Let us give Mr C a Complement

ESC101: Fundamentals of Computing

Purushottam Kar

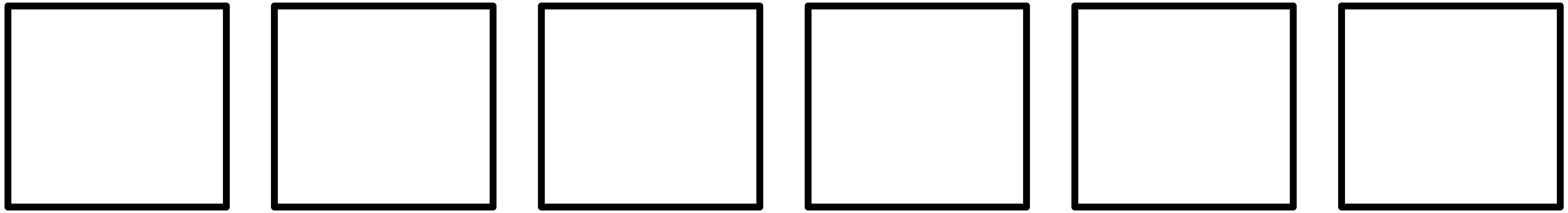
The Binary Number System

The Binary Number System

Just two digits – 0 and 1

The Binary Number System

Just two digits – 0 and 1



The Binary Number System

2

Just two digits – 0 and 1

1					

The Binary Number System

Just two digits – 0 and 1

1	0				

The Binary Number System

Just two digits – 0 and 1

1	0	1			

The Binary Number System

Just two digits – 0 and 1

1	0	1	1		

The Binary Number System

Just two digits – 0 and 1

1	0	1	1	0	

The Binary Number System

Just two digits – 0 and 1

1	0	1	1	0	1

The Binary Number System

Just two digits – 0 and 1

1	0	1	1	0	1

$= 1 \times 2^0 + 0 \times 2^1 + 1 \times 2^2 + 1 \times 2^3 + 0 \times 2^4 + 1 \times 2^5$

The Binary Number System

Just two digits – 0 and 1

1	0	1	1	0	1

$= 1 \times 2^0 + 0 \times 2^1 + 1 \times 2^2 + 1 \times 2^3 + 0 \times 2^4 + 1 \times 2^5$

$= 1 + 0 + 4 + 8 + 0 + 32$

The Binary Number System

Just two digits – 0 and 1

1	0	1	1	0	1

$= 1 \times 2^0 + 0 \times 2^1 + 1 \times 2^2 + 1 \times 2^3 + 0 \times 2^4 + 1 \times 2^5$

$= 1 + 0 + 4 + 8 + 0 + 32$

$= 45$

The Binary Number System

Just two digits – 0 and 1

1	0	1	1	0	1

$= 1 \times 2^0 + 0 \times 2^1 + 1 \times 2^2 + 1 \times 2^3 + 0 \times 2^4 + 1 \times 2^5$

$= 1 + 0 + 4 + 8 + 0 + 32$

$= 45$

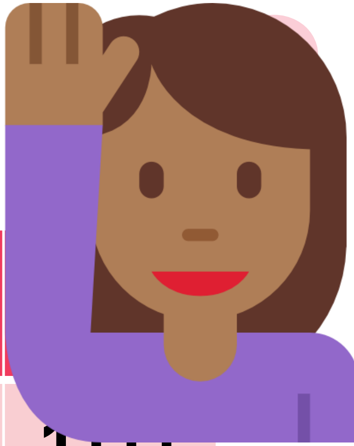
We will today see how the binary system is used to represent negative and fractional numbers

Bitwise Operators

Bitwise Operators

Operation	C Code	a	b	c	d	e	f
BITWISE AND	c = a & b	0000	1111	0000	1111	1111	1111
BITWISE OR	d = a | b	0101	1100	0100	1101	1001	1010
BITWISE XOR	e = a ^ b	1010	1110	1010	1110	0100	0101
BITWISE COMPLEMENT	f = ~a	1001	0111	0001	1111	1110	0110

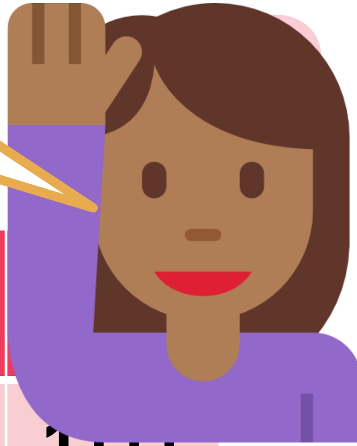
Bitwise Operators



Operation	C Code	a	b	c	d	e	
BITWISE AND	c = a & b	0000	1111	0000	1111	1111	1111
BITWISE OR	d = a | b	0101	1100	0100	1101	1001	1010
BITWISE XOR	e = a ^ b	1010	1110	1010	1110	0100	0101
BITWISE COMPLEMENT	f = ~a	1001	0111	0001	1111	1110	0110

Bitwise Operato

Note: the word is complement and not compliment ☺

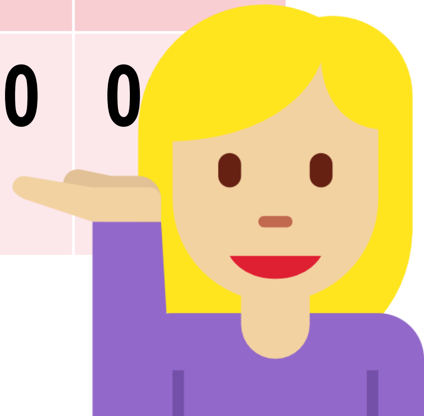


Operation	C Code	a	b	c	d	e	
BITWISE AND	c = a & b	0000	1111	0000	1111	1111	1111
BITWISE OR	d = a | b	0101	1100	0100	1101	1001	1010
BITWISE XOR	e = a ^ b	1010	1110	1010	1110	0100	0101
BITWISE COMPLEMENT	f = ~a	1001	0111	0001	1111	1110	0110

Bitwise Operat

Note: the word is complement and not compliment ☺

Operation	C Code	a	b	c	d	e	
BITWISE AND	c = a & b	0000	1111	0000	1111	1111	1111
BITWISE OR	d = a | b	0101	1100	0100	1101	1001	1010
BITWISE XOR	e = a ^ b	1010	1110	1010	1110	0100	0101
BITWISE COMPLEMENT	f = ~a	1001	0111	0001	1111	1110	0



Bitwise Operato

Note: the word is complement and not compliment ☺

Operation	C Code	a	b	c	d	e	
BITWISE AND	c = a & b	0000	1111	0000	1111	1111	1111
BITWISE OR	d = a | b	0101	1100	0100	1101	1001	1010
BITWISE XOR	e = a ^ b	1010	1110	1010	1110	0100	0101
BITWISE COMPLEMENT	f = ~a	1					0

The word compliment means to praise someone, complement means to enhance something

Bitwise And Operator &

Bitwise And Operator &

The output of bitwise AND is 1 if the corresponding bits of two operands is 1. If either bit of an operand is 0, the result of corresponding bit is evaluated to 0

Bitwise And Operator &

The output of bitwise AND is 1 if the corresponding bits of two operands is 1. If either bit of an operand is 0, the result of corresponding bit is evaluated to 0

In C Programming, bitwise AND operator is denoted by &

Bitwise And Operator &

The output of bitwise AND is 1 if the corresponding bits of two operands is 1. If either bit of an operand is 0, the result of corresponding bit is evaluated to 0

In C Programming, bitwise AND operator is denoted by &

```
12 = 00001100 (In Binary)
25 = 00011001 (In Binary)
Bitwise AND of 12 and 25
  0000 1100
& 0001 1001
  _________
  0000 1000  = 8 (In decimal)
```

Bitwise And Operator &

The output of bitwise AND is 1 if the corresponding bits of two operands is 1. If either bit of an operand is 0, the result of corresponding bit is evaluated to 0

In C Programming, bitwise AND operator is denoted by &

```
12 = 00001100 (In Binary)
25 = 00011001 (In Binary)
Bitwise AND of 12 and 25
   0000 1100
& 0001 1001
  ___________
   0000 1000  = 8 (In decimal)
```

```c
#include <stdio.h>
int main(){
    int a = 12, b = 25;
    printf("Output = %d", a & b);
    return 0;
}
```

Bitwise OR Operator |

Bitwise OR Operator |

The output of bitwise OR is 1 if at least one corresponding bit of two operands is 1

Bitwise OR Operator |

The output of bitwise OR is 1 if at least one corresponding bit of two operands is 1

In C Programming, bitwise OR operator is denoted by |

Bitwise OR Operator |

The output of bitwise OR is 1 if at least one corresponding bit of two operands is 1

In C Programming, bitwise OR operator is denoted by |

```
12 = 00001100 (In Binary)
25 = 00011001 (In Binary)
Bitwise OR of 12 and 25
   0000 1100
|  0001 1001
   _________
   00011101  = 29 (In decimal)
```

Bitwise OR Operator |

The output of bitwise OR is 1 if at least one corresponding bit of two operands is 1

In C Programming, bitwise OR operator is denoted by |

```
12 = 00001100 (In Binary)
25 = 00011001 (In Binary)
Bitwise OR of 12 and 25
   0000 1100
|  0001 1001
   ________
   00011101  = 29 (In decimal)
```

```c
#include <stdio.h>
int main(){
    int a = 12, b = 25;
    printf("Output = %d", a | b);
    return 0;
}
```

Bitwise XOR Operator ^

Bitwise XOR Operator ^

The result of bitwise XOR operator is 1 if the corresponding bits of two operands are opposite i.e. one is 1 and the other is 0

Bitwise XOR Operator ^

The result of bitwise XOR operator is 1 if the corresponding bits of two operands are opposite i.e. one is 1 and the other is 0

In C Programming, bitwise XOR operator is denoted by ^

Bitwise XOR Operator ^

The result of bitwise XOR operator is 1 if the corresponding bits of two operands are opposite i.e. one is 1 and the other is 0

In C Programming, bitwise XOR operator is denoted by ^

```
12 = 00001100 (In Binary)
25 = 00011001 (In Binary)
Bitwise XOR of 12 and 25
   00001100
^ 00011001

  ________
  00010101  = 21 (In decimal)
```

Bitwise XOR Operator ^

The result of bitwise XOR operator is 1 if the corresponding bits of two operands are opposite i.e. one is 1 and the other is 0

In C Programming, bitwise XOR operator is denoted by ^

```
12 = 00001100 (In Binary)
25 = 00011001 (In Binary)
Bitwise XOR of 12 and 25
   00001100
^ 00011001
  ________
  00010101  = 21 (In decimal)
```

```c
#include <stdio.h>
int main(){
    int a = 12, b = 25;
    printf("Output = %d", a^b);
    return 0;
}
```

Bitwise Complement Operator ~

Bitwise Complement Operator ~

A unary operator that simply flips each bit of the input

Bitwise Complement Operator ~

A unary operator that simply flips each bit of the input

In C Programming, bitwise complement operator is denoted by ~

Bitwise Complement Operator ~

A unary operator that simply flips each bit of the input

In C Programming, bitwise complement operator is denoted by ~

```
12 = 0000 0000  0000 0000  0000 0000  0000 1100
Bitwise complement of 12 00001100
~ 0000 0000  0000 0000  0000 0000  0000 1100
  ________________________________________
  1111 1111   1111  1111    1111 1111   1111  0011
= -13 (decimal)
```

Bitwise Complement Operator ~

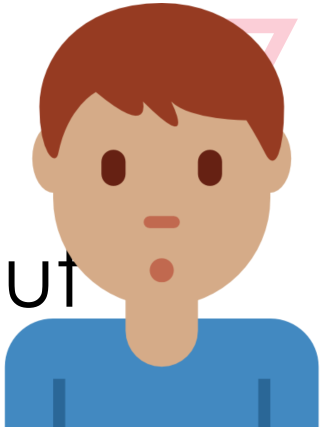
A unary operator that simply flips each bit of the input

In C Programming, bitwise complement operator is denoted by ~

```
12 = 0000 0000  0000 0000  0000 0000  0000 1100
Bitwise complement of 12 00001100
~ 0000 0000  0000 0000  0000 0000  0000 1100
  ___________________________________________
  1111 1111   1111  1111    1111 1111   1111  0011
= -13 (decimal)
```

```
#include <stdio.h>
int main(){
    int a = 12;
    printf("Output = %d", ~a);
    return 0;
}
```

Bitwise Complement Operator ~



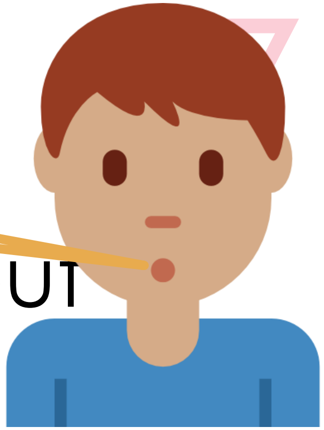
A unary operator that simply flips each bit of the input

In C Programming, bitwise complement operator is denoted by ~

```
12 = 0000 0000  0000 0000  0000 0000  0000 1100
Bitwise complement of 12 00001100
~ 0000 0000  0000 0000  0000 0000  0000 1100
  ___________________________________________
  1111 1111   1111  1111    1111 1111   1111  0011
= -13 (decimal)
```

```c
#include <stdio.h>
int main(){
    int a = 12;
    printf("Output = %d", ~a);
    return 0;
}
```

Bitwise Complement



A unary operator that simply flips each bit of the input

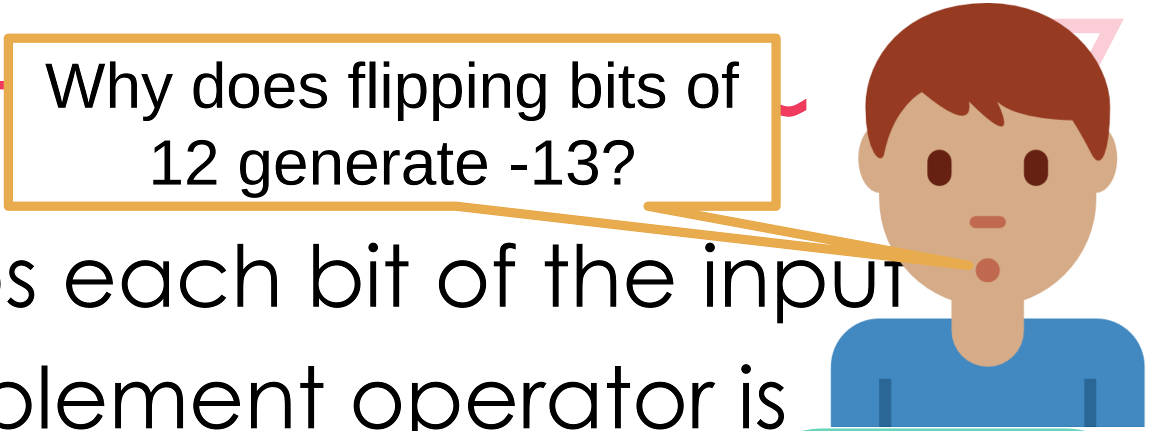
In C Programming, bitwise complement operator is denoted by ~

12 = 0000 0000 0000 0000 0000 0000 0000 1100
Bitwise complement of 12 00001100
~ 0000 0000 0000 0000 0000 0000 0000 1100

1111 1111   1111 1111   1111 1111   1111 0011
= -13 (decimal)

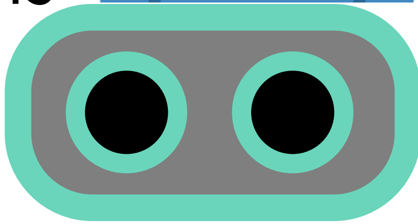
```
#include <stdio.h>
int main(){
    int a = 12;
    printf("Output = %d", ~a);
    return 0;
}
```

Bitwise Complement



A unary operator that simply flips each bit of the input

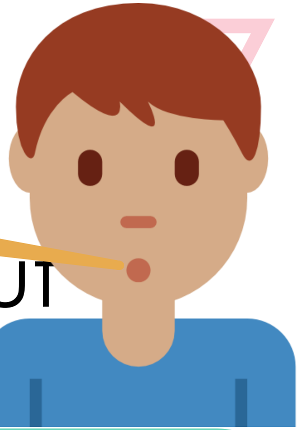
In C Programming, bitwise complement operator is denoted by ~



```
12 = 0000 0000  0000 0000  0000 0000  0000 1100
Bitwise complement of 12 00001100
~ 0000 0000  0000 0000  0000 0000  0000 1100
  ____________________________________________
  1111 1111   1111  1111    1111 1111   1111  0011
= -13 (decimal)
```

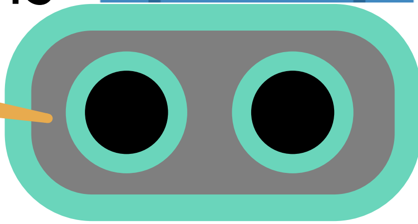
```c
#include <stdio.h>
int main(){
    int a = 12;
    printf("Output = %d", ~a);
    return 0;
}
```

Bitwise Complement



A unary operator that simply flips each bit of the input

In C Programming, bitw is
denoted by ~



12 = 0000 0000 0000 0000 0000 0000 0000 1100
Bitwise complement of 12 00001100
~ 0000 0000 0000 0000 0000 0000 0000 1100

1111 1111 1111 1111 1111 1111 1111 0011
= -13 (decimal)

```
#include <stdio.h>
int main(){
    int a = 12;
    printf("Output = %d", ~a);
    return 0;
}
```

Right Shift Operator >>

Right Shift Operator >>

Right shift operator shifts all bits towards right by a certain number of locations

Right Shift Operator >>

Right shift operator shifts all bits towards right by a certain number of locations

Bits that “fall off” from the right most end are lost

Right Shift Operator >>

Right shift operator shifts all bits towards right by a certain number of locations

Bits that "fall off" from the right most end are lost

Blank spaces in the leftmost positions are filled with 0s

Right Shift Operator >>

Right shift operator shifts all bits towards right by a certain number of locations

Bits that "fall off" from the right most end are lost

Blank spaces in the leftmost positions are filled with 0s

212 = 0000 0000   0000 0000   0000 0000   1101 0100

Right Shift Operator >>

Right shift operator shifts all bits towards right by a certain number of locations

Bits that "fall off" from the right most end are lost

Blank spaces in the leftmost positions are filled with 0s

212 = 0000 0000 0000 0000 0000 0000 1101 0100

212 >> 0 = 0000 0000 0000 0000 0000 0000 1101 0100

Right Shift Operator >>

Right shift operator shifts all bits towards right by a certain number of locations

Bits that "fall off" from the right most end are lost

Blank spaces in the leftmost positions are filled with 0s

212 = 0000 0000 0000 0000 0000 0000 1101 0100

212 >> 0 = 0000 0000 0000 0000 0000 0000 1101 0100

212 >> 4 = 0000 0000 0000 0000 0000 0000 0000 1101

Right Shift Operator >>

Right shift operator shifts all bits towards right by a certain number of locations

Bits that "fall off" from the right most end are lost

Blank spaces in the leftmost positions are filled with 0s

212 = 0000 0000 0000 0000 0000 0000 1101 0100

212 >> 0 = 0000 0000 0000 0000 0000 0000 1101 0100

212 >> 4 = 0000 0000 0000 0000 0000 0000 0000 1101

212 >> 6 = 0000 0000 0000 0000 0000 0000 0000 0011

Right Shift Operator >>

Right shift operator shifts all bits towards right by a certain number of locations

Bits that “fall off” from the right most end are lost

Blank spaces in the leftmost positions are filled with 0s

212 = 0000 0000   0000 0000   0000 0000   1101 0100

212 >> 0 = 0000 0000   0000 0000   0000 0000   1101 0100

212 >> 4 = 0000 0000   0000 0000   0000 0000   0000 1101

212 >> 6 = 0000 0000   0000 0000   0000 0000   0000 0011

212 >> 3 = 0000 0000   0000 0000   0000 0000   0001 1010

Right Shift Operator >>

Right shift operator shifts all bits towards right by a certain number of locations

Bits that "fall off" from the right most end are lost

Blank spaces in the leftmost positions are filled with 0s

212 = 0000 0000 0000 0000 0000 0000 1101 0100

212 >> 0 = 0000 0000 0000 0000 0000 0000 1101 0100

212 >> 4 = 0000 0000 0000 0000 0000 0000 0000 1101

212 >> 6 = 0000 0000 0000 0000 0000 0000 0000 0011

212 >> 3 = 0000 0000 0000 0000 0000 0000 0001 1010

Right shift by k is equivalent to integer division with 2^k ☺

Left Shift Operator <<

Left Shift Operator <<

Left shift operator shifts all bits towards left by a certain number of locations

Left Shift Operator <<

Left shift operator shifts all bits towards left by a certain number of locations

Bits that “fall off” from the left most end are lost

Left Shift Operator <<

Left shift operator shifts all bits towards left by a certain number of locations

Bits that “fall off” from the left most end are lost

Blank spaces in the right positions are filled with 0s

Left Shift Operator <<

Left shift operator shifts all bits towards left by a certain number of locations

Bits that "fall off" from the left most end are lost

Blank spaces in the right positions are filled with 0s

212 = 0000 0000   0000 0000   0000 0000   1101 0100

Left Shift Operator <<

Left shift operator shifts all bits towards left by a certain number of locations

Bits that “fall off” from the left most end are lost

Blank spaces in the right positions are filled with 0s

212 = 0000 0000   0000 0000   0000 0000   1101 0100

212 << 0 = 0000 0000   0000 0000   0000 0000   1101 0100

Left Shift Operator <<

Left shift operator shifts all bits towards left by a certain number of locations

Bits that "fall off" from the left most end are lost

Blank spaces in the right positions are filled with 0s

212 = 0000 0000 0000 0000 0000 0000 1101 0100

212 << 0 = 0000 0000 0000 0000 0000 0000 1101 0100

212 << 4 = 0000 0000 0000 0000 0000 1101 0100 0000

Left Shift Operator <<

Left shift operator shifts all bits towards left by a certain number of locations

Bits that “fall off” from the left most end are lost

Blank spaces in the right positions are filled with 0s

212 = 0000 0000 0000 0000 0000 0000 1101 0100

212 << 0 = 0000 0000 0000 0000 0000 0000 1101 0100

212 << 4 = 0000 0000 0000 0000 0000 1101 0100 0000

212 << 6 = 0000 0000 0000 0000 0011 0101 0000 0000

Left Shift Operator <<

Left shift operator shifts all bits towards left by a certain number of locations

Bits that “fall off” from the left most end are lost

Blank spaces in the right positions are filled with 0s

212 = 0000 0000 0000 0000 0000 0000 1101 0100

212 << 0 = 0000 0000 0000 0000 0000 0000 1101 0100

212 << 4 = 0000 0000 0000 0000 0000 1101 0100 0000

212 << 6 = 0000 0000 0000 0000 0011 0101 0000 0000

212 << 28 = 0100 0000 0000 0000 0000 0000 0000 0000

Left Shift Operator <<

Left shift operator shifts all bits towards left by a certain number of locations

Bits that “fall off” from the left most end are lost

Blank spaces in the right positions are filled with 0s

212 = 0000 0000   0000 0000   0000 0000   1101 0100

212 << 0 = 0000 0000   0000 0000   0000 0000   1101 0100

212 << 4 = 0000 0000   0000 0000   0000 1101   0100 0000

212 << 6 = 0000 0000   0000 0000   0011 0101   0000 0000

212 << 28 = 0100 0000   0000 0000   0000 0000   0000 0000

Left shift by k is equivalent to integer multiplication with 2^k

Left Shift Operator <<

Left shift operator shifts all bits towards left by a certain number of locations

Bits that "fall off" from the left most end are lost

Blank spaces in the right positions are filled with 0s

212 = 0000 0000 0000 0000 0000 0000 1101 0100

212 << 0 = 0000 0000 0000 0000 0000 0000 1101 0100

212 << 4 = 0000 0000 0000 0000 0000 1101 0100 0000

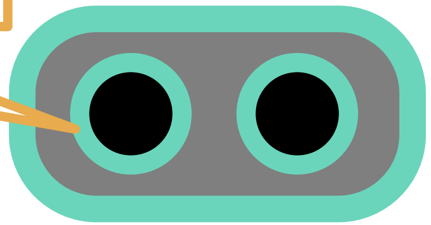
212 << 6 = 0000 0000 0000 0000 0011 0101 0000 0000

212 << 28 = 0100 0000 0000 0000 0000 0000 0000 0000

Left shift by k is equivalent to integer multiplication with 2^k

Left Shift Op

However, be warned. If you left shift too much, multiplication with 2^k may give you a number too large to be represented by an int. You may get garbage results.



Left shift operator … certain number of locations

Bits that “fall off” from the left most end are lost

Blank spaces in the right positions are filled with 0s

212 = 0000 0000 0000 0000 0000 0000 1101 0100

212 << 0 = 0000 0000 0000 0000 0000 0000 1101 0100

212 << 4 = 0000 0000 0000 0000 0000 1101 0100 0000

212 << 6 = 0000 0000 0000 0000 0011 0101 0000 0000

212 << 28 = 0100 0000 0000 0000 0000 0000 0000 0000

Left shift by k is equivalent to integer multiplication with 2^k

Example use of bitwise operators

Example use of bitwise operators

Can use "masks" to extract certain bits of a number

Example use of bitwise operators

Can use “masks” to extract certain bits of a number

Suppose I want to look at the last 6 bits of a number a

Example use of bitwise operators

Can use “masks” to extract certain bits of a number

Suppose I want to look at the last 6 bits of a number a

Create a mask with only last bits set to 1 and take & with a

Example use of bitwise operators

Can use “masks” to extract certain bits of a number

Suppose I want to look at the last 6 bits of a number a

Create a mask with only last bits set to 1 and take & with a

```
int a = 427;
int p = 1;
int q = p << 6;
int m = q – 1;
int r = a & m;
printf("%d", r); // 43
```

Example use of bitwise operators

Can use "masks" to extract certain bits of a number

Suppose I want to look at the last 6 bits of a number a

Create a mask with only last bits set to 1 and take & with a

```
a = 0000 0000  0000 0000  0000 0001  1010 1011
p = 0000 0000  0000 0000  0000 0000  0000 0001
q = 0000 0000  0000 0000  0000 0000  0100 0000
m = 0000 0000  0000 0000  0000 0000  0011 1111
r =  0000 0000  0000 0000  0000 0000  0010 1011
```

```
int a = 427;
int p = 1;
int q = p << 6;
int m = q – 1;
int r = a & m;
printf("%d", r); // 43
```

BODMAS table has more members

Operator Name	Symbol/Sign	Associativity
Brackets (array subscript), Post increment/decrement, structure field selection (dot and arrow)	(), [] ++, --, ., ->	Left
Unary negation, Pre-increment/decrement, NOT, (de)reference, sizeof, bitwise complement	-, ++, --, !, *, &, sizeof, ~	Right
Multiplication/division/remainder	*, /, %	Left
Addition/subtraction	+, -	Left
Bitwise left shift and right shift	<<, >>	
Relational	<, <=, >, >=	Left
Relational	==, !=	Left
Bitwise AND	&	Left
Bitwise XOR	^	Left
Bitwise OR	|	Left
Logical AND	&&	Left
Logical OR	||	Left
Ternary Conditional	? :	Right
Assignment, Compound assignment	=, +=, -=, *=, /=, %=, &=, ^=, |=, <<=, >>=	Right

Operator Name	Symbol/Sign	Associativity
Brackets (array subscript), Post increment/decrement, structure field selection (dot and arrow)	(), [] ++, --, ., ->	Left
Unary negation, Pre-increment/decrement, NOT, (de)reference, sizeof, bitwise complement	-, ++, --, !, *, &, sizeof, ~	Right
Multiplication/division/remainder	*, /, %	Left
Addition/subtraction	+, -	Left
Bitwise left shift and right shift	<<, >>	
Relational	<, <=, >, >=	Left
Relational	==, !=	Left
Bitwise AND	&	Left
Bitwise XOR	^	Left
Bitwise OR	|	Left
Logical AND	&&	Left
Logical OR	| |	Left
Ternary Conditional	? :	Right
Assignment, Compound assignment	=, +=, -=, *=, /=, %=, &=, ^=, |=, <<=, >>=	Right

HIGH

LOW

One's Compliment

One’s Compliment

The one’s compliment of a binary number) is simply the bitwise compliment of that binary number

One’s Compliment

The one’s compliment of a binary number) is simply the bitwise compliment of that binary number

A long time ago (25-30 years ago) one’s compliment used to be used to represent negative numbers

One's Compliment

The one's compliment of a binary number) is simply the bitwise compliment of that binary number

A long time ago (25-30 years ago) one's compliment used to be used to represent negative numbers

35 as a 4 byte int is represented as
0000 0000   0000 0000   0000 0000   0010 0011

One's Compliment

The one's compliment of a binary number) is simply the bitwise compliment of that binary number

A long time ago (25-30 years ago) one's compliment used to be used to represent negative numbers

35 as a 4 byte int is represented as
0000 0000   0000 0000   0000 0000   0010 0011

So, in those old computers, -35 used to be represented as
1111 1111   1111 1111   1111 1111   1101 1100

One’s Compliment

The one’s compliment of a binary number) is simply the bitwise compliment of that binary number

A long time ago (25-30 years ago) one’s compliment used to be used to represent negative numbers

35 as a 4 byte int is represented as
0000 0000 0000 0000 0000 0000 0010 0011

So, in those old computers, -35 used to be represented as
1111 1111 1111 1111 1111 1111 1101 1100

Note that b + ~b = 11111111 11111111 11111111 11111111

One’s Compliment

The[illegible] bit[illegible]

If we have n bits, then using one’s compliment, we can represent numbers between $-(2^{n-1} - 1)$ and $+(2^{n-1} - 1)$

A long time ago (25-30 years ago) one’s compliment used to be used to represent negative numbers

35 as a 4 byte int is represented as
0000 0000 0000 0000 0000 0000 0010 0011

So, in those old computers, -35 used to be represented as
1111 1111 1111 1111 1111 1111 1101 1100

Note that b + ~b = 11111111 11111111 11111111 11111111

One's Compliment

Th
bit

If we have n bits, then using one's compliment, we can represent numbers between $-(2^{n-1} - 1)$ and $+(2^{n-1} - 1)$

The first bit acts as a sign bit – if the first bit is 1, it is treated as a negative number, if the first bit is 0, it is treated as a positive number

0000 0000 0000 0000 0000 0000 0010 0011

So, in those old computers, -35 used to be represented as
1111 1111 1111 1111 1111 1111 1101 1100

Note that b + ~b = 11111111 11111111 11111111 11111111

One’s Compliment

If we have n bits, then using one’s compliment, we can represent numbers between $-(2^{n-1} - 1)$ and $+(2^{n-1} - 1)$

The first bit acts as a sign bit – if the first bit is 1, it is treated as a negative number, if the first bit is 0, it is treated as a positive number

Largest positive number is 01111111 11111111 11111111 11111111
Smallest negative number is 1000000 00000000 00000000 00000000
Weird thing – negative 0 ☺
11111111 11111111 11111111 11111111

One’s Compliment

If we have n bits, then using one’s compliment, we can represent numbers between $-(2^{n-1} - 1)$ and $+(2^{n-1} - 1)$

The first bit acts as a sign bit – if the first bit is 1, it is treated as a negative number, if the first bit is 0, it is treated as a positive number

Largest positive number is 01111111 11111111 11111111 11111111
Smallest negative number is 1000000 00000000 00000000 00000000
Weird thing – negative 0 ☺
11111111 11111111 11111111 11111111

Used no more. These days, computers use two’s compliment to represent negative numbers

Two's Compliment

Two's Compliment

Advantages over one's compliment

Two's Compliment

Advantages over one's compliment

- Arithmetic circuits inside microprocessors do not need to look at sign bits

Two's Compliment

Advantages over one's compliment

- Arithmetic circuits inside microprocessors do not need to look at sign bits
- There is no weirdness about positive zero and negative zero

Two's Compliment

Advantages over one's compliment

- Arithmetic circuits inside microprocessors do not need to look at sign bits
- There is no weirdness about positive zero and negative zero

Two's compliment of an n-bit binary number is the number which when added to this number, gives 2^n

Two’s Compliment

Advantages over one’s compliment

- Arithmetic circuits inside microprocessors do not need to look at sign bits
- There is no weirdness about positive zero and negative zero

Two’s compliment of an n-bit binary number is the number which when added to this number, gives 2^n

This means two’s compliment of b is $2^n – b$

Two's Compliment

Advantages over one's compliment

- Arithmetic circuits inside microprocessors do not need to look at sign bits
- There is no weirdness about positive zero and negative zero

Two's compliment of an n-bit binary number is the number which when added to this number, gives 2^n

This means two's compliment of b is $2^n - b$

Recall that $b + \sim b$ = all ones = $2^n - 1$ i.e. two's compliment of b is $2^n - b = \sim b + 1$

Two's Compliment

- Advantages over one's compliment
  - Arithmetic circuits inside microprocessors do not need to look at sign bits
  - There is no weirdness about positive zero and negative zero
- Two's compliment of an n-bit binary number is the number which when added to this number, gives 2^n
- This means two's compliment of b is $2^n - b$
- Recall that $b + \sim b$ = all ones = $2^n - 1$ i.e. two's compliment of b is $2^n - b = \sim b + 1$
- Easier way of calculating two's compliment – take the one's compliment and add 1 to the binary string ☺

Two's Compliment

- Advantages over one's compliment
  - Arithmetic circuits inside microprocessors do not need to look at sign bits
  - There is no weirdness about positive zero and negative zero
- Two's compliment of an n-bit binary number is the number which when added to this number, gives 2^n
- This means two's compliment of b is $2^n – b$
- Recall that $b + \sim b$ = all ones = $2^n – 1$ i.e. two's compliment of b is $2^n – b = \sim b + 1$
- Easier way of calculating two's compliment – take the one's compliment and add 1 to the binary string ☺
- So for any integer n, bitwise complement of n will be -(n+1)

Two’s Compliment

These days, the two’s complement of a number represents the negative of that number e.g. -9 is represented as two’s complement of 9

There is no weirdness about positive zero and negative zero

Two’s compliment of an n-bit binary number is the number which when added to this number, gives 2^n

This means two’s compliment of b is $2^n – b$

Recall that $b + \sim b$ = all ones = $2^n – 1$ i.e. two’s compliment of b is $2^n – b = \sim b + 1$

Easier way of calculating two’s compliment – take the one’s compliment and add 1 to the binary string ☺

So for any integer n, bitwise complement of n will be -(n+1)

Two's Compliment

These days, the two's complement of a number represents the negative of that number e.g. -9 is represented as two's complement of 9

If we have n bits, then using one's compliment, we can represent numbers between -2^{n-1} and $+(2^{n-1} - 1)$

This means two's compliment of b is $2^n - b$

Recall that $b + \sim b$ = all ones = $2^n - 1$ i.e. two's compliment of b is $2^n - b = \sim b + 1$

Easier way of calculating two's compliment – take the one's compliment and add 1 to the binary string ☺

So for any integer n, bitwise complement of n will be -(n+1)

Two's Compliment

These days, the two's complement of a number represents the negative of that number e.g. -9 is represented as two's complement of 9

If we have n bits, then using one's compliment, we can represent numbers between -2^{n-1} and $+(2^{n-1} - 1)$

Tw ... nber
wh

The first bit acts as a sign bit – if the first bit is 1, it is treated as a negative number, if the first bit is 0, it is treated as a positive number

of b is $2^n - b = \sim b + 1$

Easier way of calculating two's compliment – take the one's compliment and add 1 to the binary string ☺

So for any integer n, bitwise complement of n will be -(n+1)

Two's Compliment

These days, the two's complement of a number represents the negative of that number e.g. -9 is represented as two's complement of 9

If we have n bits, then using one's compliment, we can represent numbers between -2^{n-1} and $+(2^{n-1} - 1)$

The first bit acts as a sign bit – if the first bit is 1, it is treated as a negative number, if the first bit is 0, it is treated as a positive number

Largest positive number is 01111111 11111111 11111111 11111111
Smallest negative number is 1000000 00000000 00000000 00000000
11111111 11111111 11111111 11111111 now represents

Two's Compliment

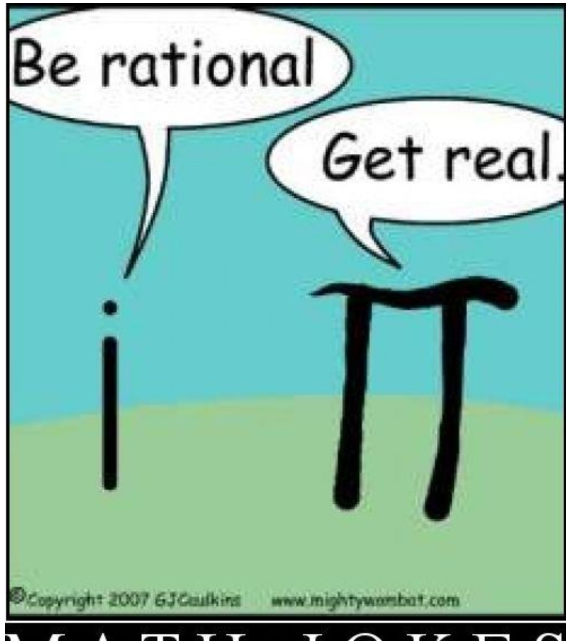
These days, the two's complement of a number represents the negative of that number e.g. -9 is represented as two's complement of 9

If we have n bits, then using one's compliment, we can represent numbers between -2^{n-1} and $+(2^{n-1} - 1)$

The first bit acts as a sign bit – if the first bit is 1, it is treated as a negative number, if the first bit is 0, it is treated as a positive number

Largest positive number is 01111111 11111111 11111111 11111111
Smallest negative number is 1000000 00000000 00000000 00000000
11111111 11111111 11111111 11111111 now represents -1 ☺

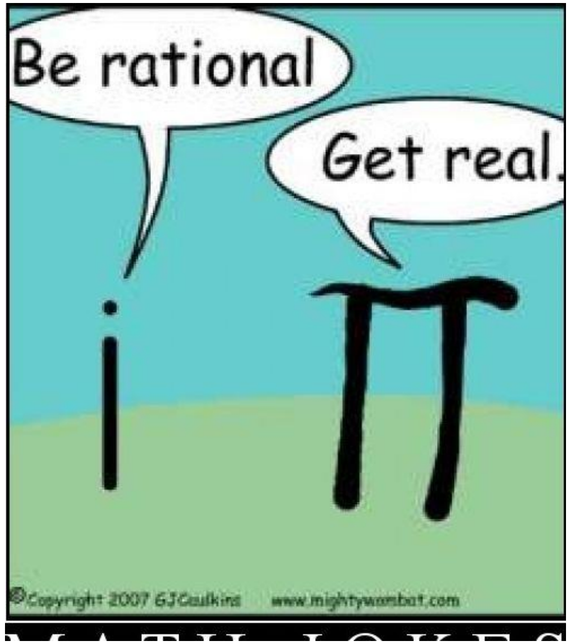
Floating Point Representation



Floating Point Representation

Have to represent three things

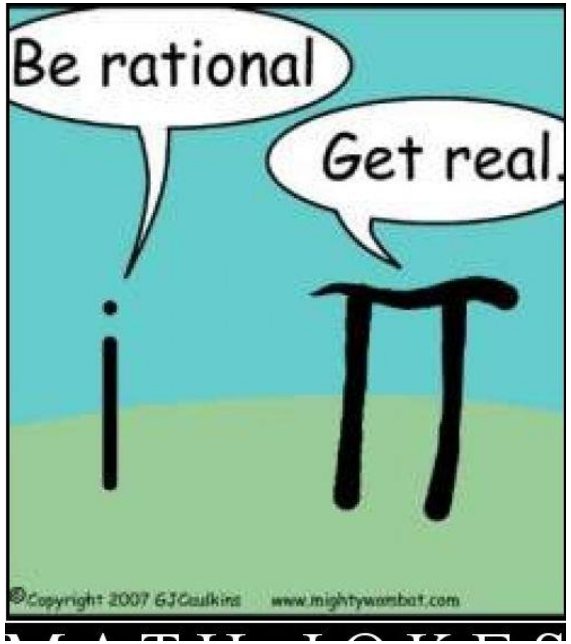
- Sign: uses 1 bit
- Mantissa: uses 23 bits
- Exponent: uses 8 bits



Floating Point Representation

Have to represent three things

- Sign: uses 1 bit
- Mantissa: uses 23 bits
- Exponent: uses 8 bits



Base 2: d_i 's are binary digits 0 or 1.

$$\pm d_0.d_1d_2\cdots d_{p-1}\times 2^e$$

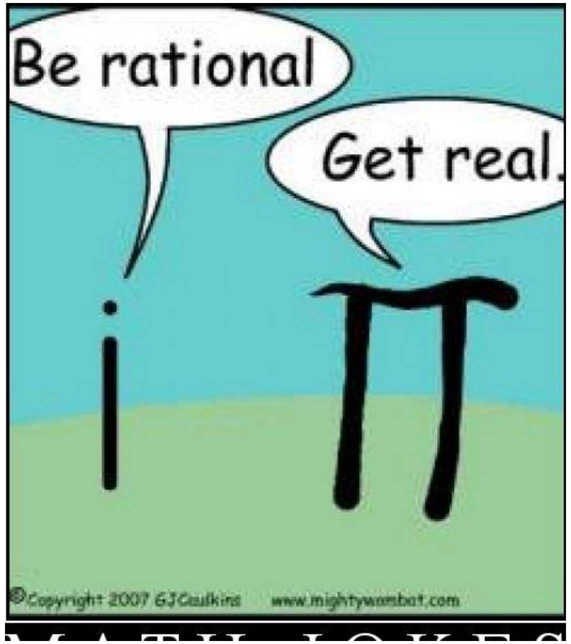
Its value is

$$\pm(d_0+d_1 2^{-1}+d_2{}^{-2}+\cdots+d_{p-1}2^{-(p-1)})2^e$$

Floating Point Representation

Have to represent three things

- Sign: uses 1 bit
- Mantissa: uses 23 bits
- Exponent: uses 8 bits



Base 2: d_i 's are binary digits 0 or 1.

$$\pm d_0.d_1 d_2 \cdots d_{p-1} \times 2^e$$

Its value is

$$\pm(d_0 + d_1 2^{-1} + d_2{}^{-2} + \cdots + d_{p-1} 2^{-(p-1)})2^e$$

1. p is called the precision of the representation.
2. Here base of the representation is 2. In general, the base could be different from 2. e.g., scientific calculators typically use base 10.

IEEE 754 Floating Point Representation

IEEE 754 Floating Point Standard

s	e=exponent	m=mantissa
1 bit	8 bits	23 bits

$$\text{number} = (-1)^{s} * (1.m) * 2^{e-127}$$

Range of Representation

- Base 2: d_i 's are binary digits 0 or 1.

$$\pm d_0.d_1d_2 \cdots d_{p-1} \times 2^e$$

- E.g., for number $(0.1)_{10}$ (i.e., 0.1 in decimal), p =24, the representation is

  $(1.10011001100110011001100) \text{ X } 2^{-4}$

1/16 + 1/32 + 1/256 + 1/512 = 51/512, ….

$(0.1)_{10}$ is not "exactly" represented in binary.

Real numbers cannot be exactly represented with finite precision and limited exponent.

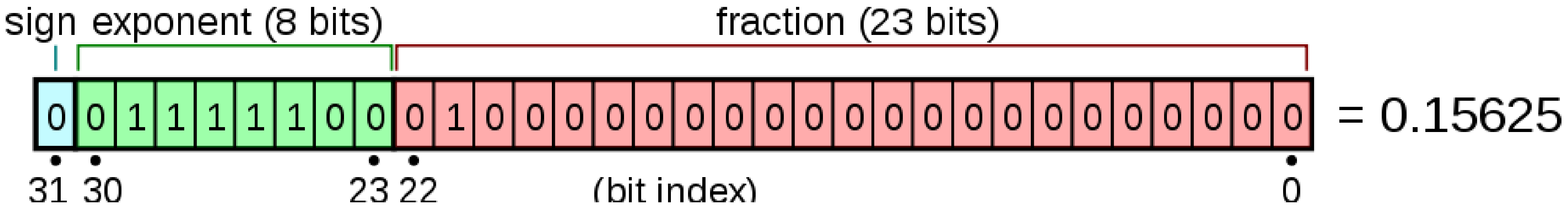
Single-precision (float)

0	0110 1000	101 0101 0100 0011 0100 0010

- Sign: 0 => positive
- Exponent:
  - $0110\ 1000_{two} = 104_{ten}$
  - Bias adjustment: 104 - 127 = -23
- Significand:
  - $1 + 1x2^{-1} + 0x2^{-2} + 1x2^{-3} + 0x2^{-4} + 1x2^{-5} + \ldots$
    $=1+2^{-1}+2^{-3}+2^{-5}+2^{-7}+2^{-9}+2^{-14}+2^{-15}+2^{-17}+2^{-22}$
    $= 1.0 + 0.666115$
- Represents: $1.666115{*}2^{-23} \sim 1.986{*}10^{-7}$

This is what you're using when you are invoking *float*

Imprecise Representation



Next number: $(1 + \frac{1}{4} + 2^{-23})2^{124-127} = 0.15625+2^{-26}$

0	0111100	01000000000000000000001

Given number: 0.15625

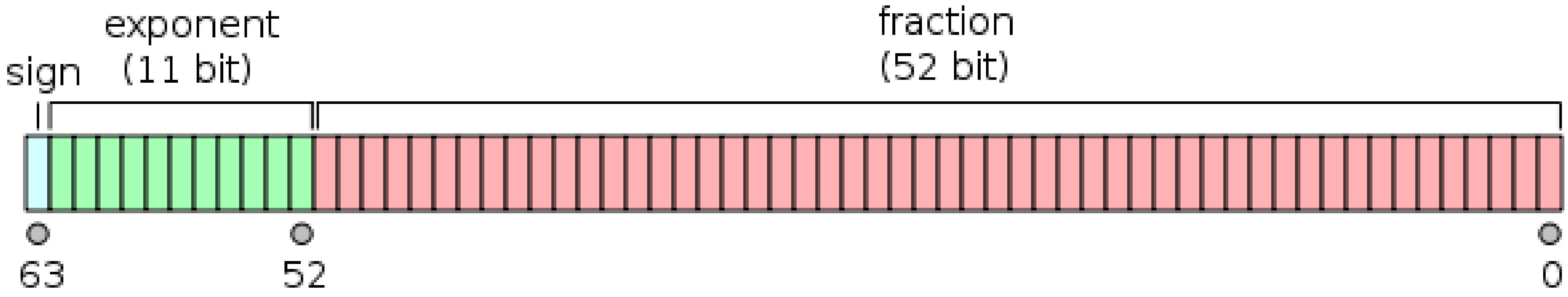
Previous number: $(1+1/8+1/16+\ldots 2^{-23})2^{-3}$
$= (1+1/4-2^{-23})2^{-3} = 0.15625-2^{-26}$

0	0111100	00111111111111111111111

Real numbers in-between 0.15625 and $0.15625 + 2^{-26}$ may be all represented as 0.15625.
Discrete representation: leads to approximation errors.

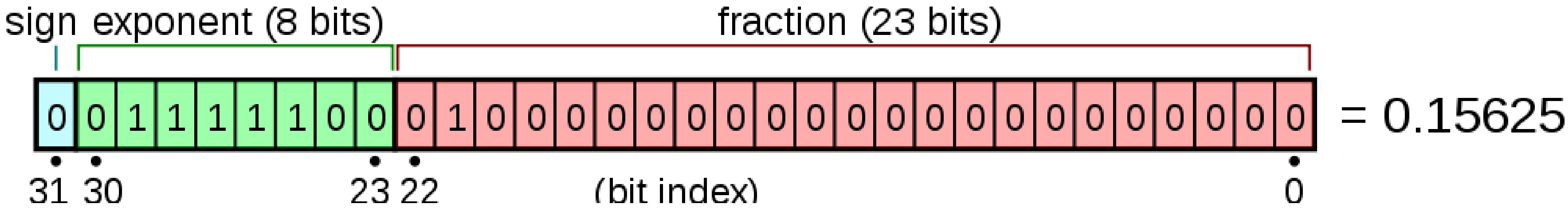
Double Precision (double)

Same logic as single precision, but with 64 bits of memory



Underflow and Overflow

Consider IEEE single precision.



Numbers smaller than 2^{-128} are indistinguishable from 0.

Such numbers occurring in calculations are said to be in underflow and are treated as 0.

Numbers that are larger than 2^{128} cannot be represented.

Such numbers occurring in calculations are said to have overflowed.

Warning: IEEE representation allows for +0.00 and –0.00. Be careful.