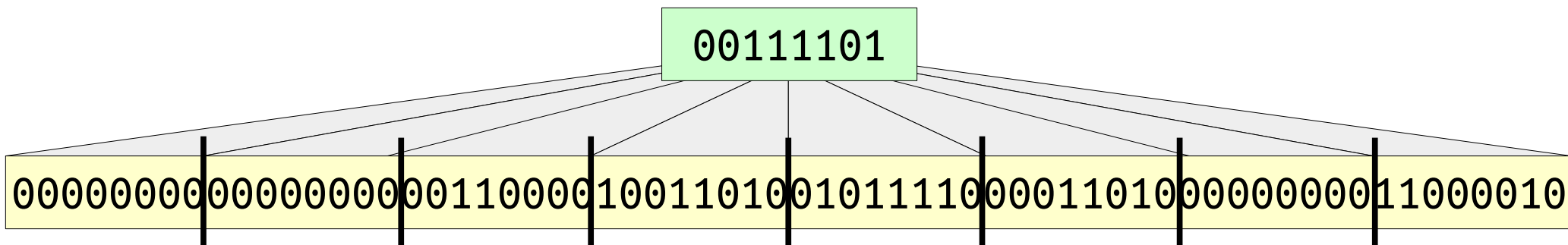
MSB for $w^{1/2}$ Bits

- **Claim:** This is an engineering problem at this point.
- **Option 1:** Split the powers of two into two different machine words and do two rank calculations.
- **Option 2:** Special-case the most-significant bit to reduce the number of bits to check.
- Either way, we find that the work done here is $O(1)$ machine operations, with no dependency on the word size w !

000000001000000010000000100000001000000010000001000000010000000010000000010000000010000000

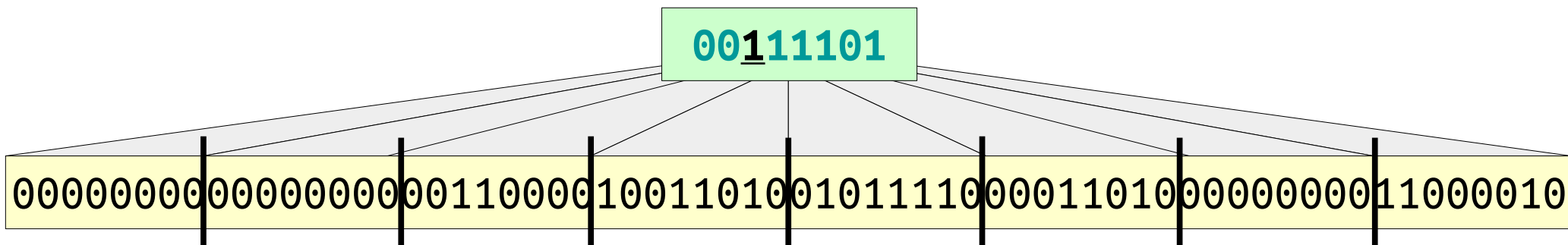
A Nice Decomposition

- We need to see how to
 - solve $\text{msb}(n)$ for integers with $w^{1/2}$ bits, and
 - replace each block with a bit indicating whether that block contains a 1.



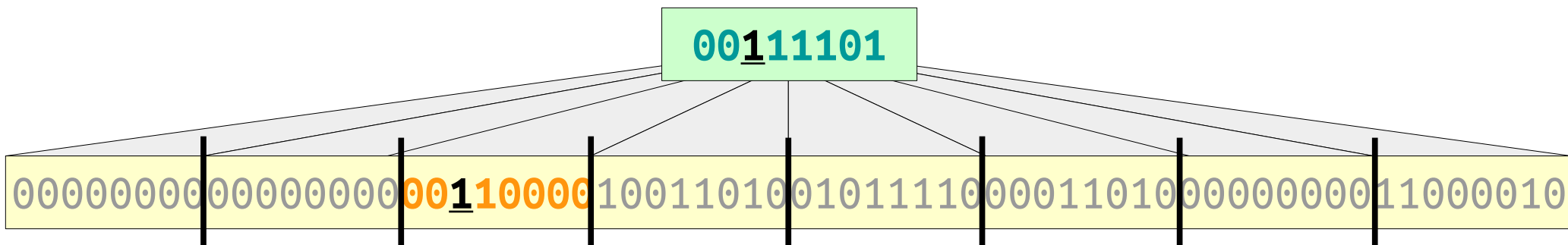
A Nice Decomposition

- We need to see how to
 - solve $\text{msb}(n)$ for integers with $w^{1/2}$ bits, and
 - replace each block with a bit indicating whether that block contains a 1.



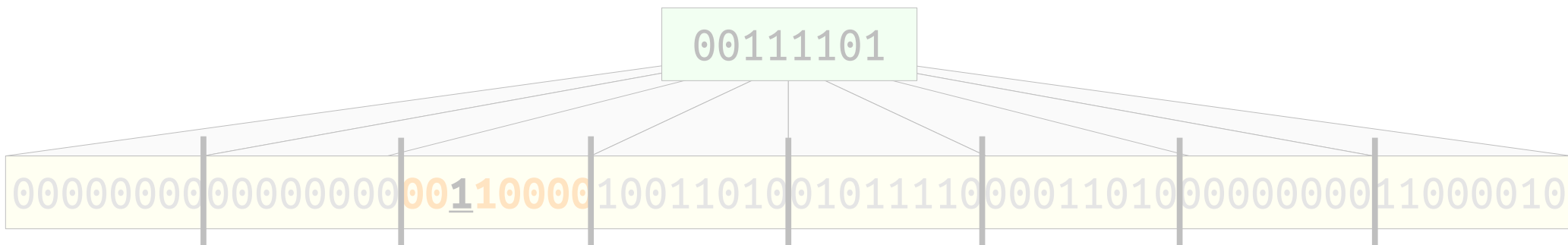
A Nice Decomposition

- We need to see how to
 - solve $\text{msb}(n)$ for integers with $w^{1/2}$ bits, and
 - replace each block with a bit indicating whether that block contains a 1.



A Nice Decomposition

- We need to see how to
 - solve $\text{msb}(n)$ for integers with $w^{1/2}$ bits, and
 - replace each block with a bit indicating whether that block contains a 1.



Identifying Active Blocks

b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8
00000000	00011001	11110000	10011010	10000000	00011010	11101110	11000010

Observation: A block contains a 1 bit if its first bit is 1 or its lower 7 bits contain a 1.

Identifying Active Blocks

b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8
00000000	00011001	11110000	10011010	10000000	00011010	11101110	11000010

Observation: A block contains a 1 bit if its first bit is 1 or its lower 7 bits contain a 1.

Identifying Active Blocks

b_1 b_2 b_3 b_4 b_5 b_6 b_7 b_8

00000000 00011001 11110000 10011010 10000000 00011010 11101110 11000010

10000000 10000000 10000000 10000000 10000000 10000000 10000000 10000000

00000000 00000000 10000000 10000000 10000000 00000000 10000000 10000000

Observation: A block contains a 1 bit if its first bit is 1 or its lower 7 bits contain a 1.

Identifying Active Blocks

b_1 b_2 b_3 b_4 b_5 b_6 b_7 b_8

00000000 00011001 11110000 10011010 10000000 00011010 11101110 11000010

00000000 00000000 10000000 10000000 10000000 00000000 10000000 10000000

High bit set?

Observation: A block contains a 1 bit if its first bit is 1 or its lower 7 bits contain a 1.

Identifying Active Blocks

b_1 b_2 b_3 b_4 b_5 b_6 b_7 b_8

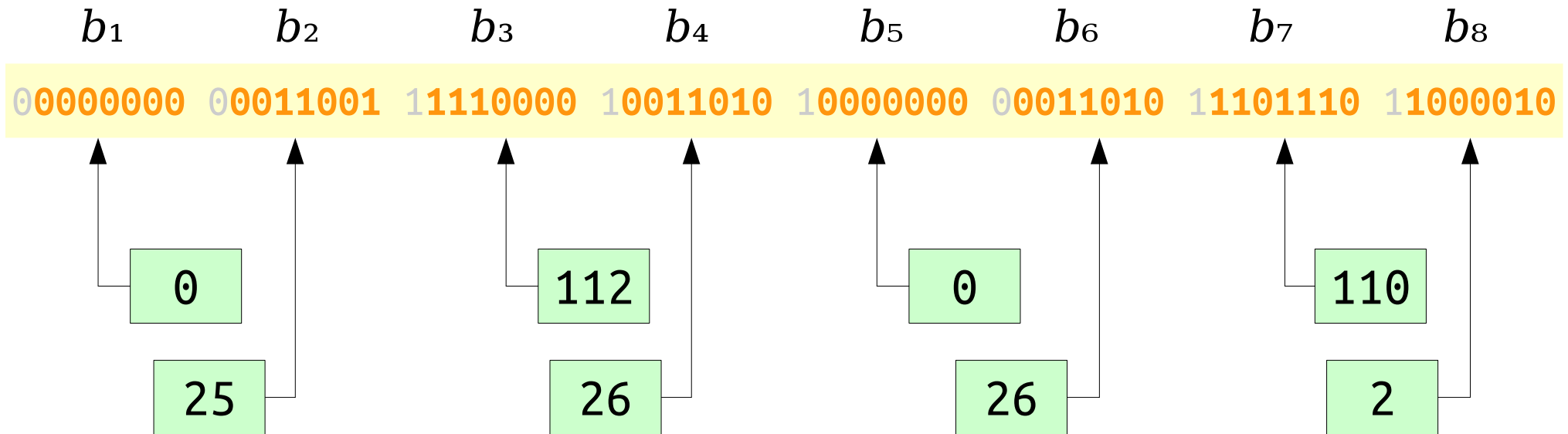
00000000 00011001 11110000 10011010 10000000 00011010 11101110 11000010

00000000 00000000 10000000 10000000 10000000 00000000 10000000 10000000

High bit set?

Observation: A block contains a 1 bit if its first bit is 1 or its lower 7 bits contain a 1.

Identifying Active Blocks



00000000 00000000 10000000 10000000 10000000 00000000 10000000 10000000

A number's lower 7 bits contain a 1 if and only if the numeric value of those bits is at least 1.

Observation: A block contains a 1 bit if its first bit is 1 or its lower 7 bits contain a 1.

Identifying Active Blocks

b_1 b_2 b_3 b_4 b_5 b_6 b_7 b_8

00000000 00011001 11110000 10011010 10000000 00011010 11101110 11000010

00000001 00000001 00000001 00000001 00000001 00000001 00000001 00000001

00000000 00000000 10000000 10000000 10000000 00000000 10000000 10000000

High bit set?

Observation: A block contains a 1 bit if its first bit is 1 or its lower 7 bits contain a 1.

Identifying Active Blocks

b_1 b_2 b_3 b_4 b_5 b_6 b_7 b_8

10000000 10011001 11110000 10011010 10000000 10011010 11101110 11000010

00000001 00000001 00000001 00000001 00000001 00000001 00000001 00000001

01111111 10011000 11101111 10011001 01111111 10011001 11101101 11000001

00000000 00000000 10000000 10000000 10000000 00000000 10000000 10000000

High bit set?

Observation: A block contains a 1 bit if its first bit is 1 or its lower 7 bits contain a 1.

Identifying Active Blocks

b_1 b_2 b_3 b_4 b_5 b_6 b_7 b_8

10000000 10011001 11110000 10011010 10000000 10011010 11101110 11000010

00000001 00000001 00000001 00000001 00000001 00000001 00000001 00000001

01111111 10011000 11101111 10011001 01111111 10011001 11101101 11000001

00000000 00000000 10000000 10000000 10000000 00000000 10000000 10000000

High bit set?

Observation: A block contains a 1 bit if its first bit is 1 or its lower 7 bits contain a 1.

Identifying Active Blocks

b_1 b_2 b_3 b_4 b_5 b_6 b_7 b_8

10000000 10011001 11110000 10011010 10000000 10011010 11101110 11000010

00000001 00000001 00000001 00000001 00000001 00000001 00000001 00000001

00000000 10000000 10000000 10000000 00000000 10000000 10000000 10000000

Low bits set?

00000000 00000000 10000000 10000000 10000000 00000000 10000000 10000000

High bit set?

Observation: A block contains a 1 bit if its first bit is 1 or its lower 7 bits contain a 1.

Identifying Active Blocks

b_1 b_2 b_3 b_4 b_5 b_6 b_7 b_8

00000000 00011001 11110000 10011010 10000000 00011010 11101110 11000010

00000000 10000000 10000000 10000000 00000000 10000000 10000000 10000000

Low bits set?

00000000 00000000 10000000 10000000 10000000 00000000 10000000 10000000

High bit set?

Observation: A block contains a 1 bit if its first bit is 1 or its lower 7 bits contain a 1.

Identifying Active Blocks

b_1 b_2 b_3 b_4 b_5 b_6 b_7 b_8

00000000 00011001 11110000 10011010 10000000 00011010 11101110 11000010

00000000 10000000 10000000 10000000 10000000 10000000 10000000 10000000

Any bits set?

00000000 10000000 10000000 10000000 00000000 10000000 10000000 10000000

Low bits set?

00000000 00000000 10000000 10000000 10000000 00000000 10000000 10000000

High bit set?

Observation: A block contains a 1 bit if its first bit is 1 or its lower 7 bits contain a 1.

Identifying Active Blocks

- **Idea:** Adapt the shifting technique we used to compute ranks.
- Instead of shifting the bits on top of one another, shift the bits next to one another:

a00000000b00000000c00000000d00000000

+

a

b

c

d

Fundamental Primitive: *Parallel Pack*

1. Perform a *parallel tile* with an appropriate multiplier to place all leading bits adjacent to one another.
2. Use a bitmask and bitshift to isolate those bits.

Assuming the multiplier for part (1) and the mask and shift for part (2) can be computed in time $O(1)$, this takes time $O(1)$.

Putting It All Together

- Use a bitmask to identify all blocks whose high bit is set.
- Use a *parallel tile* and a *parallel compare* to identify all blocks with a 1 bit aside from the first.
- Use a *parallel pack* to pack those bits together.
- Use a *parallel rank* to determine the highest of those bits set, which gives the block index.
- Use a *parallel rank* to determine the highest bit set within that block.

The Finished Product

- I've posted a link to a working implementation of this algorithm for 64-bit integers on the course website.
- Feel free to check it out - it's really magical seeing all the techniques come together!

Next Time

- ***Patricia Codes***
 - Compressing a small number of big integers into a small number of small integers.
- ***Fusion Trees***
 - Combining all these techniques together!