Introduction to Programming

Week – *11*, Lecture – *1*

Assorted Topics in C – Part 1

SAURABH SRIVASTAVA

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

IIT KANPUR

Recap – Boolean Algebra

The basics of Boolean Algebra can be summarized as follows:

Recap – Boolean Algebra

The basics of Boolean Algebra can be summarized as follows:

- The variables in Boolean Algebra can only take one of the two values – 0 or 1

Recap – Boolean Algebra

The basics of Boolean Algebra can be summarized as follows:

- The variables in Boolean Algebra can only take one of the two values – 0 or 1
- The *AND* function is defined over two or more Boolean variables as
  - `The AND function has a value 1, if and only if, all variables have the value 1`
  - `Otherwise, the function has a value 0`

Recap – Boolean Algebra

The basics of Boolean Algebra can be summarized as follows:

- The variables in Boolean Algebra can only take one of the two values – 0 or 1
- The *AND* function is defined over two or more Boolean variables as
  - `The AND function has a value 1, if and only if, all variables have the value 1`
  - `Otherwise, the function has a value 0`
- The *OR* function is defined over two or more Boolean variables as
  - `The OR function has a value 0, if and only if, all variables have the value 0`
  - `Otherwise, the function has a value 1`

Recap – Boolean Algebra

The basics of Boolean Algebra can be summarized as follows:

- The variables in Boolean Algebra can only take one of the two values – 0 or 1
- The *AND* function is defined over two or more Boolean variables as
  - The AND function has a value 1, if and only if, all variables have the value 1
  - Otherwise, the function has a value 0
- The *OR* function is defined over two or more Boolean variables as
  - The OR function has a value 0, if and only if, all variables have the value 0
  - Otherwise, the function has a value 1
- The *NOT* function is defined over one Boolean variable as
  - The NOT function has a value 1, if the variable has the value 0
  - Otherwise, the function has a value 0 (i.e., when the variable has the value 1)

Recap – Logical Operators in C

Logical operators performed operations over expressions

Recap – Logical Operators in C

Logical operators performed operations over expressions

The Logical AND operator (`&&`) performs *ANDing* of two *C expressions*

Recap – Logical Operators in C

Logical operators performed operations over expressions

The Logical AND operator (`&&`) performs *ANDing* of two *C expressions*

- For two expressions, `e1` and `e2`, `(e1 && e2)` is `1`, if and only if both expressions are `1`…
- … otherwise, it is `0`

Recap – Logical Operators in C

Logical operators performed operations over expressions

The Logical AND operator (`&&`) performs *ANDing* of two *C expressions*

- For two expressions, `e1` and `e2`, `(e1 && e2)` is `1`, if and only if both expressions are `1`…
- … otherwise, it is `0`

The Logical OR operator (`||`) performs *ORing* of two *C expressions*

Recap – Logical Operators in C

Logical operators performed operations over expressions

The Logical AND operator (`&&`) performs *ANDing* of two *C expressions*

- For two expressions, `e1` and `e2`, `(e1 && e2)` is `1`, if and only if both expressions are `1`…
- … otherwise, it is `0`

The Logical OR operator (`||`) performs *ORing* of two *C expressions*

- For two expressions, `e1` and `e2`, `(e1 || e2)` is `0`, if and only if both expressions are `0`…
- … otherwise, it is `1`

Recap – Logical Operators in C

Logical operators performed operations over expressions

The Logical AND operator (`&&`) performs *ANDing* of two *C expressions*

- For two expressions, `e1` and `e2`, `(e1 && e2)` is `1`, if and only if both expressions are `1`…
- … otherwise, it is `0`

The Logical OR operator (`||`) performs *ORing* of two *C expressions*

- For two expressions, `e1` and `e2`, `(e1 || e2)` is `0`, if and only if both expressions are `0`…
- … otherwise, it is `1`

The Logical NOT operator (`!`) applies the NOT operation or *inversion* of a *C expression*

Recap – Logical Operators in C

Logical operators performed operations over expressions

The Logical AND operator (`&&`) performs *ANDing* of two *C expressions*

- For two expressions, `e1` and `e2`, `(e1 && e2)` is `1`, if and only if both expressions are `1`…
- … otherwise, it is `0`

The Logical OR operator (`||`) performs *ORing* of two *C expressions*

- For two expressions, `e1` and `e2`, `(e1 || e2)` is `0`, if and only if both expressions are `0`…
- … otherwise, it is `1`

The Logical NOT operator (`!`) applies the NOT operation or *inversion* of a *C expression*

- For an expression, `e`, (`!e`) is `0`, if and only if, it evaluates to a non-zero value…
- … otherwise, it is `1` (i.e. when `e` evaluates to zero)

Bitwise Operators in C (1/2)

The Logical operators work over expressions and they produce `0` or `1` as output

Bitwise Operators in C (1/2)

The Logical operators work over expressions and they produce `0` or `1` as output

The Bitwise operators work over integer operands and produce another integer as output…

- … which need not necessarily be `0` or `1`

Bitwise Operators in C (1/2)

The Logical operators work over expressions and they produce `0` or `1` as output

The Bitwise operators work over integer operands and produce another integer as output…

- … which need not necessarily be `0` or `1`
- These operators are applicable to integer operands only

Bitwise Operators in C (1/2)

The Logical operators work over expressions and they produce `0` or `1` as output

The Bitwise operators work over integer operands and produce another integer as output…

- … which need not necessarily be `0` or `1`
- These operators are applicable to integer operands only
- Also, it is **not recommended** to use them with negative integers (as the output may be unpredictable)

Bitwise Operators in C (1/2)

The Logical operators work over expressions and they produce `0` or `1` as output

The Bitwise operators work over integer operands and produce another integer as output…

- … which need not necessarily be `0` or `1`
- These operators are applicable to integer operands only
- Also, it is **not recommended** to use them with negative integers (as the output may be unpredictable)

The Bitwise operators, as the term suggests, work at individual bit level, i.e. bit-wise

- The result for a particular bit is not affected by the results of any other bits

Bitwise Operators in C (1/2)

The Logical operators work over expressions and they produce `0` or `1` as output

The Bitwise operators work over integer operands and produce another integer as output…

- … which need not necessarily be `0` or `1`
- These operators are applicable to integer operands only
- Also, it is **not recommended** to use them with negative integers (as the output may be unpredictable)

The Bitwise operators, as the term suggests, work at individual bit level, i.e. bit-wise

- The result for a particular bit is not affected by the results of any other bits

The three Logical operators have their corresponding Bitwise operators too…

- … i.e. Bitwise AND (`&`), Bitwise OR (`|`) and Bitwise NOT (`~`)

Bitwise Operators in C (1/2)

The Logical operators work over expressions and they produce `0` or `1` as output

The Bitwise operators work over integer operands and produce another integer as output…

- … which need not necessarily be `0` or `1`
- These operators are applicable to integer operands only
- Also, it is **not recommended** to use them with negative integers (as the output may be unpredictable)

The Bitwise operators, as the term suggests, work at individual bit level, i.e. bit-wise

- The result for a particular bit is not affected by the results of any other bits

The three Logical operators have their corresponding Bitwise operators too…

- … i.e. Bitwise AND (`&`), Bitwise OR (`|`) and Bitwise NOT (`~`)

The XOR operator (`^`) provides the implementation of another Boolean function with the same name

Bitwise Operators in C (1/2)

The Logical operators work over expressions and they produce `0` or `1` as output

The Bitwise operators work over integer operands and produce another integer as output…

- … which need not necessarily be `0` or `1`
- These operators are applicable to integer operands only
- Also, it is **not recommended** to use them with negative integers (as the output may be unpredictable)

The Bitwise operators, as the term suggests, work at individual bit level, i.e. bit-wise

- The result for a particular bit is not affected by the results of any other bits

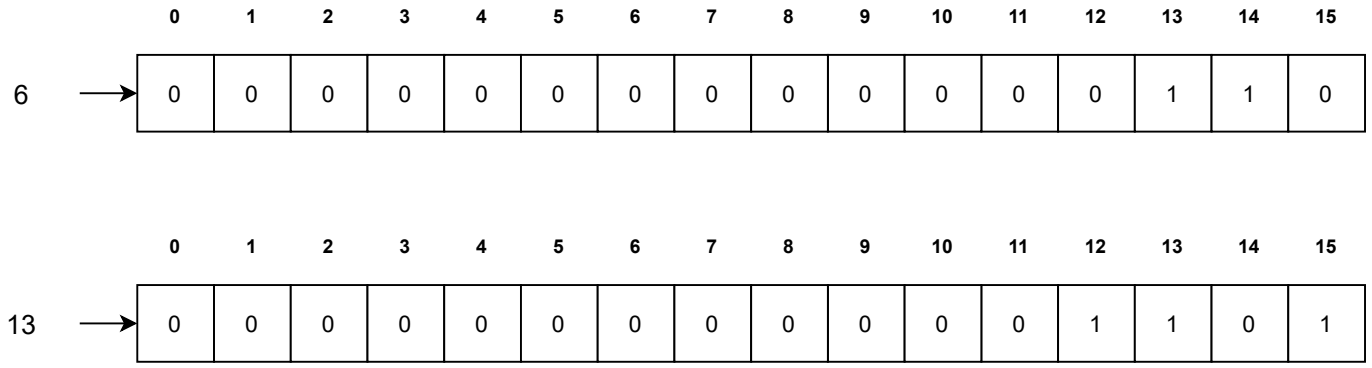
The three Logical operators have their corresponding Bitwise operators too…

- … i.e. Bitwise AND (`&`), Bitwise OR (`|`) and Bitwise NOT (`~`)

The XOR operator (`^`) provides the implementation of another Boolean function with the same name

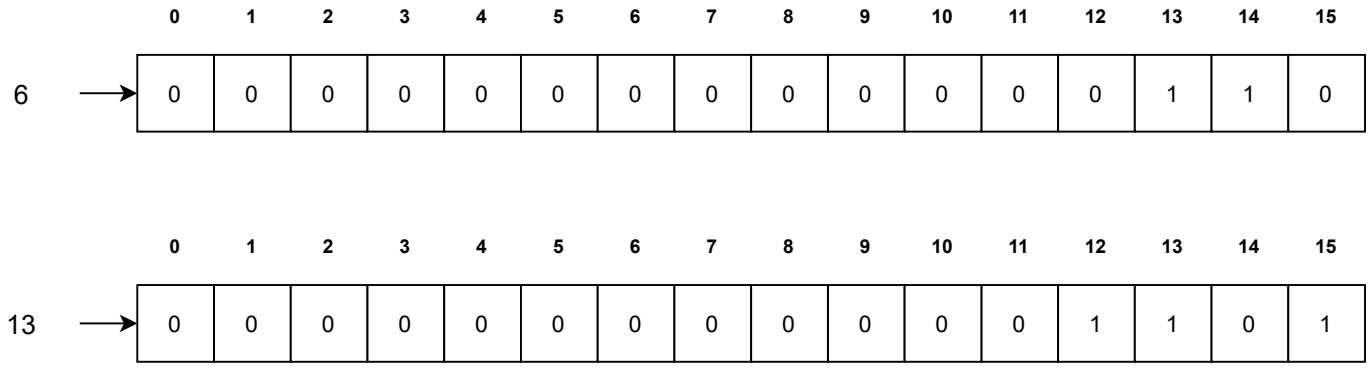
- XOR stands for *Exclusive OR*, which outputs a 1, if either of the two inputs (but not both) is `1`

Bitwise Representation of integers



Assume that we have two positive integers as operands

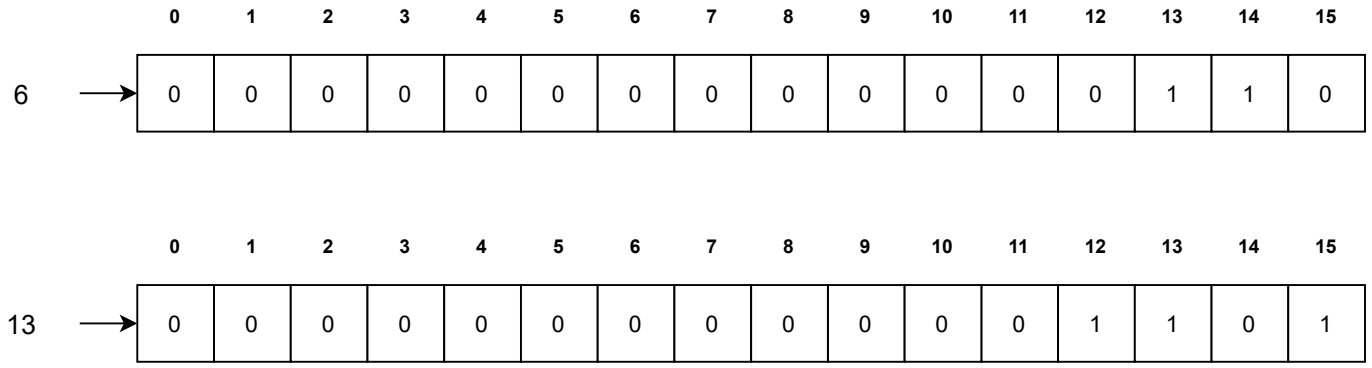
Bitwise Representation of integers



Assume that we have two positive integers as operands

Also assume that they are represented in 16 bits

Bitwise Representation of integers

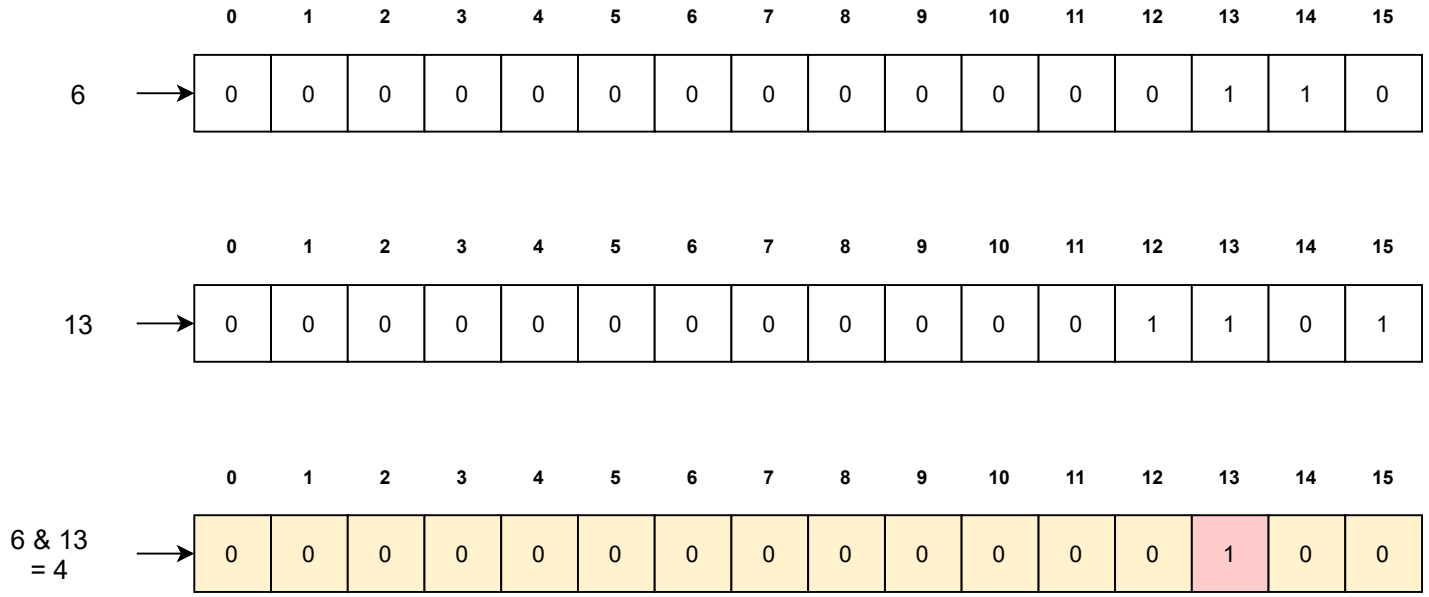


Assume that we have two positive integers as operands

Also assume that they are represented in 16 bits

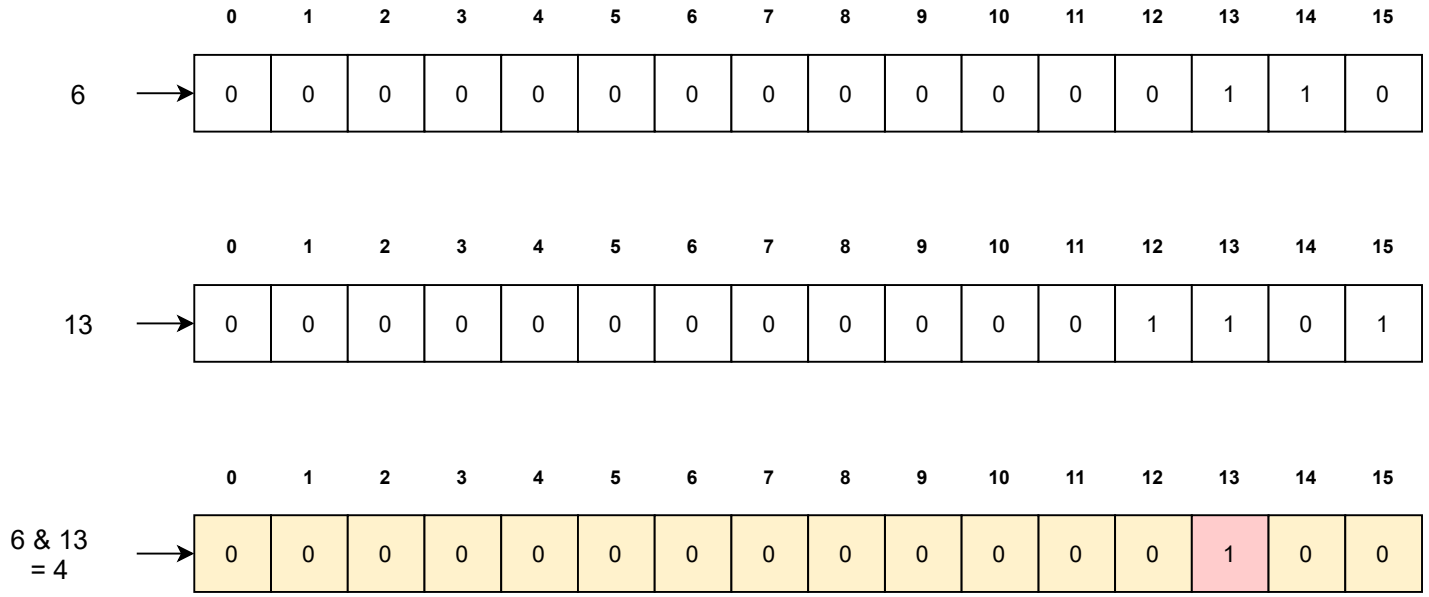
If the integers are unsigned, this is simply their representation

Bitwise AND Operator



The `&` operator performs *ANDing* of i^{th} bits of the two integers in parallel

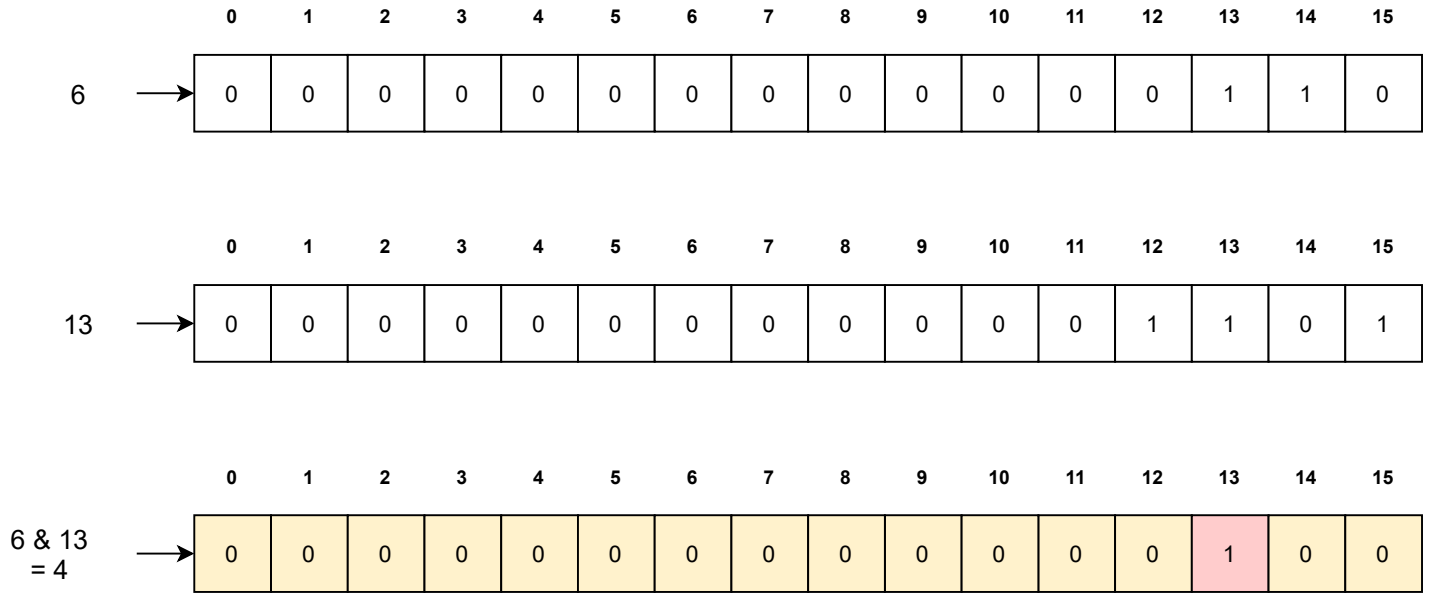
Bitwise AND Operator



The `&` operator performs *ANDing* of i^{th} bits of the two integers in parallel

Here, the only value of `i`, where both input bits are `1` is bit$_{13}$

Bitwise AND Operator

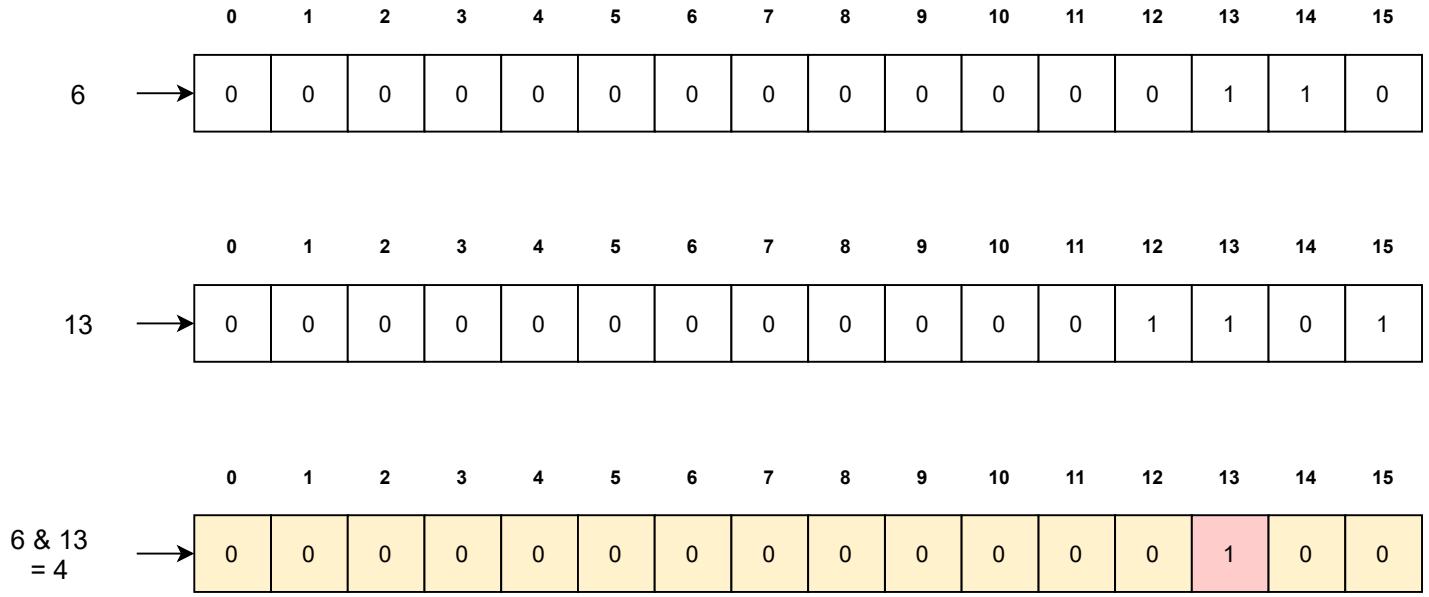


The `&` operator performs *ANDing* of i^{th} bits of the two integers in parallel

Here, the only value of i, where both input bits are 1 is bit_{13}

Thus, except bit_{13}, all other bits in the output are 0

Bitwise AND Operator



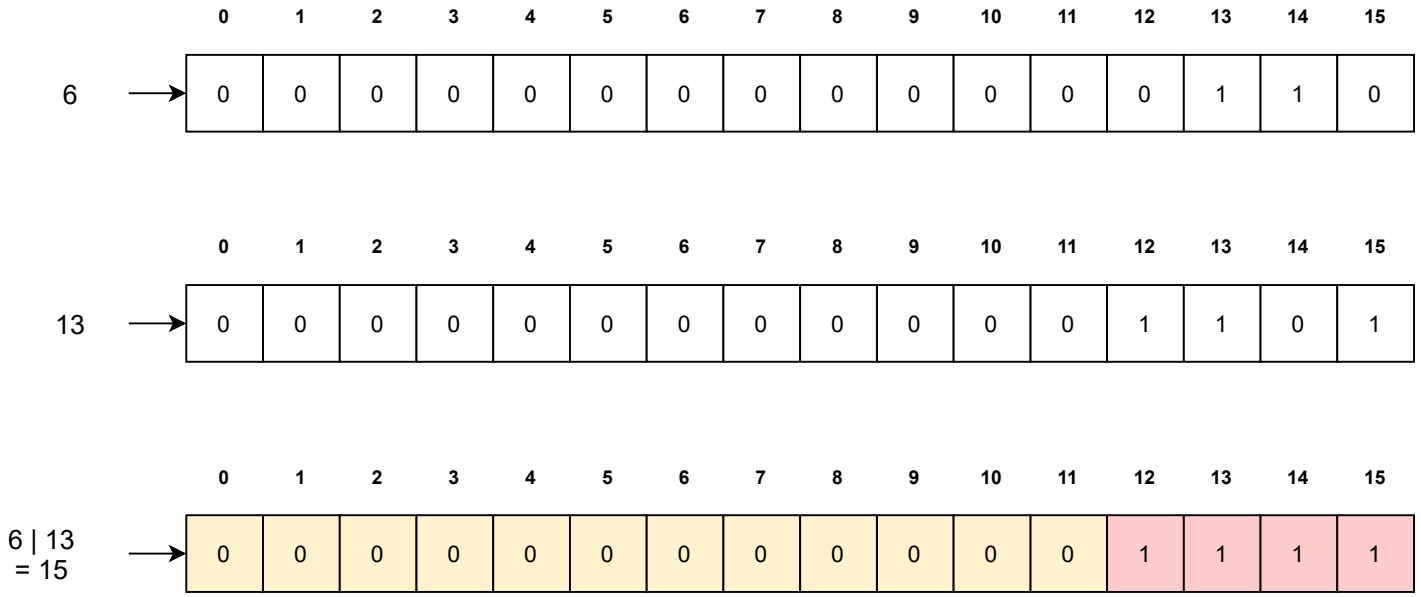
The `&` operator performs *ANDing* of i^{th} bits of the two integers in parallel

Here, the only value of i, where both input bits are `1` is bit_{13}

Thus, except bit_{13}, all other bits in the output are `0`

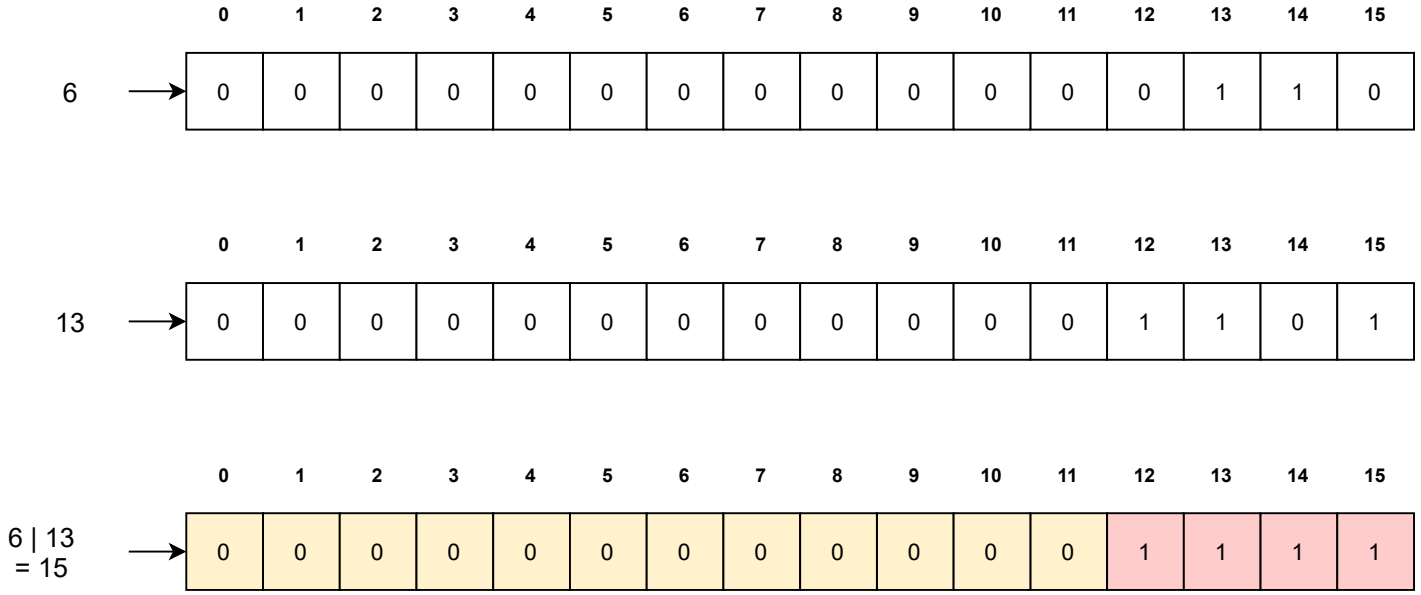
The output too, is an integer in 16 bits

Bitwise OR Operator



Similarly, | operator performs *ORing* of i^{th} bits of the two integers in parallel

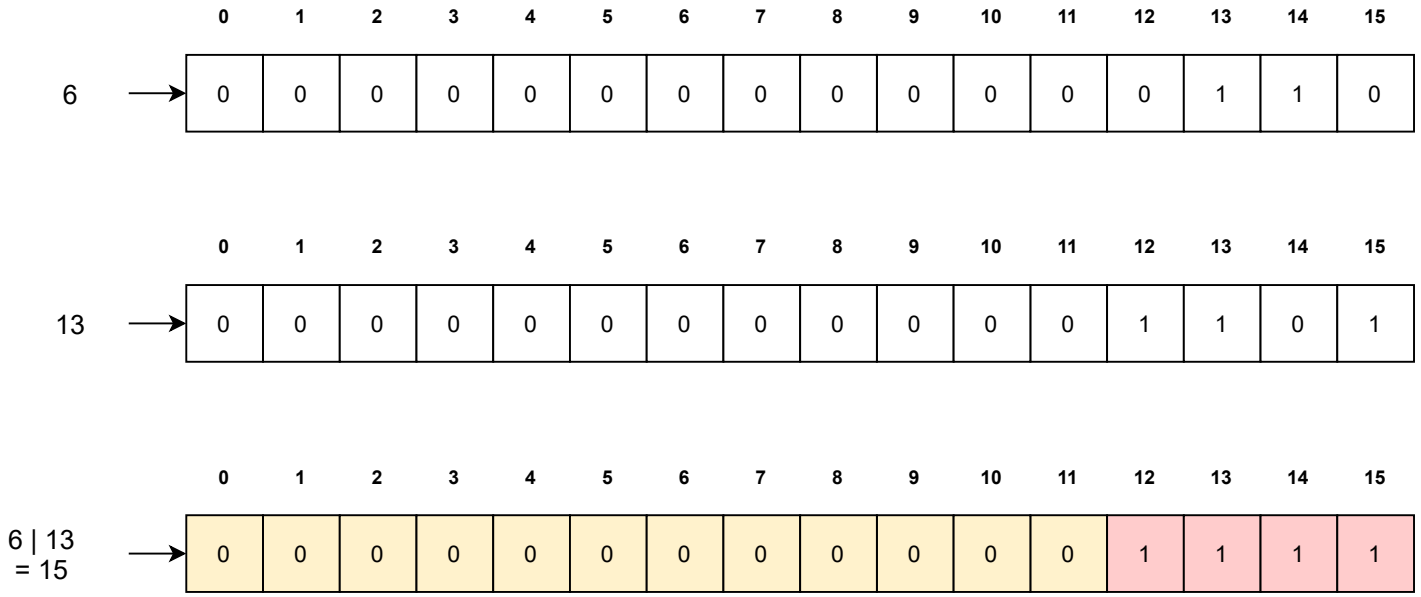
Bitwise OR Operator



Similarly, | operator performs *ORing* of i^{th} bits of the two integers in parallel

Here, bits 12 to 15 have 1s in either one or both operands

Bitwise OR Operator

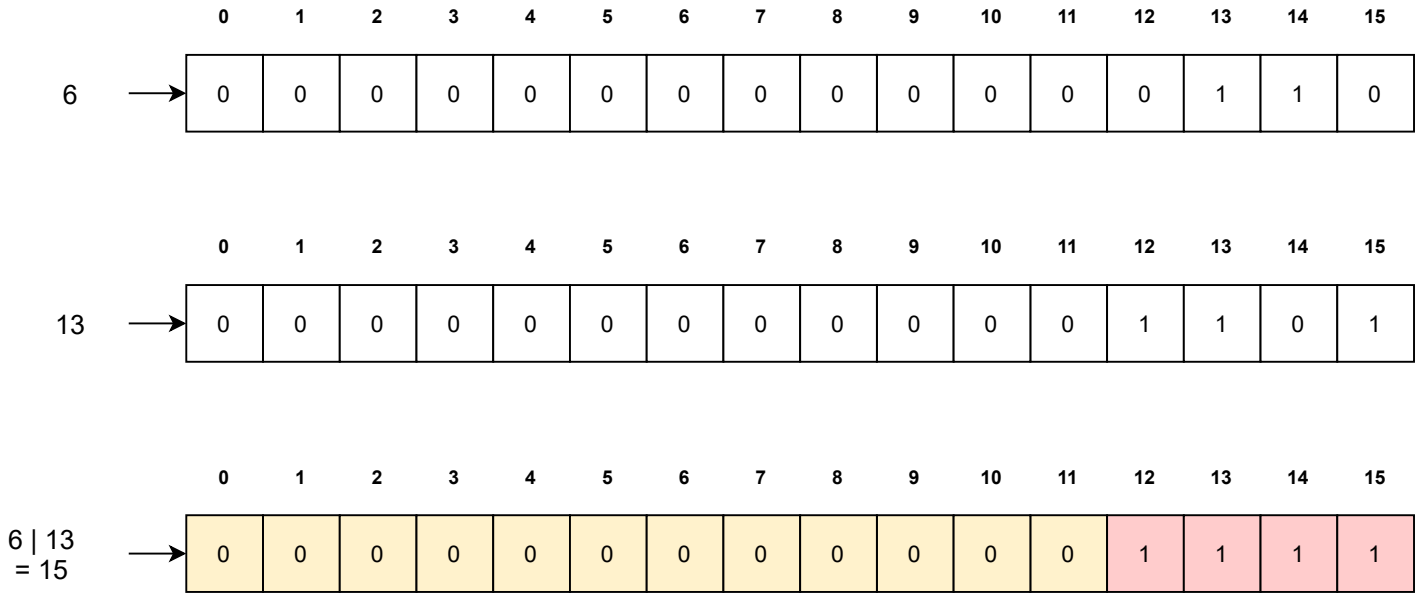


Similarly, `|` operator performs *ORing* of i^{th} bits of the two integers in parallel

Here, bits *12* to *15* have `1`s in either one or both operands

Thus, these bits become `1` in the output, while rest are `0`

Bitwise OR Operator



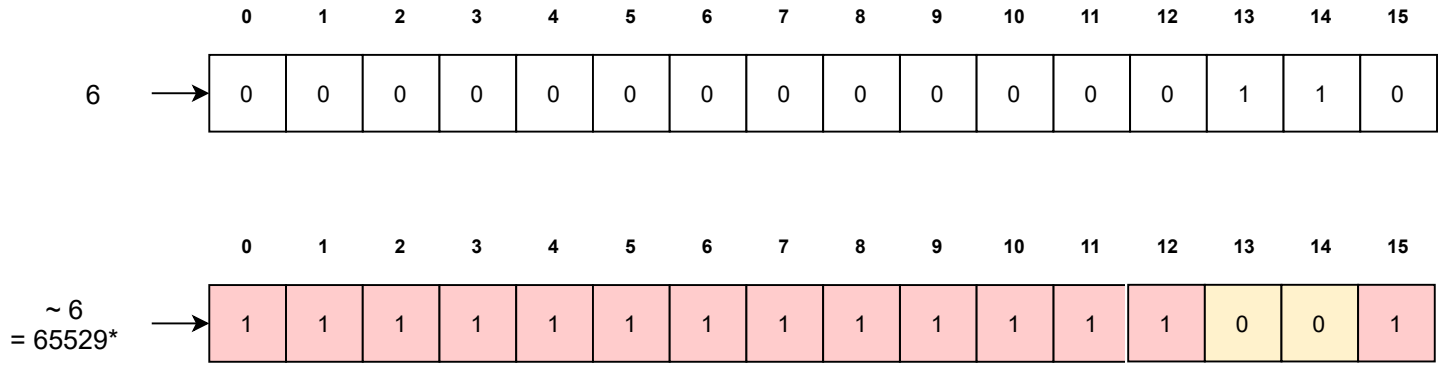
Similarly, `|` operator performs *ORing* of i^{th} bits of the two integers in parallel

Here, bits *12* to *15* have `1`s in either one or both operands

Thus, these bits become `1` in the output, while rest are `0`

Again, the output can be interpreted as an integer as well

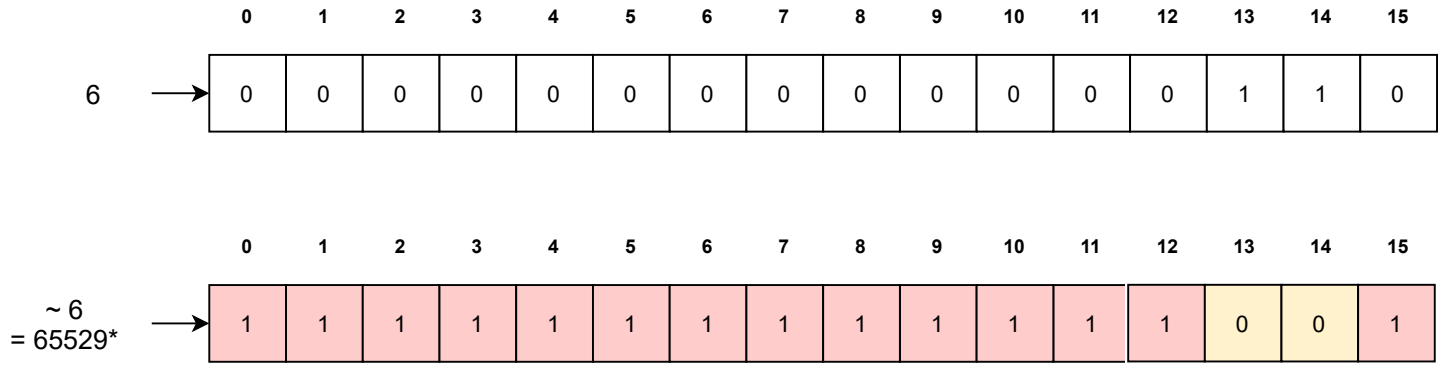
Bitwise NOT Operator



The ~ operator applies to a single operand, and it simply “flips” all the bits

* When interpreted as an unsigned integer; if it is interpreted as a signed integer in 2's complement representation, it represents -7

Bitwise NOT Operator

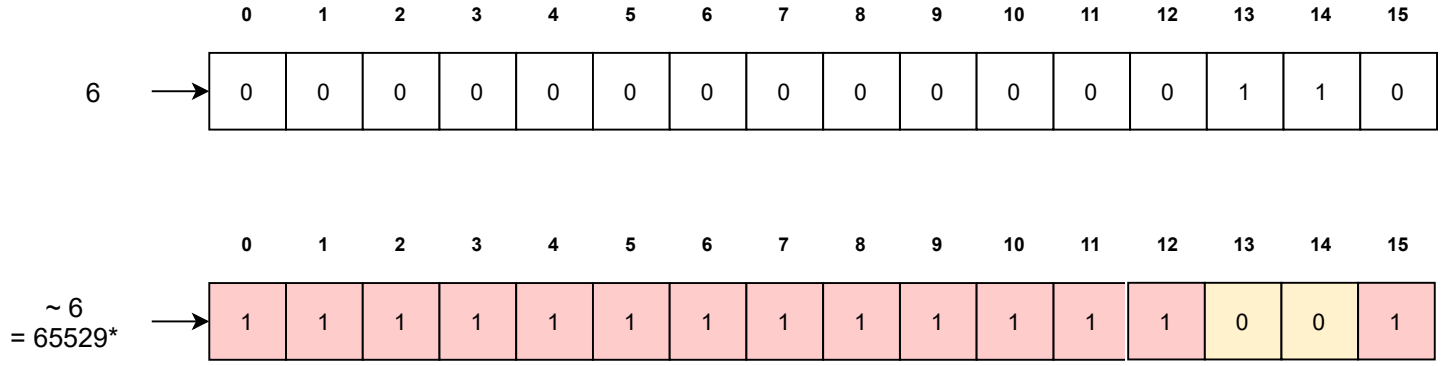


The ~ operator applies to a single operand, and it simply “flips” all the bits

Remember, the flip will also flip the leftmost bit

* When interpreted as an unsigned integer; if it is interpreted as a signed integer in 2's complement representation, it represents -7

Bitwise NOT Operator



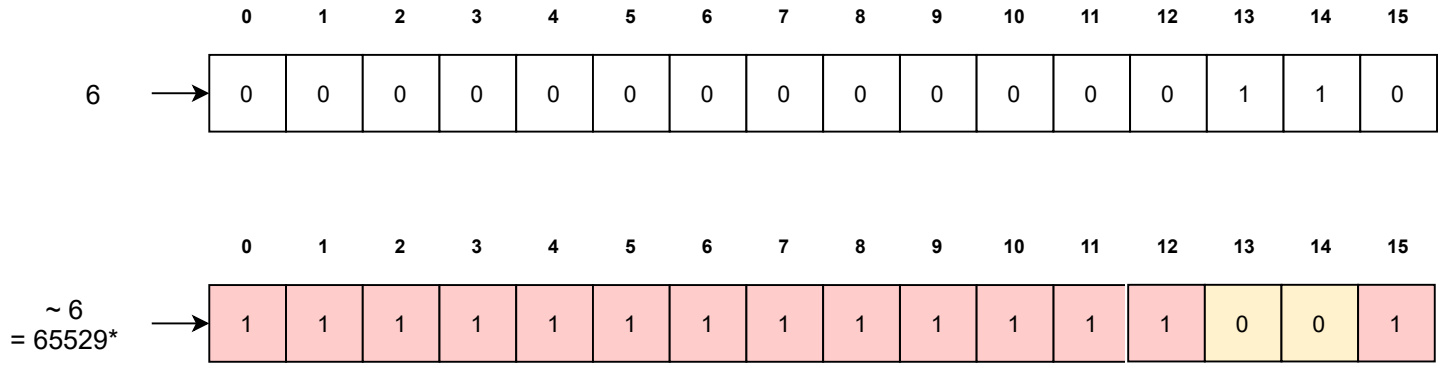
The ~ operator applies to a single operand, and it simply “flips” all the bits

Remember, the flip will also flip the leftmost bit

Thus, the interpretation of the result may be representation specific

* When interpreted as an unsigned integer; if it is interpreted as a signed integer in 2's complement representation, it represents -7

Bitwise NOT Operator



* When interpreted as an unsigned integer; if it is interpreted as a signed integer in 2's complement representation, it represents -7

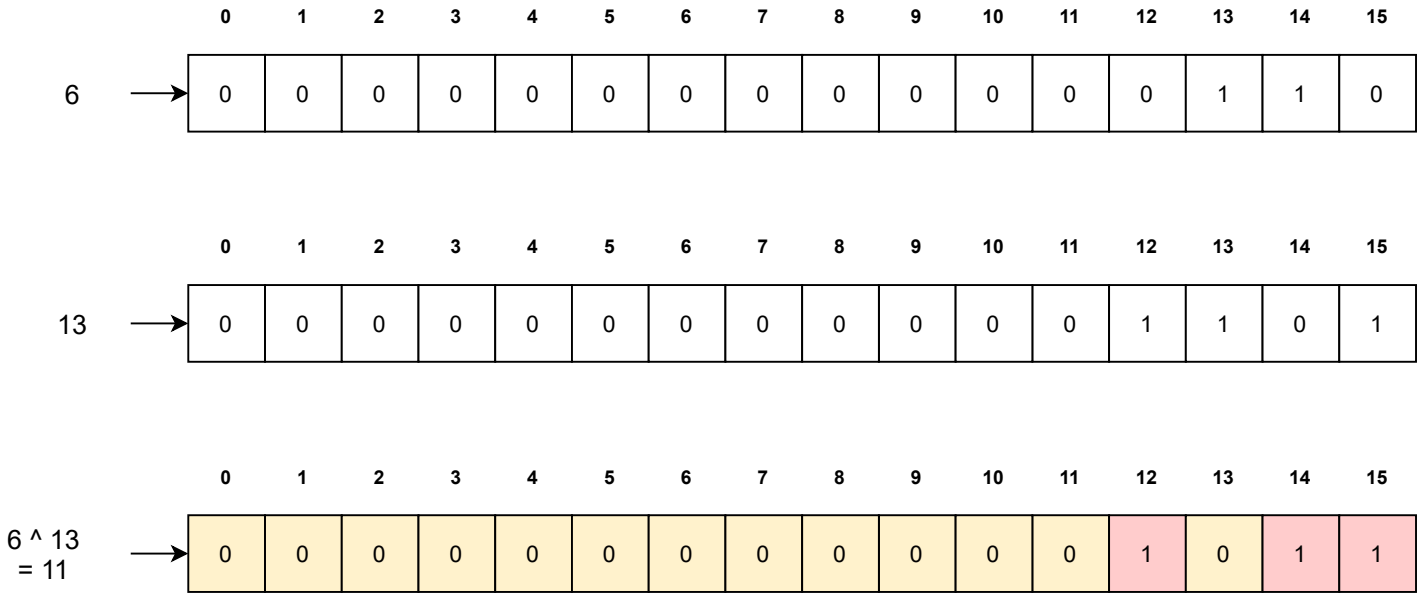
The ~ operator applies to a single operand, and it simply "flips" all the bits

Remember, the flip will also flip the leftmost bit

Thus, the interpretation of the result may be representation specific

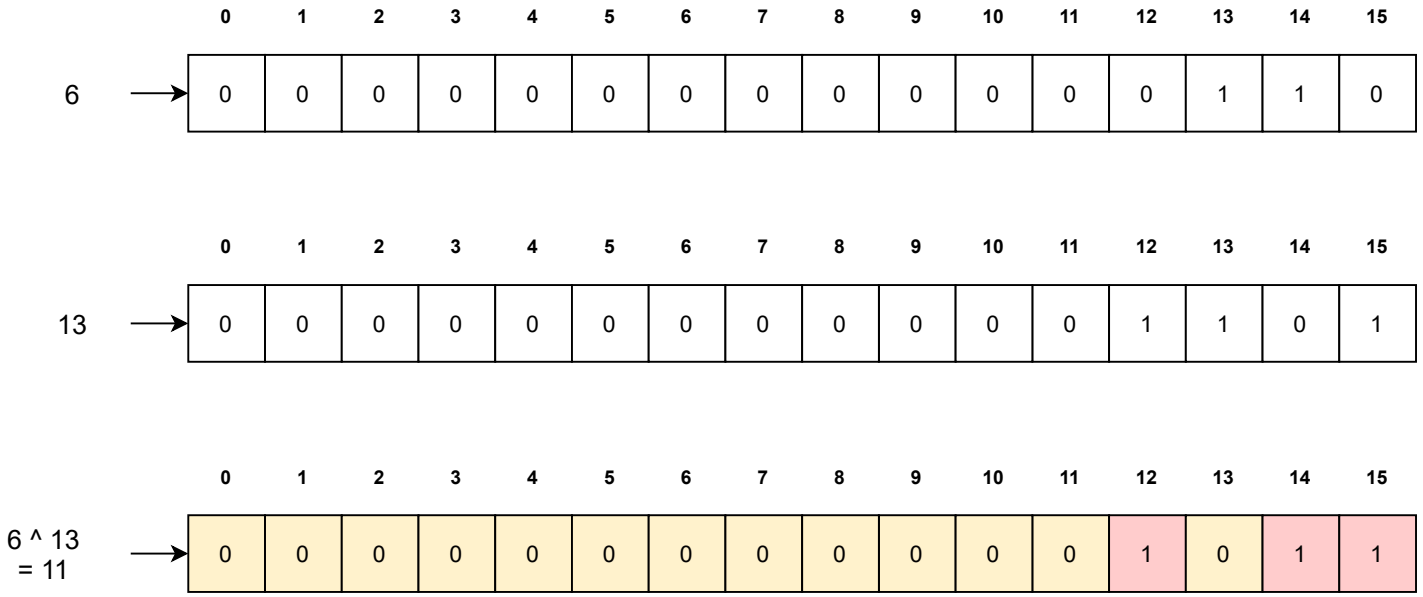
Based on the representation, the sign might also be flipped (from + to − and vice versa)

Bitwise XOR Operator



The ^ operator looks for the "number of input 1s"

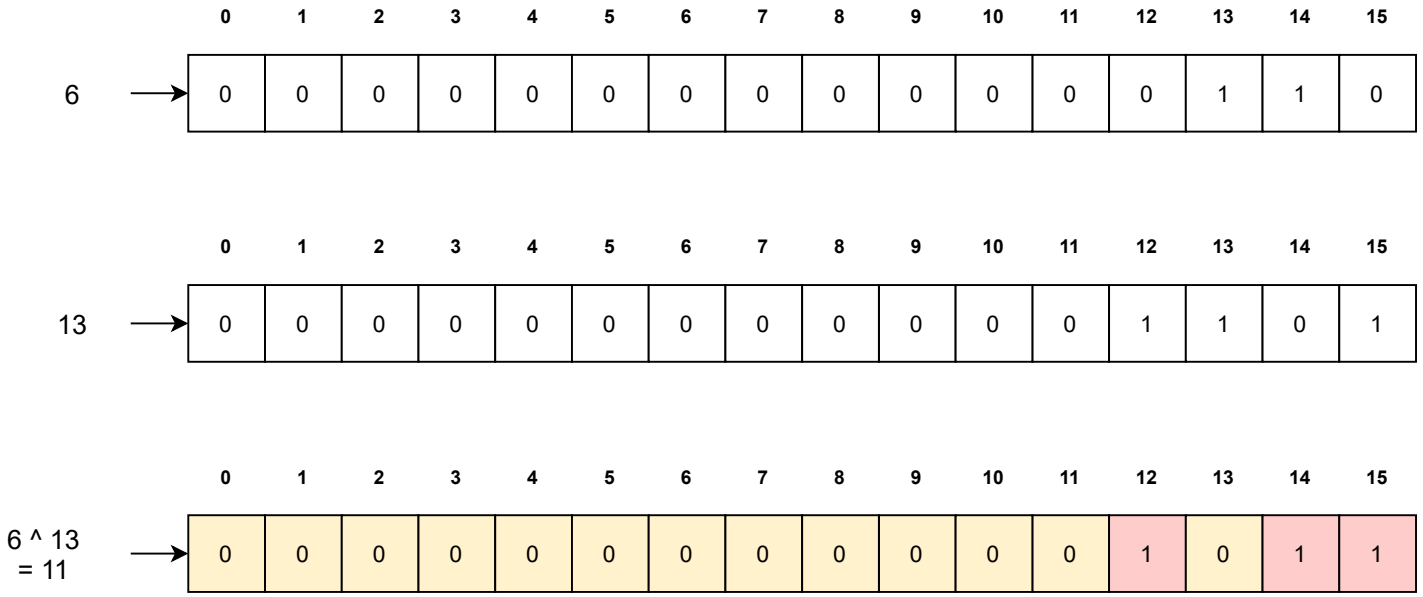
Bitwise XOR Operator



The ^ operator looks for the "number of input 1s"

It produces 0 as output, when the number of input 1s are 0 or 2

Bitwise XOR Operator

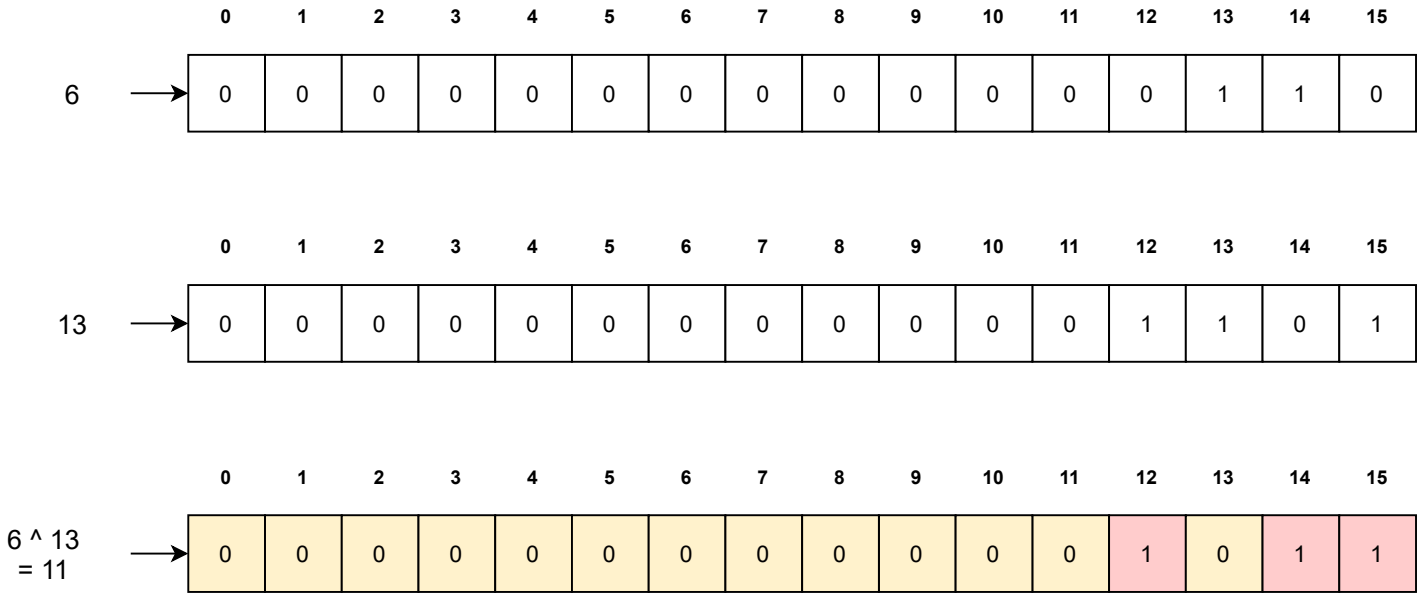


The ^ operator looks for the "number of input 1s"

It produces 0 as output, when the number of input 1s are 0 or 2

When "exactly" one of the two input bits is 1, the output is 1

Bitwise XOR Operator



The ^ operator looks for the "number of input 1s"

It produces 0 as output, when the number of input 1s are 0 or 2

When "exactly" one of the two input bits is 1, the output is 1

It differs from |, where bit$_{13}$ would have been 1

Bitwise Operators in C (2/2)

There are two more bitwise operators in C, which are essentially, arithmetic operators

Bitwise Operators in C (2/2)

There are two more bitwise operators in C, which are essentially, arithmetic operators

Both these operators, apply to a single integer, though they take two operands

Bitwise Operators in C (2/2)

There are two more bitwise operators in C, which are essentially, arithmetic operators

Both these operators, apply to a single integer, though they take two operands

- These operators, called the shift operators, shift bit pattern of the integer to either left or right

Bitwise Operators in C (2/2)

There are two more bitwise operators in C, which are essentially, arithmetic operators

Both these operators, apply to a single integer, though they take two operands

- These operators, called the shift operators, shift bit pattern of the integer to either left or right
- This results in the loss of some bits from one side, and `0` being added to the bit pattern on the other side

Bitwise Operators in C (2/2)

There are two more bitwise operators in C, which are essentially, arithmetic operators

Both these operators, apply to a single integer, though they take two operands

- These operators, called the shift operators, shift bit pattern of the integer to either left or right
- This results in the loss of some bits from one side, and `0` being added to the bit pattern on the other side

The left shift operator (`<<`) *shifts* the bit pattern of the first operand *to the left*…

- … by the number of positions equal to the second operand

Bitwise Operators in C (2/2)

There are two more bitwise operators in C, which are essentially, arithmetic operators

Both these operators, apply to a single integer, though they take two operands

- These operators, called the shift operators, shift bit pattern of the integer to either left or right
- This results in the loss of some bits from one side, and `0` being added to the bit pattern on the other side

The left shift operator (`<<`) *shifts* the bit pattern of the first operand *to the left*…

- … by the number of positions equal to the second operand
- For instance, `6 << 2` means “shift the bit pattern of 6 to the left by 2 positions”

Bitwise Operators in C (2/2)

There are two more bitwise operators in C, which are essentially, arithmetic operators

Both these operators, apply to a single integer, though they take two operands

- These operators, called the shift operators, shift bit pattern of the integer to either left or right
- This results in the loss of some bits from one side, and `0` being added to the bit pattern on the other side

The left shift operator (`<<`) *shifts* the bit pattern of the first operand *to the left*…

- … by the number of positions equal to the second operand
- For instance, `6 << 2` means "shift the bit pattern of 6 to the left by 2 positions"
- The left shift of a bit pattern essentially represents *multiplication*

Bitwise Operators in C (2/2)

There are two more bitwise operators in C, which are essentially, arithmetic operators

Both these operators, apply to a single integer, though they take two operands

- These operators, called the shift operators, shift bit pattern of the integer to either left or right
- This results in the loss of some bits from one side, and `0` being added to the bit pattern on the other side

The left shift operator (`<<`) *shifts* the bit pattern of the first operand *to the left*…

- … by the number of positions equal to the second operand
- For instance, `6 << 2` means "shift the bit pattern of 6 to the left by 2 positions"
- The left shift of a bit pattern essentially represents *multiplication*

The right shift operator (`>>`) *shifts* the bit pattern of the first operand *to the right*…

- … by the number of positions equal to the second operand

Bitwise Operators in C (2/2)

There are two more bitwise operators in C, which are essentially, arithmetic operators

Both these operators, apply to a single integer, though they take two operands

- These operators, called the shift operators, shift bit pattern of the integer to either left or right
- This results in the loss of some bits from one side, and `0` being added to the bit pattern on the other side

The left shift operator (`<<`) *shifts* the bit pattern of the first operand *to the left*…

- … by the number of positions equal to the second operand
- For instance, `6 << 2` means "shift the bit pattern of 6 to the left by 2 positions"
- The left shift of a bit pattern essentially represents *multiplication*

The right shift operator (`>>`) *shifts* the bit pattern of the first operand *to the right*…

- … by the number of positions equal to the second operand
- For instance, `6 >> 2` means "shift the bit pattern of 6 to the right by 2 positions"

Bitwise Operators in C (2/2)

There are two more bitwise operators in C, which are essentially, arithmetic operators

Both these operators, apply to a single integer, though they take two operands

- These operators, called the shift operators, shift bit pattern of the integer to either left or right
- This results in the loss of some bits from one side, and `0` being added to the bit pattern on the other side

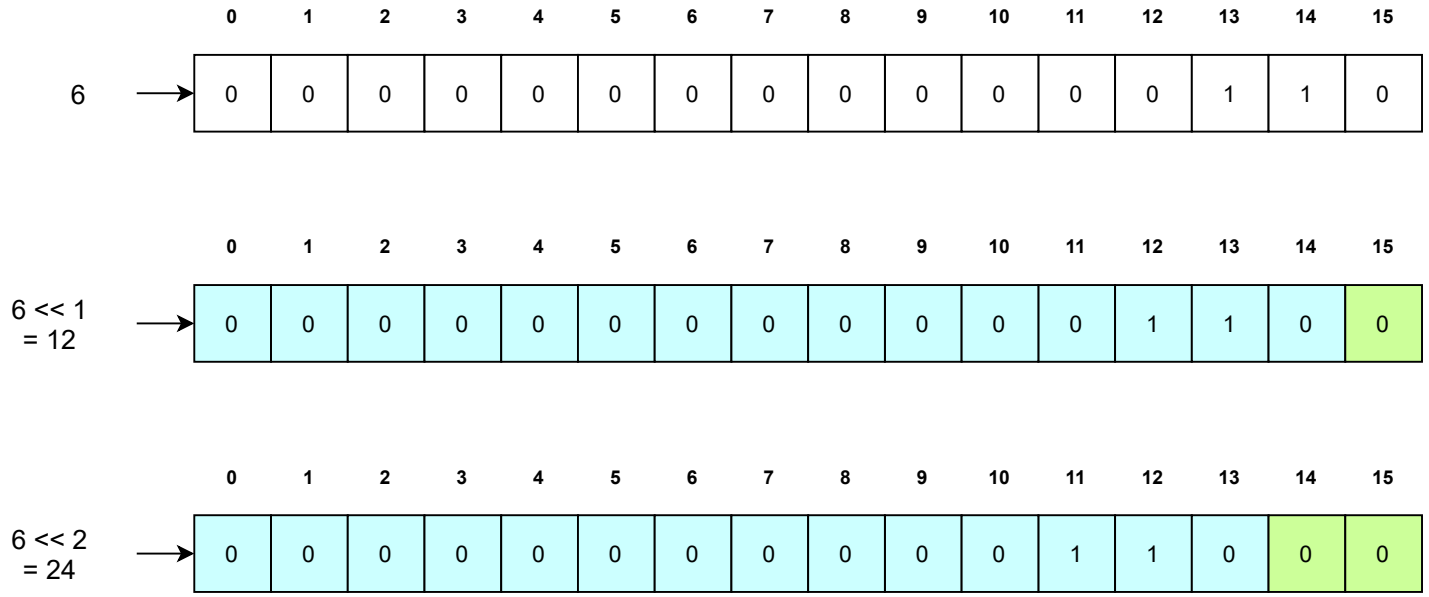
The left shift operator (`<<`) *shifts* the bit pattern of the first operand *to the left*…

- … by the number of positions equal to the second operand
- For instance, `6 << 2` means "shift the bit pattern of 6 to the left by 2 positions"
- The left shift of a bit pattern essentially represents *multiplication*

The right shift operator (`>>`) *shifts* the bit pattern of the first operand *to the right*…

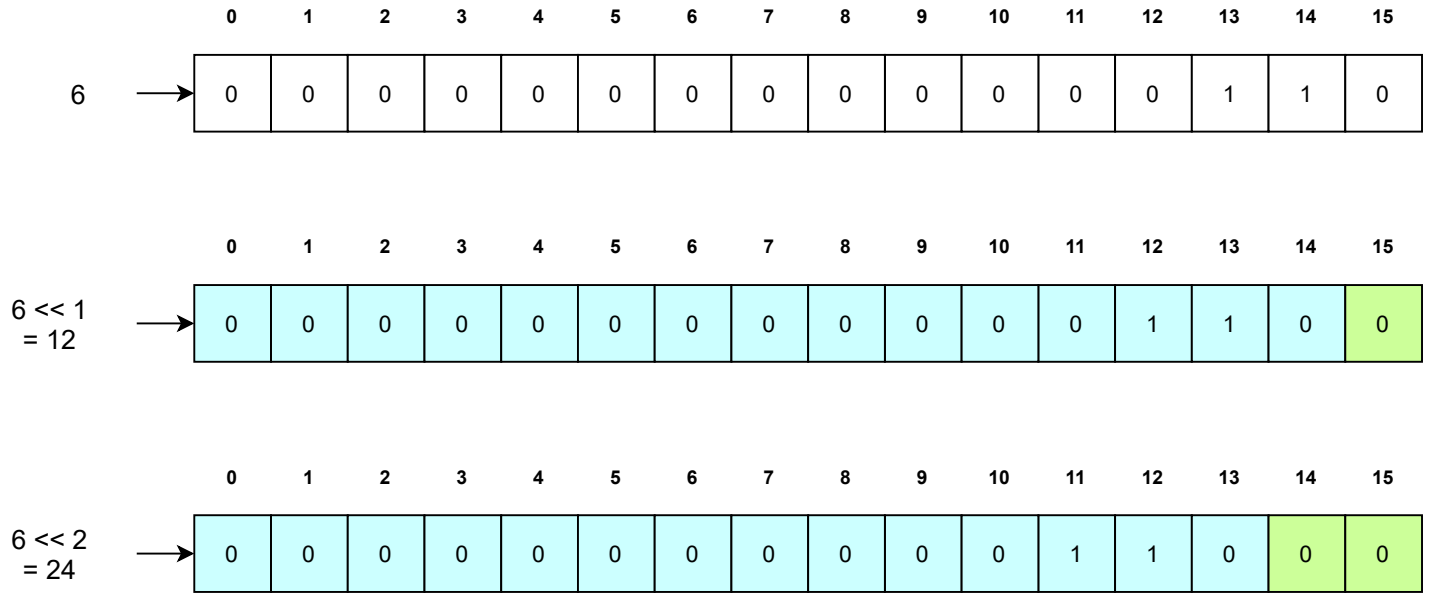
- … by the number of positions equal to the second operand
- For instance, `6 >> 2` means "shift the bit pattern of 6 to the right by 2 positions"
- The right shift of a bit pattern essentially represents *division*

Left Shift Operator



The << operator removes bits from the left and adds 0s from the right

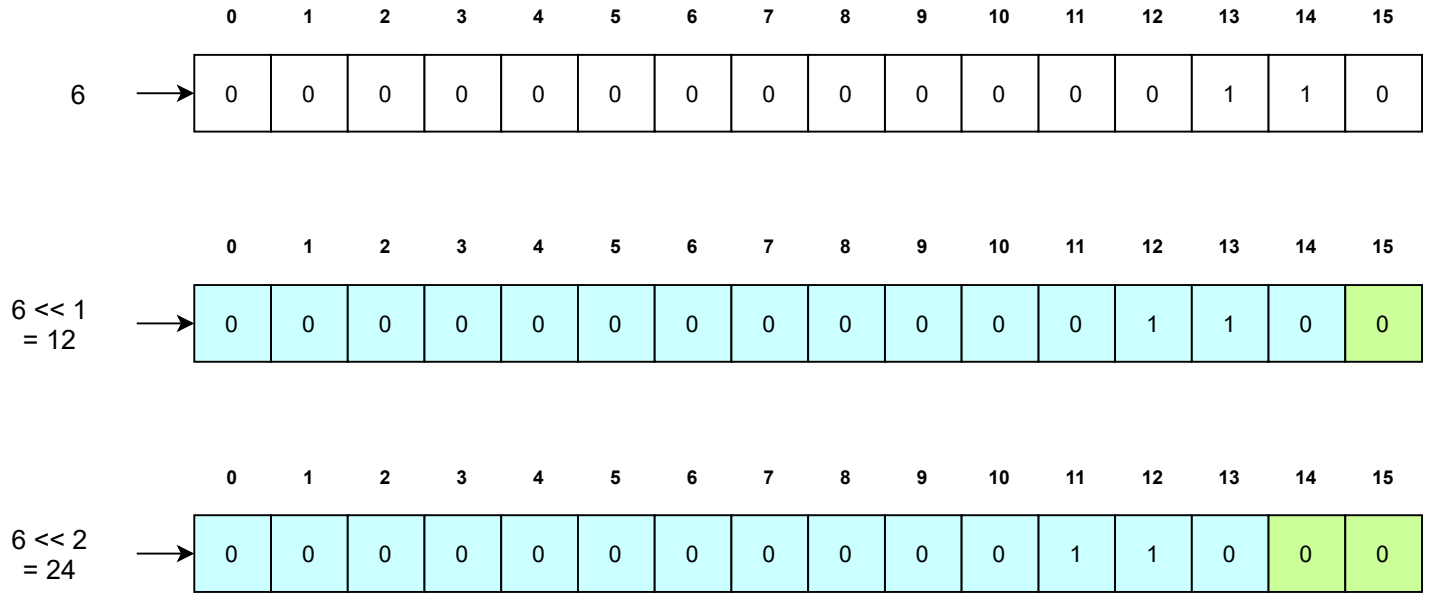
Left Shift Operator



The $<<$ operator removes bits from the left and adds 0s from the right

A left shift by k positions of an integer means its multiplication with 2^k

Left Shift Operator

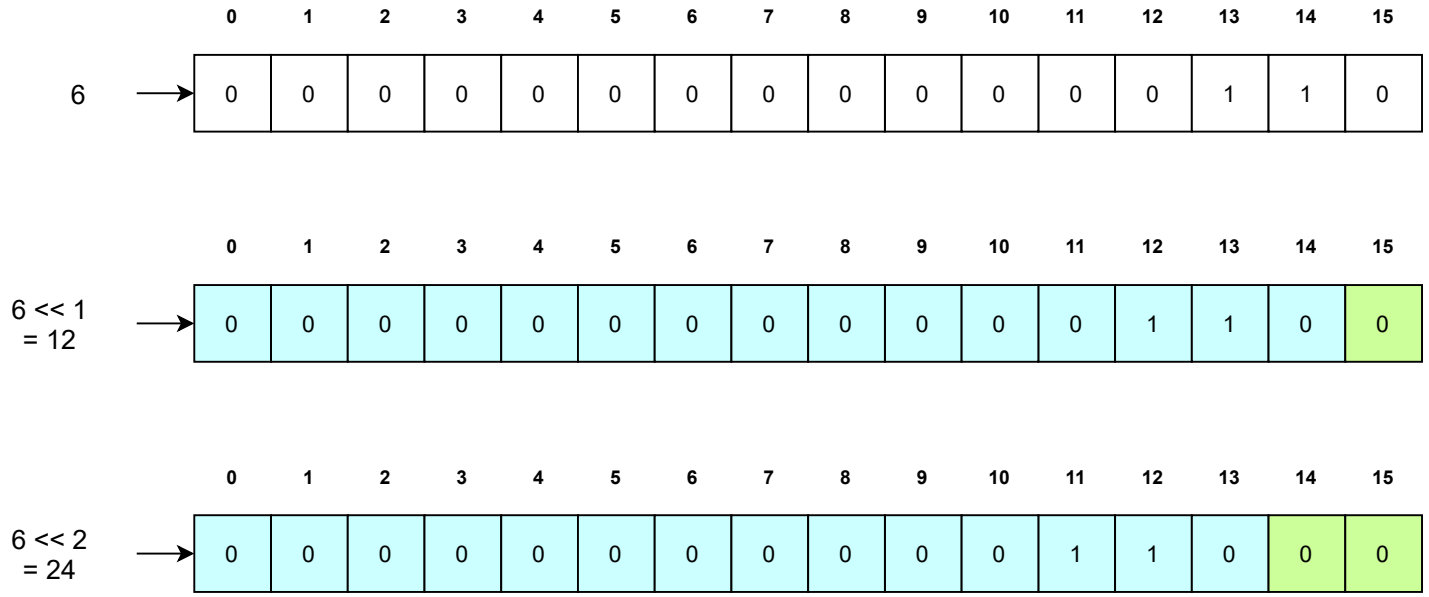


The $<<$ operator removes bits from the left and adds 0s from the right

A left shift by k positions of an integer means its multiplication with 2^k

Since the leftmost bit will be removed, interpretation is representation-specific

Left Shift Operator



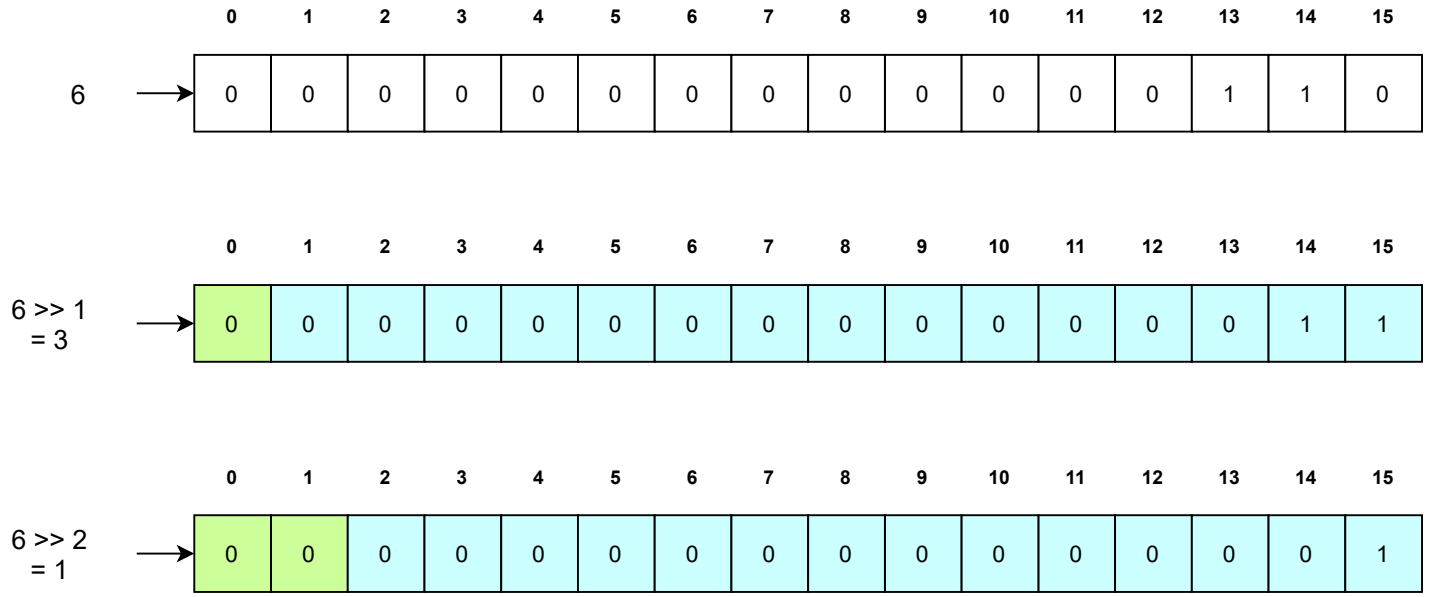
The $<<$ operator removes bits from the left and adds 0s from the right

A left shift by k positions of an integer means its multiplication with 2^k

Since the leftmost bit will be removed, interpretation is representation-specific

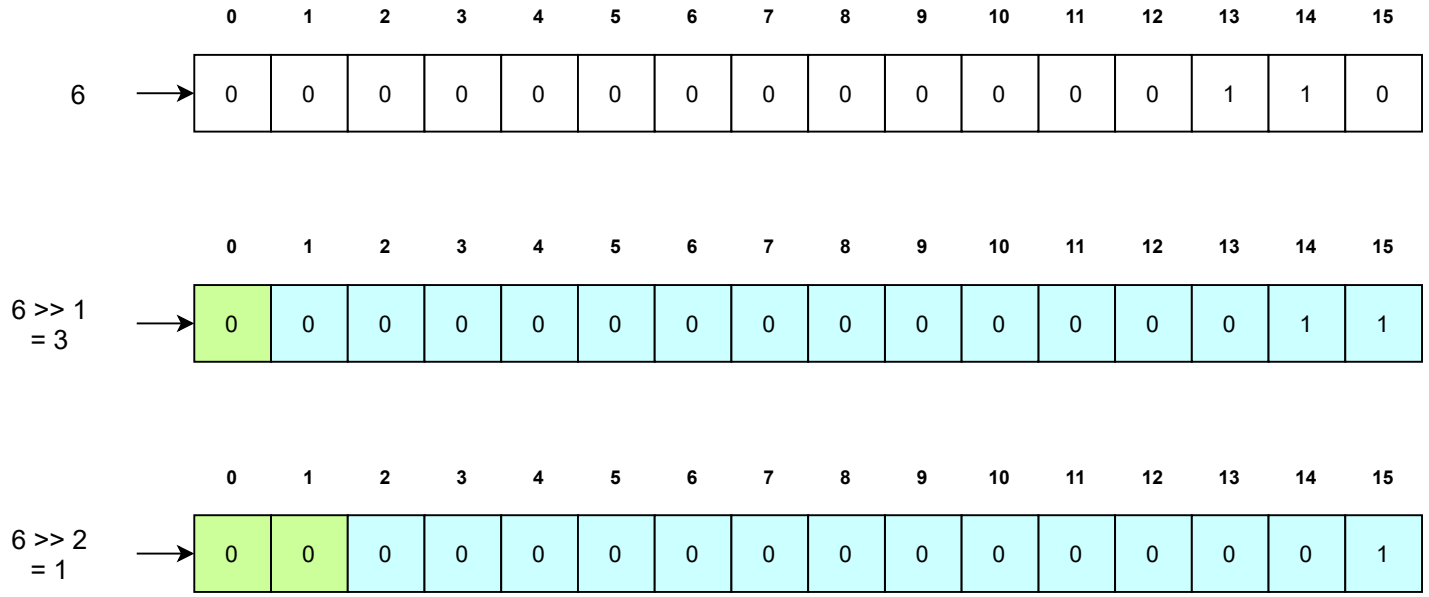
Based on the representation, the sign might also be flipped (from $+$ to $-$)

Right Shift Operator



The >> operator removes bits from the right and adds 0s from the right

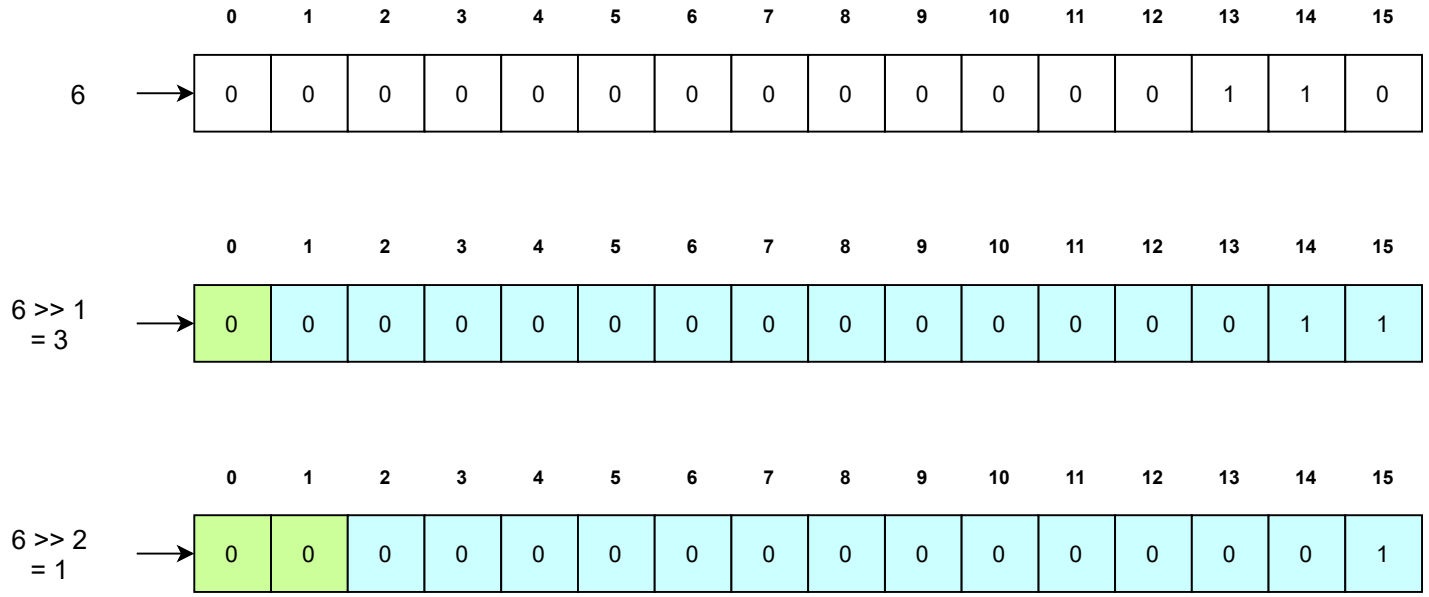
Right Shift Operator



The >> operator removes bits from the right and adds 0s from the right

A right shift by k positions of an integer means its division by 2^k

Right Shift Operator

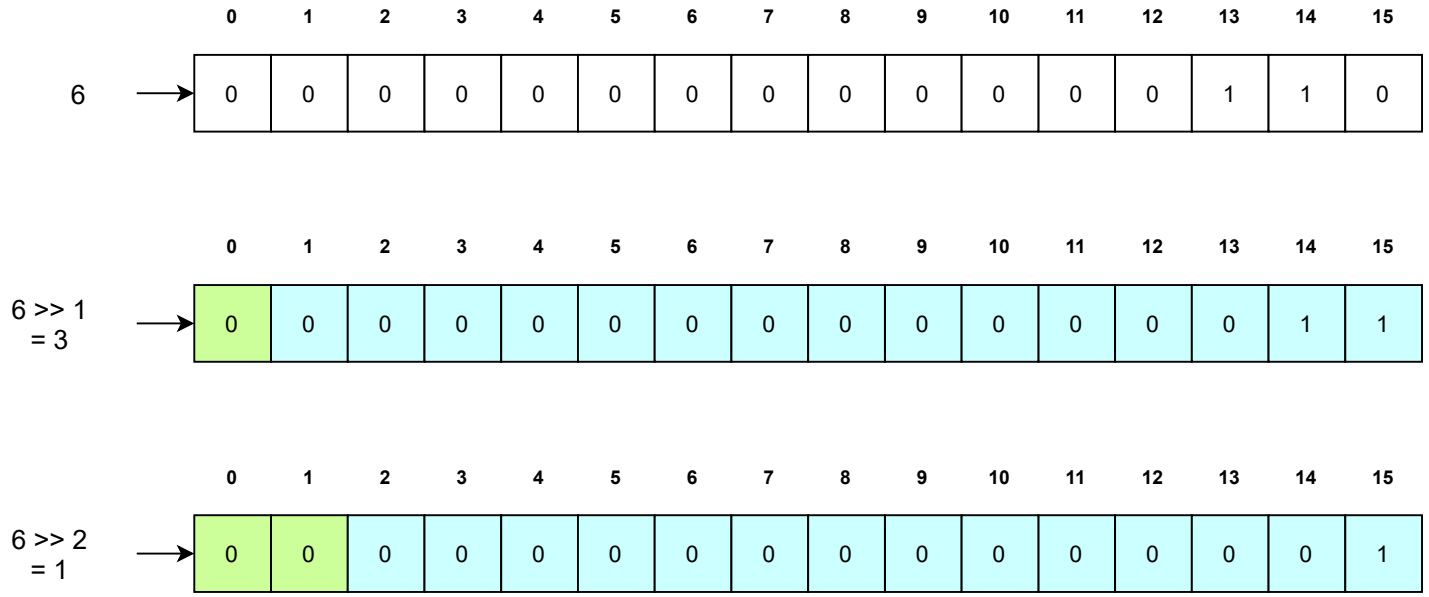


The $>>$ operator removes bits from the right and adds 0s from the right

A right shift by k positions of an integer means its division by 2^k

Since the leftmost bit will become 0, interpretation is representation-specific

Right Shift Operator



The $>>$ operator removes bits from the right and adds 0s from the right

A right shift by k positions of an integer means its division by 2^k

Since the leftmost bit will become 0, interpretation is representation-specific

Based on the representation, the sign might also be flipped (from $-$ to $+$)

The customary example

```c
#include<stdio.h>

int main()
{
        unsigned short op1 = 6;
        unsigned short op2 = 13;

        printf("This is a simple demonstration of bitwise operators\n");

        printf("%hu&%hu = %hu\n", op1, op2, op1&op2);
        printf("%hu|%hu = %hu\n", op1, op2, op1|op2);
        printf("~%hu = %hu in unsigned form\n", op1, ~op1);
        printf("~%hu = %hi in signed 2's complement form\n", op1, ~op1);
        printf("%hu<<1 = %hu\n", op1, op1<<1);
        printf("%hu<<2 = %hu\n", op1, op1<<2);
        printf("%hu>>1 = %hu\n", op1, op1>>1);
        printf("%hu>>2 = %hu\n", op1, op1>>2);
}
```

The customary example

```c
#include<stdio.h>

int main()
{
        unsigned short op1 = 6;
        unsigned short op2 = 13;

        printf("This is a simple demonstration of bitwise operators\n");

        printf("%hu&%hu = %hu\n", op1, op2, op1&op2);
        printf("%hu|%hu = %hu\n", op1, op2, op1|op2);
        printf("~%hu = %hu in unsigned form\n", op1, ~op1);
        printf("~%hu = %hi in signed 2's complement form\n", op1, ~op1);
        printf("%hu<<1 = %hu\n", op1, op1<<1);
        printf("%hu<<2 = %hu\n", op1, op1<<2);
        printf("%hu>>1 = %hu\n", op1, op1>>1);
        printf("%hu>>2 = %hu\n", op1, op1>>2);
}
```

Just the code to reproduce the examples in the slide

The customary example

```
saurabh@saurabh-VirtualBox:~/C/examples/Week 11$ gcc TheBitwiseWorld.c
saurabh@saurabh-VirtualBox:~/C/examples/Week 11$ ./a.out
This is a simple demonstration of bitwise operators
6&13 = 4
6|13 = 15
~6 = 65529 in unsigned form
~6 = -7 in signed 2's complement form
6<<1 = 12
6<<2 = 24
6>>1 = 3
6>>2 = 1
saurabh@saurabh-VirtualBox:~/C/examples/Week 11$
```

The customary example

```
saurabh@saurabh-VirtualBox:~/C/examples/Week 11$ gcc TheBitwiseWorld.c
saurabh@saurabh-VirtualBox:~/C/examples/Week 11$ ./a.out
This is a simple demonstration of bitwise operators
6&13 = 4
6|13 = 15
~6 = 65529 in unsigned form
~6 = -7 in signed 2's complement form
6<<1 = 12
6<<2 = 24
6>>1 = 3
6>>2 = 1
saurabh@saurabh-VirtualBox:~/C/examples/Week 11$
```

A confirmation of the results that we saw in the slides

Some important points

The bitwise operators are applicable to integer types only…

- … i.e. `char`, `short`, `int` and `long`

Some important points

The bitwise operators are applicable to integer types only…

- … i.e. `char`, `short`, `int` and `long`

Many operations on bitwise operators are left as “undefined” in the standards

- This means that the output may differ from one platform to the other for such cases

Some important points

The bitwise operators are applicable to integer types only…

- … i.e. `char`, `short`, `int` and `long`

Many operations on bitwise operators are left as “undefined” in the standards

- This means that the output may differ from one platform to the other for such cases

Bitwise operators may produce results that may not be “intuitive”

Some important points

The bitwise operators are applicable to integer types only…

- … i.e. `char`, `short`, `int` and `long`

Many operations on bitwise operators are left as “undefined” in the standards

- This means that the output may differ from one platform to the other for such cases

Bitwise operators may produce results that may not be “intuitive”

- For example, the Bitwise NOT operator may flip the sign of the integer operand…
- … and the Shift operators may result in overflow or underflow

Some important points

The bitwise operators are applicable to integer types only…

- … i.e. `char`, `short`, `int` and `long`

Many operations on bitwise operators are left as “undefined” in the standards

- This means that the output may differ from one platform to the other for such cases

Bitwise operators may produce results that may not be “intuitive”

- For example, the Bitwise NOT operator may flip the sign of the integer operand…
- … and the Shift operators may result in overflow or underflow

Bitwise operators are usually used in cases where performance is of utmost importance

Some important points

The bitwise operators are applicable to integer types only…

- … i.e. `char`, `short`, `int` and `long`

Many operations on bitwise operators are left as “undefined” in the standards

- This means that the output may differ from one platform to the other for such cases

Bitwise operators may produce results that may not be “intuitive”

- For example, the Bitwise NOT operator may flip the sign of the integer operand…
- … and the Shift operators may result in overflow or underflow

Bitwise operators are usually used in cases where performance is of utmost importance

- If a particular computation can be formulated in terms of bitwise operators…
- … there is a good chance that this formulation runs faster as compared to any other equivalent formulation

Some important points

The bitwise operators are applicable to integer types only…

- … i.e. `char`, `short`, `int` and `long`

Many operations on bitwise operators are left as “undefined” in the standards

- This means that the output may differ from one platform to the other for such cases

Bitwise operators may produce results that may not be “intuitive”

- For example, the Bitwise NOT operator may flip the sign of the integer operand…
- … and the Shift operators may result in overflow or underflow

Bitwise operators are usually used in cases where performance is of utmost importance

- If a particular computation can be formulated in terms of bitwise operators…
- … there is a good chance that this formulation runs faster as compared to any other equivalent formulation
- You’ll understand the reasons for the same during your course on Computer Organisation

Homework !!

Read more about bitwise operators, especially if you wish to use them in your code

- This compact article is a nice read: https://www.geeksforgeeks.org/bitwise-operators-in-c-cpp/

Probably the most useful bitwise operator is the ^ operator

- Open some of the links provided on the above page to get an idea of its utility