

Bit Arithmetics

CS143: lecture 15

Byron Zhong, July 1

Unsigned Arithmetics

- unsigned means without sign (+/-). In C, `unsigned int x = 4`
- Why unsigned?
 - 32 bits means 2^{32} choices
 - If we don't allow negative number, we have range $[0, 2^{32} - 1]$
 - If we have negative numbers, the maximum has to be smaller.
- Most things that we care about are non-negative
 - counts, lengths, indices, dimensions...

Unsigned Arithmetics

Operators

- Arithmetic operators: $+$, $-$, $*$, $/$
- Bitwise operators: $\&$, $|$, $\wedge$, $\sim$
- Shifts: $\ll$, $\gg$

Unsigned Arithmetics

```
unsigned int n = 45;
```

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128

1101 0011 1100 0001 0010 0100 1001 1111

& 0101 0101 0101 0101 0101 0101 0101 0101

Unsigned Arithmetics

```
unsigned int n = 45;
```

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128

1101 0011 1100 0001 0010 0100 1001 1111

& 0101 0101 0101 0101 0101 0101 0101 0101

1

Unsigned Arithmetics

```
unsigned int n = 45;
```

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128

1101 0011 1100 0001 0010 0100 1001 1111

& 0101 0101 0101 0101 0101 0101 0101 0101

11

Unsigned Arithmetics

```
unsigned int n = 45;
```

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128

1101 0011 1100 0001 0010 0100 1001 1111

& 0101 0101 0101 0101 0101 0101 0101 0101

111

Unsigned Arithmetics

```
unsigned int n = 45;
```

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128

1101 0011 1100 0001 0010 0100 1001

& 0101 0101 0101 0101 0101 0101 0101

0111

0111

```
unsigned int n = 45;
```

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128

	1101	0011	1100	0001	0010	0100	1001	1111	
&	0101	0101	0101	0101	0101	0101	0101	0101	
								1	0111

```
unsigned int n = 45;
```

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128

	1101	0011	1100	0001	0010	0100	1001	1111
&	0101	0101	0101	0101	0101	0101	0101	0101
							01	0111

Unsigned Arithmetics

```
unsigned int n = 45;
```

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128

1101 0011 1100 0001 0010 0100 1001 1111

& 0101 0101 0101 0101 0101 0101 0101 0101

0101 0001 0100 0001 0000 0100 0001 0101

Unsigned Arithmetics

```
unsigned int n = 45;
```

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128

1101 0011 1100 0001 0010 0100 1001 1111

| 0101 0101 0101 0101 0101 0101 0101 0101

1

Unsigned Arithmetics

```
unsigned int n = 45;
```

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128

1101 0011 1100 0001 0010 0100 1001 1111

| 0101 0101 0101 0101 0101 0101 0101 0101

11

Unsigned Arithmetics

```
unsigned int n = 45;
```

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128

1101 0011 1100 0001 0010 0100 1001 1111

| 0101 0101 0101 0101 0101 0101 0101 0101

111

Unsigned Arithmetics

```
unsigned int n = 45;
```

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128

1101 0011 1100 0001 0010 0100 1001

| 0101 0101 0101 0101 0101 0101 0101

1111

1111

```
unsigned int n = 45;
```

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128

	1101	0011	1100	0001	0010	0100	1001	1111
	0101	0101	0101	0101	0101	0101	0101	0101
								1 1111

Unsigned Arithmetics

```
unsigned int n = 45;
```

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128

1101 0011 1100 0001 0010 0100 1001 1111

| 0101 0101 0101 0101 0101 0101 0101 0101

01 1111

Unsigned Arithmetics

```
unsigned int n = 45;
```

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128

1101 0011 1100 0001 0010 0100 1001 1111

| 0101 0101 0101 0101 0101 0101 0101 0101

101 1111

Unsigned Arithmetics

```
unsigned int n = 45;
```

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128

1101 0011 1100 0001 0010 0100 1001 1111

| 0101 0101 0101 0101 0101 0101 0101 0101

1101 0111 1101 0101 0111 0101 1101 1111

Unsigned Arithmetics

```
unsigned int n = 45;
```

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128

~ 1101 0011 1100 0001 0010 0100 1001 1111

Unsigned Arithmetics

```
unsigned int n = 45;
```

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128

~ 1101 0011 1100 0001 0010 0100 1001 1111

0000

Unsigned Arithmetics

```
unsigned int n = 45;
```

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128

~ 1101 0011 1100 0001 0010 0100 1001 1111

0110 0000

Unsigned Arithmetics

```
unsigned int n = 45;
```

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128

~ 1101 0011 1100 0001 0010 0100 1001 1111

1011 0110 0000

Unsigned Arithmetics

```
unsigned int n = 45;
```

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128

~ 1101 0011 1100 0001 0010 0100 1001 1111

0010 1100 0011 1110 1101 1011 0110 0000

Unsigned Arithmetics

```
unsigned int n = 45;
```

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128

Exclusive-Or (XOR): 1 if the bits are different
0 if the bits are the same.

1101 0011 1100 0001 0010 0100 1001 1111

^

0101 0101 0101 0101 0101 0101 0101 0101

```
unsigned int n = 45;
```

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128

	1101	0011	1100	0001	0010	0100	1001	1111
^	0101	0101	0101	0101	0101	0101	0101	0101
								0

```
unsigned int n = 45;
```

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128

[illegible]

```
unsigned int n = 45;
```

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128

	1101	0011	1100	0001	0010	0100	1001	1111
^	0101	0101	0101	0101	0101	0101	0101	0101
								010

Unsigned Arithmetics

```
unsigned int n = 45;
```

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128

1101 0011 1100 0001 0010 0100 1001 1111

^ 0101 0101 0101 0101 0101 0101 0101 0101

1010

Unsigned Arithmetics

```
unsigned int n = 45;
```

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128

1101 0011 1100 0001 0010 0100 1001 1111

^

0101 0101 0101 0101 0101 0101 0101 0101

0

1010

```
unsigned int n = 45;
```

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128

	1101	0011	1100	0001	0010	0100	1001	1111
^	0101	0101	0101	0101	0101	0101	0101	0101
							10	1010

Unsigned Arithmetics

```
unsigned int n = 45;
```

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128

1101 0011 1100 0001 0010 0100 1001 1111

^

0101 0101 0101 0101 0101 0101 0101 0101

100 1010

Unsigned Arithmetics

```
unsigned int n = 45;
```

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128

1101 0011 1100 0001 0010 0100 1001 1111

^ 0101 0101 0101 0101 0101 0101 0101 0101

1000 0110 1001 0100 0111 0001 1100 1010

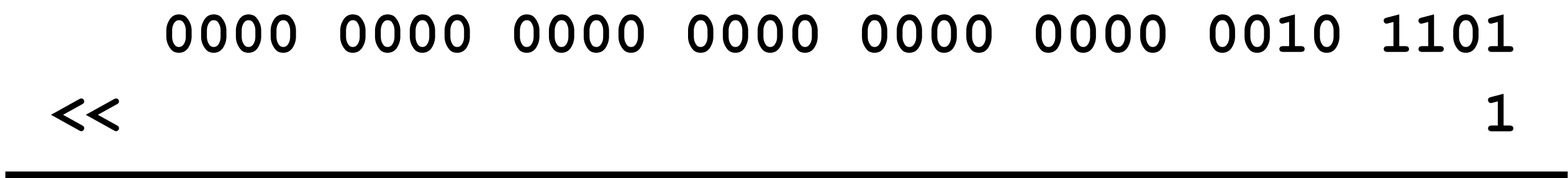
Unsigned Arithmetics

- $\sim$ is not the same as $!$, $\&$ is not the same as $\&\&$, $|$ is not the same as $||$
 - $!$, $\&\&$, $||$ are *Boolean* operators -- they treat zero as false and non-zero as true
 - $\sim$, $\&$, $|$ are *bitwise* operators -- they operate on each bit.

Unsigned Arithmetics

```
unsigned int n = 45;
```

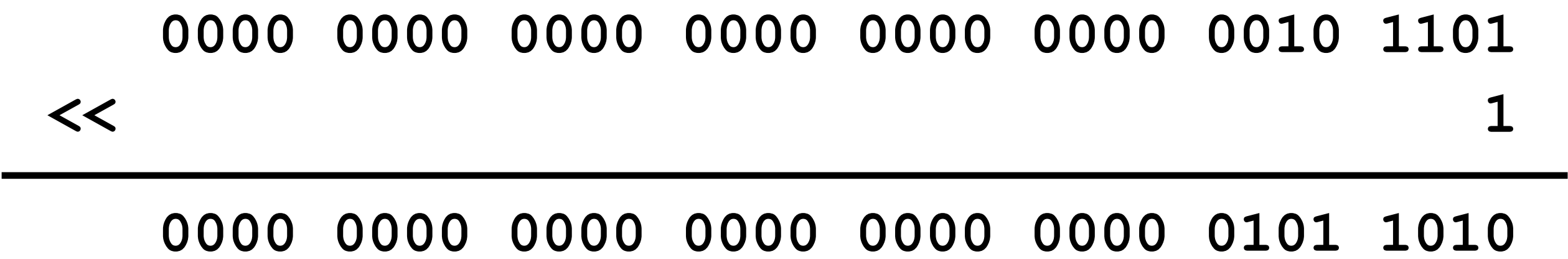
n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128



Unsigned Arithmetics

```
unsigned int n = 45;
```

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128



Unsigned Arithmetics

```
unsigned int n = 45;
```

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128

0000 0000 0000 0000 0000 0000 0010 1101

<<1

0000 0000 0000 0000 0000 0000 0101 1010 = 0x5A

Unsigned Arithmetics

```
unsigned int n = 45;
```

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128

0000 0000 0000 0000 0000 0000 0010 1101

<<1

0000 0000 0000 0000 0000 0000 0101 1010

= 0x5A = 90

Unsigned Arithmetics

```
unsigned int n = 45;
```

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128

0000 0000 0000 0000 0000 0000 0010 1101

<<1

0000 0000 0000 0000 0000 0000 0101 1010

<<16

= 0x5A = 90

Unsigned Arithmetics

```
unsigned int n = 45;
```

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128

0000 0000 0000 0000 0000 0000 0010 1101

<<

1

0000 0000 0000 0000 0000 0000 0101 1010

<<

16

0000 0000 0101 1010 0000 0000 0000 0000

= 0x5A = 90

Unsigned Arithmetics

```
unsigned int n = 45;
```

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128

0000 0000 0000 0000 0000 0000 0010 1101

<<

1

0000 0000 0000 0000 0000 0000 0101 1010

<<

16

0000 0000 0101 1010 0000 0000 0000 0000

<<

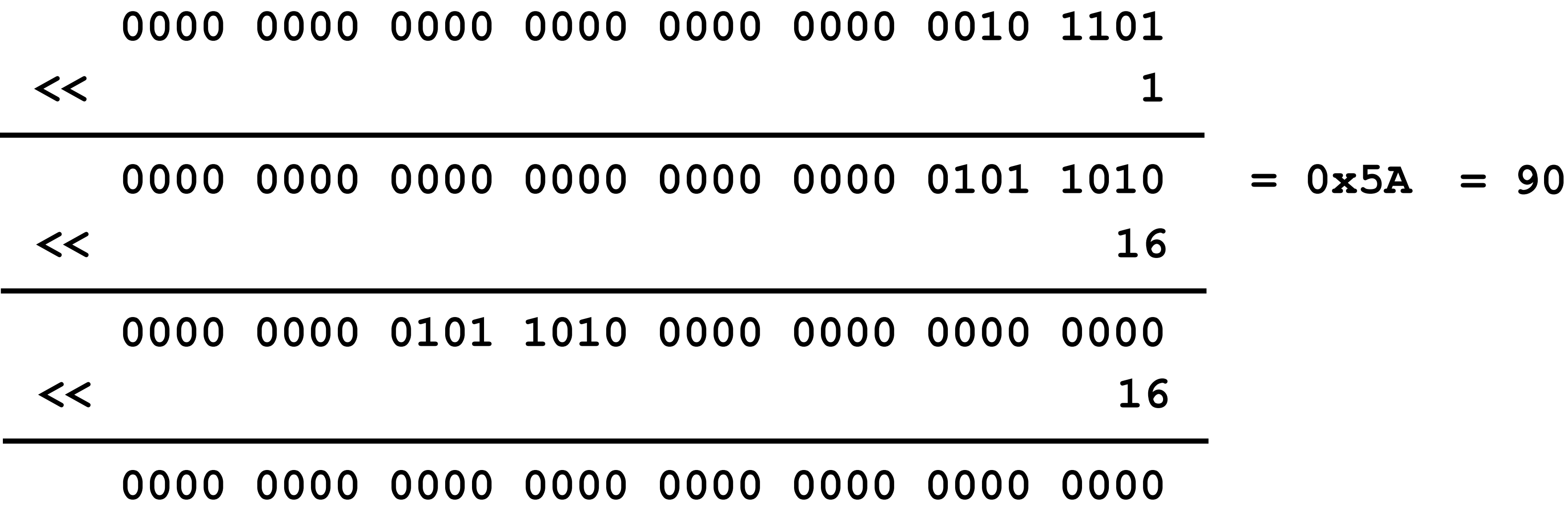
16

= 0x5A = 90

Unsigned Arithmetics

```
unsigned int n = 45;
```

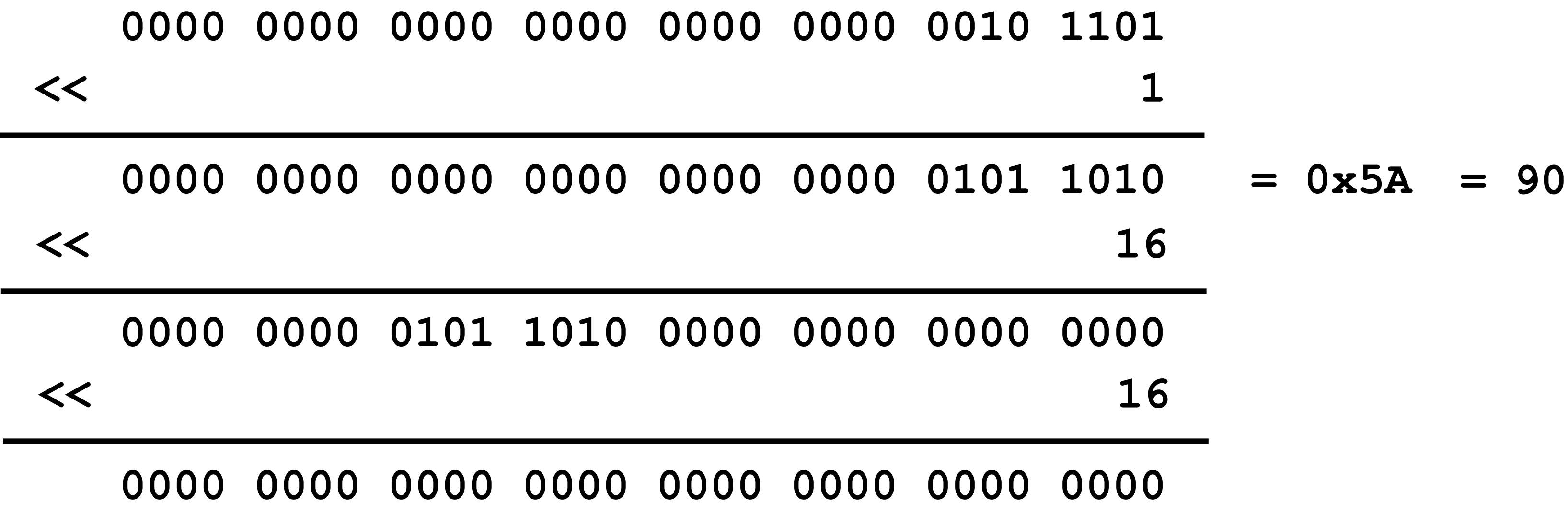
n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128



Unsigned Arithmetics

```
unsigned int n = 45;
```

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128

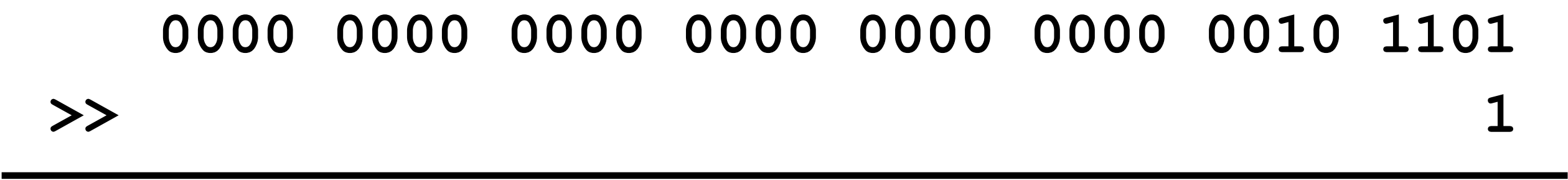


- When shift overflows, the bit is lost.

Unsigned Arithmetics

```
unsigned int n = 45;
```

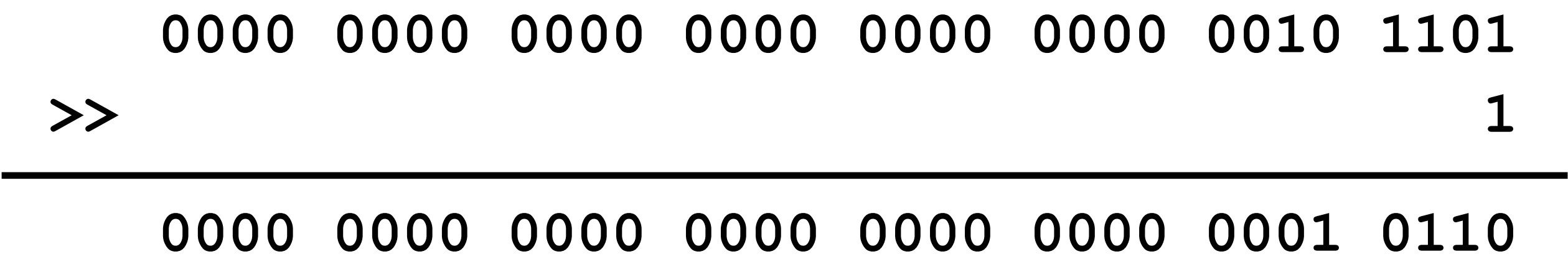
n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128



Unsigned Arithmetics

```
unsigned int n = 45;
```

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128



Unsigned Arithmetics

```
unsigned int n = 45;
```

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128

0000 0000 0000 0000 0000 0000 0010 1101

>>1

0000 0000 0000 0000 0000 0000 0001 0110

= 0x16

Unsigned Arithmetics

```
unsigned int n = 45;
```

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128

0000 0000 0000 0000 0000 0000 0010 1101

>> 1

0000 0000 0000 0000 0000 0000 0001 0110

= 0x16 = 22

Unsigned Arithmetics

```
unsigned int n = 45;
```

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128

0000 0000 0000 0000 0000 0000 0010 1101

>>

1

0000 0000 0000 0000 0000 0000 0001 0110

>>

8

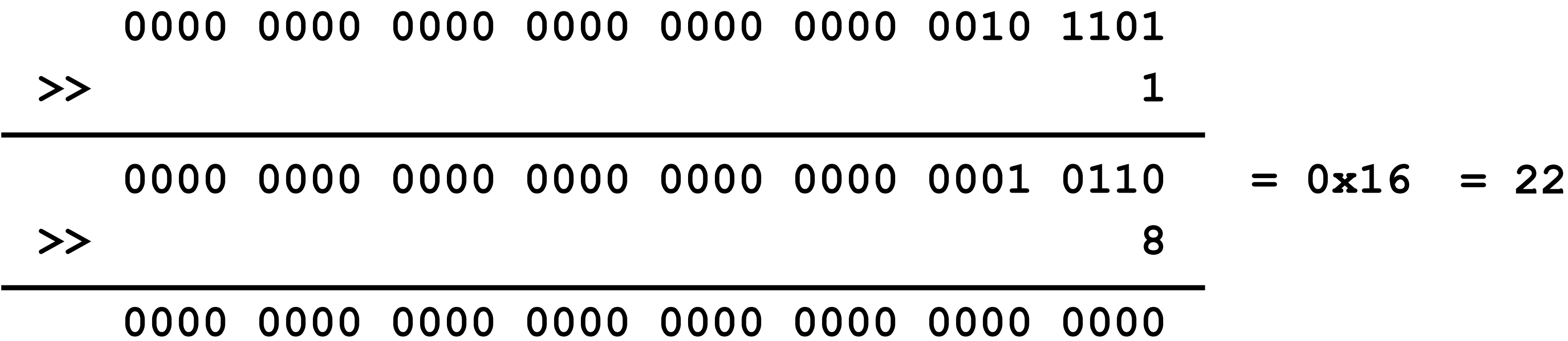
0000 0000 0000 0000 0000 0000 0000 0000

= 0x16 = 22

Unsigned Arithmetics

```
unsigned int n = 45;
```

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128



- When shift underflows, the bit is lost.

Bit-packing

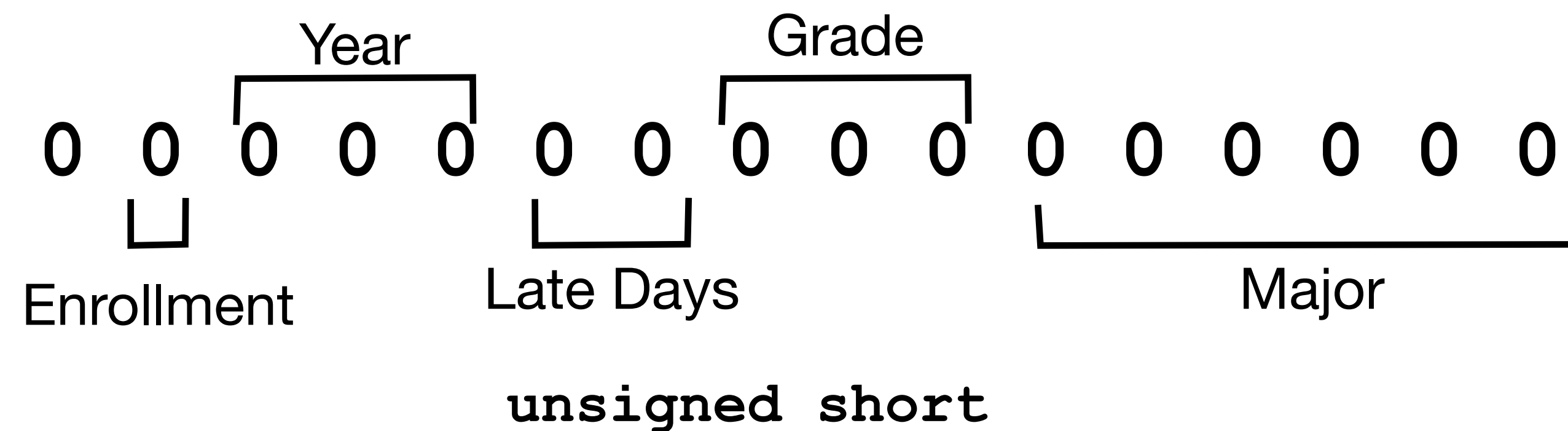
```
struct student {  
    int is_enrolled;    /* is student formally enrolled in this class? */  
    int year;  
    int late_days_used;  
    char letter_grade;  
    char major[32];  
};
```

Bit-packing

- Enrollment: 2 choices (1 bit)
- Year: 5 choices [1, 2, 3, 4, beyond-4] (3 bits)
- Late Days: 4 choices (2 bits)
- Letter grade: 5 choices (3 bits)
- Major: 53 majors (6 bits)
- None of these need 32 bit integers!
- $1 + 3 + 2 + 3 + 6 = 15$, in fact we can fit the entire record inside a short.

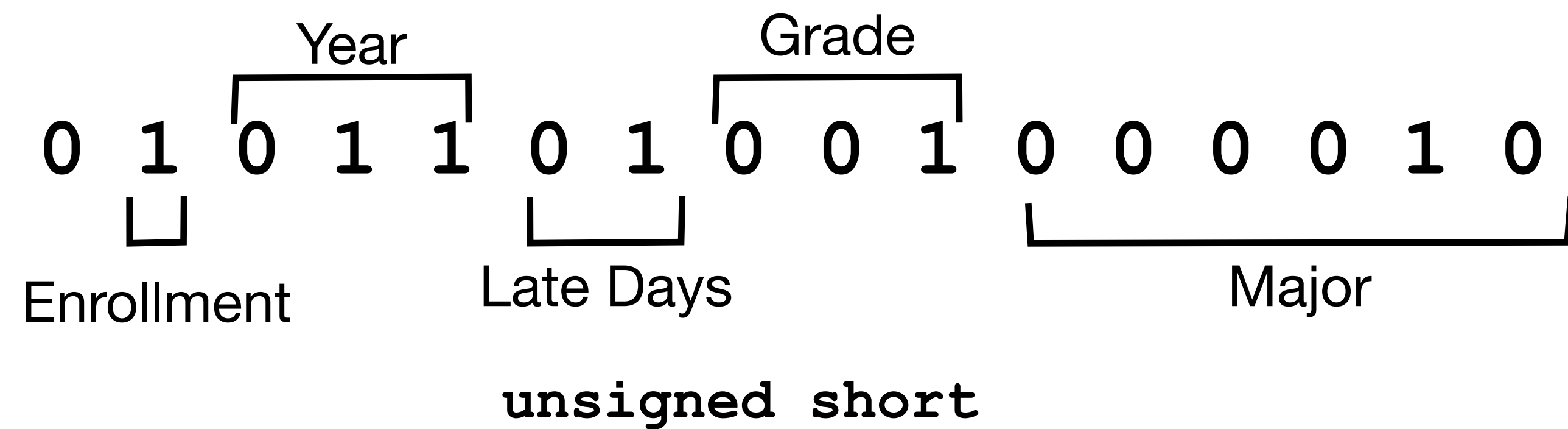
```
struct student {  
    int is_enrolled;  
    int year;  
    int late_days_used;  
    char letter_grade;  
    char major[32];  
};
```

Bit-packing



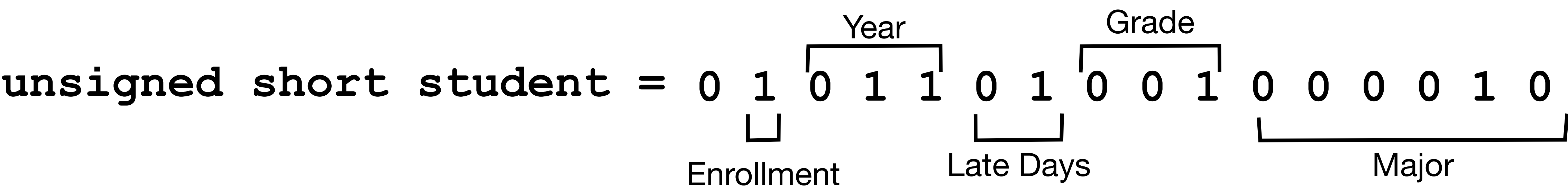
- Enrollment: 2 choices (1 bit)
- Year: 6 choices [0, 1, 2, 3, 4, beyond-4] (3 bits)
- Late Days: 4 choices (2 bits)
- Letter grade: 5 choices (3 bits)
- Major: 53 majors (6 bits)

Bit-packing



- Enrolled
- Year 3
- 1 late day
- 1st choice of grade
- 2nd choice of major

Bit-packing



Bit-packing

`unsigned short student =` 0 1 0 1 1 0 1 0 0 1 0 0 0 0 1 0

The diagram illustrates the bit fields for the `student` variable. The 16 bits are grouped as follows:

- Enrollment:** Bits 1 and 2 (01)
- Year:** Bits 3, 4, and 5 (011)
- Late Days:** Bits 6 and 7 (01)
- Grade:** Bits 8, 9, and 10 (001)
- Major:** Bits 11, 12, 13, 14, and 15 (00001)

`unsigned short mask =`

Bit-packing

unsigned short student = 0 1 0 1 1 0 1 0 0 1 0 0 0 0 1 0

Enrollment Late Days Year Grade Major

unsigned short mask = 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0

Bit-packing

`unsigned short student =` 0 1 0 1 1 0 1 0 0 1 0 0 0 0 1 0

Enrollment Late Days Year Grade Major

`unsigned short mask =` 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0

```
int year = mask & student;
```

Bit-packing

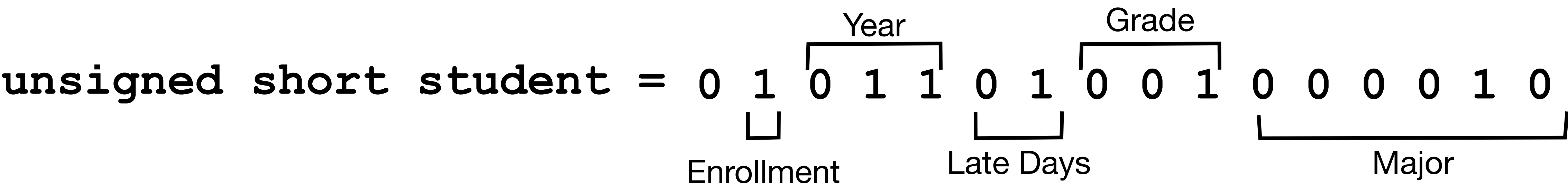
unsigned short student = 0 1 0 1 1 0 1 0 0 1 0 0 0 0 1 0

Enrollment Late Days Year Grade Major

unsigned short mask = 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0

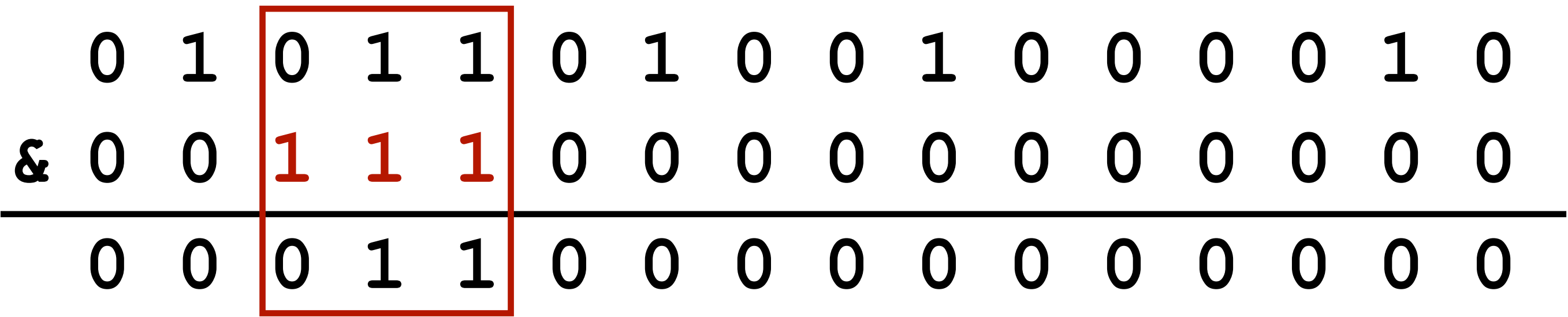
[illegible]

Bit-packing



`unsigned short mask =` 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0

```
int year = mask & student;
```



```
year = year >> 11;
```

Bit-packing

unsigned short student = 0 1 0 1 1 0 1 0 0 1 0 0 0 0 1 0

Enrollment Late Days Year Grade Major

unsigned short mask = 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0

Diagram illustrating the bit manipulation process for the code:

```
int year = mask & student;
```

The diagram shows the bitwise AND operation between the `mask` and `student` variables. The `mask` is represented by the first row of bits (0 1 0 1 1 0 1 0 0 1 0 0 0 0 1 0), and the `student` is represented by the second row of bits (0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0). The result of the AND operation is shown in the third row (0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0). The bits 1 1 1 in the `student` row and 0 1 1 in the result row are highlighted with red boxes.

```
year = year >> 11;
```

The diagram shows the right shift operation. The result from the previous step (0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0) is shifted right by 11 positions, resulting in the final value (0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0). The bits 1 1 in the final value are highlighted with a red box.

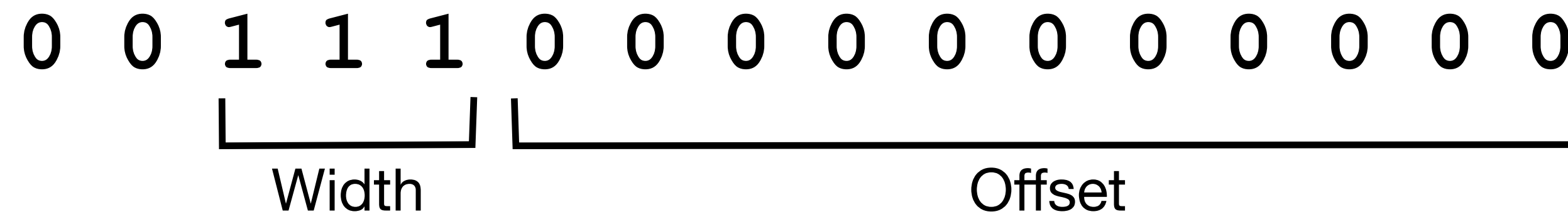
Bit-packing

0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0

```
unsigned short mask = 14336;
```

```
unsigned short mask = 0x7 << 11;
```

Bit-packing



```
unsigned short mask = 14336;
```

```
unsigned short mask = 0x7 << 11;
```

```
unsigned short mask = ~0u >> (16 - width) << offset;
```

Future Note: we need to write 0u instead of 0 because by default, 0 is signed, so >> is going to perform a signed right shift.

Bit-packing

0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0

└───┬──────────────────────────────────┘

Width Offset

```
unsigned short mask = ~0 >> (16 - width) << offset;
```

~ 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Bit-packing

0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0

└───┬──────────────────────────┘

Width Offset

```
unsigned short mask = ~0 >> (16 - width) << offset;
```

~ 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

Bit-packing

0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0

└───┬──────────────────────────┘

Width Offset

```
unsigned short mask = ~0 >> (16 - width) << offset;
```

~ 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

>> 13

Bit-packing

0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0

└───┬──────────────────────────┘

Width Offset

```
unsigned short mask = ~0 >> (16 - width) << offset;
```

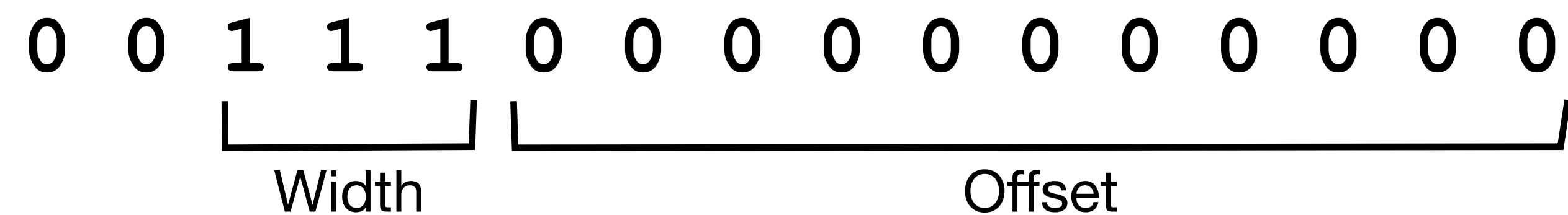
~ 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

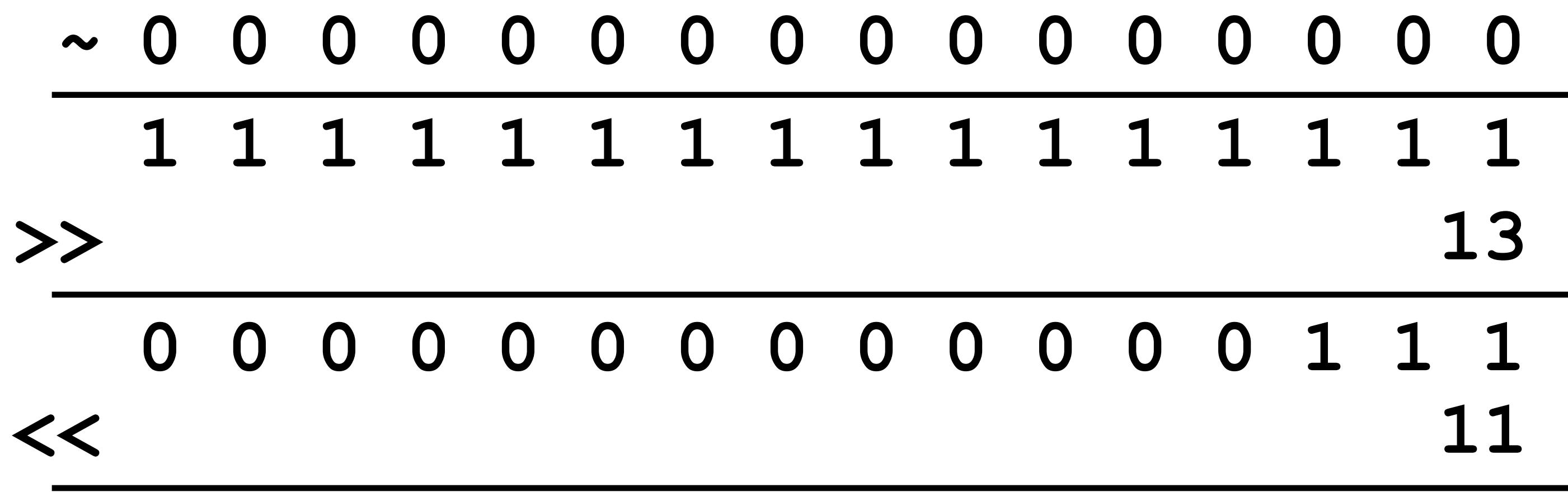
>> 13

0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1

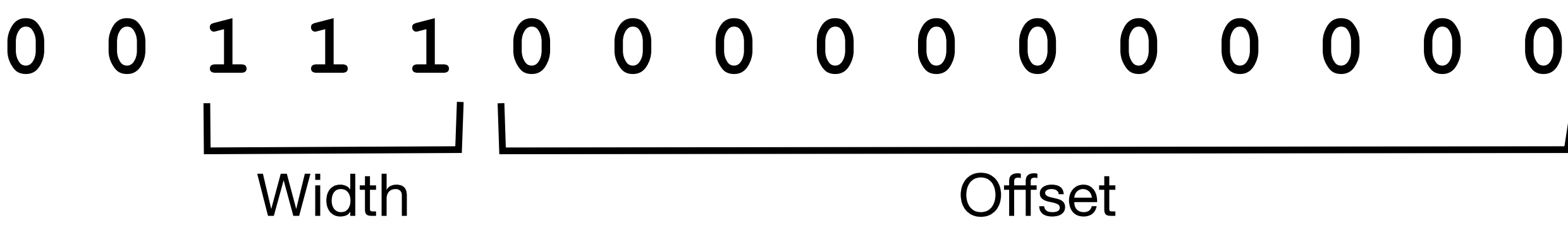
Bit-packing



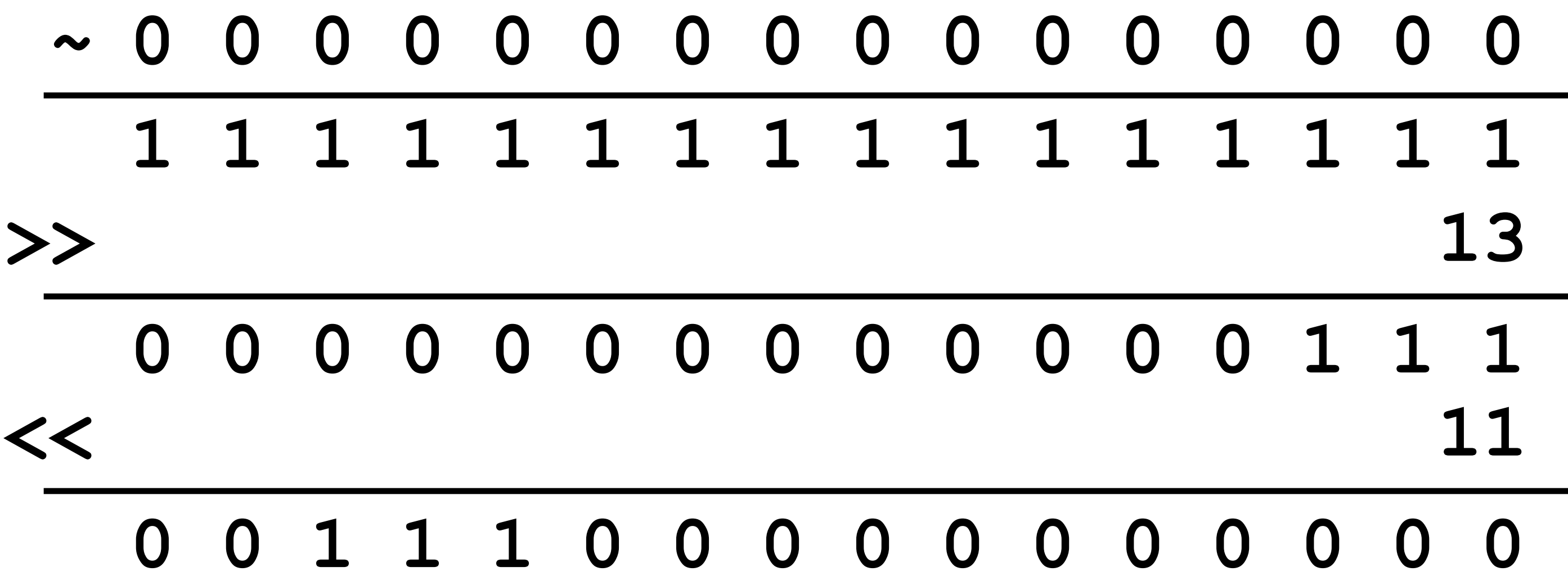
```
unsigned short mask = ~0 >> (16 - width) << offset;
```



Bit-packing



```
unsigned short mask = ~0 >> (16 - width) << offset;
```



Bit-packing

```
unsigned short student = ...;  
unsigned short year_mask = ~0 >> (16 - 3) << 11;  
  
unsigned int year = (student & year_mask) >> 11;
```

Bit-packing

`unsigned short student =` 0 1 0 1 1 0 1 0 0 1 0 0 0 0 1 0

Enrollment Year Late Days Grade Major

`unsigned short mask =` 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0

`unsigned short year =` 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0

Bit-packing

`unsigned short student =` 0 1 0 1 1 0 1 0 0 1 0 0 0 0 1 0

Enrollment Year Late Days Grade Major

`unsigned short mask =` 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0

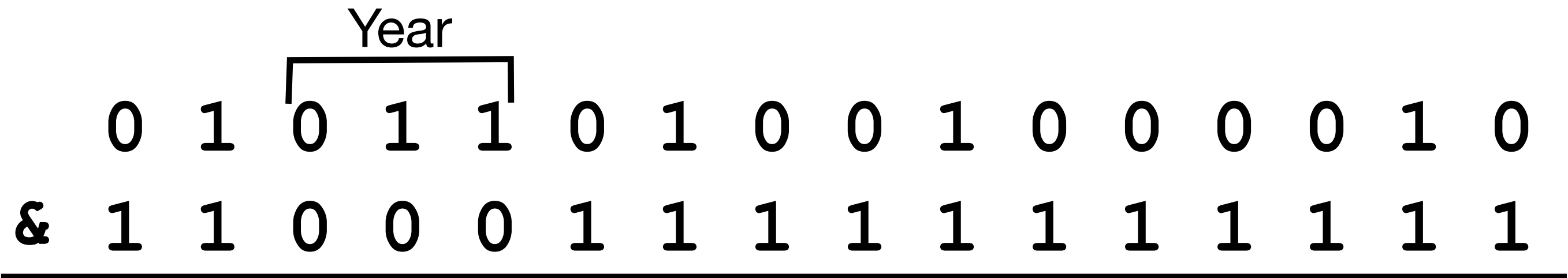
`unsigned short year =` 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0

`year = year << 11;` 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0

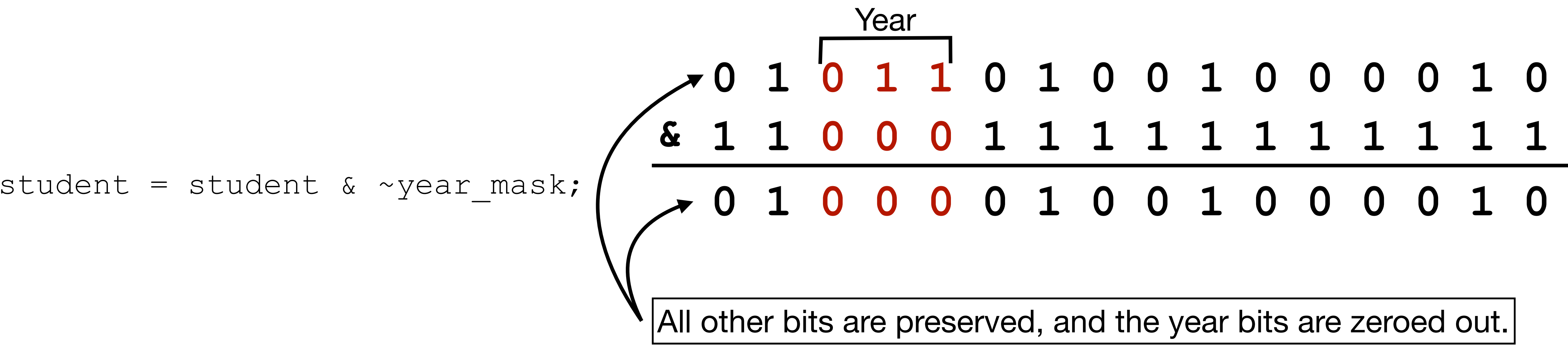
??

Bit-packing

```
student = student & ~year_mask;
```



Bit-packing



Bit-packing

```
student = student & ~year_mask;
```

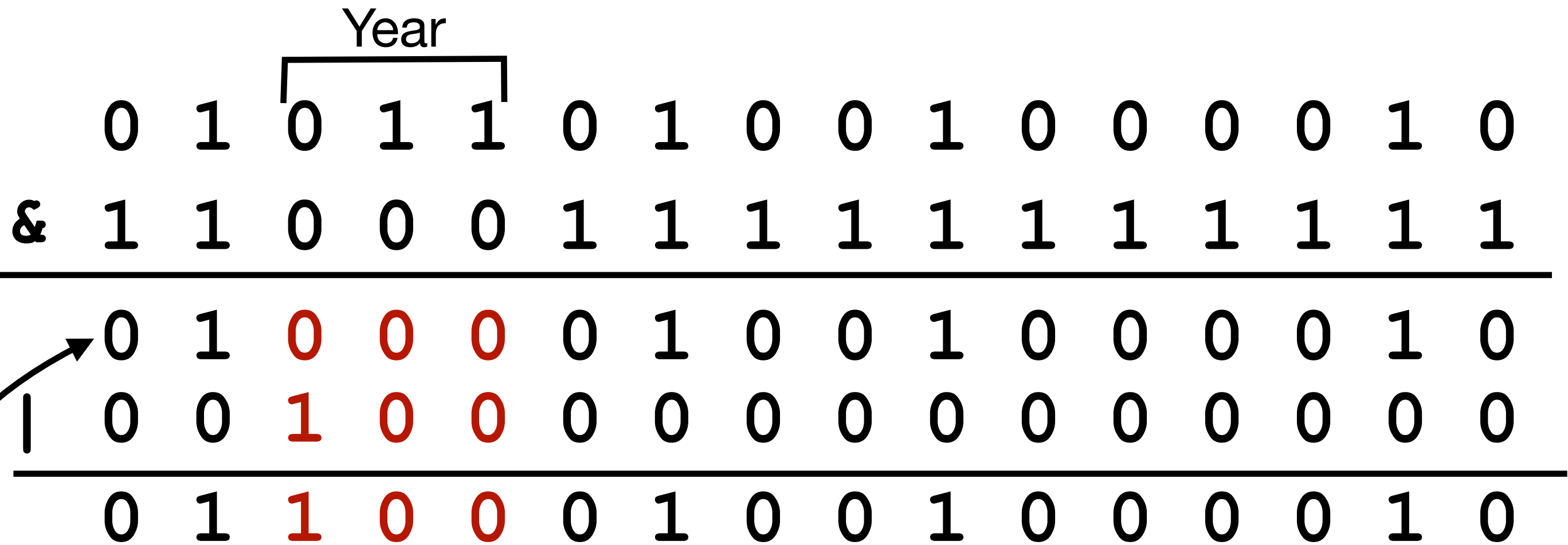
```
student = student | (year << 11);
```

[illegible]

Bit-packing

```
student = student & ~year_mask;
```

```
student = student | (year << 11);
```



All other bits are preserved, and the year bits are updated.

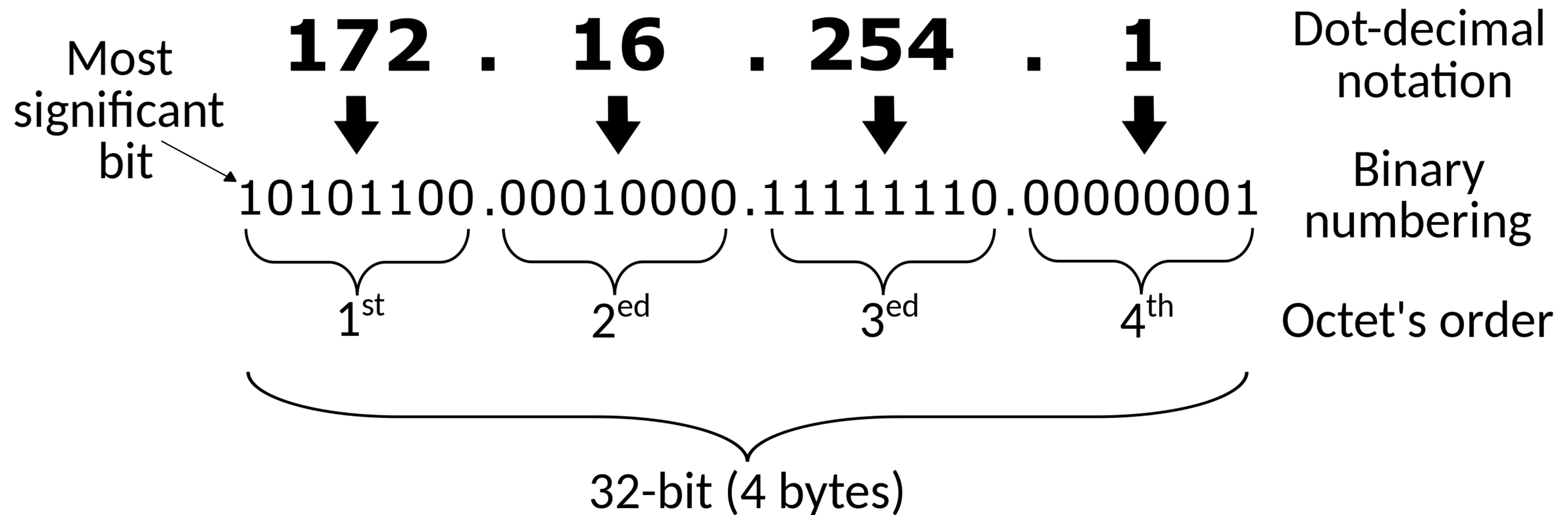
Bit-packing

Summary

- A mask has the bit fields of interest set 1 and all others 0.
- `& mask` gets the bits
- `& ~mask` clears out the bits
- `~0u >> (16 - width) << offset` creates a 16-bit mask with `width` and `offset`.
- After the bits are zeroed out, `|` is used to write new bits in the field.
- Does anyone use this irl?

Bit-packing

Example: IP Address



Bit-packing

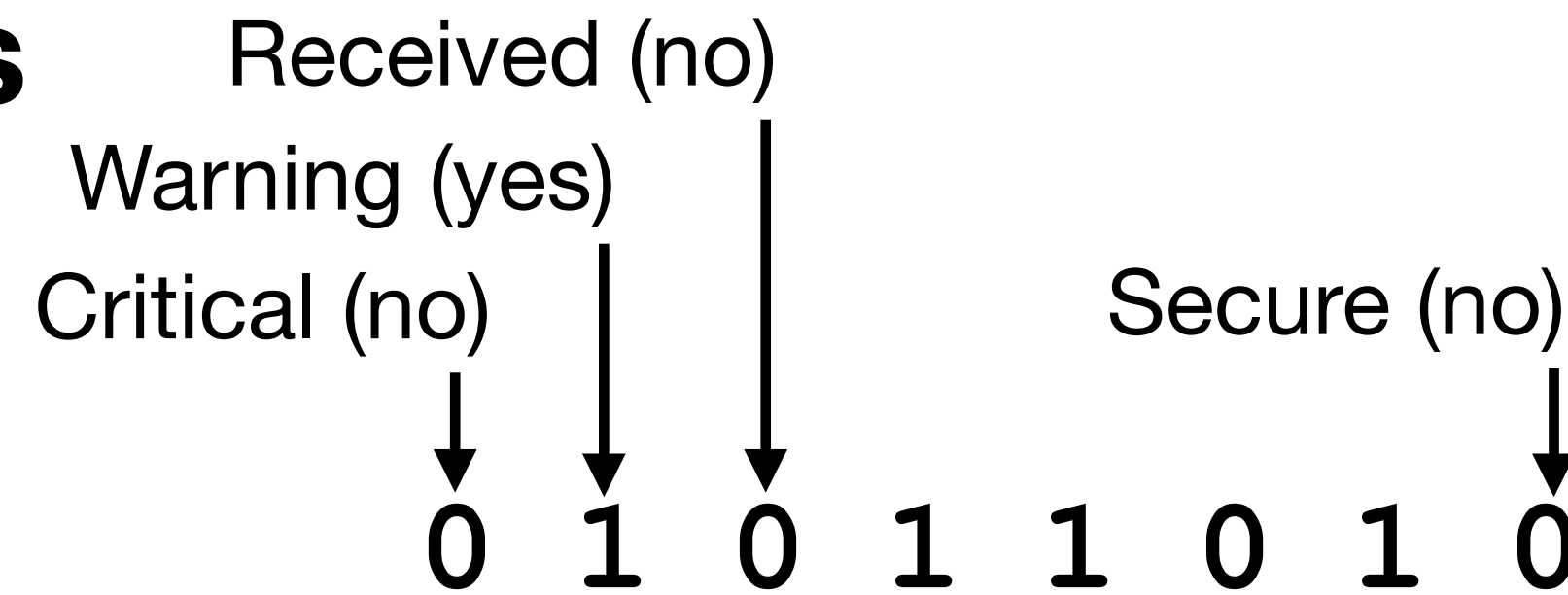
Example: IP Address

IP address	192.168.1.2
Subnet mask	255.255.255.0
Router	192.168.1.1

What is this mask trying to get?

Bit-packing

Example: Flag Bits

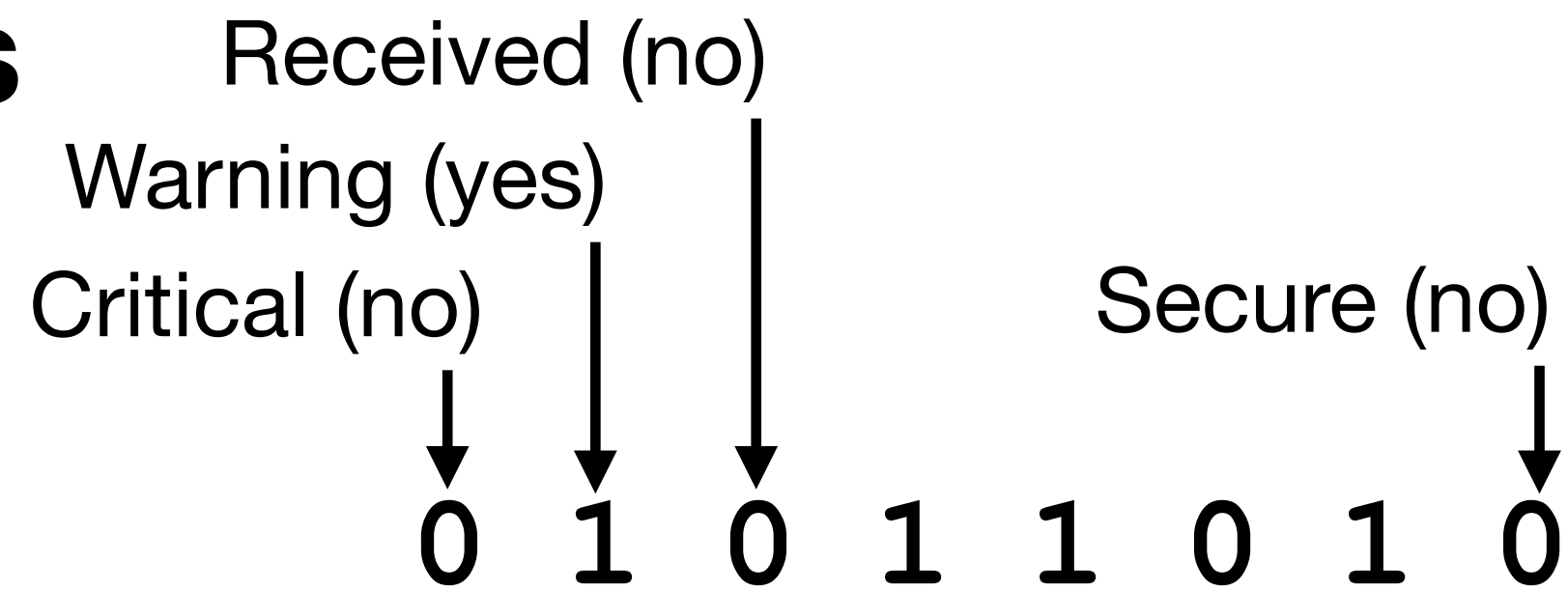


Each bit answers one yes/no question -- 8 Booleans in a byte!

```
const uint8_t CRITICAL = 0x80;    -> 100000000
const uint8_t WARNING  = 0x40;    -> 010000000
const uint8_t RECEIVED = 0x20;    -> 001000000
...
const uint8_t SECURE   = 0x01;    -> 000000001
```

Bit-packing

Example: Flag Bits



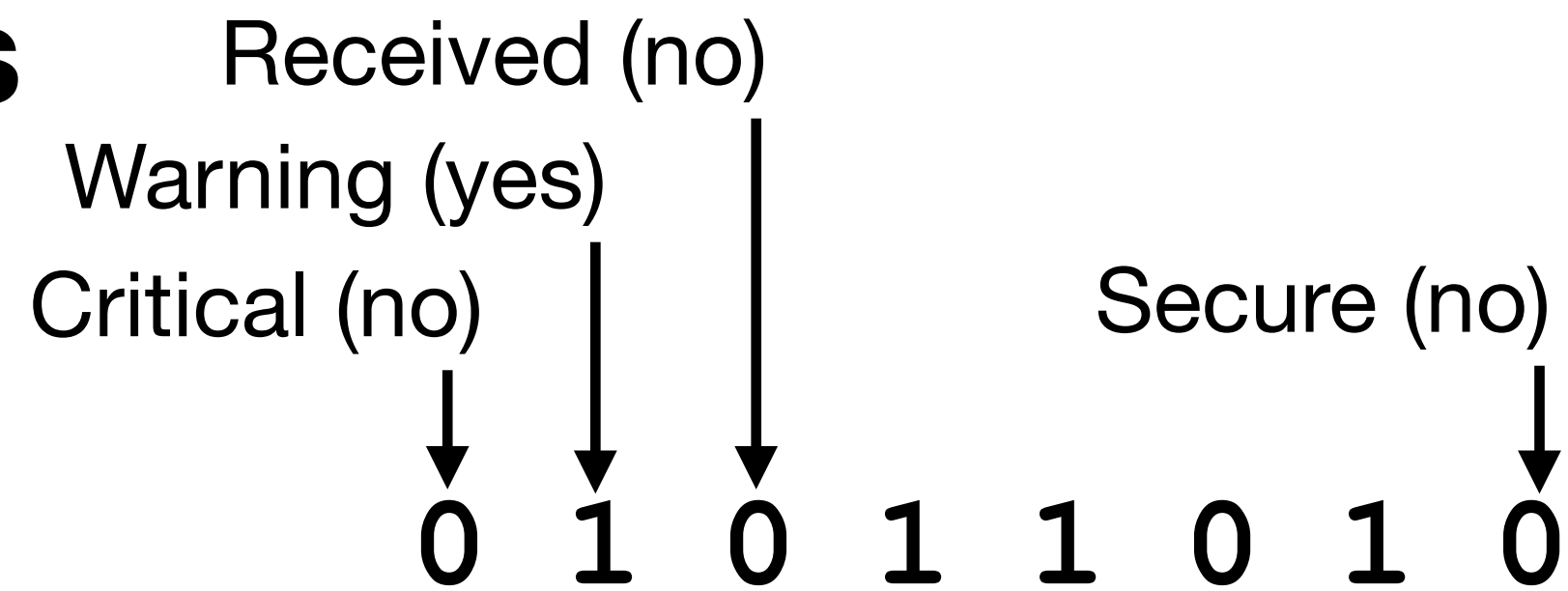
```
const uint8_t CRITICAL = 0x80;    -> 10000000
const uint8_t WARNING  = 0x40;    -> 01000000
const uint8_t RECEIVED = 0x20;    -> 00100000
...
const uint8_t SECURE   = 0x01;    -> 00000001
```

```
if (flags & CRITICAL) { ... }
```

This is non-zero if and only if the first bit is set.

Bit-packing

Example: Flag Bits



```
const uint8_t CRITICAL = 0x80;    -> 100000000
const uint8_t WARNING  = 0x40;    -> 010000000
const uint8_t RECEIVED = 0x20;    -> 001000000
...
const uint8_t SECURE   = 0x01;    -> 000000001
```

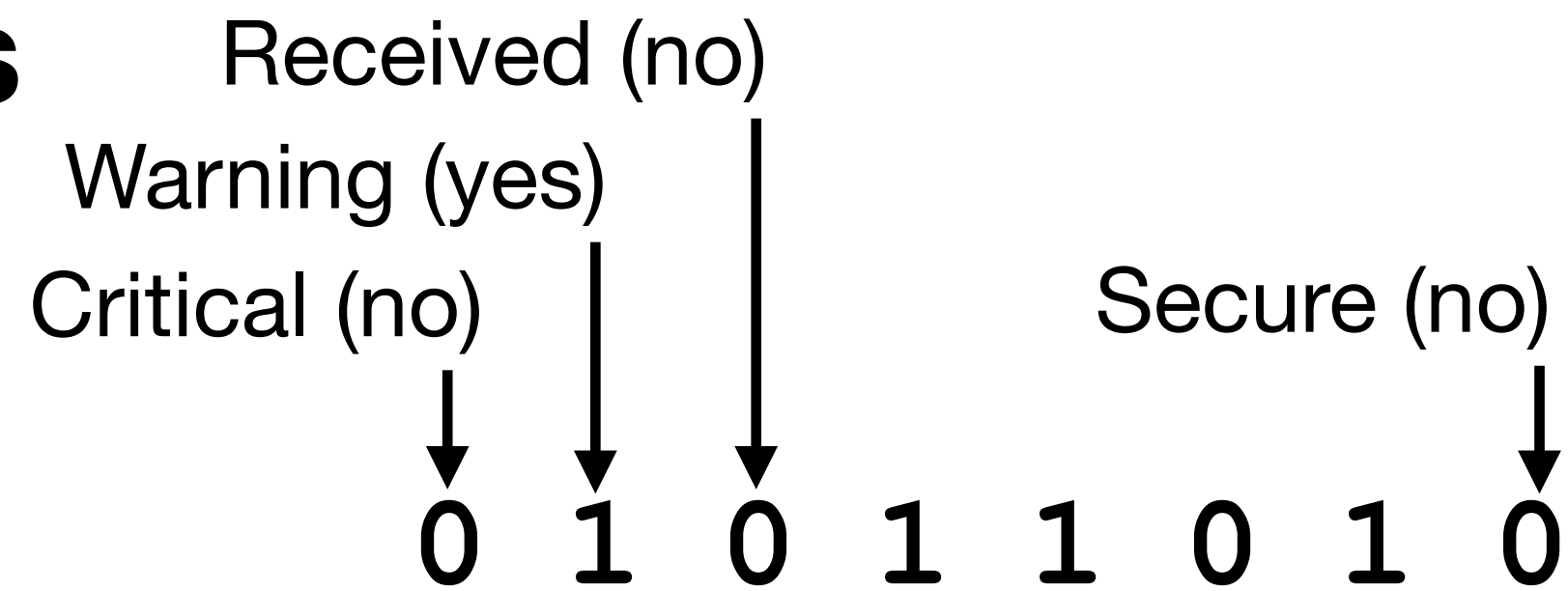
```
if (flags & (CRITICAL | SECURE)) { ... }
```

10000001

This is non-zero if either the first bit or the last bit is set.

Bit-packing

Example: Flag Bits



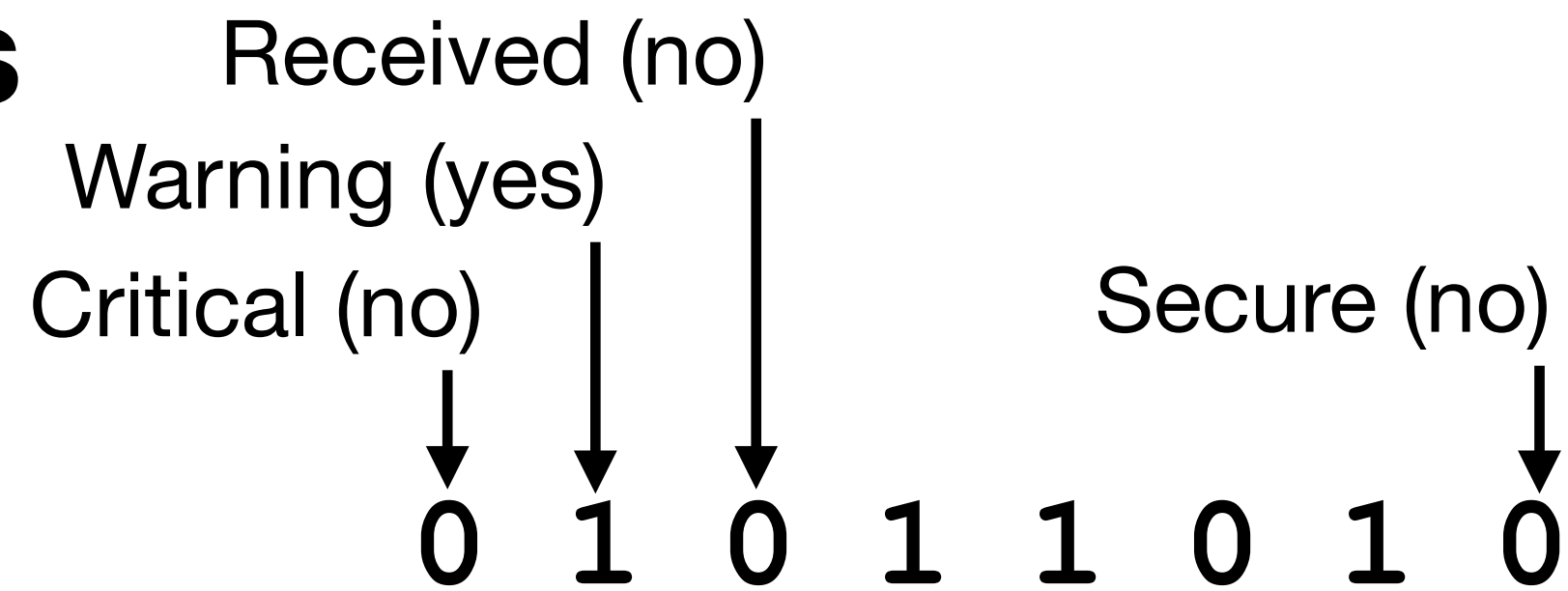
```
const uint8_t CRITICAL = 0x80;    -> 100000000
const uint8_t WARNING  = 0x40;    -> 010000000
const uint8_t RECEIVED = 0x20;    -> 001000000
...
const uint8_t SECURE   = 0x01;    -> 000000001
```

```
flags |= RECEIVED;
```

This sets the RECEIVED bit.

Bit-packing

Example: Flag Bits



```
const uint8_t CRITICAL = 0x80;  -> 100000000
const uint8_t WARNING  = 0x40;  -> 010000000
const uint8_t RECEIVED = 0x20;  -> 001000000
...
const uint8_t SECURE   = 0x01;  -> 000000001
```

```
flags &= ~SECURE;
```

This unsets the SECURE bit.

Interlude: Command-Line Arguments

and ++, --