

# Negative Bit Representation Outline

1. Negative Bit Representation Outline
2. Negative Integers
3. Representing Negativity #1
4. Representing Negativity #2
5. Which Bit for the Sign?
6. Sign-Value
7. Disadvantages of Sign-Value
8. One's Complement
9. Disadvantages of One's Complement
10. Two's Complement
11. Advantages of Two's Complement #1
12. Advantages of Two's Complement #2
13. Advantages of Two's Complement #3
14. Advantages of Two's Complement #4
15. Advantages of Two's Complement #5
16. Range of Two's Complement Values #1
17. Range of Two's Complement Values #2
18. Range of Two's Complement Values #3
19. Range of Two's Complement Values #4
20. Range of Two's Complement Values #5
21. Overflow #1
22. Overflow #2
23. Underflow #1
24. Underflow #2
25. Overflow Example #1
26. Overflow Example #2



# Negative Integers

In the first slide packet on binary representation, we saw how non-negative integer values like 97 are represented in memory.

What if, instead of having 97, we had -97?

We need a way to represent negative integers.



# Representing Negativity #1

For starters, we need a way to represent whether an integer is negative or positive or zero.

We can think of this as a binary question:

a number is either negative or **non-negative**.

So, we could add an extra bit to every byte in RAM, to represent the sign (negative or non-negative).

Of course, then we'd have 9-bit bytes, and computer hardware much prefers numbers of components that are powers of 2.

Plus, we'd need extra circuitry in our CPUs to properly do the calculations with negative numbers, which would cost more.

So, this isn't a good solution.



# Representing Negativity #2

For starters, we need a way to represent whether an integer is negative or positive or zero.

We can think of this as a binary question: a number is either negative or **non-negative**.

We've already decided that we don't want to have an additional bit whose only role is to represent the sign.

So, we can simply pick a bit in the binary representation of the integer and decide that it's going to be the sign bit.

Which bit should we pick?



# Which Bit for the Sign?

Which bit should we pick to represent the sign?

Well, we want to pick the bit that we're least likely to use in real life, so that it's not a big waste to use it as a sign bit.

In real life, we're much more likely to deal with very small numbers (for example, 0, 1, 13, 97) than very large numbers (for example, 4001431453).

So, we pick the leftmost bit, called the **most significant bit**, and decide that it'll be our sign bit – because that's the bit that we'll miss the least.



# Sign-Value

Okay, now we have our sign bit.

Here is the first of three ways to represent negative integers.

1. **Sign-Value**: To get the negative version of a positive number, set the sign bit to 1, and leave all the other bits unchanged:

$$97 = \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|} \hline 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ \hline \end{array}$$

$$-97 = \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|} \hline 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ \hline \end{array}$$



# Disadvantages of Sign-Value

An unfortunate feature of Sign-Value representation is that there are two ways of representing the value zero: all bits set to zero, and all bits except the sign bit set to zero.

0 =	<b>0</b>	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0 =	<b>1</b>	0	0	0	0	0	0	0	0	0	0	0	0	0	0

This makes the math a bit confusing.

More importantly, when performing arithmetic operations, we need to treat negative operands as special cases.

So, our CPU needs a lot more circuitry to do arithmetic, which can be expensive.



# One's Complement

2. **One's Complement**: To get the negative version of a positive number, invert (**complement**) all bits; that is, all 1's become 0's and vice versa.

$$97 = \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|} \hline 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ \hline \end{array}$$

$$-97 = \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|} \hline 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\ \hline \end{array}$$





# Disadvantages of One's Complement

An unfortunate feature of One's Complement representation is that there are two ways of representing the value zero: all bits set to zero, and all bits set to one.

0 = 

0	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
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

This makes the math a bit confusing.

More importantly, when performing arithmetic operations, we need to treat negative operands as special cases.

So, our CPU needs a lot more circuitry to do arithmetic, which can be expensive.



# Two's Complement

3. **Two's Complement**: To get the negative version of a positive number, invert all bits **and then add 1**.  
If the addition causes a **carry** bit past the most significant bit, then discard the high carry:

$$97 = \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|} \hline 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ \hline \end{array}$$

$$\begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|} \hline \sim 97 = 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\ \hline & & & & & & & & & & & & & & & + 1 \\ \hline \end{array}$$

$$-97 = \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|} \hline 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ \hline \end{array}$$



# Advantages of Two's Complement #1

In Two's Complement representation, the value zero is uniquely represented by having all bits set to zero:

$$0 = \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|} \hline 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline \end{array}$$

$$\sim 0 = \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|} \hline 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ \hline \end{array}$$

~~X~~ Discard the high carry. + 1

$$-0 = \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|} \hline 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline \end{array}$$



# Advantages of Two's Complement #2

In Two's Complement representation, when you perform an arithmetic operation (for example, addition, subtraction, multiplication, division) on two signed integers, you can use **exactly the same method** as if you had two unsigned integers (that is, non-negative integers with no sign bit) ...

... **EXCEPT**, you throw away the high carry (or the high **borrow** for subtraction).

So, the only extra capabilities you need are:

- a. flipping all the bits (which it turns out you need for other things);
- b. throwing away the high carry (or the high borrow).

That's **MUCH CHEAPER** than the extra circuitry needed for Sign-Magnitude or One's Complement.



# Advantages of Two's Complement #3

This property of Two's Complement representation is so incredibly handy that virtually every general-purpose computer available today uses Two's Complement.

Why?

Because, with Two's Complement, we don't need special algorithms (and therefore extra circuitry) for arithmetic operations that involve negative values.



# Advantages of Two's Complement #4

Using Two's Complement to do arithmetic:

97 = 

0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

-97 = 

1	1	1	1	1	1	1	1	1	1	0	0	1	1	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---



0 = 

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---



# Advantages of Two's Complement #5

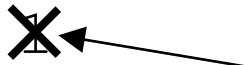
Using Two's Complement to do arithmetic:

45 = 

0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

-14 = 

1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---



31 = 

0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---



# Range of Two's Complement Values #1

When we represent negative integers in Two's Complement notation, the range of numbers that can be represented in  $b$  bits is:

$$-(2^{b-1}) \dots (+2^{b-1} - 1)$$

For example, the range of numbers that can be represented in 8 bits is:

$$-(2^7) \dots (+2^7 - 1) = -128 \dots 127$$

Likewise, the range of numbers that can be represented in 16 bits is:

$$-(2^{15}) \dots (+2^{15} - 1) = -32,768 \dots 32,767$$

How do we know this?





# Range of Two's Complement Values #2

When we represent negative integers in Two's Complement notation, the range of numbers that can be represented in  $b$  bits is:

$$-(2^{b-1}) \dots (+2^{b-1} - 1)$$

How do we know this?

Here's the biggest number that can be represented in 16 bits:

0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

If we add one to it, we get:

1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

But this number is negative (that is, it has a 1 in the sign bit).

So, mathematically, it's smaller than the previous number, not bigger, and therefore the previous number is the biggest.



# Range of Two's Complement Values #3

Here's the biggest number that can be represented in 16 bits:

0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

If we add one to it, we get:

1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

But this number is negative.

If this number were “unsigned”, then it'd be  $2^{15}$ , which is  $2^{16-1}$ , which is  $2^{b-1}$ , where  $b$  (the number of bits) is 16.

Therefore, the largest number that can be represented in  $b$  bits in Two's Complement must be  $2^{b-1}-1$ .



# Range of Two's Complement Values #4

So what's the smallest negative integer (that is, the negative integer with the greatest absolute value)?

Well, can we represent the negative of  $2^{b-1} - 1$ ?

$$+(2^{b-1} - 1) = \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|} \hline 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ \hline \end{array}$$

$$\sim(2^{b-1} - 1) = \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|} \hline 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline \end{array}$$

+ 1

$$-(2^{b-1} - 1) = \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|} \hline 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ \hline \end{array}$$



# Range of Two's Complement Values #5

We can represent  $-(2^{b-1} - 1)$ .

So, can we represent the negative of  $-(2^{b-1})$ ?

$$-(2^{b-1} - 1) = \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|} \hline 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ \hline \end{array} - 1$$

$$-(2^{b-1}) = \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|} \hline 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline \end{array}$$

Since the sign didn't change, it worked!

But, if we tried subtracting again, we'd borrow and the sign would change, indicating failure.

So,  $-(2^{b-1})$  is the lowest number that can be represented (that is, the negative number with the highest absolute value).



# Overflow #1

When we're working with a value that's near the upper limit of what can be represented in Two's Complement for the given number of bits, we sometimes perform an operation that should result in a positive value but instead produces a negative value.

Such an event is called *overflow*.



# Overflow #2

Consider the following addition in 8-bit Two's Complement representation:

$$\begin{array}{r} +127 = \boxed{0} \boxed{1} \boxed{1} \boxed{1} \boxed{1} \boxed{1} \boxed{1} \boxed{1} \\ + 1 \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad + 1 \\ \hline +128? = \boxed{1} \boxed{0} \boxed{0} \boxed{0} \boxed{0} \boxed{0} \boxed{0} \boxed{0} \end{array}$$

Notice that the result should be +128,  
but because the leftmost bit is 1, it's actually -128.  
This is **overflow**: an arithmetic operation that should have  
a positive result goes over the top to become negative.



# Underflow #1

When we're working with a value that's near the lower limit of what can be represented in Two's Complement for the given number of bits, we sometimes perform an operation that should result in a negative value but instead produces a positive value.

Such an event is called *underflow*.



## Underflow #2

Consider the following addition in 8-bit Two's Complement representation:

$$\begin{array}{r} -128 = \boxed{1} \boxed{0} \boxed{0} \boxed{0} \boxed{0} \boxed{0} \boxed{0} \boxed{0} \\ - 1 \phantom{=} \phantom{=} \phantom{=} \phantom{=} \phantom{=} \phantom{=} \phantom{=} \boxed{1} \\ \hline -129? = \boxed{0} \boxed{1} \boxed{1} \boxed{1} \boxed{1} \boxed{1} \boxed{1} \boxed{1} \end{array}$$

Notice that the result should be -129,  
but because the leftmost bit is 1, it's actually +127.  
This is **underflow**: an arithmetic operation that should have  
a negative result goes under the bottom to become positive.





# Overflow Example

“17. Inserted debug statements into anomdtb.f90, discovered that a sum-of-squared variable is becoming very, very negative! Key output from the debug statements:

```
OpEn= 16.00, OpTotSq= 4142182.00, OpTot= 7126.00
DataA val = 93, OpTotSq= 8649.00
DataA val = 172, OpTotSq= 38233.00
DataA val = 950, OpTotSq= 940733.00
DataA val = 797, OpTotSq= 1575942.00
DataA val = 293, OpTotSq= 1661791.00
DataA val = 83, OpTotSq= 1668680.00
DataA val = 860, OpTotSq= 2408280.00
DataA val = 222, OpTotSq= 2457564.00
DataA val = 452, OpTotSq= 2661868.00
DataA val = 561, OpTotSq= 2976589.00
DataA val = 49920, OpTotSq=-1799984256.00
DataA val = 547, OpTotSq=-1799684992.00
DataA val = 672, OpTotSq=-1799233408.00
DataA val = 710, OpTotSq=-1798729344.00
DataA val = 211, OpTotSq=-1798684800.00
DataA val = 403, OpTotSq=-1798522368.00
OpEn= 16.00, OpTotSq=-1798522368.00,
OpTot=56946.00
forrtl: error (75): floating point exception
IOT trap (core dumped)”
```

“..so the data value is unfeasibly large, but why does the sum-of-squares parameter OpTotSq go negative?!!

Probable answer: the high value is pushing beyond the single-precision default for Fortran reals?”

<https://gist.github.com/markmarino/9e1afcad683e7832ad38d149ba157feb>



# Overflow Example (cont'd)

“17. Inserted debug statements into anomdtb.f90, discovered that a sum-of-squared variable is becoming very, very negative! Key output from the debug statements:

```
OpEn= 16.00, OpTotSq= 4142182.00, OpTot= 7126.00
DataA val = 93, OpTotSq= 8649.00
DataA val = 172, OpTotSq= 38233.00
DataA val = 950, OpTotSq= 940733.00
DataA val = 797, OpTotSq= 1575942.00
DataA val = 293, OpTotSq= 1661791.00
DataA val = 83, OpTotSq= 1668680.00
DataA val = 860, OpTotSq= 2408280.00
DataA val = 222, OpTotSq= 2457564.00
DataA val = 452, OpTotSq= 2661868.00
DataA val = 561, OpTotSq= 2976589.00
DataA val = 49920, OpTotSq=-1799984256.00
DataA val = 547, OpTotSq=-1799684992.00
DataA val = 672, OpTotSq=-1799233408.00
DataA val = 710, OpTotSq=-1798729344.00
DataA val = 211, OpTotSq=-1798684800.00
DataA val = 403, OpTotSq=-1798522368.00
OpEn= 16.00, OpTotSq=-1798522368.00,
OpTot=56946.00
fortrl: error (75): floating point exception
IOT trap (core dumped)”
```

“..so the data value is unfeasibly large, but why does the sum-of-squares parameter OpTotSq go negative?!!

Probable answer: the high value is pushing beyond the single-precision default for Fortran reals?”

## Likely actual answer:

The code used an integer to do the arithmetic, then multiplied by a floating point number before outputting -- but it got an overflow.

<https://gist.github.com/markmarino/9e1afcad683e7832ad38d149ba157feb>

