

Fast Scans

Niv Dayan - CSC2525: Research Topics in Database Management

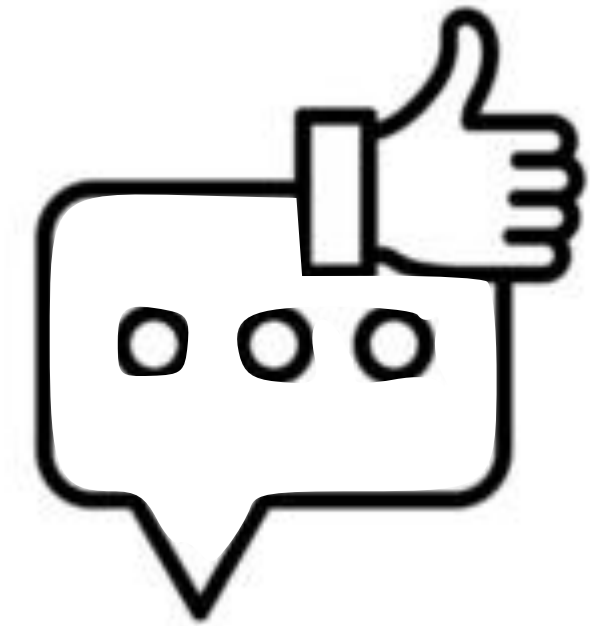
Logistics



Projects due
on April 9



Oral Exams
April 8-10



Course
Evaluation



BitWeaving
SIGMOD 2013



Column-Sketches
SIGMOD 2018

Assume column-store
(Each column in table is separate)

Birth Year

1970
1981
2000
1976
...

Name

...
...
...
...
...

Address

...
...
...
...
...

Birth Year (**BY**)

1970

1981

2000

1976

...

Example Query

Select * where **BY** < **X** and **BY** > **Y**

Birth Year

1970

1981

2000

1976

...



How to encode?

Birth Year

4 byte integer

1970

000000000000000000 0000011110110010

1981

000000000000000000 0000011110111101

2000

000000000000000000 0000011111010000

1976

000000000000000000 0000011110111000

...

Birth Year

4 byte integer

1970

000000000000000000 0000011110110010

1981

000000000000000000 0000011110111101

2000

000000000000000000 0000011111010000

1976

000000000000000000 0000011110111000

...

Problem?

Birth Year

4 byte integer

1970

00000000000000000000 0000011110110010

1981

00000000000000000000 0000011110111101

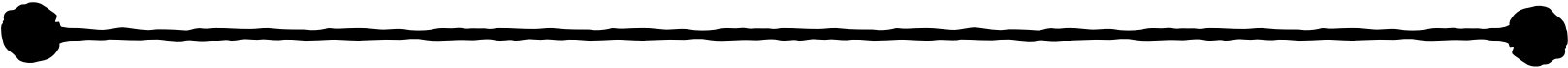
2000

00000000000000000000 0000011111010000

1976

00000000000000000000 0000011110111000

...



Space wastage

Birth Year

4 byte integer

1970

00000000000000000000 0000011110110010

1981

00000000000000000000 0000011110111101

2000

00000000000000000000 0000011111010000

1976

00000000000000000000 0000011110111000

...



Space wastage

Worse scan performance!

Birth Year

2 byte integer

1970

0000011110110010

1981

0000011110111101

2000

0000011110100000

1976

0000011110111000

...

Birth Year

1970

1981

2000

1976

...

2 byte integer

0000011110110010

0000011110111101

0000011110100000

0000011110111000

2x faster :)

Birth Year

1970

1981

2000

1976

...

2 byte integer

0000011110110010

0000011110111101

0000011110100000

0000011110111000

Problem?

Birth Year

1970

1981

2000

1976

...

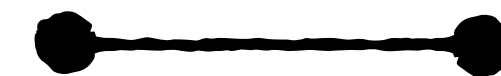
2 byte integer

0000011110110010

0000011110111101

0000011110100000

0000011110111000



Leading zeros

Birth Year

1970

1981

2000

1976

...

2 byte integer

0000011110110010

0000011110111101

0000011110100000

0000011110111000



Narrow range
e.g., 1900 - 2025

Birth Year

1970

1981

2000

1976

...

2 byte integer

0000011110110010

0000011110111101

0000011110100000

0000011110111000



Narrow range

e.g., 1900 - 2025

Most values aren't used

Birth Year

1970

1981

2000

1976

...

2 byte integer

00000**11110110010**

00000**11110111101**

00000**11111010000**

00000**11110111000**

Solutions? :)

Solutions from CSC443

0
1
1
0

**Bit-vector
encoding**



**Run-Length
Encoding**



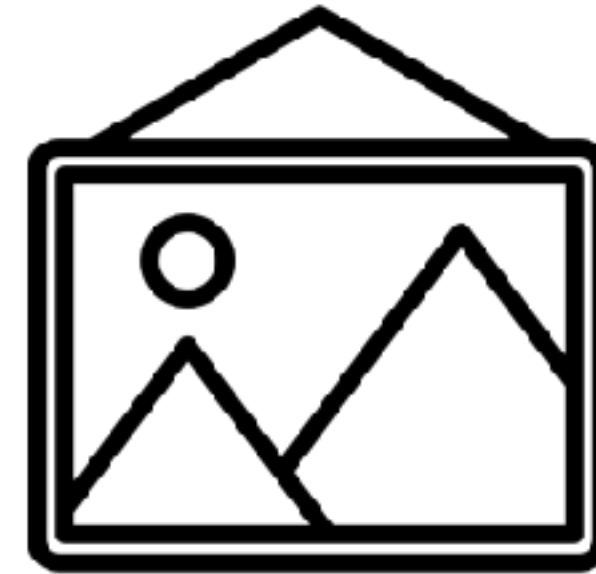
**Dictionary
Encoding**

0
1
1
1
0

**Bit-vector
encoding**



**Run-Length
Encoding**

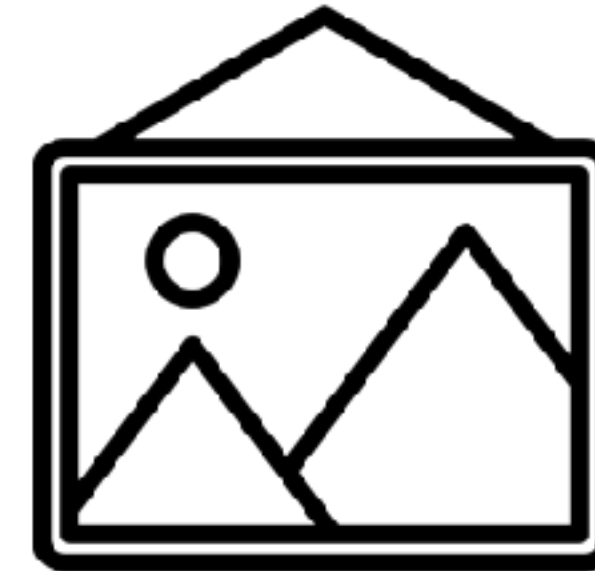


**Frame of
Reference
Encoding**



**Dictionary
Encoding**

0
1
1
0



**Bit-vector
encoding**

**Run-Length
Encoding**

Frame of
Reference
Encoding

Dictionary
Encoding

**Less Relevant
for today**

Quick Review



**Frame of
Reference
Encoding**



**Dictionary
Encoding**

Frame of Reference Encoding

Birth Year

1970

1981

2000

1976

...

Frame of Reference Encoding

Birth Year

1970

1981

2000

1976

...



Find min
e.g., 1970

Frame of Reference Encoding

Birth Year		Store differences
1970	- 1970 =	0
1981	- 1970 =	11
2000	- 1970 =	30
1976	- 1970 =	6
...		

Frame of Reference Encoding

Birth Year

Store differences

1970

00000000

1981

00001011

2000

00011110

1976

00000110

...

Down to 1 byte :)

Frame of Reference Encoding

Birth Year

Store differences

1970

00000000

1981

00001011

2000

00011110

1976

00000110

...



Still wasting some space & speed

Quick Review



Frame of
Reference
Encoding



**Dictionary
Encoding**

Dictionary Encoding

Birth Year

1970

1981

2000

1970

1970

2000

Birth Year

1970

1981

2000

1970

1970

2000

Dictionary

1970	0
1981	1
2000	2

Birth Year

Dictionary

Compressed column

1970

1970 0

0

1981

1981 1

1

2000

2000 2

2

1970

1

1970

1

2000

2

Birth Year	Order-Preserving Dictionary	Compressed column
1970	1970 0	0
1981	1981 1	1
2000	2000 2	2
1970		1
1970		1
2000		2

Birth Year	Order-Preserving Dictionary		Compressed column
1970	Sorted	1970 0	Sorted 0
1981		1981 1	1
2000		2000 2	2
1970			1
1970			1
2000			2

Birth Year	Order-Preserving Dictionary		Compressed column
1970	Sorted ↓	1970 0	0
1981		1981 1	1
2000		2000 2	2
1970			1
1970			1
2000			2

Select * where year > 1981

Birth Year	Order-Preserving Dictionary		Compressed column
1970	1970	0	0
1981	1981	1	1
2000	2000	2	2
1970			1
1970			1
2000			2

Replace with code

Select * where year > **1**

Birth Year	Order-Preserving Dictionary		Compressed column
1970	1970	0	0000 0000
1981	1981	1	0000 0001
2000	2000	2	0000 0010
1970			0000 0001
1970			0000 0001
2000			0000 0010
			↑
			Encode as 1 byte...

Quick Review



**Frame of
Reference
Encoding**



**Dictionary
Encoding**

Problems

**Checking
Whole Codes**



**Wasted
Space**



Checking Whole Codes

Dec

1

10

7

11

Checking Whole Codes

Dec	Binary
------------	---------------

1	0001
---	------

10	1010
----	------

7	0111
---	------

11	1011
----	------

Dec	Binary
-----	--------

1	0001
---	------

10	1010
----	------

7	0111
---	------

11	1011
----	------

Select * where $c \geq 12$ (1100)

		Dec	Binary
Scan each code	→	1	0001
		10	1010
		7	0111
		11	1011

Select * where c $\geq$ 12 (1100)

Problem?

Dec	Binary
-----	--------

1	0001
---	------

10	1010
----	------

7	0111
---	------

Scan each code →

11	1011
----	------

Select * where $c \geq 12$ (1100)

Problem?

We could have known after only 2 bits per code that none qualify

Dec	Binary
1	0001
10	1010
7	0111
11	1011

Select * where $c \geq 12$ **(1100)**

Problems

Checking
Whole Codes



**Wasted
Space**



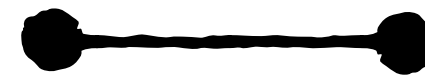
Wasted Space

00000 000

00000 101

00000 011

00000 110



**Leading zeros waste space force
traversing more data**

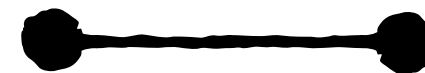
Wasted Space

00000 000

00000 101

00000 011

00000 110



Leading zeros waste space force
traversing more data

Solutions? :)

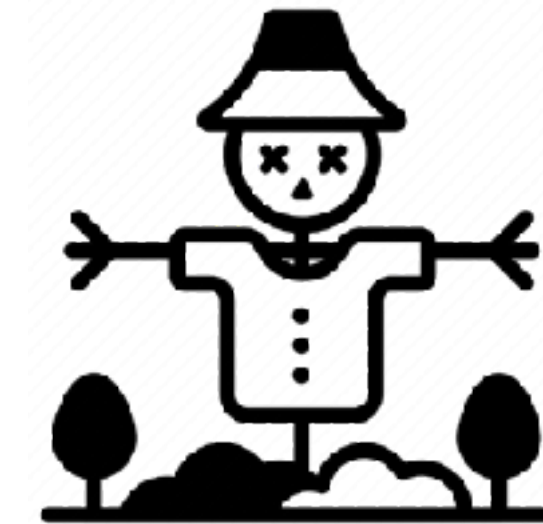
Three Straw-Man Solutions



**Dense
Packing**



Align & Pad



Pack, Align & Pad

Dense Packing

Codes:

000

101

011

110

111

010

111

Dense Packing

Codes: 000 101 011 110 111 010 111

Words: 
 8 bits 8 bits 8 bits

Assume 8 bit words - in reality 64

Dense Packing

Words:

000 101 011 110 111 010 111 ...



The diagram illustrates dense packing by showing a sequence of 3-bit words: 000, 101, 011, 110, 111, 010, 111, followed by an ellipsis. These words are packed into a single stream, represented by a horizontal line with vertical tick marks at the boundaries of each word. The words are packed such that the 1s of one word are aligned with the 0s of the next word, minimizing the number of 0s used as separators.

Dense Packing

Words: 000 101 011 110 111 010 111 ...



Problems?

Dense Packing

Words: 000 101 011 110 111 010 111 ...



The diagram illustrates dense packing by showing a sequence of 7-bit words: 000, 101, 011, 110, 111, 010, and 111, followed by an ellipsis. These words are packed into a single word boundary, which is represented by a horizontal line with vertical tick marks at the boundaries of the words. The words are packed such that the first word (000) starts at the first boundary, and the last word (111) ends at the fourth boundary, with the ellipsis indicating that the sequence continues.

Problems? **The processor reads at word granularity**

Dense Packing

Words: 000 101 011 110 111 010 111 ...



Problems? The processor reads at word granularity

Hard to run CPU instructions ($=$, $>$, $<$) on codes due to their irregular offsets within words and no alignment

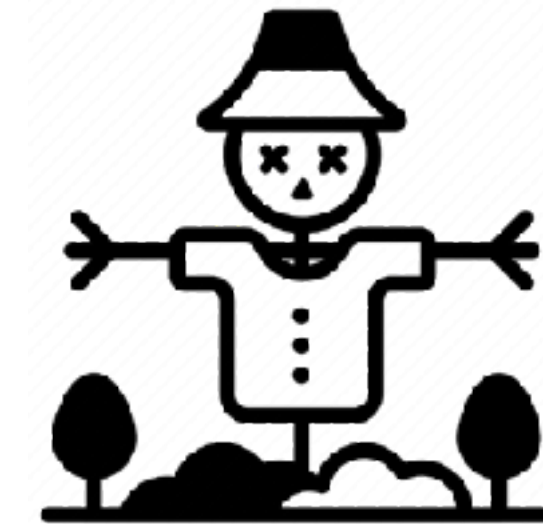
Three Straw-Man Solutions



**Dense
Packing**



Align & Pad



Pack, Align & Pad

Align & Pad (every 8 bits)

000000 000 000000 101 000000 011 000000 110 000000 111 000000 010 000000 111



Align & Pad (every 16 bits)

e.g., with 11 bit codes



The diagram illustrates the alignment and padding of three 11-bit codes. A horizontal line with four vertical tick marks at the start, after the first code, after the second code, and at the end, represents the bit stream. The first code is '10001001000' (11 bits), followed by five zeros '00000' (5 bits) to reach a total of 16 bits. The second code is '1011100000' (10 bits), followed by six zeros '00000' (6 bits) to reach a total of 16 bits. The third code is '11000011011' (11 bits), followed by five zeros '00000' (5 bits) to reach a total of 16 bits. The padding zeros are shown in light gray, while the data bits are in bold black.

00000 **10001001000** 00000 **1011100000** 00000 **11000011011**

Align & Pad (every 32 bits)

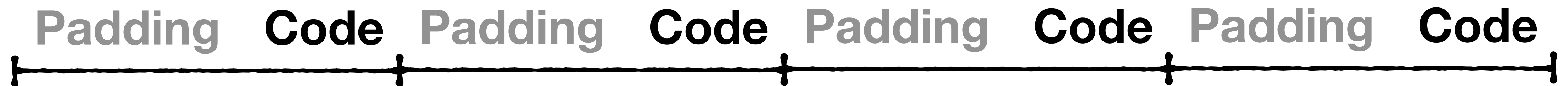
e.g., with 20 bit codes

00000000000000000000 11011010110010011110



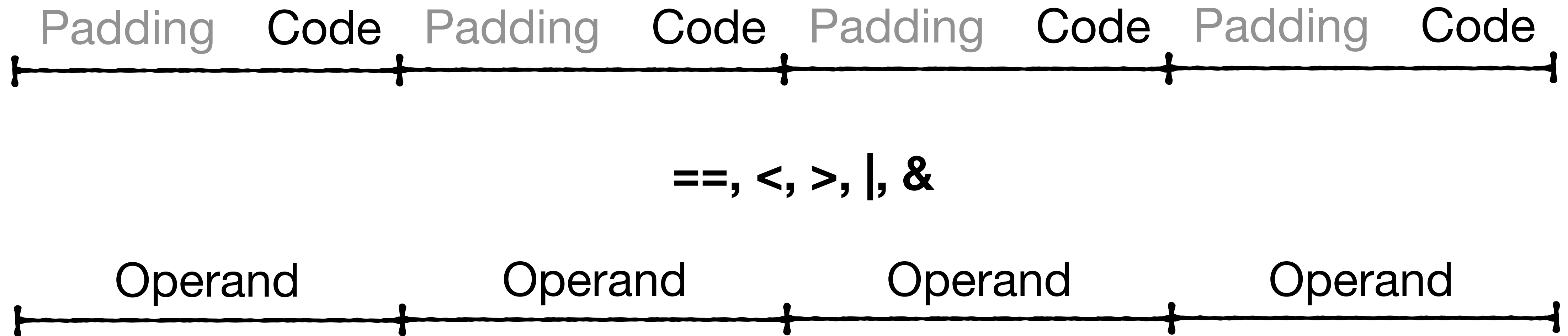
Align & Pad

Allows AVX instructions (SIMD)



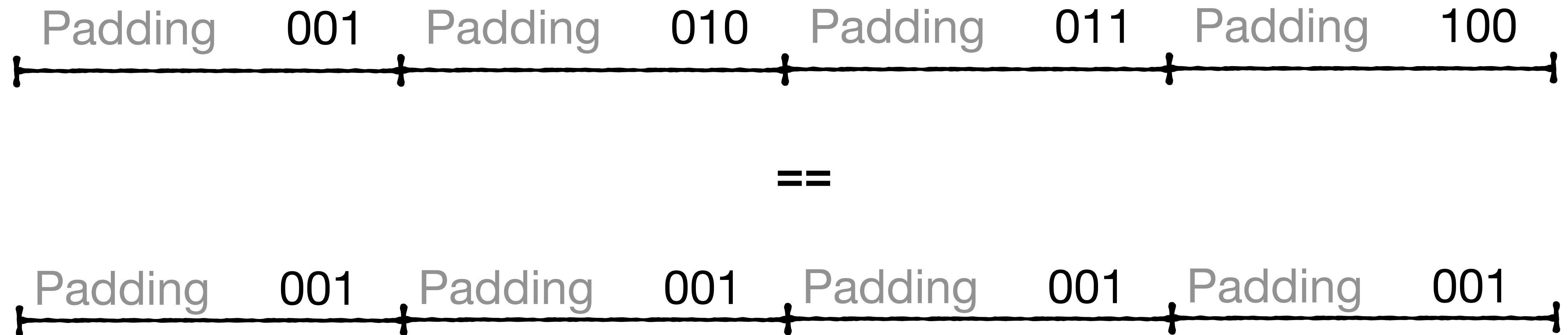
Align & Pad

Allows AVX instructions (SIMD)



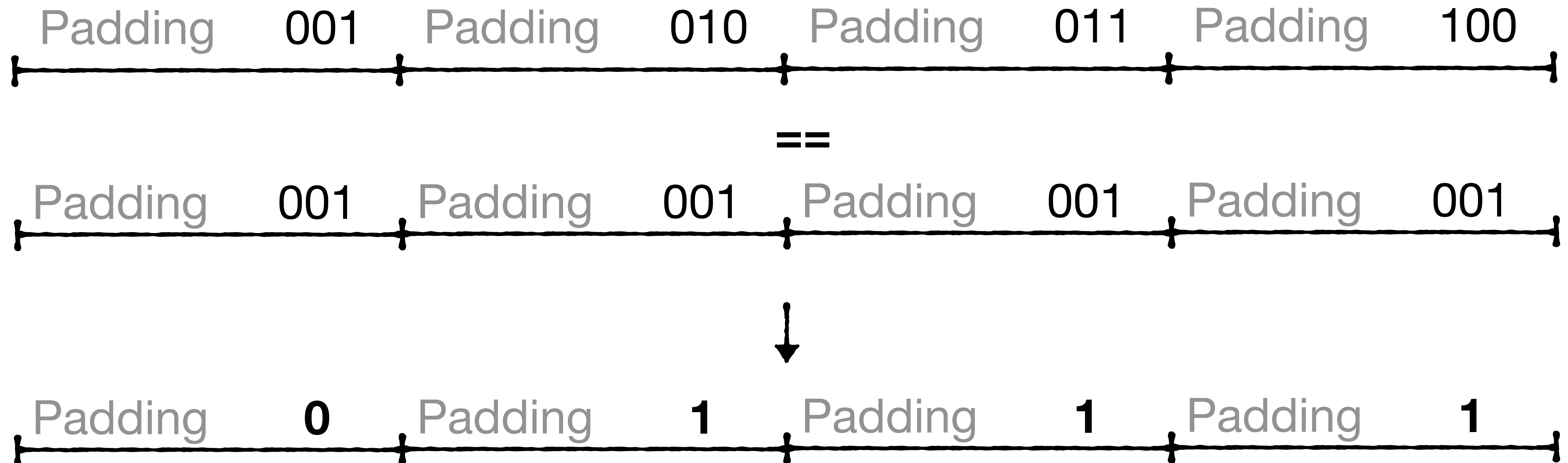
Align & Pad

Allows AVX instructions (SIMD)

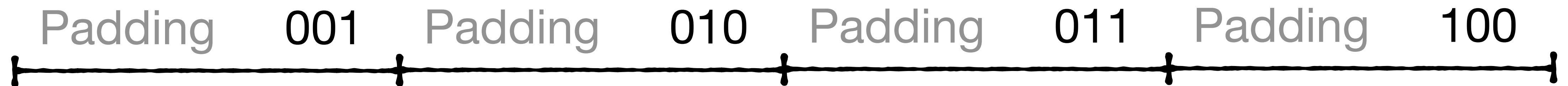


Align & Pad

Allows AVX instructions (SIMD)



Align & Pad



Padding wastes space and slows down scans

Three Straw-Man Solutions



Dense
Packing



Align & Pad



Pack, Align & Pad

Pack, Align & Pad

00000 000 00000 101 00000 011 00000 110 00000 111 00000 010 00000 111



Pack, Align & Pad

Pack only as many codes into each word as would fit

00 000 101 00 011 110 00 111 010 00000 111



wastes some space, but since word size is usually 64 bits, only a bit of space is wasted at the end

00 000 101 00 011 110 00 111 010 00000 111



wastes some space, but since word size is usually 64 bits, only a bit of space is wasted at the end

00 000 101 00 011 110 00 111 010 00000 111



Problems?

wastes some space, but since word size is usually 64 bits, only a bit of space is wasted at the end

00 000 101 00 011 110 00 111 010 00000 111



Problems?

Can't use SIMD

00 000 101 00 011 110 00 111 010 00000 111



Problems?

Can't use SIMD

Looping over each code during query is CPU-intensive

```
for (int i = 0; i < words; i++)  
    for (int j = 0; j < word_size; j += code_size)
```

00 000 101 00 011 110 00 111 010 00000 111



Problems?

Can't use SIMD

Looping over each code during query is CPU-intensive

```
for (int i = 0; i < words; i++)  
    for (int j = 0; j < word_size; j += code_size)  
        code = extract_code(j, j + code_size)
```

00 000 101 00 011 110 00 111 010 00000 111



Problems?

Can't use SIMD

Looping over each code during query is CPU-intensive

```
for (int i = 0; i < words; i++)  
    for (int j = 0; j < word_size; j += code_size)  
        code = extract_code(j, j + code_size)  
        If( selection_predicate(code) )  
            record result
```

00 000 101 00 011 110 00 111 010 00000 111



Problems?

Can't use SIMD

Looping over each code during query is CPU-intensive

```
for (int i = 0; i < words; i++)  
    for (int j = 0; j < word_size; j += code_size)  
        code = extract_code(j, j + code_size)  
        if( selection_predicate(code) )  
            record result
```

00 000 101 00 011 110 00 111 010 00000 111



Can we do better in terms of CPU?

Problems

**Wasted
Space**

**Checking
whole Codes**

BitWeaving: Fast Scans for Main Memory Data Processing

Yinan Li, Jignesh M. Patel

SIGMOD 2013



BitWeaving: Fast Scans for Main Memory Data Processing

Yinan Li, Jignesh M. Patel

SIGMOD 2013



use bitwise operations to achieve parallelism at word level

BitWeaving: Fast Scans for Main Memory Data Processing



Horizontal



Vertical

Horizontal BitWeaving

Column codes

c1 001

c2 101



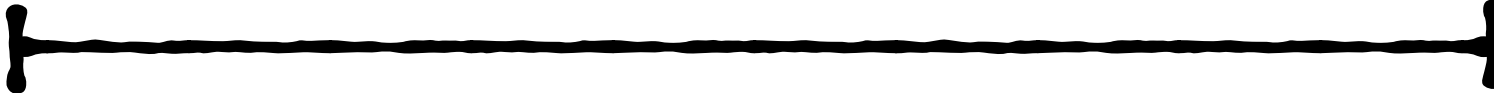
8 bit word

c1

c2

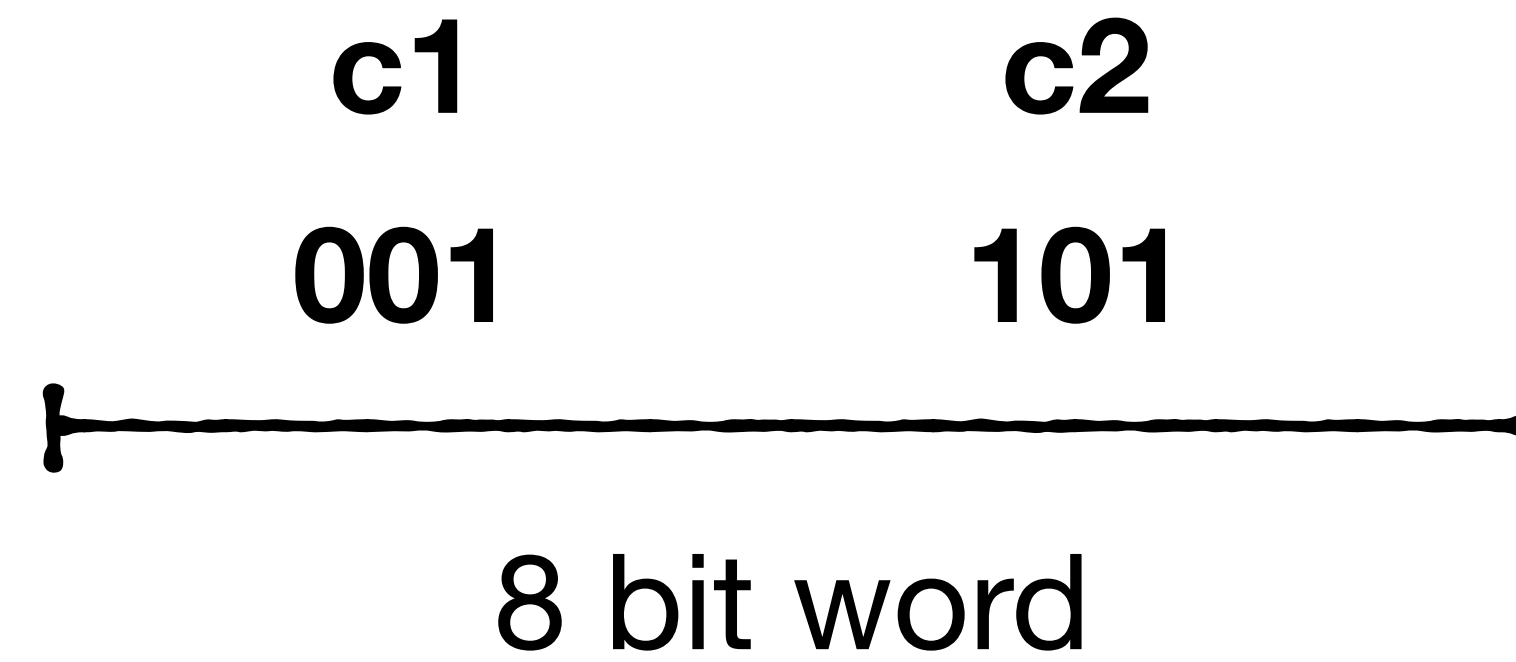
001

101

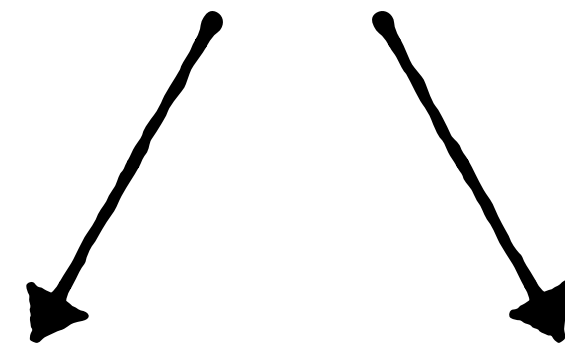


8 bit word

Similar to “Pack, Align & Pad”, except...



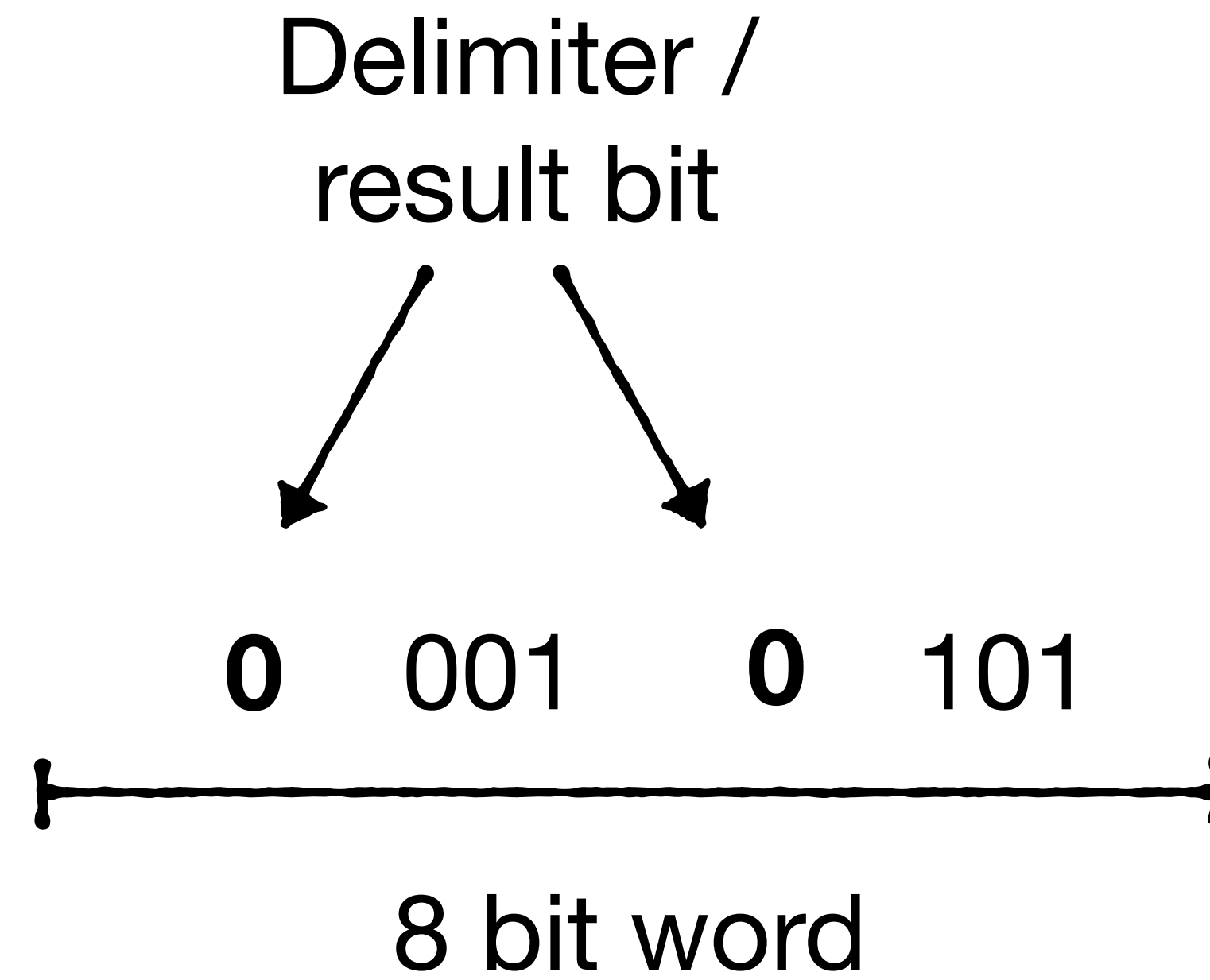
**Delimiter /
result bit**



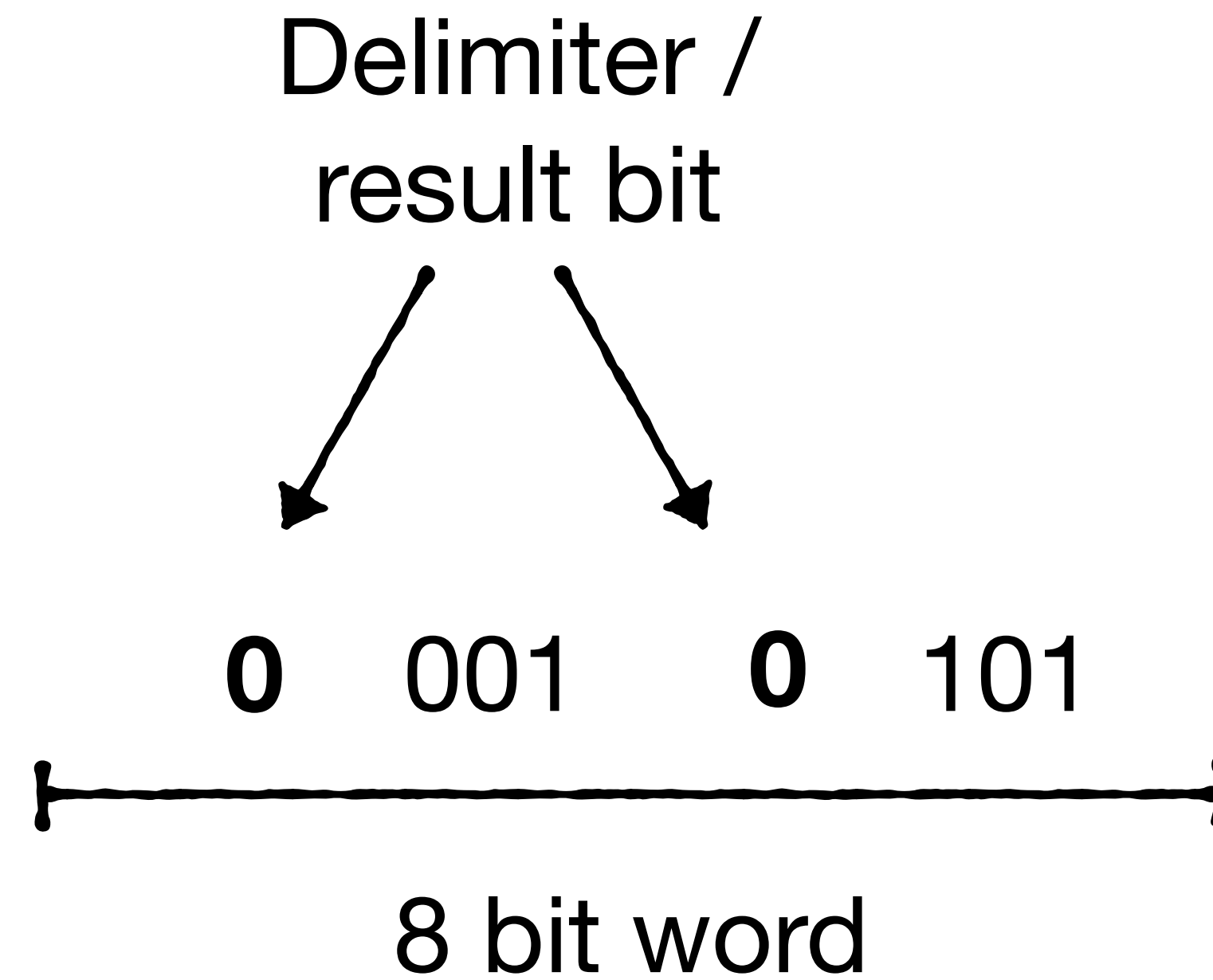
0 001 0 101



8 bit word



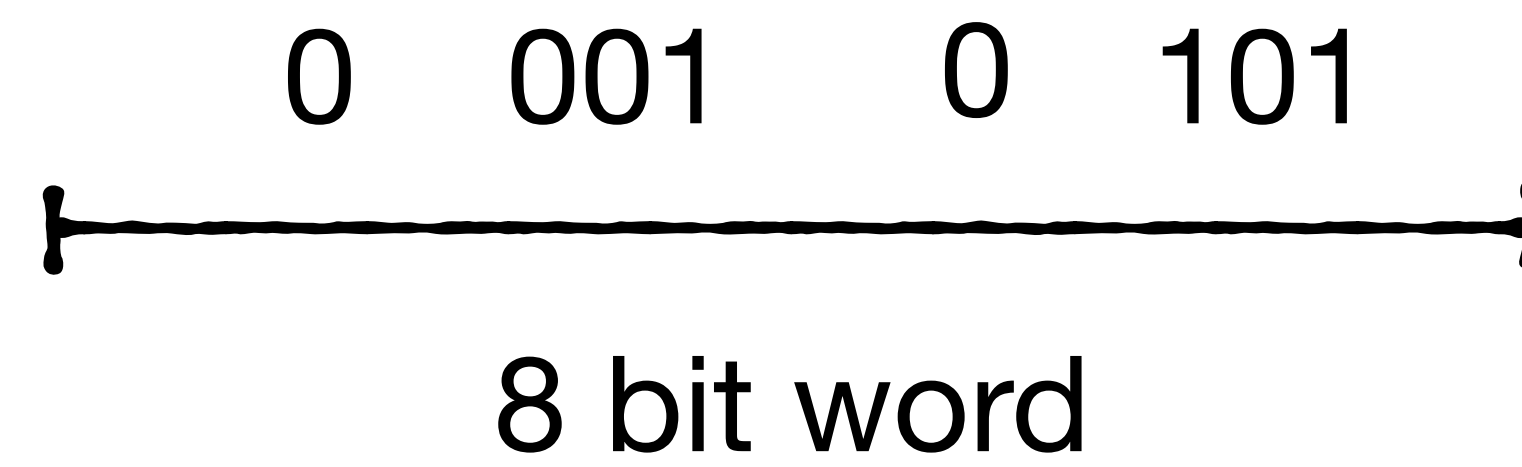
Evaluate predicate (==, <, >, |, &) on all codes in a word in parallel



Evaluate predicate (**==**, **<**, **>**, **|**, **&**) on all codes in a word in parallel

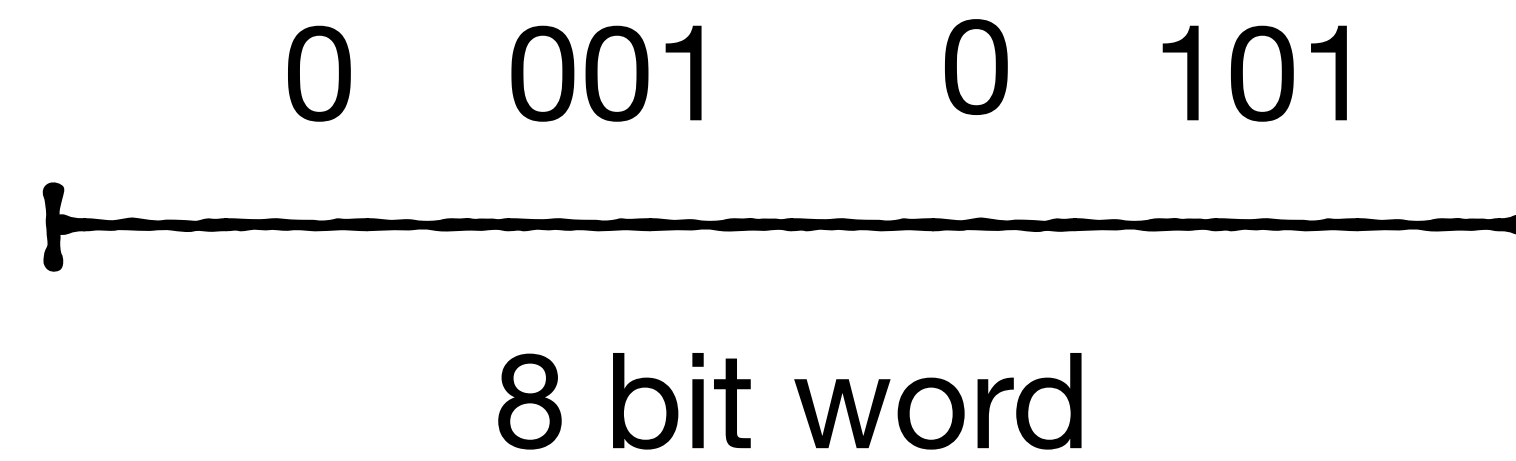
Store result for each code in result bit

Let's consider ≠

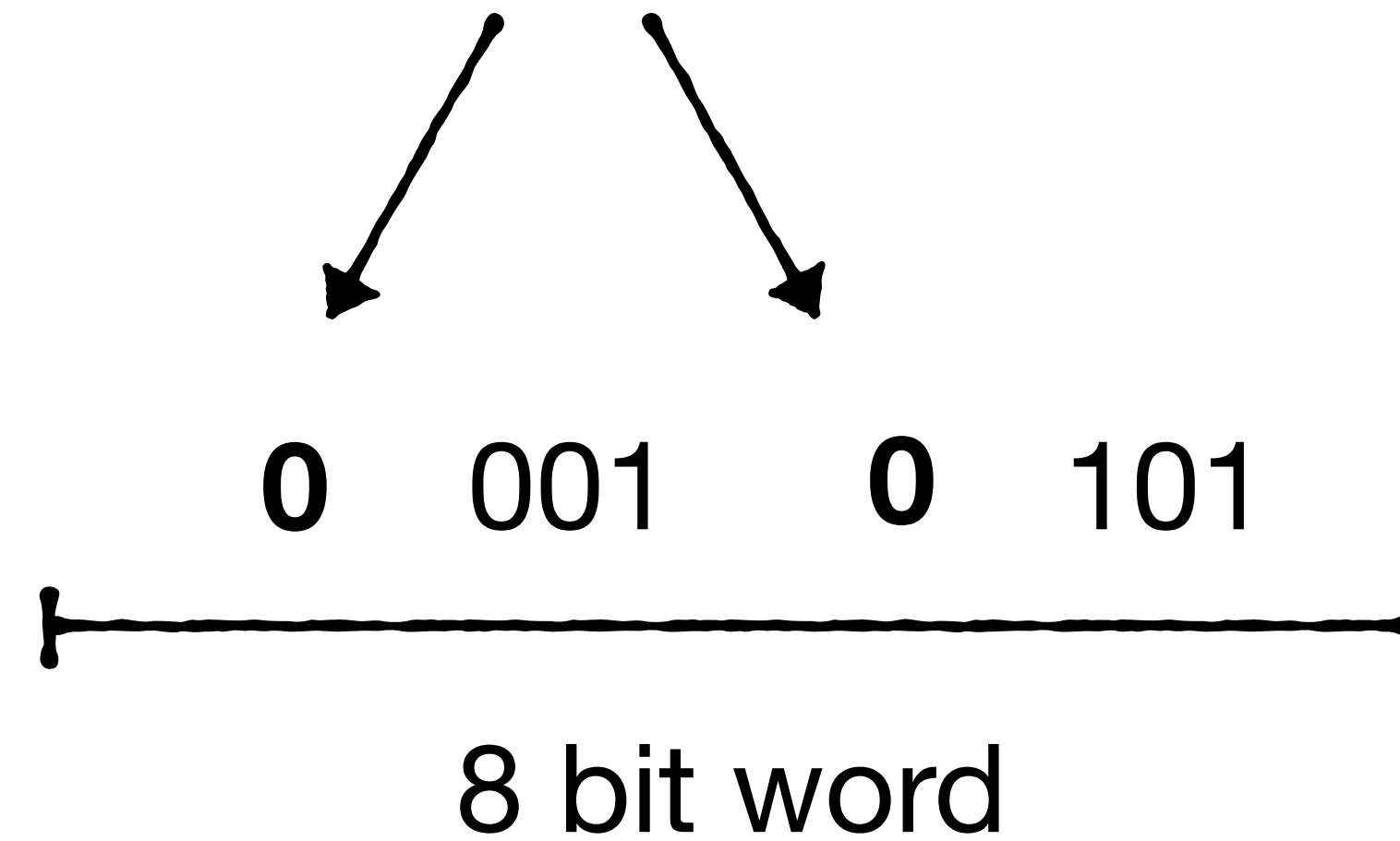


Let's consider $\neq$

e.g., Select * where $X \neq \text{constant}$



**Run bitwise operations on word such that
delimiter for each code will be 1 if unequal**



$$\mathbf{X \neq Y \quad iff \quad X \oplus Y \neq 0}$$

$$X \neq Y \quad \text{iff} \quad X \oplus Y \neq 0$$

0 0 1 1

$\oplus$

0 0 1 1

=

0 0 0 0

$$X \neq Y \quad \text{iff} \quad X \oplus Y \neq 0$$

0 0 1 1

$\oplus$

0 0 1 **0**

=

0 0 0 **1**

$$X \neq Y \quad \text{iff} \quad X \oplus Y \neq 0$$

$$\text{iff} \quad (\mathbf{X} \oplus \mathbf{Y}) + \mathbf{01}^K = \mathbf{1}^K$$

$$0 \ 0 \ 1 \ 1$$

$$\oplus$$

$$0 \ 0 \ 1 \ 0$$

$$=$$

$$0 \ 0 \ 0 \ 1$$

$$X \neq Y \quad \text{iff} \quad X \oplus Y \neq 0$$

$$\text{iff} \quad (X \oplus Y) + 01^K = 1^K$$

0 0 1 1

$\oplus$

0 0 1 0

+

0 1 1 1

=

1 0 0 0



Always 1 if $X \neq Y$

$$X \neq Y \quad \text{iff} \quad X \oplus Y \neq 0$$

$$\text{iff} \quad (X \oplus Y) + 01^K = 1^K$$

0 0 1 1

$\oplus$

0 0 1 1

+

0 1 1 1

=

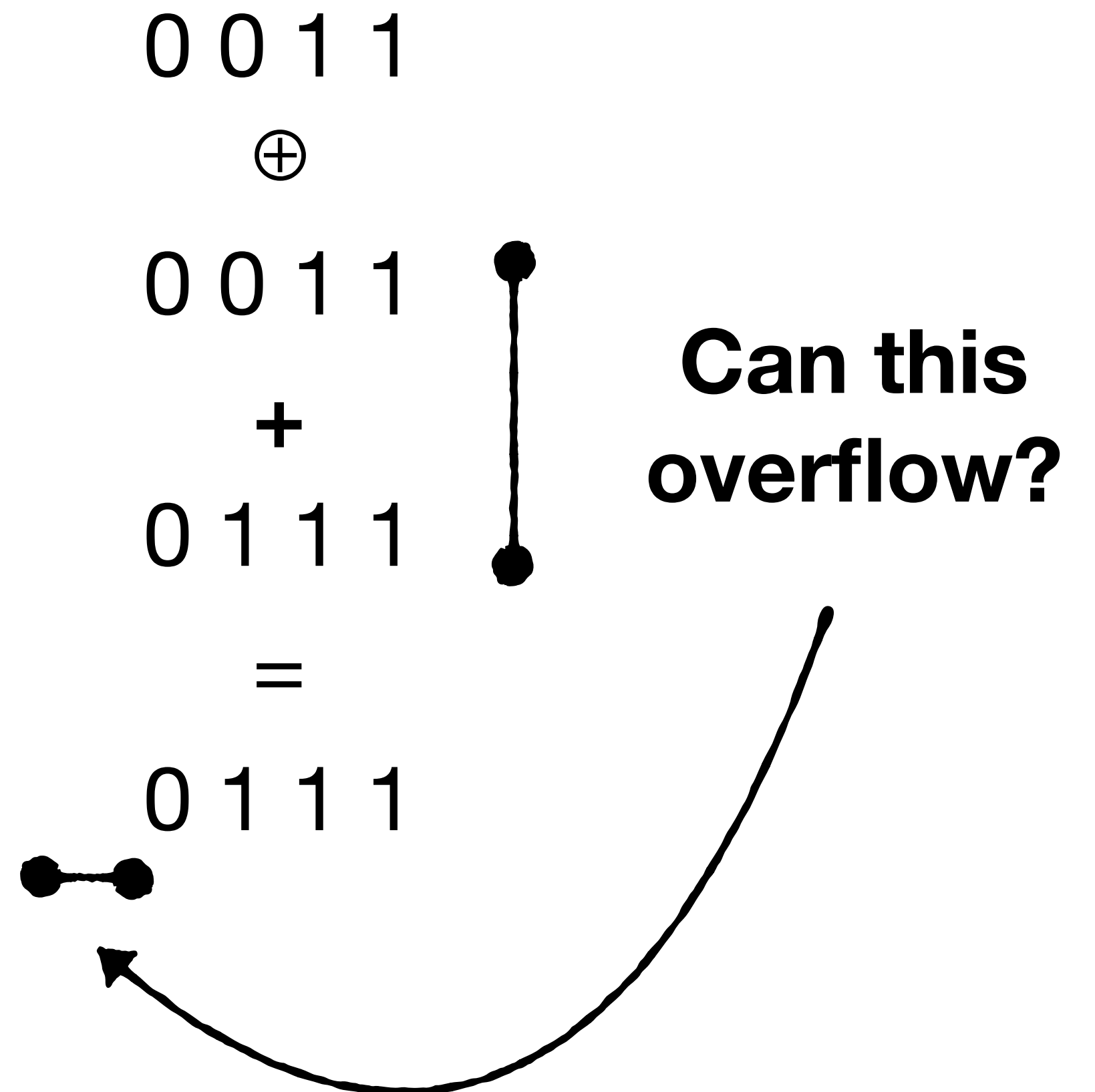
0 1 1 1



Always 0 if $X = Y$

$$X \neq Y \quad \text{iff} \quad X \oplus Y \neq 0$$

$$\text{iff} \quad (X \oplus Y) + 01^K = 1^K$$



**Do we ever require more than
1 extra bit?**

$$X \neq Y \quad \text{iff} \quad X \oplus Y \neq 0$$

$$\text{iff} \quad (X \oplus Y) + 01^K = 1^K$$

0 0 1 1

$\oplus$

0 0 1 1

+

0 1 1 1

=

0 1 1 1

Can this
overflow?

No :)

addition of 2 integers never overflows by more than one bit

e.g., 0111 + 0111 = 1110

$$X \neq Y \quad \text{iff} \quad X \oplus Y \neq 0$$

$$\text{iff} \quad (X \oplus Y) + 01^K = 1^K$$

0 0 1 1

$\oplus$

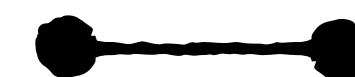
0 0 1 1

+

0 1 1 1

=

0 1 1 1



Always 1^K iff $X = Y$

$$X \neq Y \quad \text{iff} \quad X \oplus Y \neq 0$$

$$\text{iff} \quad (X \oplus Y) + 01^K = 1^K$$

0 0 1 1

$\oplus$

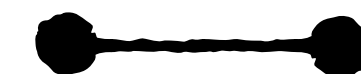
0 1 0 1

+

0 1 1 1

=

1 1 0 1



Can be anything if $X \neq Y$

$$X \neq Y \quad \text{iff} \quad X \oplus Y \neq 0$$

$$\text{iff} \quad (X \oplus Y) + 01^K = 1^K$$

$$\text{iff} \quad ((X \oplus Y) + 01^K) \wedge \mathbf{10^K} = \mathbf{10^K}$$

0 0 1 1

$\oplus$

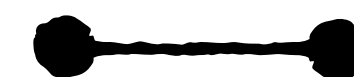
0 1 0 1

+

0 1 1 1

=

1 1 0 1



Reset

$$X \neq Y \quad \text{iff} \quad X \oplus Y \neq 0$$

$$\text{iff} \quad (X \oplus Y) + 01^K = 1^K$$

$$\text{iff} \quad ((X \oplus Y) + 01^K) \wedge 10^K = 10^K$$

0 0 1 1

$\oplus$

0 1 0 1

+

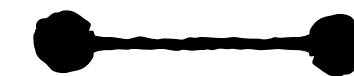
0 1 1 1

$\wedge$

1 0 0 0

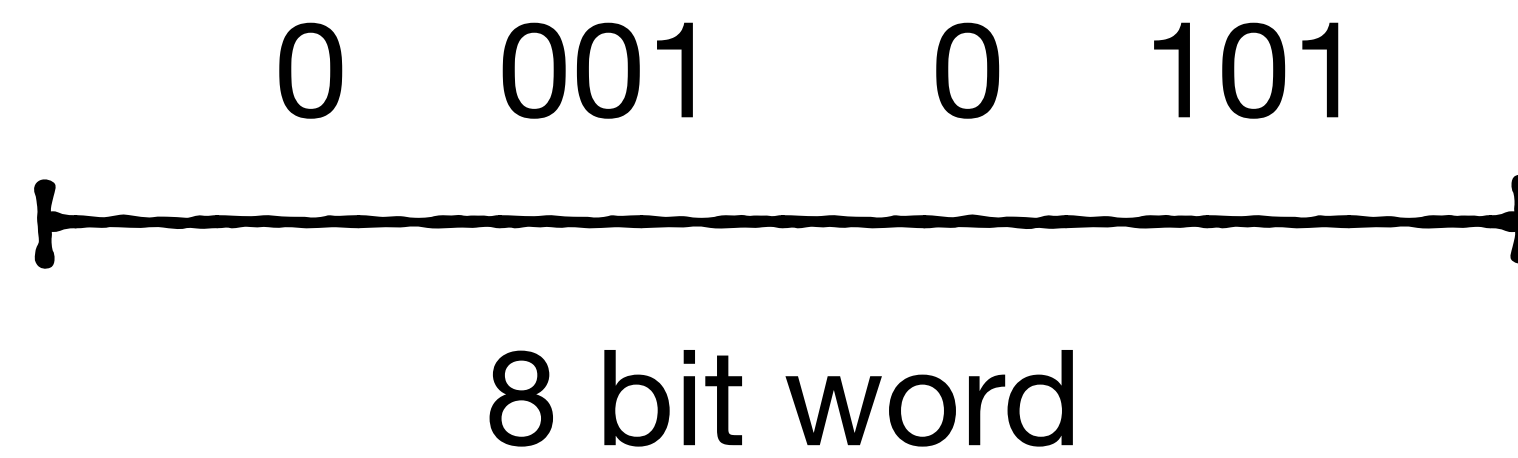
=

1 0 0 0



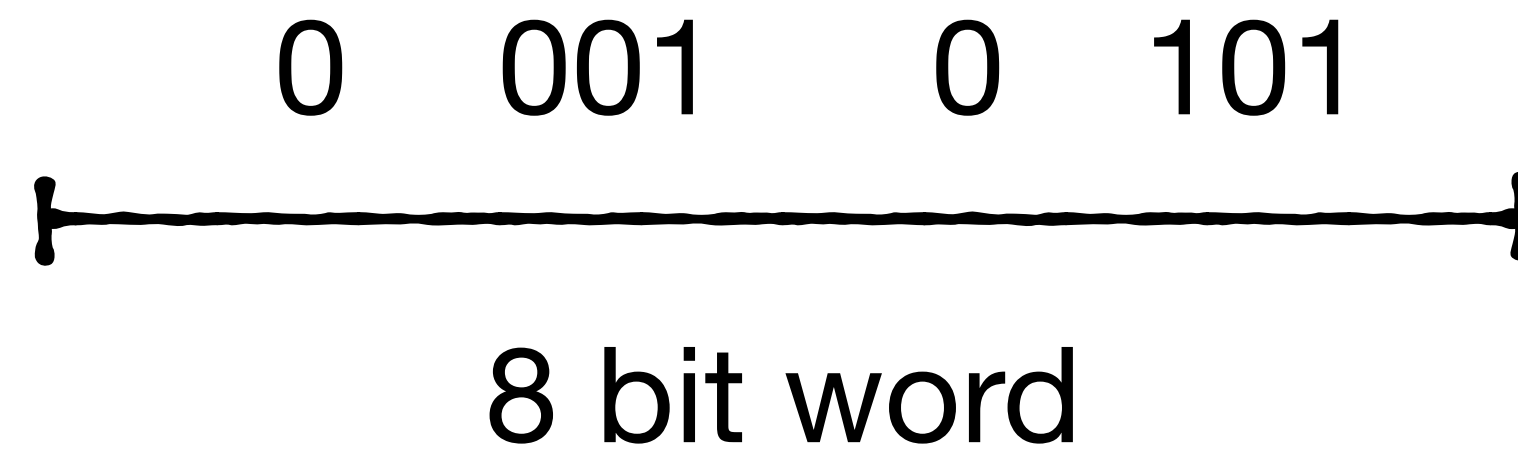
Reset

Now let's apply this in parallel on all codes in a word



Now let's apply this in parallel on all codes in a word

Select * where $X \neq 5$



Now let's apply this in parallel on all codes in a word

Select * where $X \neq 5$

word	0	001	0	101
		$\oplus$		$\oplus$
Constant (5)	0	101	0	101

Now let's apply this in parallel on all codes in a word

Select * where $X \neq 5$

word	0	001	0	101
		$\oplus$		$\oplus$
Constant (5)	0	101	0	101
		+		+
Mask	0	111	0	111

Now let's apply this in parallel on all codes in a word

Select * where $X \neq 5$

word	0	001	0	101
		$\oplus$		$\oplus$
Constant (5)	0	101	0	101
		+		+
Mask	0	111	0	111
		$\wedge$		$\wedge$
-Mask	1	000	1	000

Now let's apply this in parallel on all codes in a word

Select * where $X \neq 5$

word	0	001	0	101
		$\oplus$		$\oplus$
Constant (5)	0	101	0	101
		+		+
Mask	0	111	0	111
		$\wedge$		$\wedge$
-Mask	1	000	1	000
		=		=
Result	1	000	0	000

Can apply on multiple words in a cache line using SIMD

	word 1				word 2			
word	0	001	0	101	0	101	0	000
		$\oplus$		$\oplus$		$\oplus$		$\oplus$
Constant (5)	0	101	0	101	0	101	0	101
		+		+		+		+
Mask	0	111	0	111	0	111	0	111
		$\wedge$		$\wedge$		$\wedge$		$\wedge$
-Mask	1	000	1	000	1	000	1	000
		=		=		=		=
Result	1	000	0	000	0	000	1	000

**Word-Level
Parallelism**

**Vectorization
(SIMD)**

Word-Level Parallelism

**Within single machine word
using general purpose registers
& instructions**

Vectorization (SIMD)

**Across words in a cache line
using specialized instructions &
registers**

Word-Level Parallelism

Within single machine word
using general purpose registers
& instructions

Vectorization (SIMD)

Across words in a cache line
using specialized instructions &
registers

Compatible using careful design :)

How about range predicates?

Select * where $X < Y$

How about range predicates?

Select * where $X < Y$

$$X = Y \text{ iff } (X \oplus 01^K) + Y = 01^K$$

How about range predicates?

Select * where $X < Y$

$$X = Y \text{ iff } (X \oplus 01^K) + Y = 01^K$$

X	0 1 1 0
	$\oplus$
01^K	0 1 1 1
	+
Y	0 1 1 0

How about range predicates?

Select * where $X < Y$

$$X = Y \text{ iff } (X \oplus 01^K) + Y = 01^K$$

X	0	1	1	0
		$\oplus$		
01^K	0	1	1	1
		$+$		
Y	0	1	1	0

$\bullet$
|
 $\bullet$

0 0 0 1

How about range predicates?

Select * where $X < Y$

$$X = Y \text{ iff } (X \oplus 01^K) + Y = 01^K$$

X	0 1 1 0		0 0 0 1
	$\oplus$		
01^K	0 1 1 1		
	+		
Y	0 1 1 0		
	=		
	0 1 1 1		

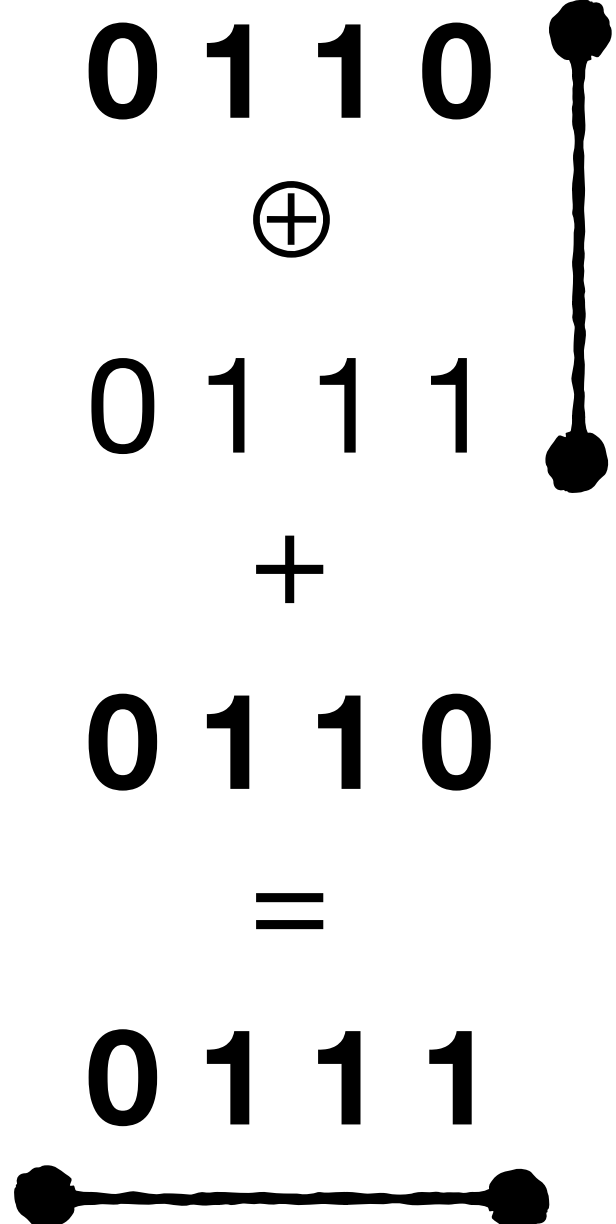
How about range predicates?

Select * where $X < Y$

$$X = Y \text{ iff } (X \oplus 01^K) + Y = 01^K$$

X	0	1	1	0	
		$\oplus$			
01 ^K	0	1	1	1	
		+			
Y	0	1	1	0	
		=			
	0	1	1	1	

0 0 0 1



Always iff $X=Y$

How about range predicates?

Select * where $X < Y$

$$X = Y \text{ iff } (X \oplus 01^K) + Y = 01^K$$

X	0	1	1	0	
		$\oplus$			0 0 0 1
01 ^K	0	1	1	1	
		+			
Y	0	1	1	0	
		=			
	0	1	1	1	

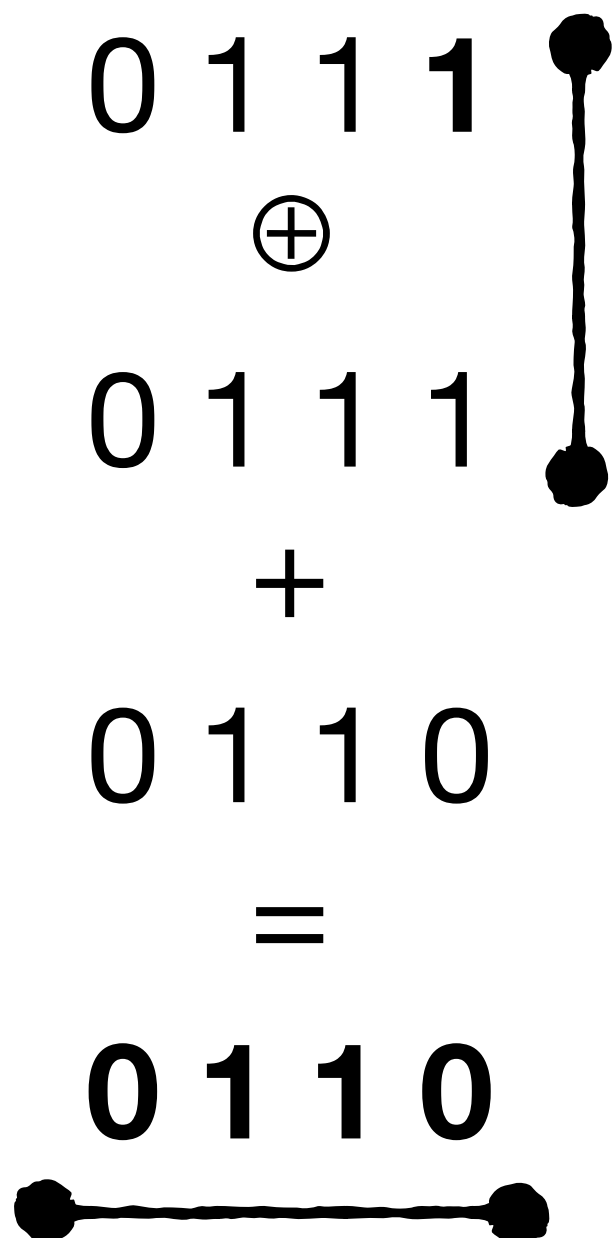
If $X > Y$, result decreases

How about range predicates?

Select * where $X < Y$

$$X = Y \text{ iff } (X \oplus 01^K) + Y = 01^K$$

X	0	1	1	1				
				$\oplus$				
01 ^K	0	1	1	1			0	0
				+				
Y	0	1	1	0				
				=				
	0	1	1	0				



If $X > Y$, result decreases

How about range predicates?

Select * where $X < Y$

$$X = Y \text{ iff } (X \oplus 01^K) + Y = 01^K$$

X	0	1	0	0	
		$\oplus$			
01 ^K	0	1	1	1	
		+			
Y	0	1	1	0	
		=			
	1	0	0	1	

Diagram illustrating the binary addition of $X \oplus 01^K$ and Y to verify the range predicate $X < Y$. The result is 1001 , which is less than 01^K (represented as 0011 in the diagram).

If $X < Y$, result increases

How about range predicates?

Select * where $X < Y$

$$X = Y \text{ iff } (X \oplus 01^K) + Y = 01^K$$

X	0	1	0	0
		$\oplus$		
01 ^K	0	1	1	1
		+		
Y	0	1	1	0
		=		
		1	0	0
		1		

↑

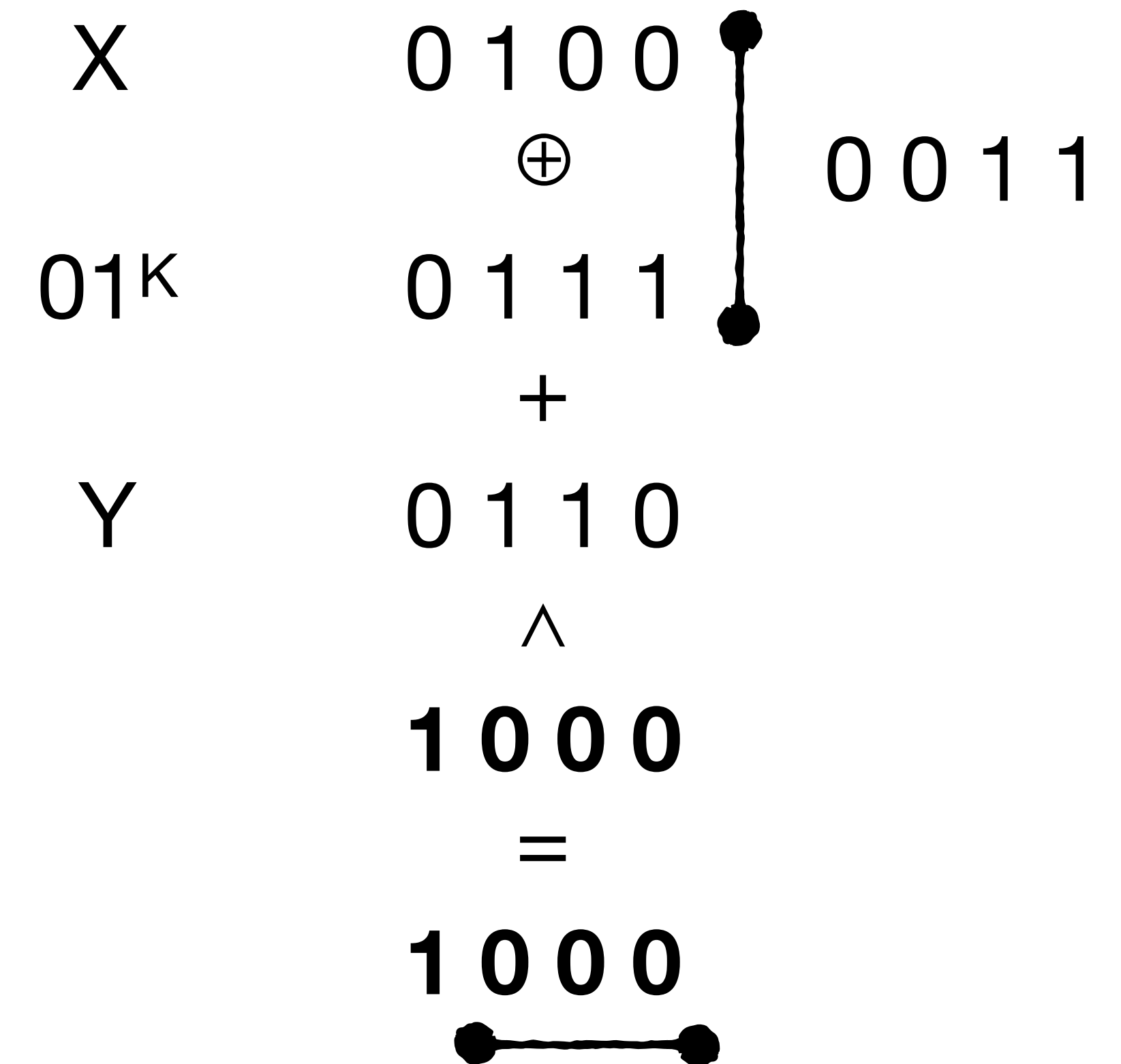
If $X < Y$, this bit is always 1

How about range predicates?

Select * where $X < Y$

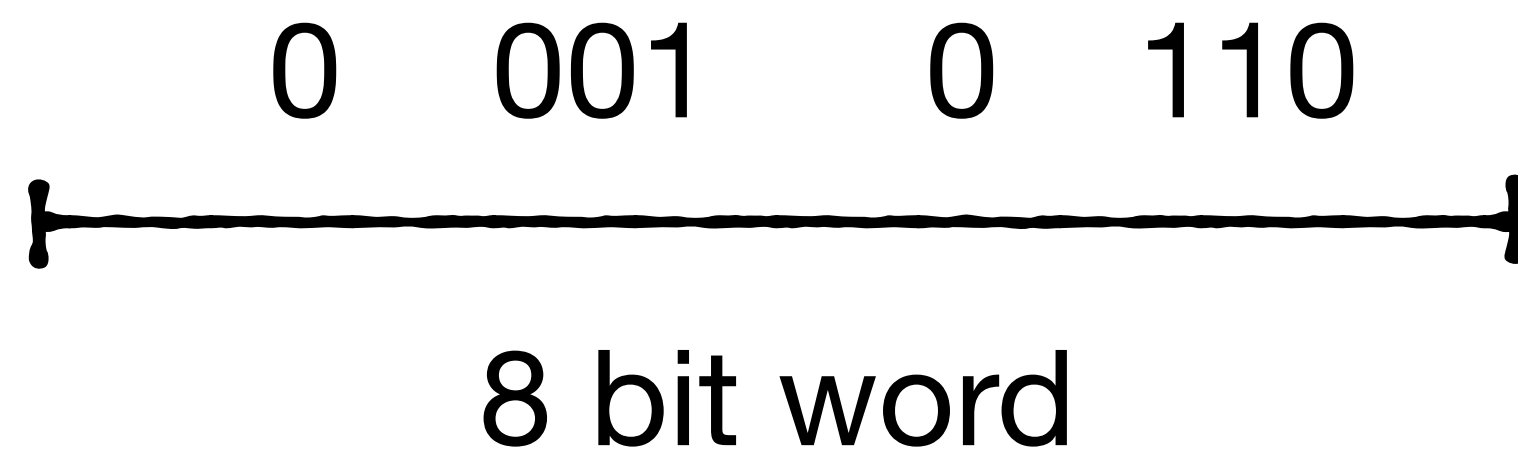
$$X = Y \text{ iff } (X \oplus 01^K) + Y = 01^K$$

$$\mathbf{X < Y \text{ iff } ((X \oplus 01^K) + Y) \wedge 10^K = 10^K}$$



Reset remaining bits

Now let's apply this in parallel on all codes in a word



Now let's apply this in parallel on all codes in a word

Select * where $X < 5$



Now let's apply this in parallel on all codes in a word

Select * where $X < 5$

word	0	001	0	110
		$\oplus$		$\oplus$
mask	0	111	0	111

Now let's apply this in parallel on all codes in a word

Select * where $X < 5$

word	0	001	0	110
		$\oplus$		$\oplus$
mask	0	111	0	111
		+		+
Constant	0	101	0	101

Now let's apply this in parallel on all codes in a word

Select * where $X < 5$

word	0	001	0	110
		$\oplus$		$\oplus$
mask	0	111	0	111
		+		+
Constant	0	101	0	101
		$\wedge$		$\wedge$
-mask	1	000	1	000

Now let's apply this in parallel on all codes in a word

Select * where $X < 5$

word $\oplus$ mask	0	110	0	001
		+		+
Constant	0	101	0	101
		$\wedge$		$\wedge$
-mask	1	000	1	000

Now let's apply this in parallel on all codes in a word

Select * where $X < 5$

Constant + word $\oplus$ mask	1	011	0	110
		$\wedge$		$\wedge$
-mask	1	000	1	000

Now let's apply this in parallel on all codes in a word

Select * where $X < 5$

$$(\text{Constant} + \text{word} \oplus \text{mask}) \wedge \neg \text{mask} = \quad \mathbf{1} \quad 000 \quad \mathbf{0} \quad 000$$

Next challenge: reducing sparse result vector into bitmap

1 000 0 000 → 1 0

Next challenge: reducing sparse result vector into bitmap

1 000 0 000 → 1 0

Bit Shifting
(From paper)

Parallel Bit Extract - PEXT
(Intel Haswell BMI2, 2013)

Bit Shifting

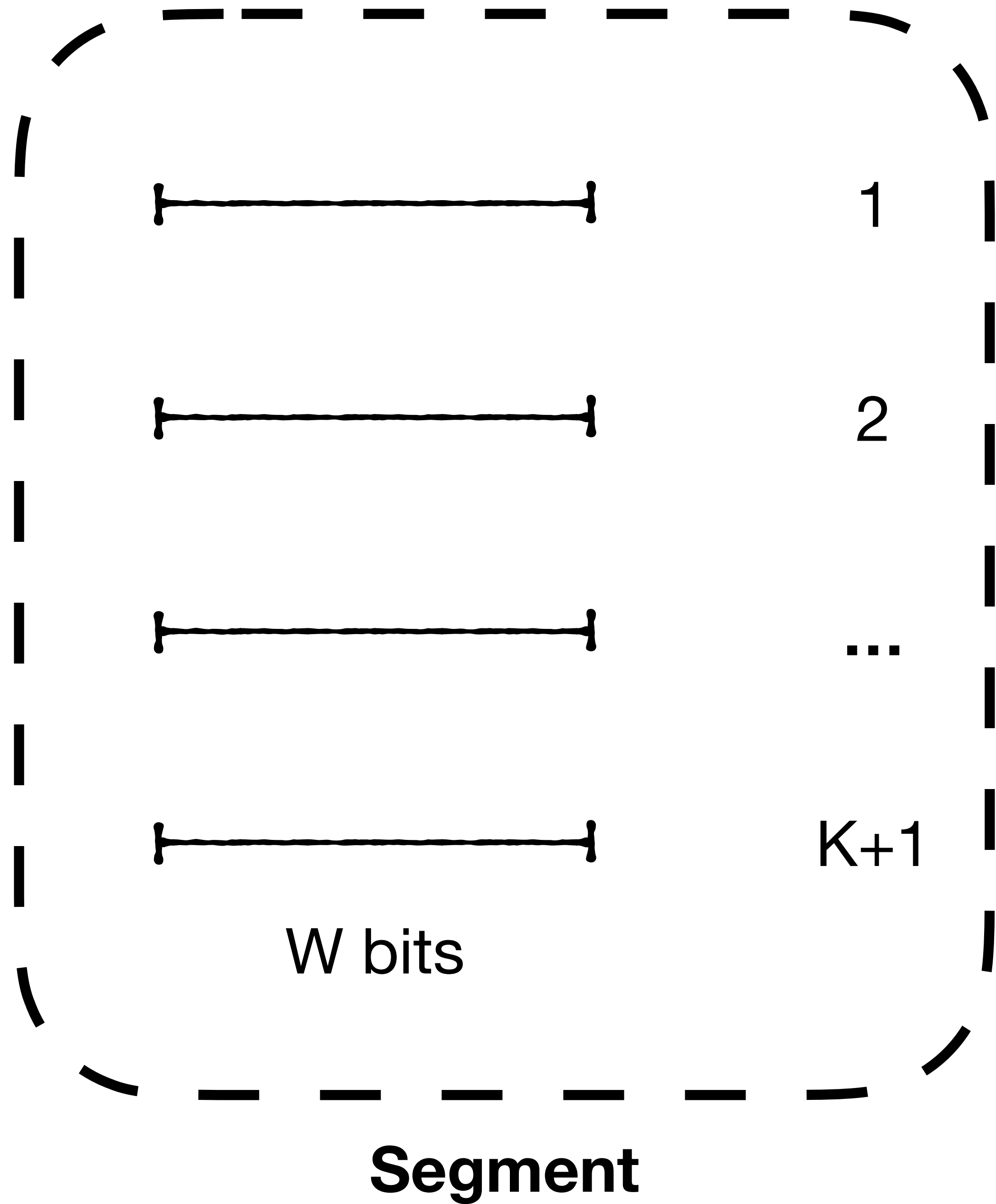
Column codes

c1	001		1
c2	101		
c3	110		2
c4	001		
c5	110		...
c6	100		
c7	000		K+1
c8	111		

W bits

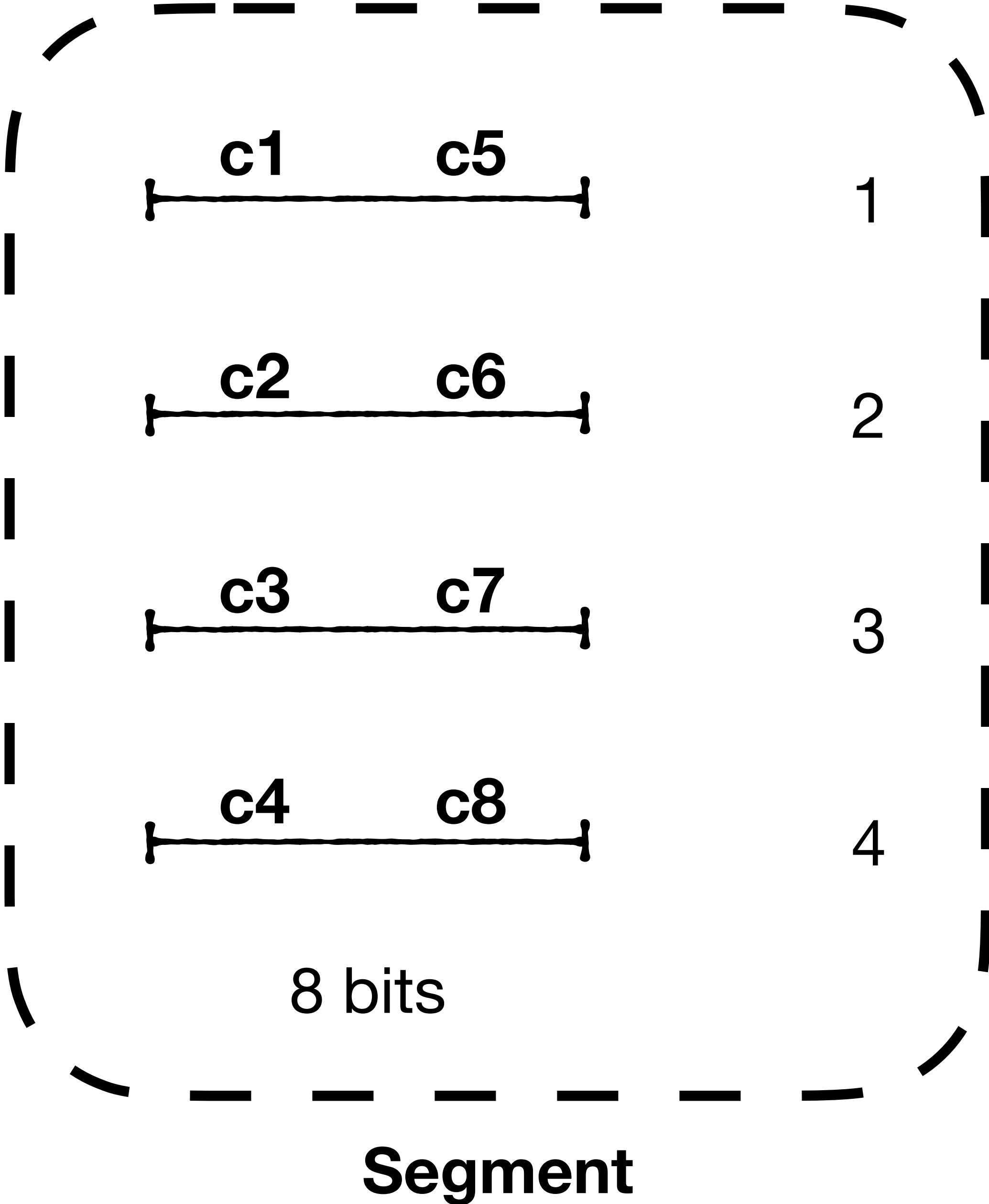
Column codes

c1	001
c2	101
c3	110
c4	001
c5	110
c6	100
c7	000
c8	111

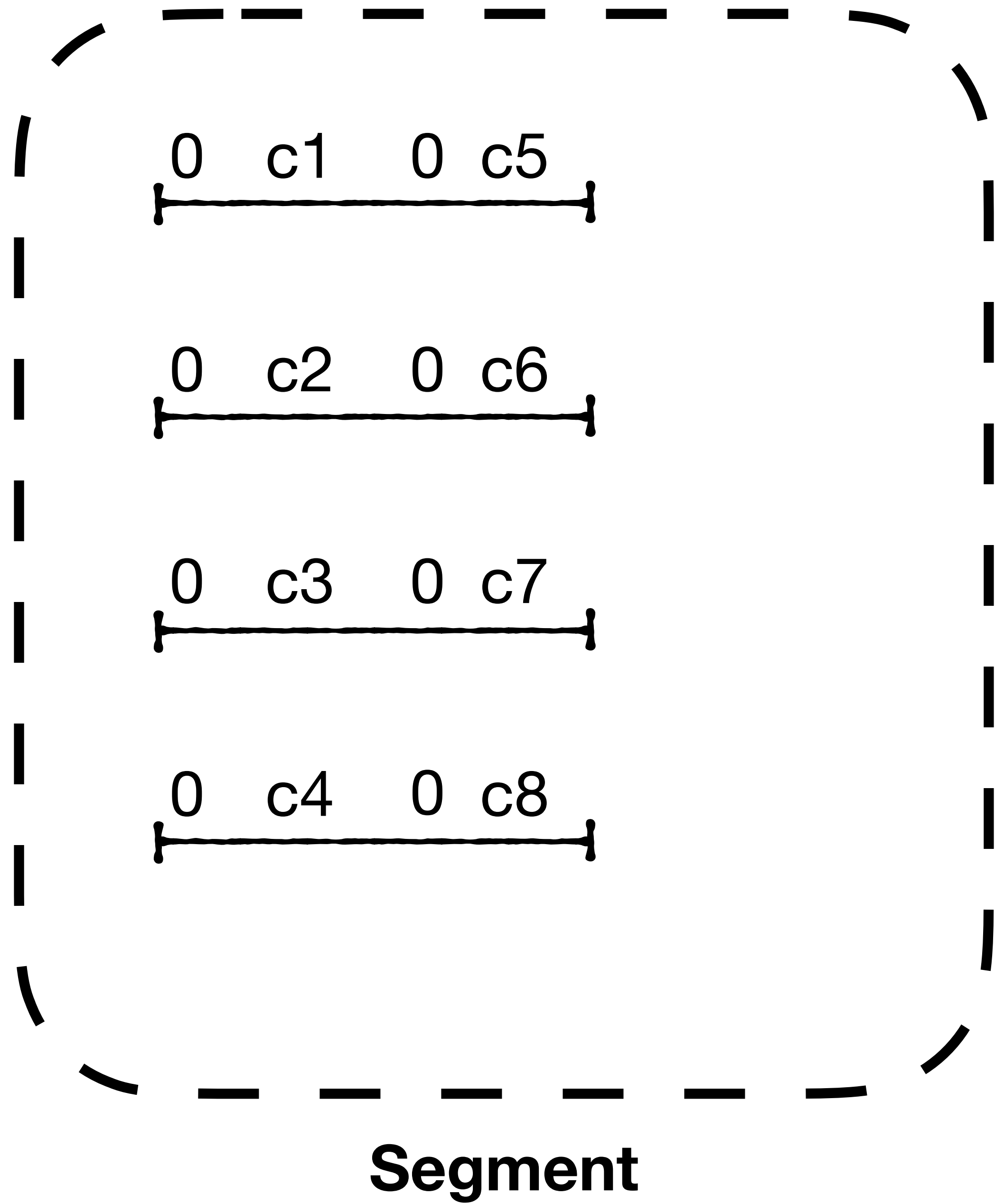


Column codes

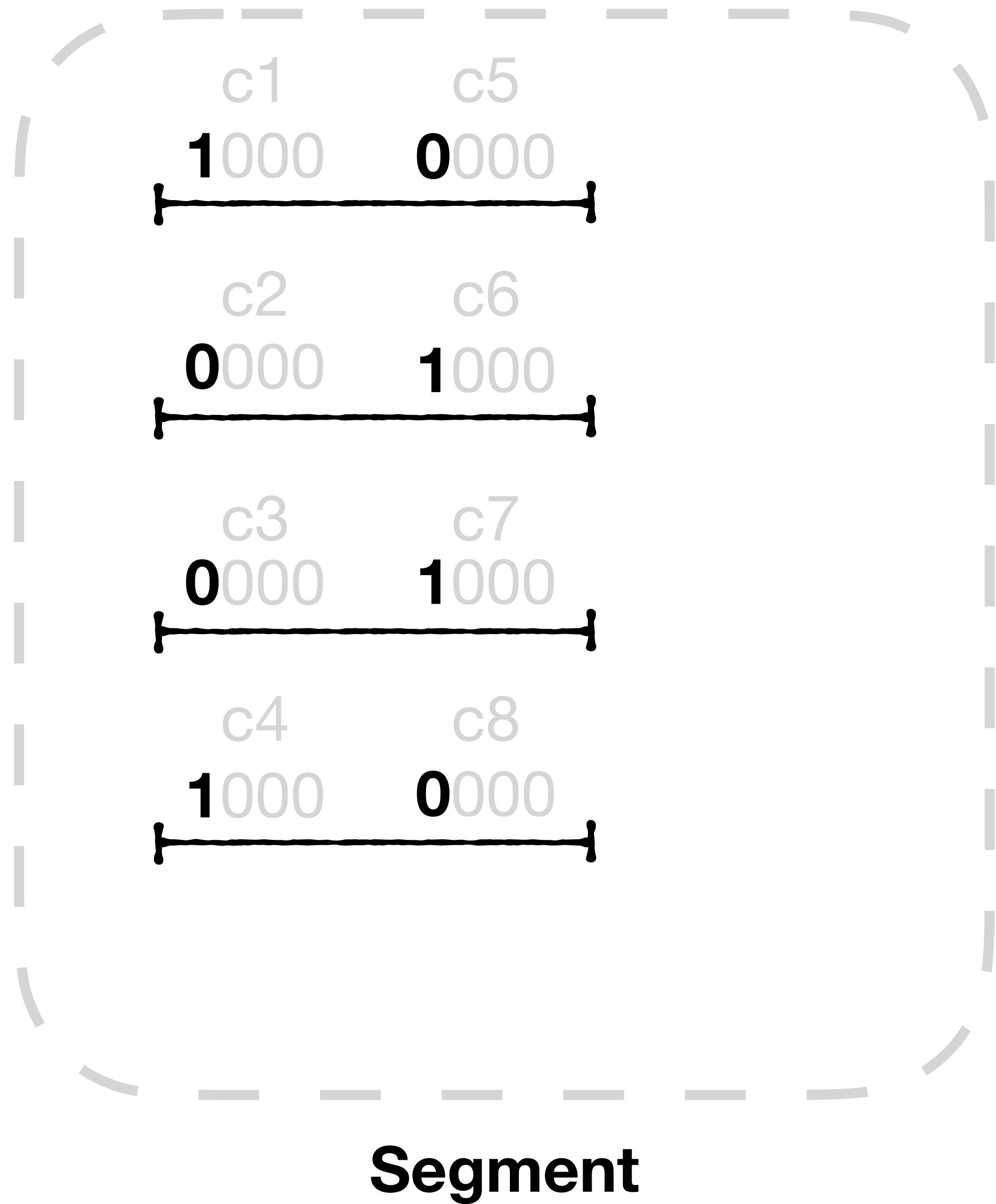
c1	001
c2	101
c3	110
c4	001
c5	110
c6	100
c7	000
c8	111



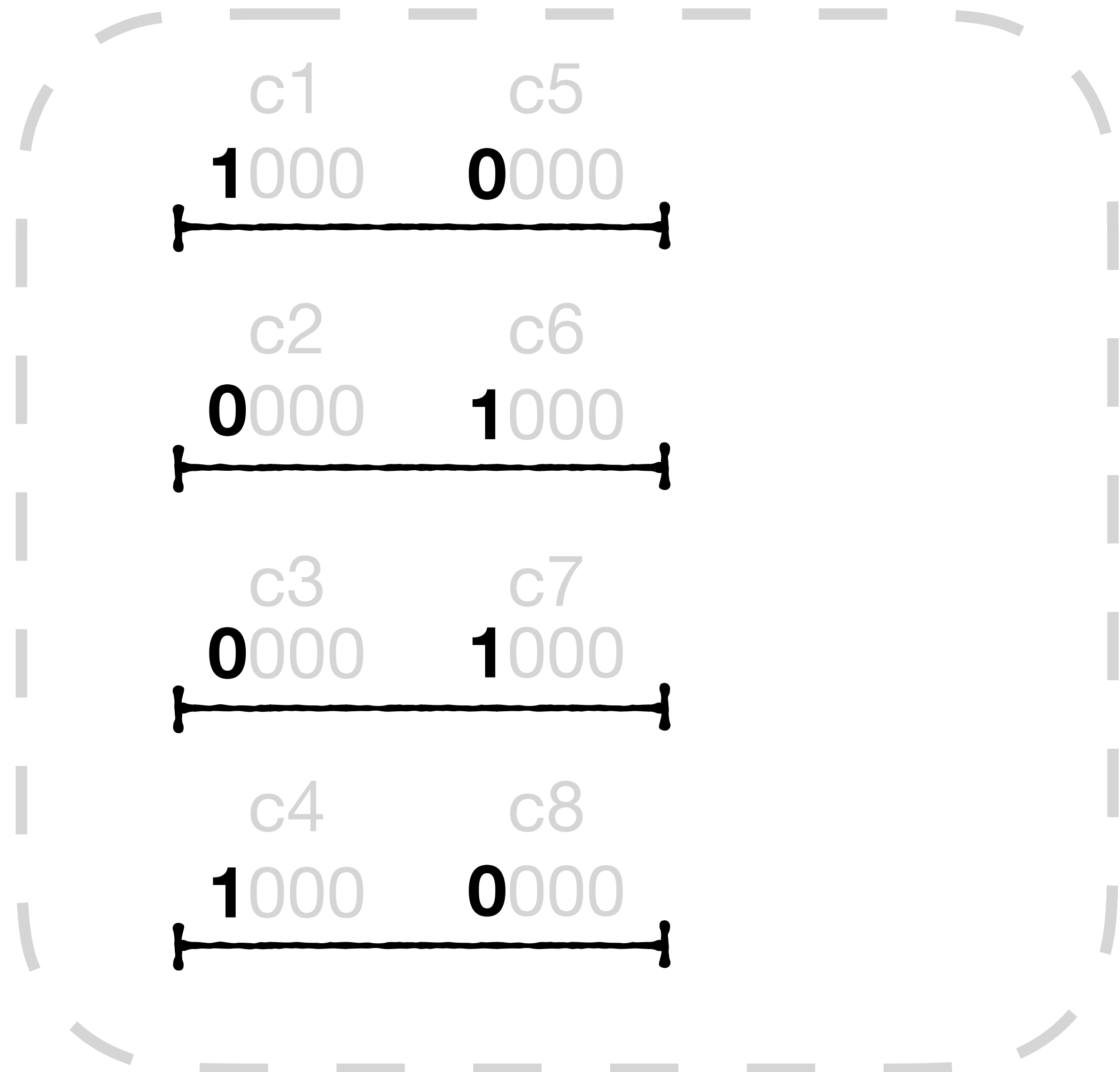
**Add delimiter /
result bit**



Suppose we run a query



Suppose we run a query



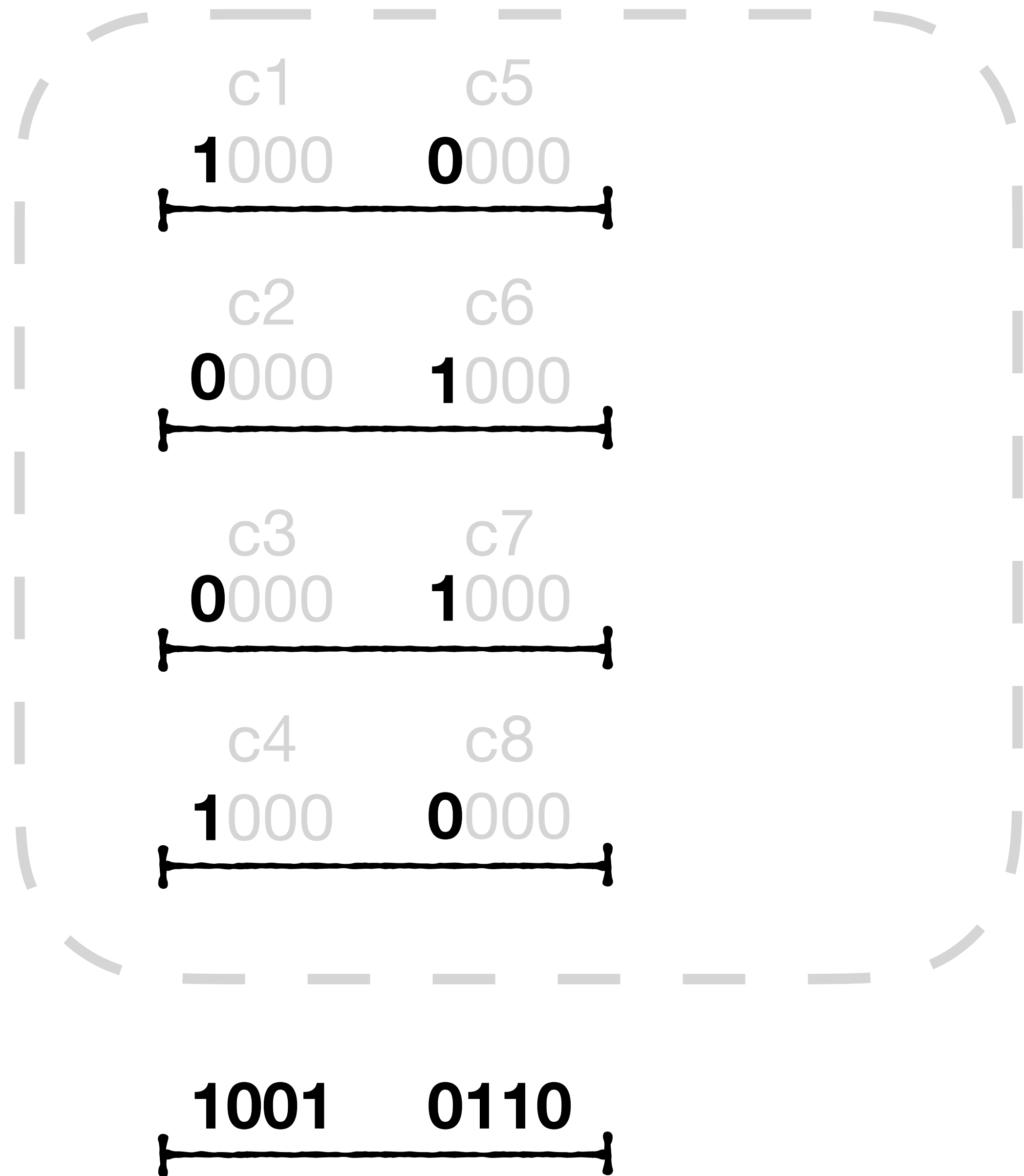
Expected result:

1001 0110

Suppose we run a query

What shifts do we do?

Expected result:



Rightwards bit shift

0



1



2



3



Expected result:



Rightwards bit shift

0

c1 c5
1000 **0**000

1

c2 c6
0**0**00 0**1**00

2

c3 c7
00**0**0 00**1**0

3

c4 c8
000**1** 0000**0**

Expected result:

1001 0110

$$\begin{array}{r}
 \begin{array}{cc}
 c1 & c5 \\
 \hline
 \mathbf{1}000 & \mathbf{0}000
 \end{array} \\
 \vee \\
 \begin{array}{cc}
 c2 & c6 \\
 \hline
 0\mathbf{0}00 & 0\mathbf{1}00
 \end{array} \\
 \vee \\
 \begin{array}{cc}
 c3 & c7 \\
 \hline
 00\mathbf{0}0 & 00\mathbf{1}0
 \end{array} \\
 \vee \\
 \begin{array}{cc}
 c4 & c8 \\
 \hline
 000\mathbf{1} & 0000\mathbf{0}
 \end{array} \\
 =
 \end{array}$$

Now “or” all the words into result:

$$\begin{array}{cc}
 \hline
 \mathbf{1}001 & \mathbf{0}110 \\
 \hline
 \end{array}$$

Bit Shifting
(From paper)

Parallel Bit Extract - PEXT
(Intel Haswell BMI2, 2013)

Parallel Bit Extract - PEXT

Extract bits at specific locations and store them contiguously

Parallel Bit Extract - PEXT

Extract bits at specific locations and store them contiguously

Mask	10100101
-------------	-----------------

Input	00101011
--------------	-----------------

Parallel Bit Extract - PEXT

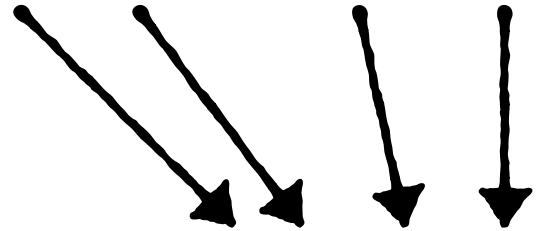
Extract bits at specific locations and store them contiguously

Mask **10100101**

Input **00101011**

Parallel Bit Extract - PEXT

Extract bits at specific locations and store them contiguously

Mask	1 0 1 00 1 0 1
Input	0 0 1 0 1 0 1 1
	
Result:	0 000 1 0 1

Horizontal BitWeaving Summary

Packs codes
efficiently into word

Uses word-level
parallelism

Does not do early
pruning

Horizontal BitWeaving Summary

Packs codes
efficiently into word

Uses word-level
parallelism

Does not do early
pruning

Some padding
at end

BitWeaving: Fast Scans for Main Memory Data Processing



Horizontal



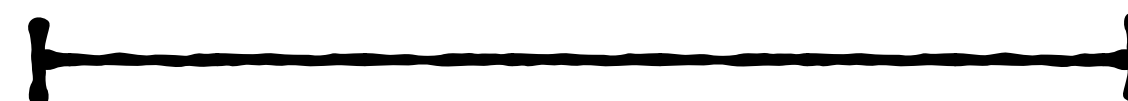
Vertical

Vertical BitWeaving

Column codes

c1 001

c2 101



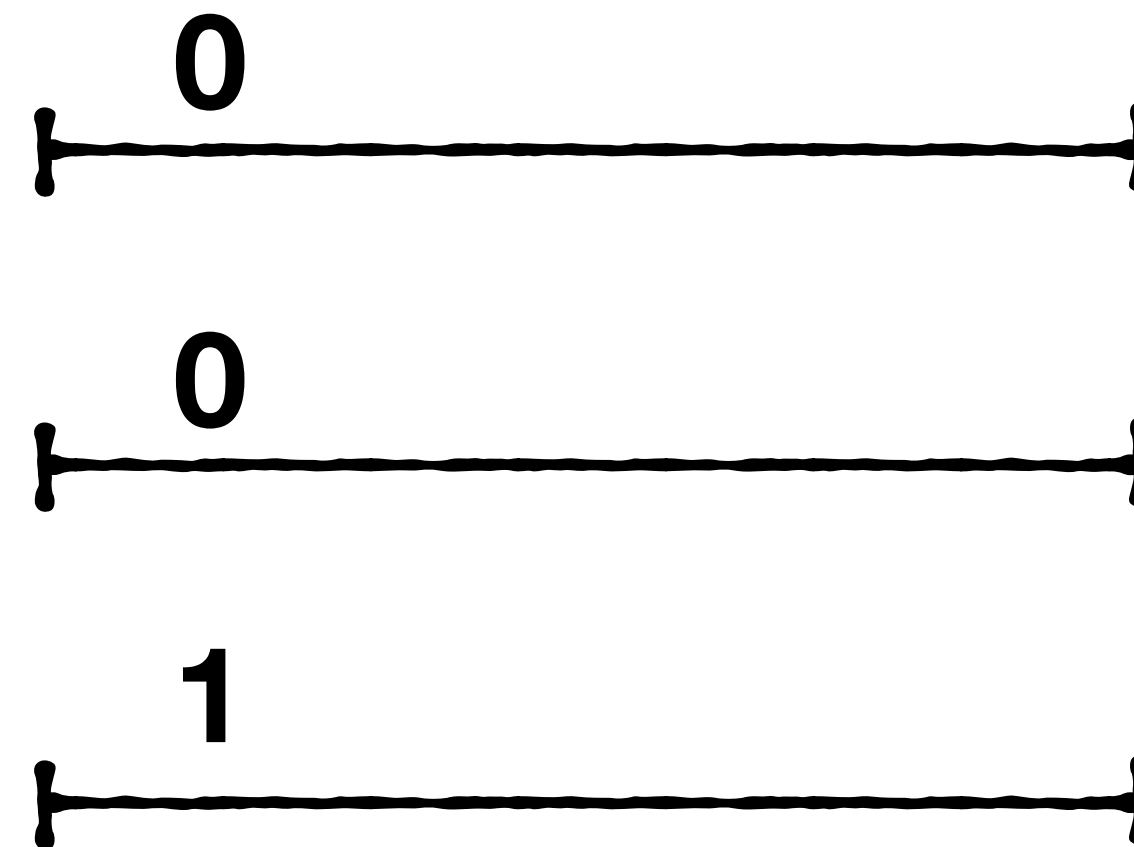
8 bit word

Vertical BitWeaving

Column codes

c1 **001**

c2 101



8 bit words

Column codes

c1 001

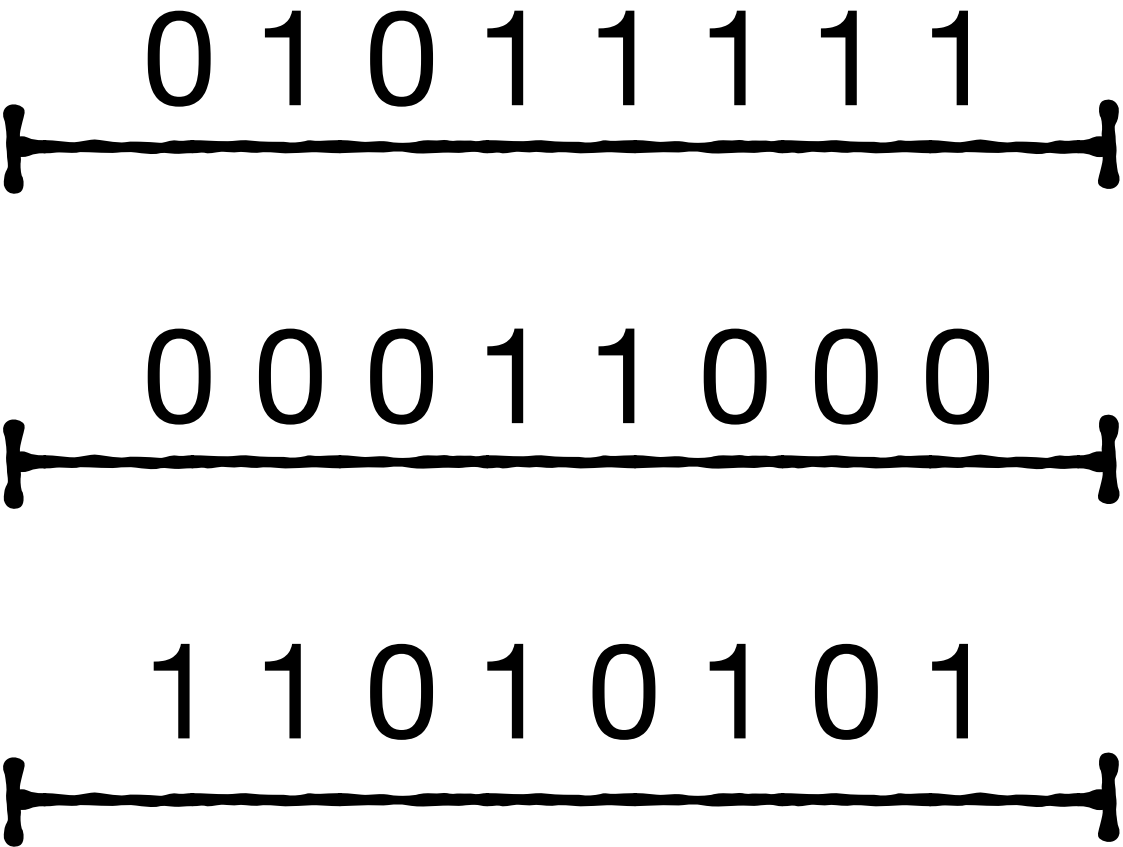
c2 101



8 bit words

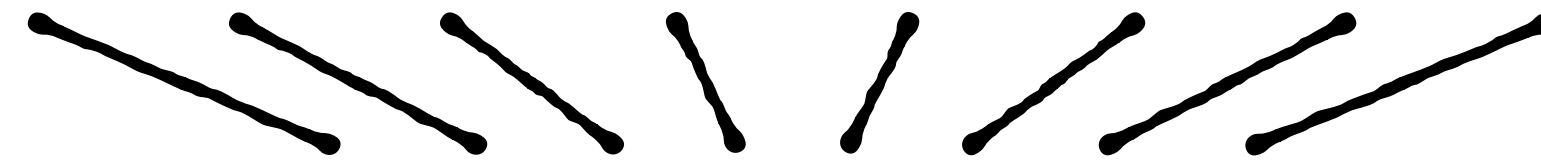
Column codes

c1	001
c2	101
c3	000
c4	111
c5	110
c6	101
c7	100
c8	101



8 bit words

c1 c2 c3 c4 c5 c6 c7 c8



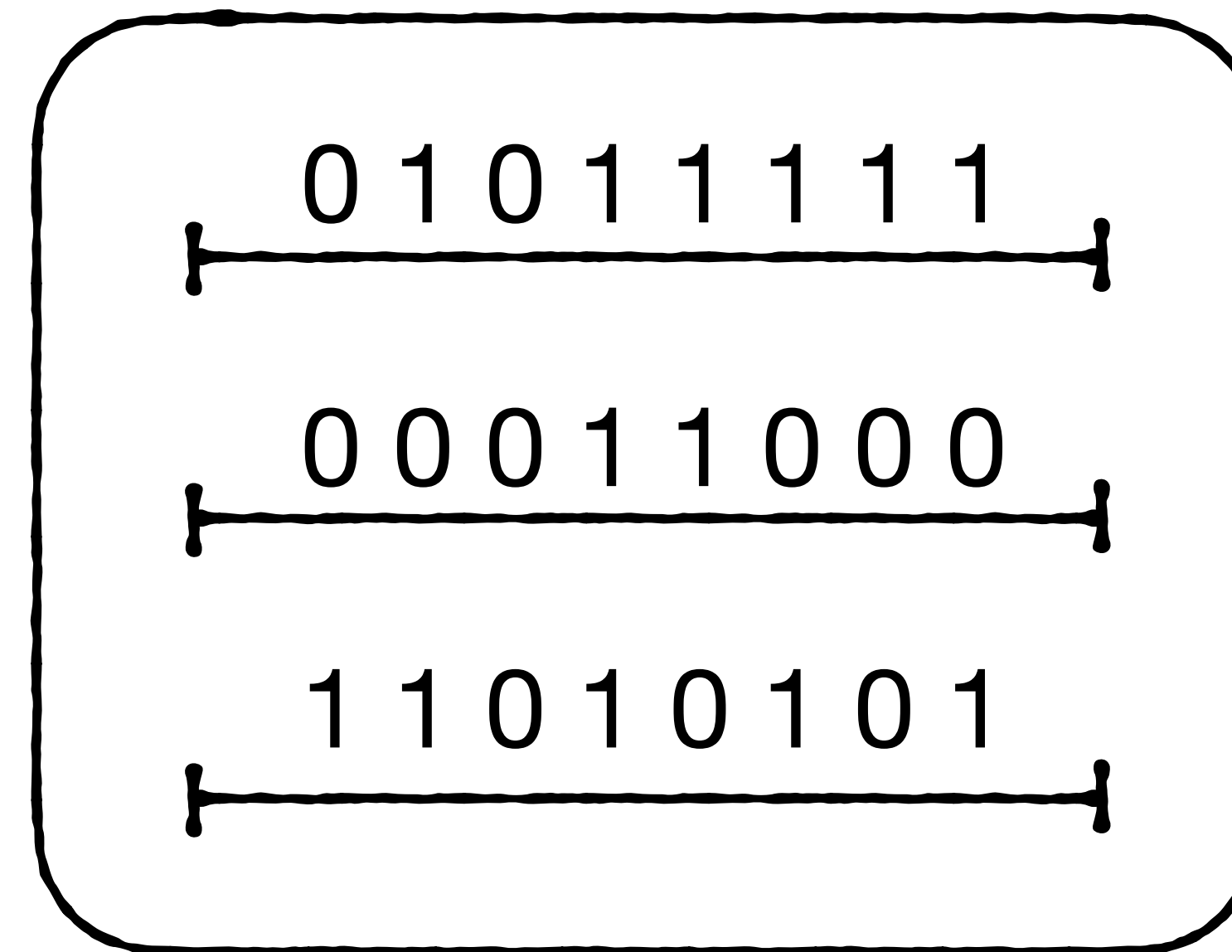
0 1 0 1 1 1 1 1

0 0 0 1 1 0 0 0

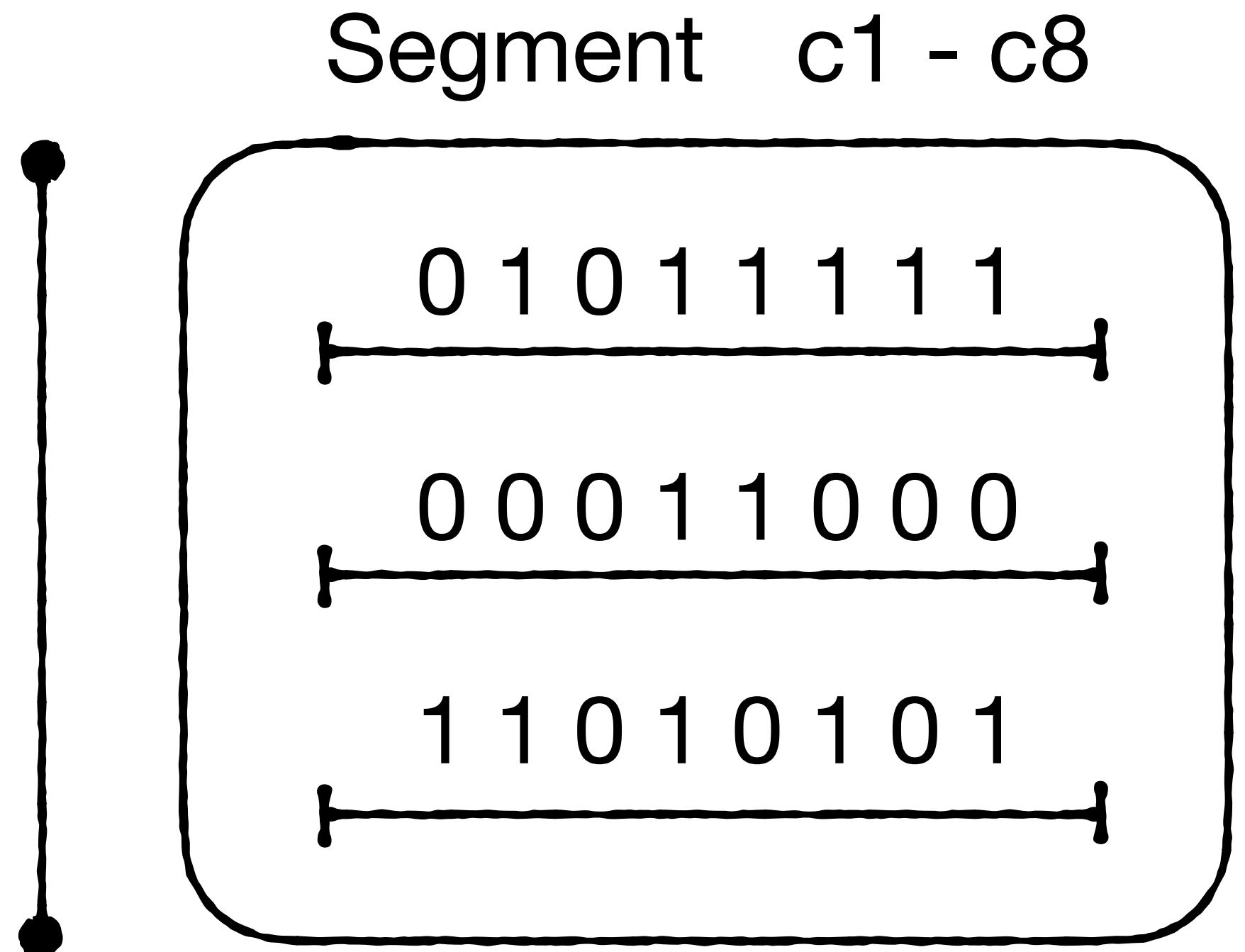
1 1 0 1 0 1 0 1

8 bit words

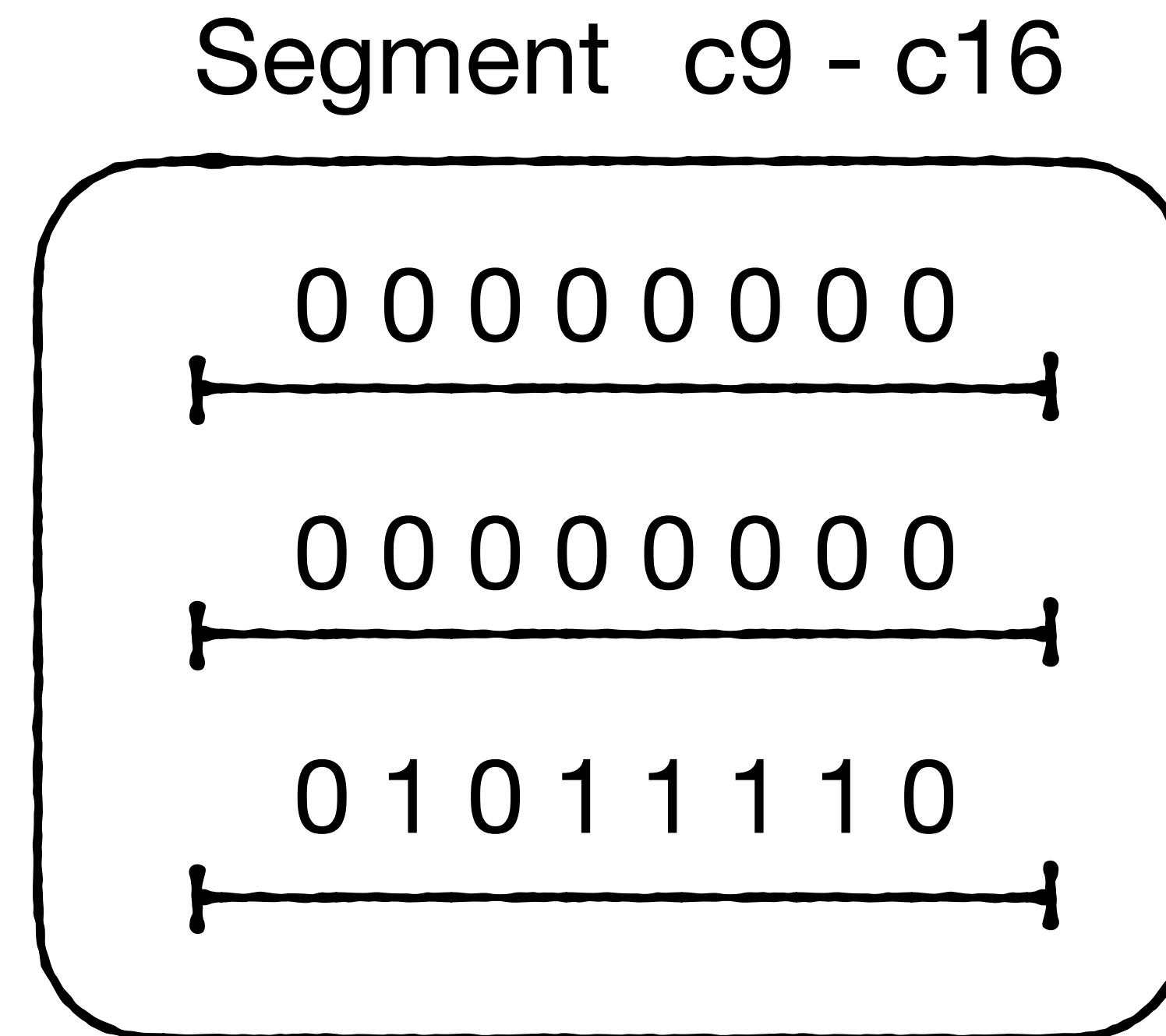
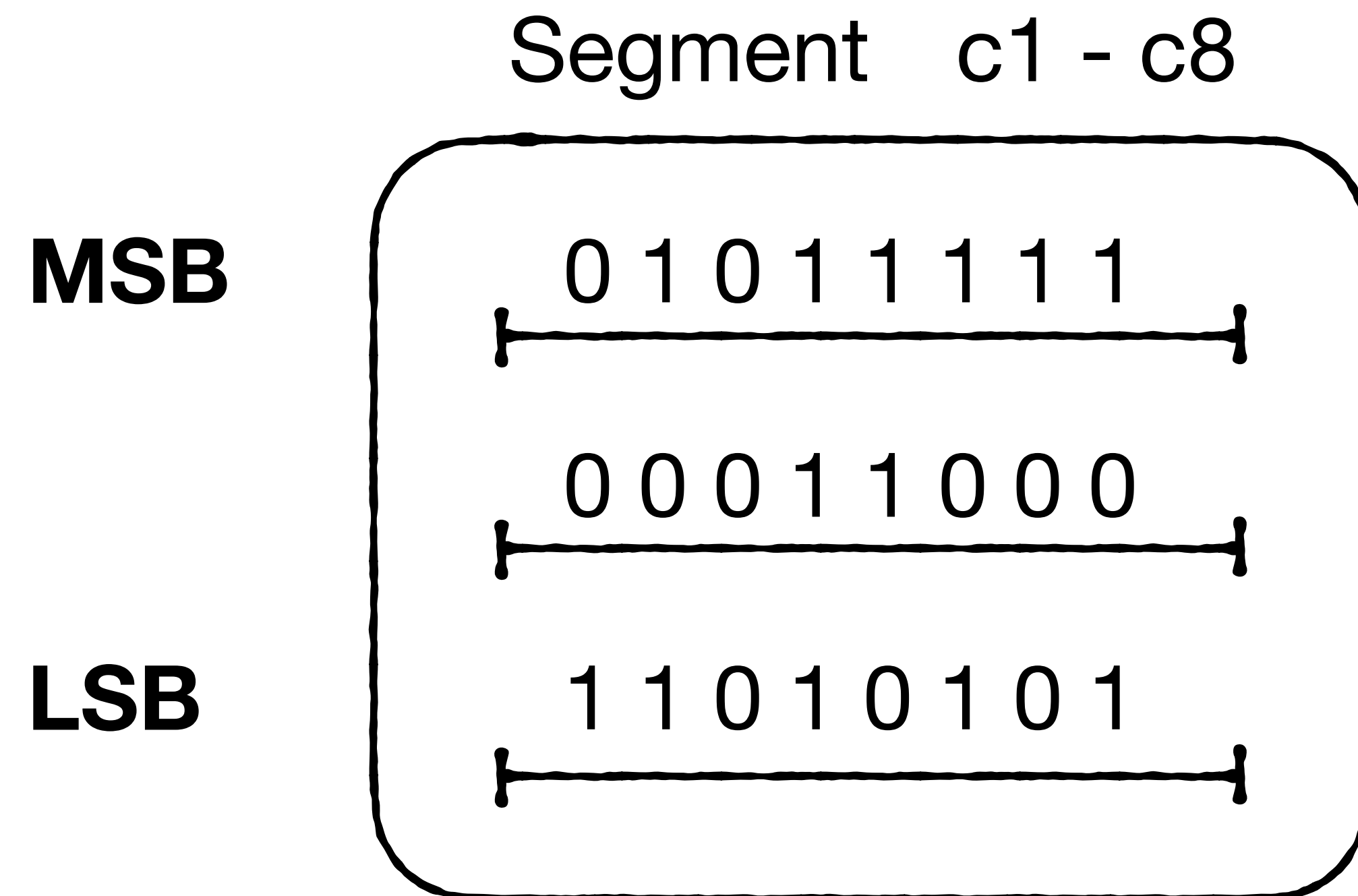
Segment c1 - c8



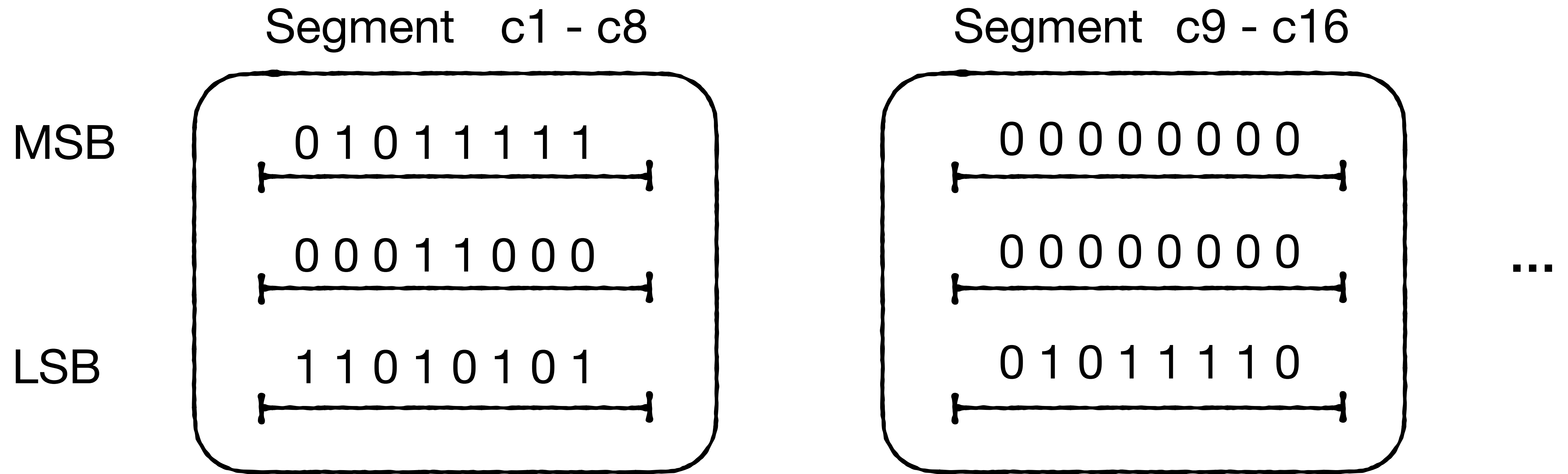
**|C| words per segment,
where |C| is code length**
e.g., 3



**|W| codes per segment,
where |W| is word length**
e.g., 8

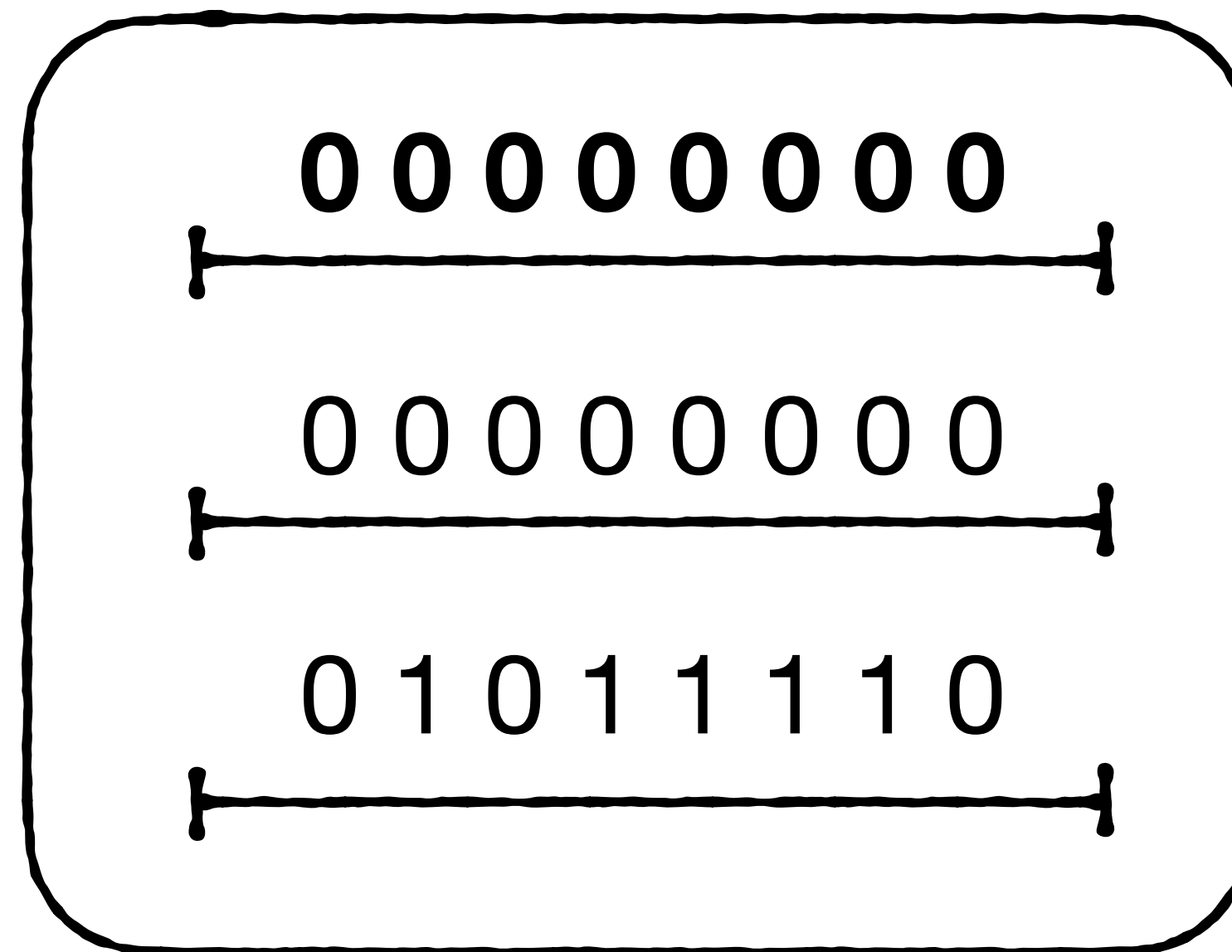


...



Select * where c > 100

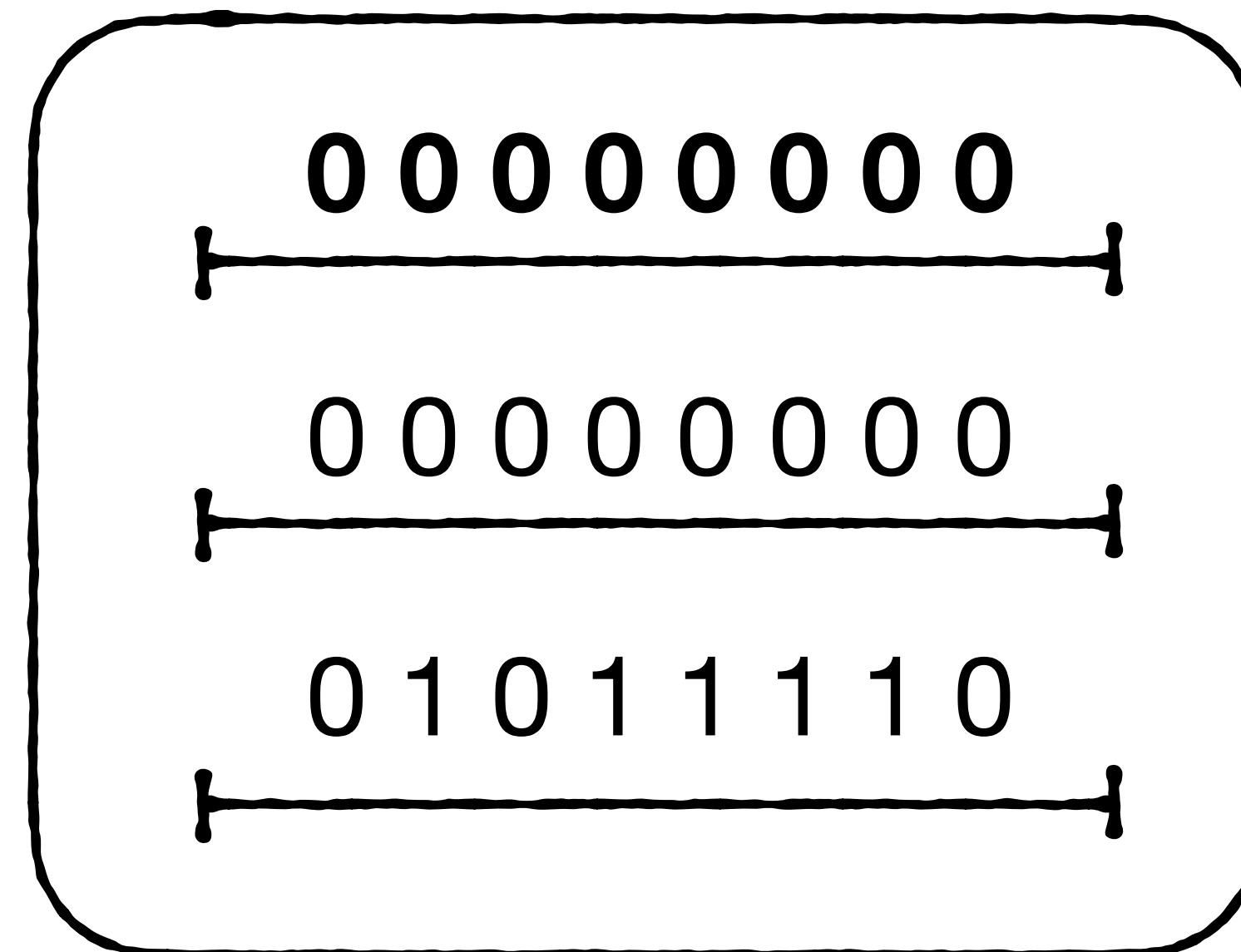
Segment c9 - c16



← Check

Select * where c > 100

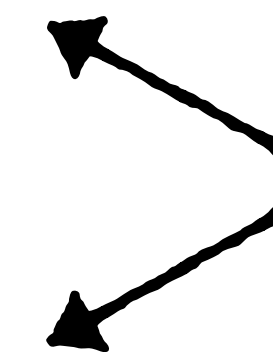
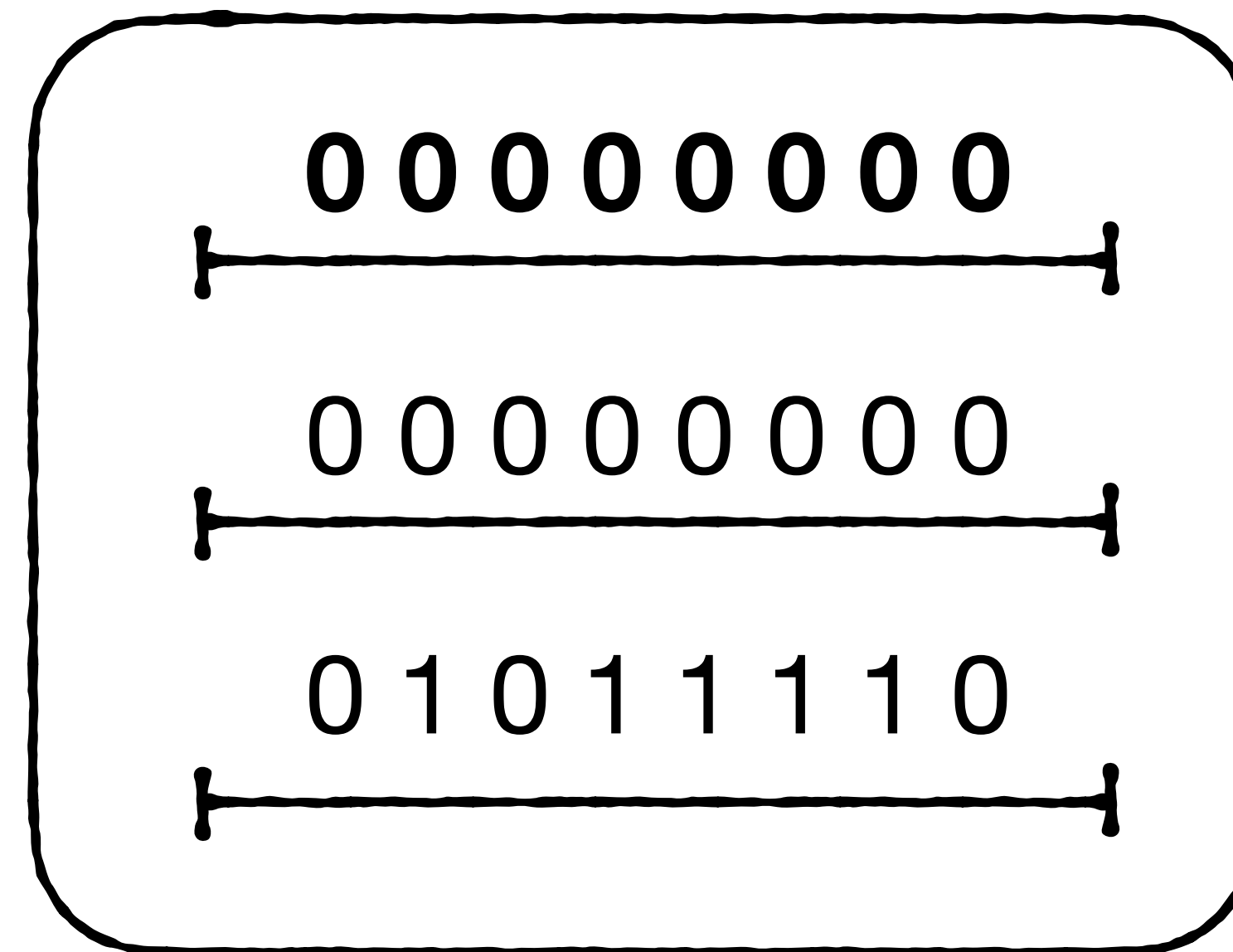
Segment c9 - c16



← Check
**No need to scan
further :)**

Select * where c > 100

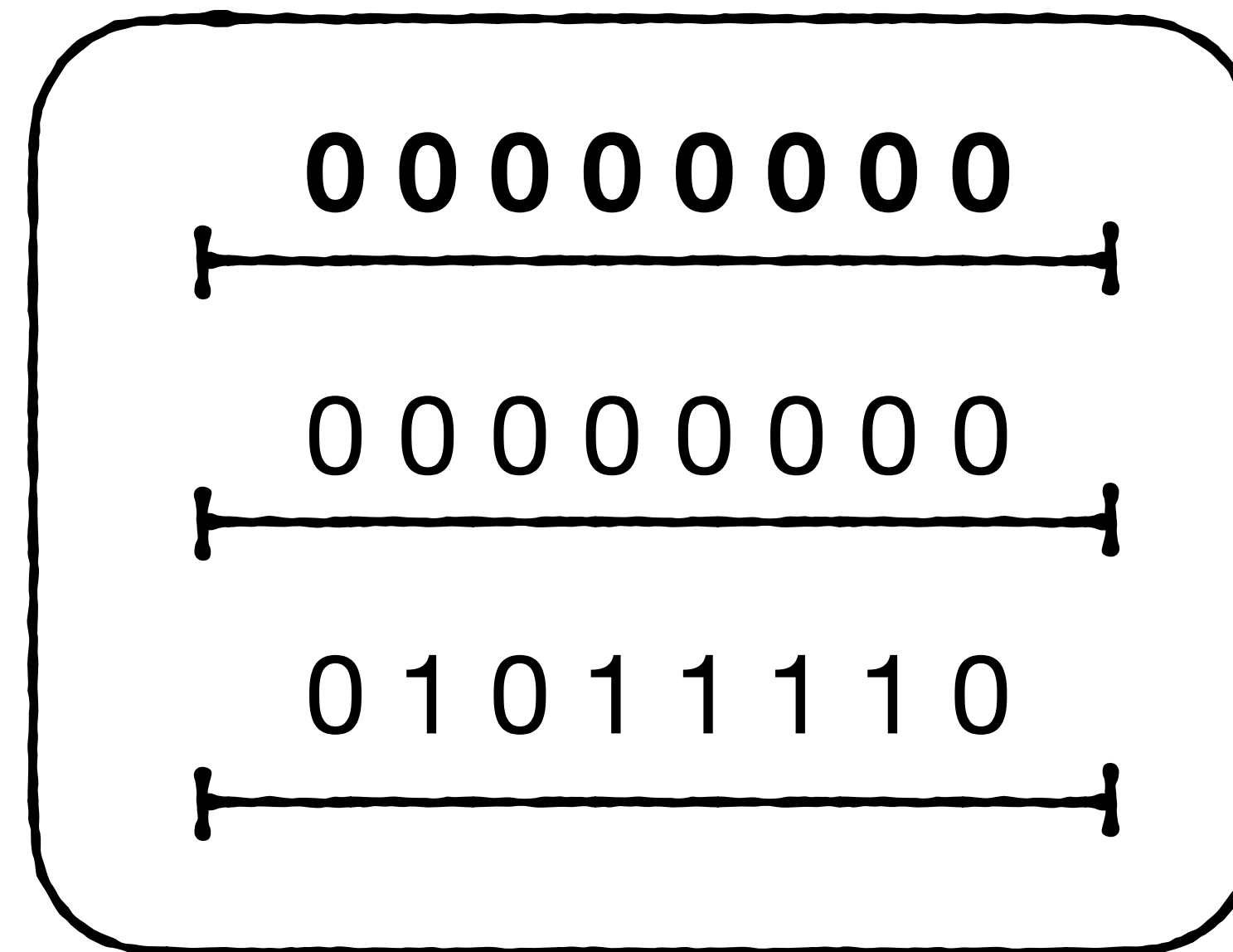
Segment c9 - c16



**But these still
pollute the cache**

Select * where c > 100

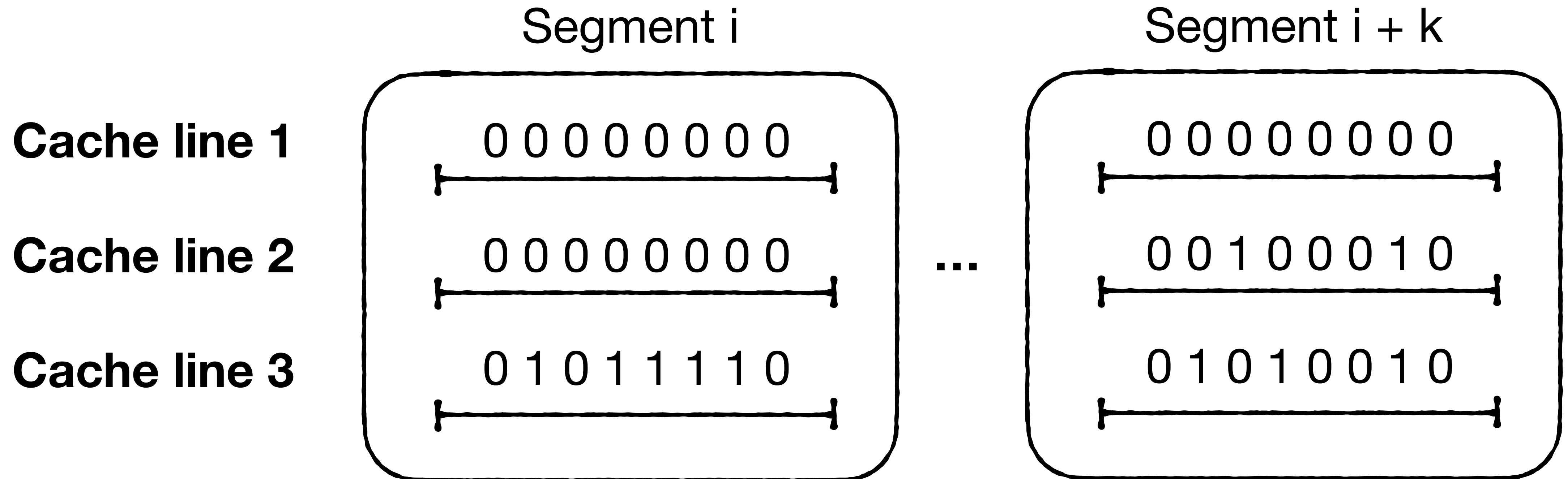
Segment c9 - c16



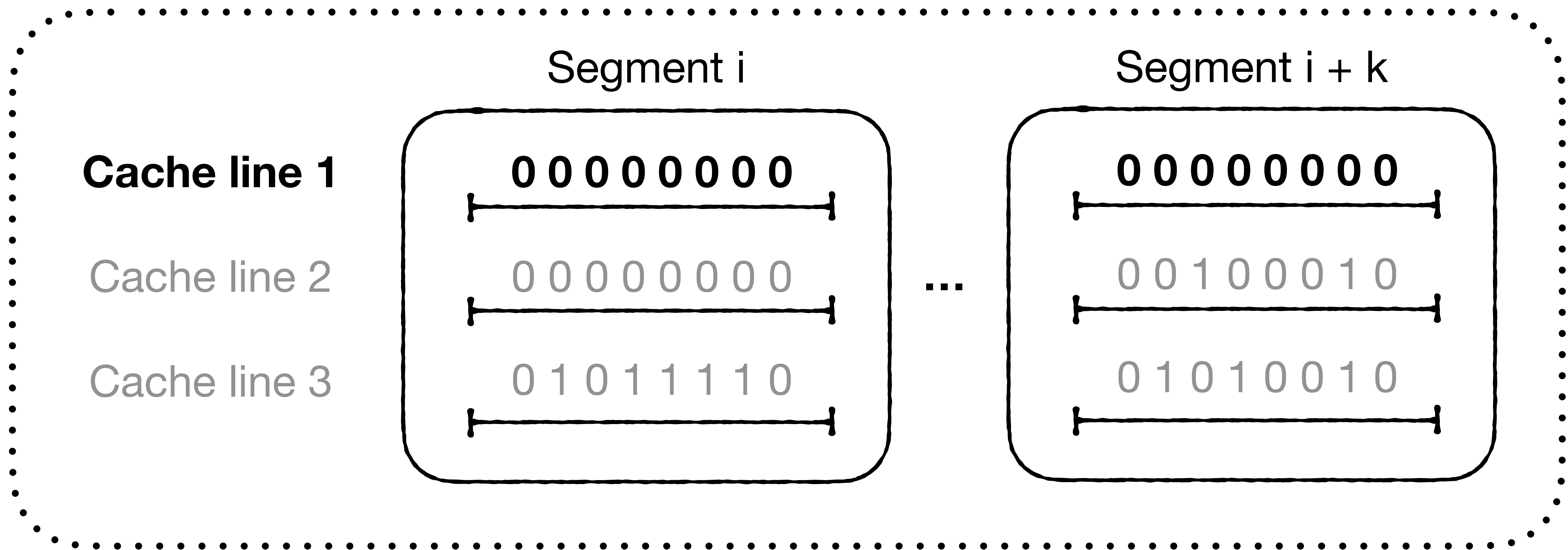
But these still
pollute the cache
**Can we further
optimize?**

Select * where c > 100

Interleave segments across cache lines :)



Interleave segments across cache lines :)



Ideally, we skip all but the first cache line in this grouping

e.g., Select * where c > 100

So far, vertical bitweaving wastes less space than horizontal (no padding or result bits)

Cache line 1

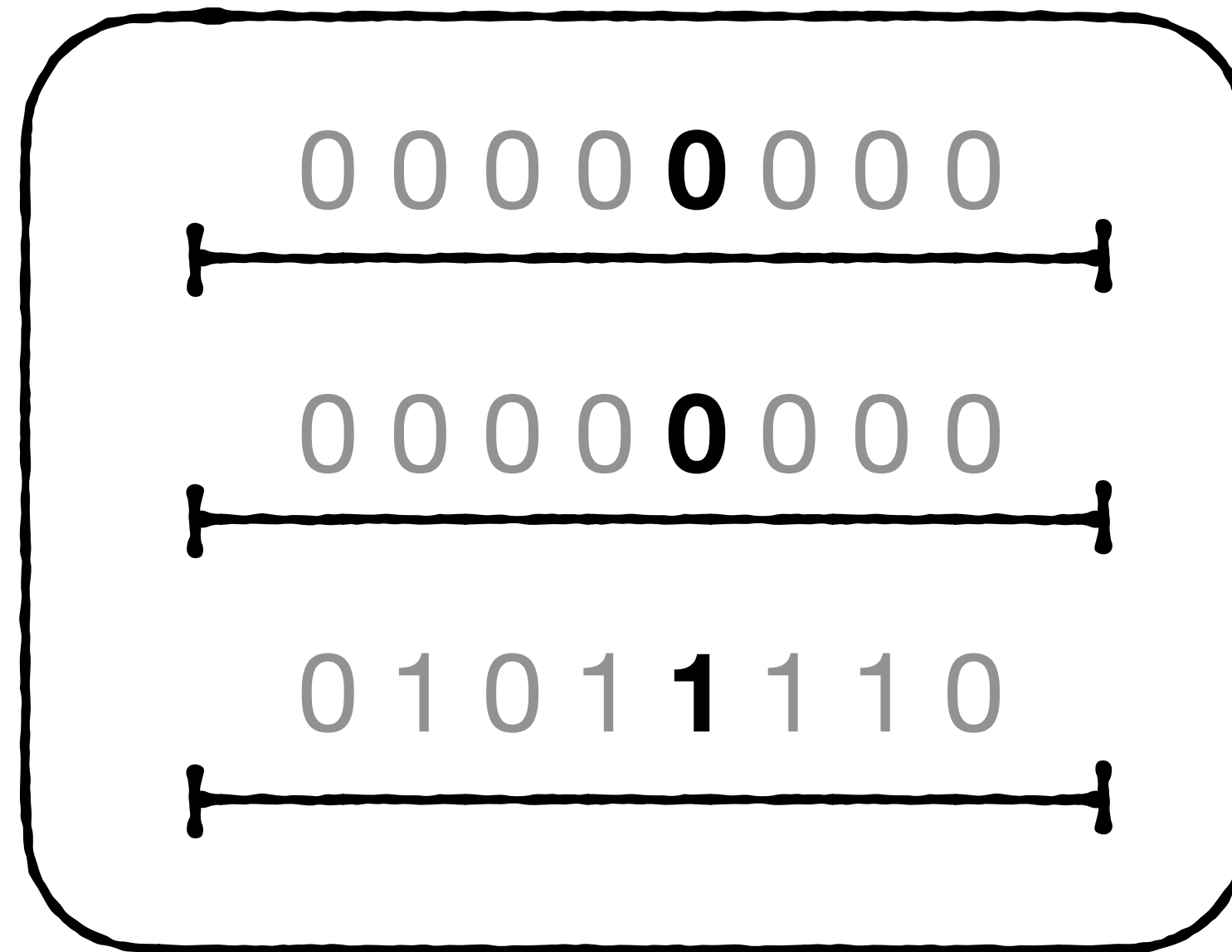
0 0 0 0 **0** 0 0 0

Cache line 2

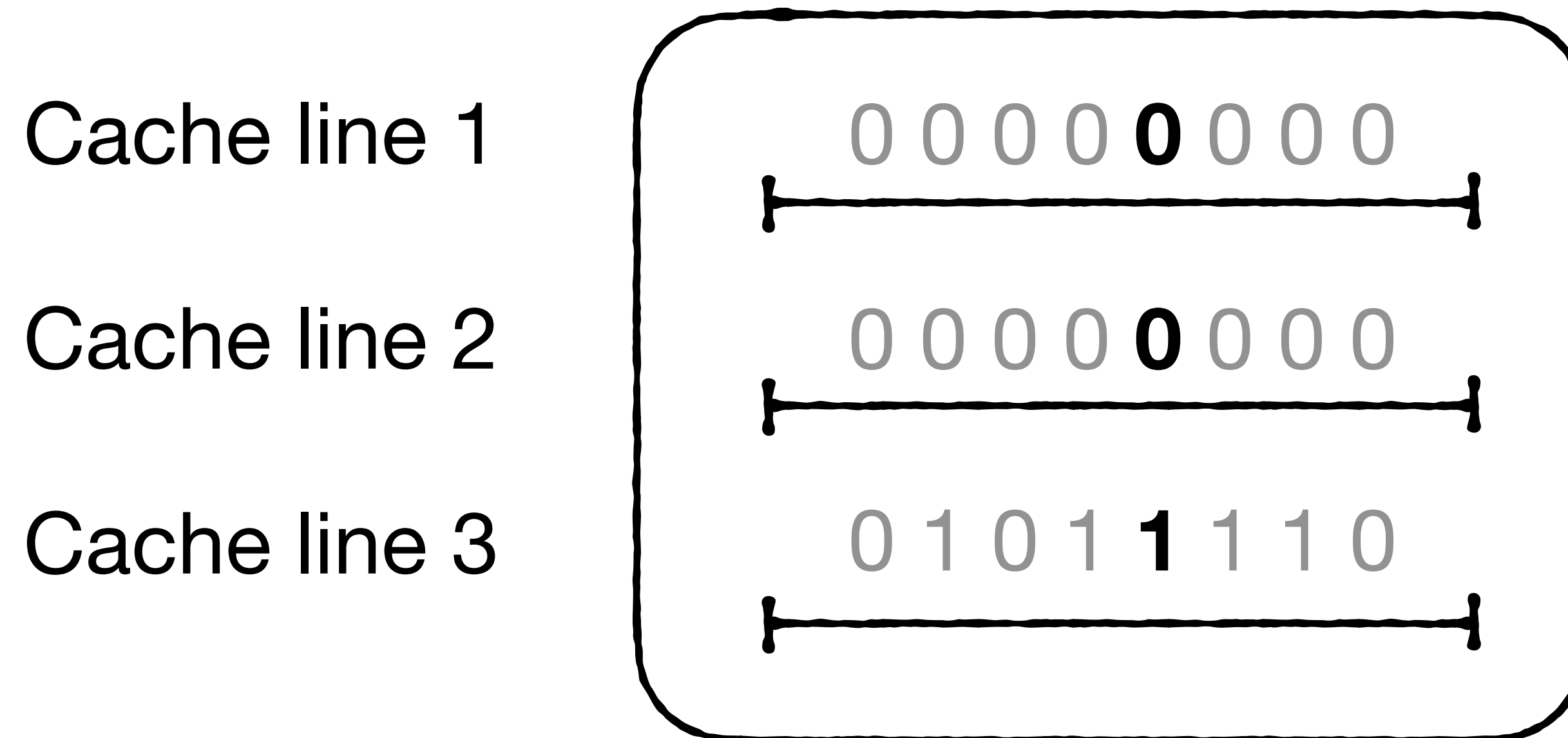
0 0 0 0 **0** 0 0 0

Cache line 3

0 1 0 1 **1** 1 1 0



So far, vertical bitweaving wastes less space than horizontal (no padding or result bits)



It also allows early pruning :)

But suppose we wish to get a value at some specific index



Cache line 1

0 0 0 0 **0** 0 0 0

Cache line 2

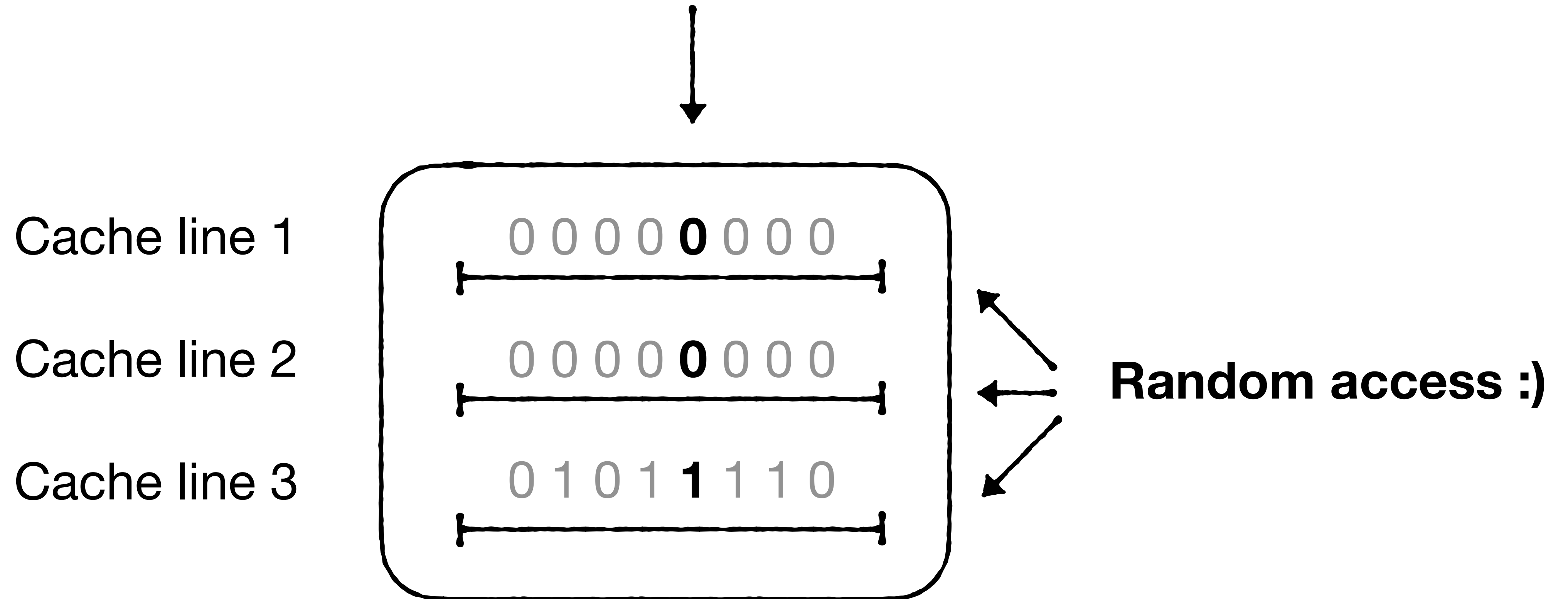
0 0 0 0 **0** 0 0 0

Cache line 3

0 1 0 1 **1** 1 1 0

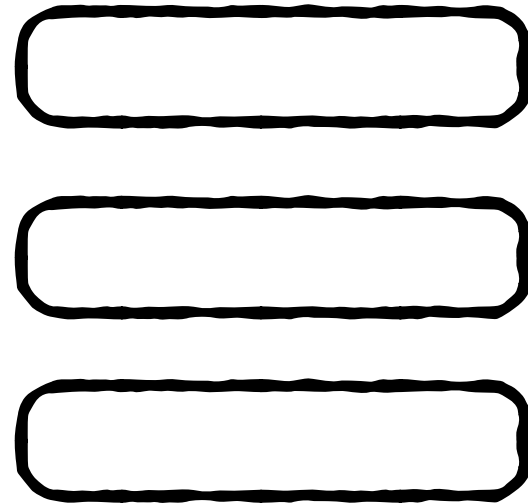
Any issue?

But suppose we wish to get a value at some specific index



BitWeaving Summary

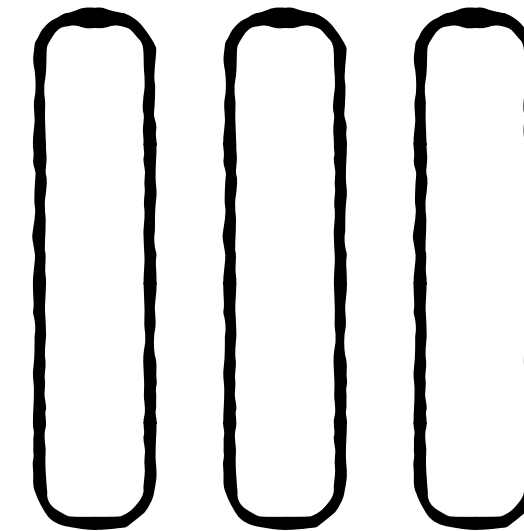
Horizontal



**More
space**

**No early
pruning**

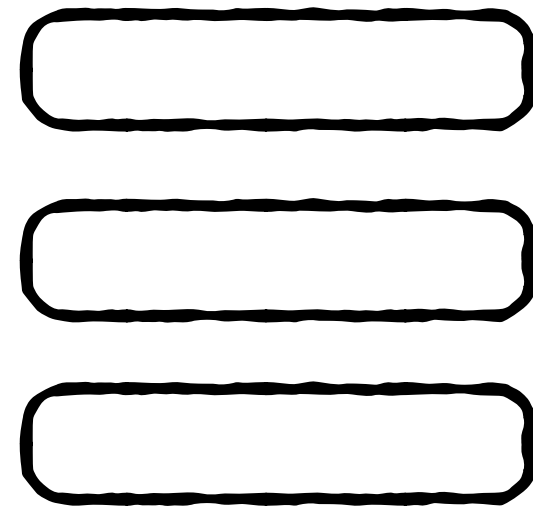
Vertical



**Selective value
reconstruction entails
random I/Os**

BitWeaving Summary

Horizontal

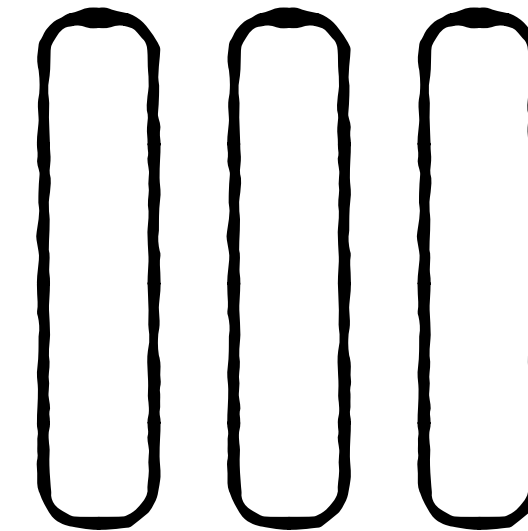


More
space

No early
pruning

Unselective queries
Larger codes

Vertical



Selective value
reconstruction entails
random I/Os

Selective queries
Smaller codes



BitWeaving
SIGMOD 2013



Column-Sketches
SIGMOD 2018