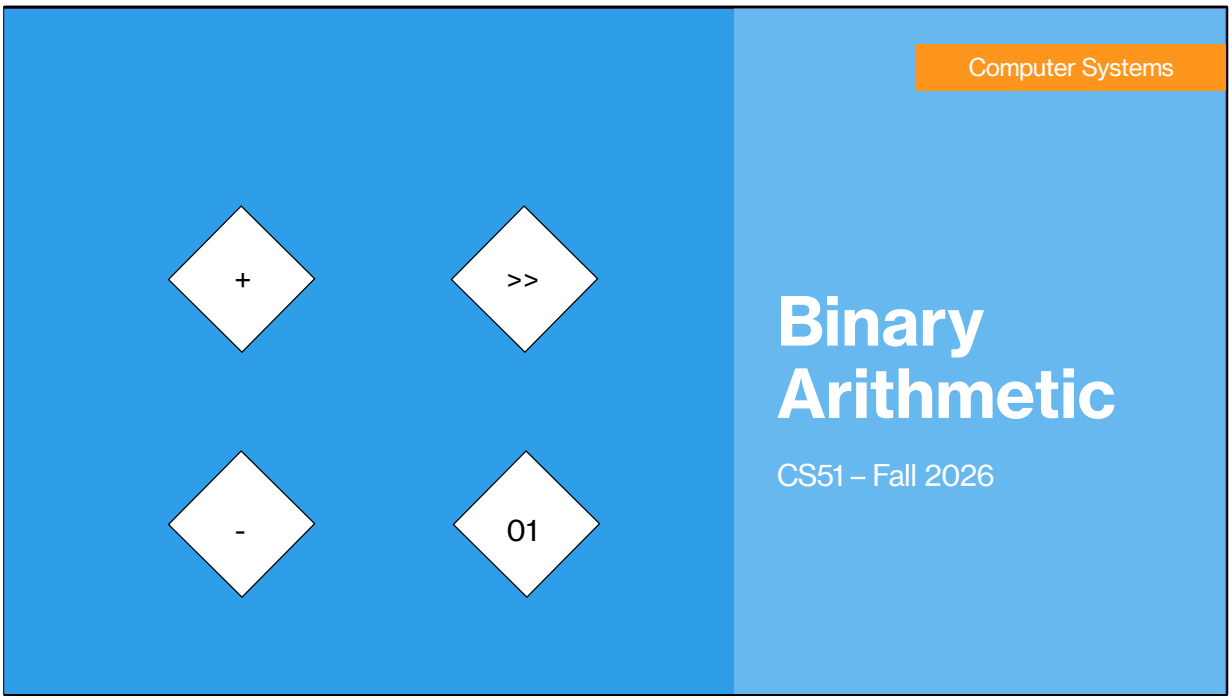


Last time, we saw that there are different numeral systems. We examined binary, decimal, and hexadecimal and learned how to convert a number from one representation to another. Today, we will focus on binary numbers alone and learn how to perform basic arithmetic with them

Addition

2

How do we perform addition in decimal?

Think

- *How do you add the numbers* $4597_{10} + 334_{10}$?

3

Let's go back to primary education. How would you add the two decimal numbers 4597 and 334?

Think

- *How do you add the numbers* $4597_{10} + 334_{10}$?

0 1 1 1 ← carry

$$\begin{array}{r} 4\,5\,9\,7_{10} \\ +\quad 3\,3\,4_{10} \\ \hline 4\,9\,3\,1_{10} \end{array}$$

4

Presumably, you started with adding the rightmost digits (the least significant digits). If the sum is greater or equal to 10, then carry a 1 and write down the sum%10. Repeat with the next digits to the left, but this time include the **carry** in the addition.

Think

- *How do you add the numbers* $223_5 + 414_5$?

5

OK you (thankfully!) remember decimal addition. How would we go about adding together these two base 5 numbers?

Think

- *How do you add the numbers* $223_5 + 414_5$?

$$
\begin{array}{r}
1\,0\,1 \leftarrow \text{carry} \\
2\,2\,3_5 \\
+\ 4\,1\,4_5 \\
\hline
1\,1\,4\,2_5
\end{array}
$$

6

Again, we'd start with the least significant digits and add them up. If the sum exceeds our base (i.e.5), we have a carry of 1.

Think

- *How do you add the numbers* $223_5 + 414_5$?

$$\begin{array}{r} 1\,0\,1 \leftarrow \text{carry} \\ 2\,2\,3_5 \\ +\ 4\,1\,4_5 \\ \hline 1\,1\,4\,2_5 \end{array}$$

- *Can you confirm this is correct by checking the decimal equivalent?*

7

Let's further convince ourselves that this is correct by checking the decimal equivalent

Think

- *How do you add the numbers* $223_5 + 414_5$?

1 0 1 ← carry	0 0 1 ← carry
2 2 3_5 = 2*25 + 2*5 + 3*1 = 63	6 3_{10}
+ 4 1 4_5 = 4*25 + 1*5 + 4*1 = 109	+ 1 0 9_{10}
1 1 4 2_5 = 1* 125 + 1*25 + 4*5 + 2*1 = 172	1 7 2_{10}

8

Remember all the work we did from last class meeting? Looks like we got it right

Think

- *How do you add the binary numbers* $1_2 + 101_2$?

9

What if we wanted to add the two binary numbers 1 and 0101?

Think

- *How do you add the binary numbers* $1_2 + 101_2$?

$$
\begin{array}{r}
0\ 0\ 1 \leftarrow \text{carry} \\
\quad 1_2 \\
+\ 1\ 0\ 1_2 \\
\hline
1\ 1\ 0_2
\end{array}
$$

10

we'd follow the same process from right to left. if we add two bits equal to 1, then we get back 0 and have a carry of 1.

Think

- *How do you add the binary numbers* $1_2 + 101_2$?

$$\begin{array}{r} 0\ 0\ 1 \leftarrow \text{carry} \\ 1_2 \\ +\ 1\ 0\ 1_2 \\ \hline 1\ 1\ 0_2 \end{array}$$

- *Can you confirm this is correct by checking the decimal equivalent?*

11

Again, can we confirm that what we did is correct by converting it to its decimal representation we are more comfortable with

Think

- *How do you add the binary numbers* $1_2 + 101_2$?

0 0 1 ← carry	0 ← carry
$1_2 = 1_{10}$	1_{10}
+ $1\ 0\ 1_2 = 4+1=5_{10}$	+ 5_{10}
$1\ 1\ 0_2 = 4+2=6_{10}$	6_{10}

- *Can you confirm this is correct by checking the decimal equivalent?*

12

Indeed, it's 1+5=6.

Think

- *How do you add the hexadecimal numbers* $11F5_{16} + 14E_{16}$?

0 0 1 1 ← carry
1 1 F 5_{16}
\+ 1 4 E $_{16}$
---
1 3 4 3_{16}

13

Indeed, it's 1+5=6.

Think

- *How do you add the hexadecimal numbers* $11F5_{16} + 14E_{16}$?

0 0 1 1 ← carry
  1 1 F 5_{16}
+ 1 4 E_{16}
---
  1 3 4 3_{16}

- *Can you confirm this is correct by checking the decimal equivalent?*

14

Can you confirm it is correct by calculating the decimal equivalent representations

Think

- *How do you add the hexadecimal numbers* $11F5_{16} + 14E_{16}$?

0 0 1 1 ← carry

1 1 F 5_{16}= 16^3+16^2+15x16+5 = 4597_{10}

\+ 1 4 E_{16} = 16^2+4x16+14 = 334_{10}

1 3 4 3_{16} = 16^3+3x16^2+4x16+3 = 4931_{10}

0 0 1 1 ← carry

4 5 9 7_{10}

\+ 3 3 4_{10}

4 9 3 1_{10}

- *Can you confirm this is correct by checking the decimal equivalent?*

15

Indeed they add up to 4931.

PRACTICE TIME - Addition

- Compute the following sums
- $0111_2 + 0101_2$
- $9B3_{16} + 38_{16}$

16

OK Let's see whether we can do addition in binary and hex.

ANSWER - Addition

- Compute the following sums
- $0111_2 + 0101_2$
- $9B3_{16} + 38_{16}$

$0111_2 + 0101_2$

0 1 11

0 11 1_2

+ 0 1 0 1_2

1 1 0 0_2

← carry →

$9B3_{16} + 38_{16}$

0 0 0

9 B 3_{16}

+ 3 8_{16}

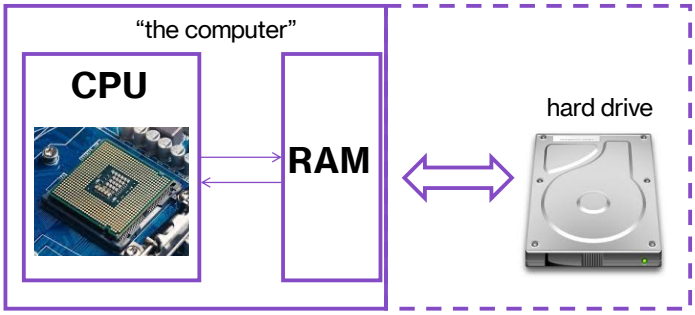
9 E B_{16}

17

Dod you get these results?

Overflow

Simplified view of a computer

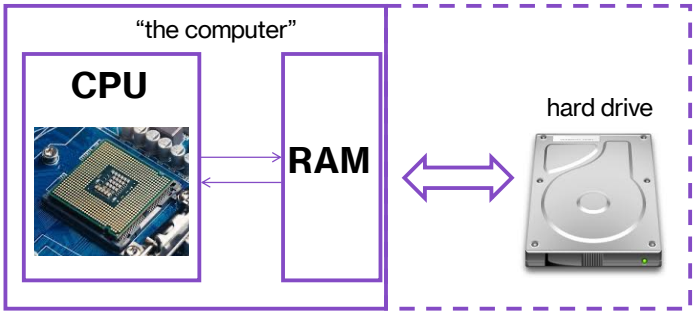


- Both RAM and the hard drive store a sequence of bits.

0100100100110100100101111000100010 10

Remember we said a computer at its basis is the CPU communicating with the RAM, that is the memory, where temporary data are stored as long as we have power. More permanent data are stored in the hard drive. Both types of memory store information in sequences of bits.

Word length

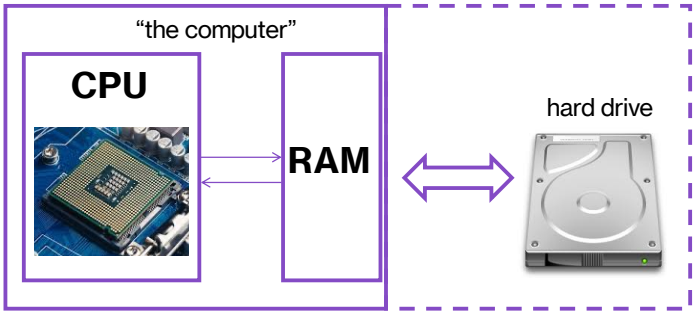


- To simplify hardware, these bits are grouped into fixed length chunks called "words", e.g., in groups of 4.

0100 1001 0011 0100 1001 0111 1000 1000 1010

To simplify hardware, we group bits into fixed length chunks called words.

Computers today



- Most computers today work with 64-bit words.

Most computers today read words in 64 bits. When I grew up, most had 32-bit words but as our computational needs go up, computer architecture has to adapt.

Think

- *What happens if we want to add the binary numbers* $1001_2 + 1101_2$ *but only have 4 bits available?*

22

What if we want to add two binary numbers and we know that the result has to fit in a fixed number of bits, e.g., can you try to add the two binary numbers 1001 and 1101?

Think

- *What happens if we want to add the binary numbers* $1001_2 + 1101_2$ *but only have 4 bits available?*

1 0 0 1 ← carry

$1\ 0\ 0\ 1_2$

$+\ 1\ 1\ \ 0\ 1_2$

$1\ 0\ 1\ 1\ 0_2$

23

You see that we would need an extra bit!

Overflow

- If we operate on a fixed number of bits, addition can **overflow** if the result is too big to fit in the available space.

1 0 0 1 ← carry

$$\begin{array}{r} 1\,0\,0\,1_2 \\ +\,1\,1\,\;0\,1_2 \\ \hline 1\,0\,1\,1\,0_2 \end{array}$$

24

This can be a problem and the problem is known as overflow.

Think

- If we operate on a fixed number of bits, addition can **overflow** if the result is too big to fit in the available space.

$$\begin{array}{r} 1\,0\,0\,1 \leftarrow \text{carry} \\ 1\,0\,0\,1_2 \\ +\,1\,1\,0\,1_2 \\ \hline 1\,0\,1\,1\,0_2 \end{array}$$

- *How can we detect overflow?*

25

How do you think we can detect overflow?

Think

- If we operate on a fixed number of bits, addition can **overflow** if the result is too big to fit in the available space.

1 0 0 1 ← carry

$$\begin{array}{r} 1\,0\,0\,1_2 \\ +\,1\,1\,0\,1_2 \\ \hline 1\,0\,1\,1\,0_2 \end{array}$$

- *How can we detect overflow?*
  - If the carry bit for the MSB is 1, then we have overflow

26

We just need to check the carry bit for the most significant (remember, left most) bit. If it's 1, then we know we are in trouble.

Signed numbers

27

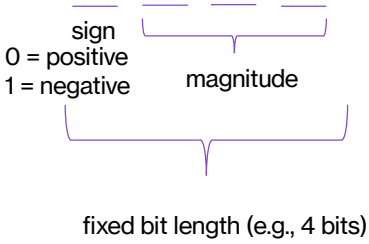
Everything we have seen so far has assume we are working with positive numbers only, including 0. Now we will move into signed numbers, that is numbers that can tell us whether they are positive or negative.

Binary numbers revisited

- *What is -6_{10} in binary?*

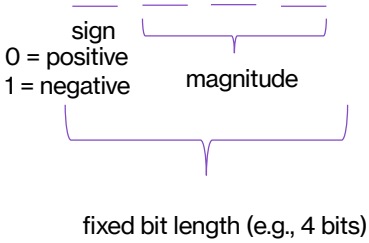
Here's a question for you. Say we have the -6 decimal number. How do we represent it in binary?

One option: sign/magnitude



Our first option is to work with a representation called sign or magnitude. If we have a fixed number of bits, say 4, we reserve the left most bit for the sign. The convention is that if it's a 0, then the number is positive. If it's 1, it's negative. The remaining 3 bits are known as the magnitude.

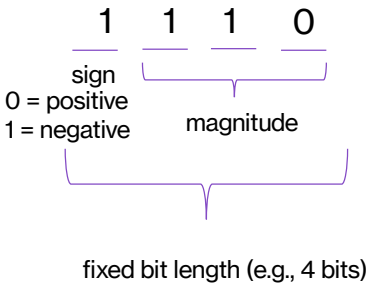
One option: sign/magnitude



What is -6_{10} in sign/magnitude?

So if we go back to -6 from decimal and want to represent it in binary using the sign/magnitude representation, what would that look like?

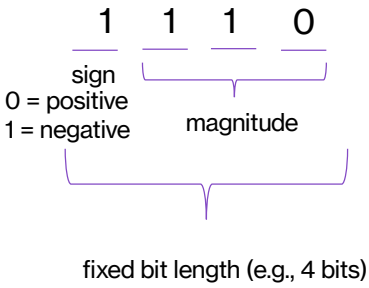
One option: sign/magnitude



What are the range of values we can support with 4 bits in sign/magnitude representation?

If we had 4 bits available and we use the sign/magnitude representation, what is the range of decimal values we can support?

One option: sign/magnitude

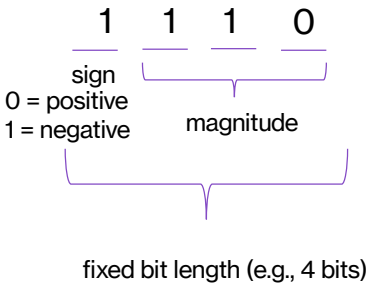


What are the range of values we can support with 4 bits in sign/magnitude representation?

1111 = -7 to 0111 = 7

The largest magnitude is 111 with 3 bits, that is 7. So we can go from -7 to 7

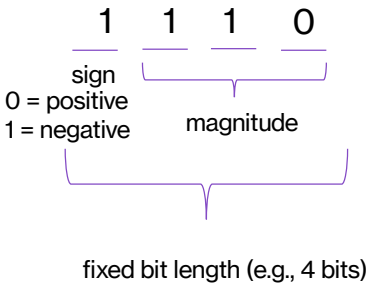
Think



Any problems?

Do you foresee any problems with this representation?

Think



Any problems?

We have two zeros, 0000 and 1000

We end up with two zeros, one for the positive and one for the negative sign.

Think

- *Try to add 47_{10} and -47_{10} using sign/magnitude representation. You can assume you have 8 bits.*

35

There's also one more problem. Try to add 47 and -47 in binary using sign/magnitude representation.

Think

- *Try to add 47_{10} and -47_{10} using sign/magnitude representation. You can assume you have 8 bits.*

00101111

+10101111

011011110 = 222_{10}

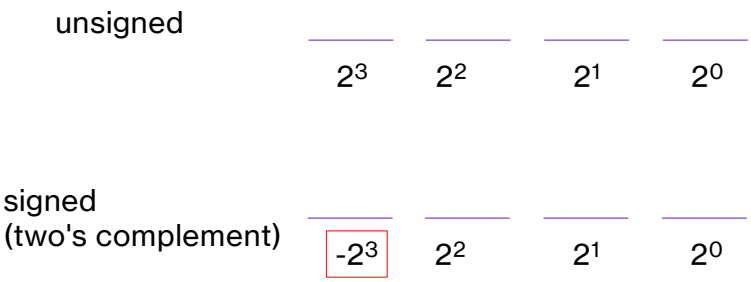
Which is nonsensical!

36

Instead of 0, we got the nonsensical 222?!

A better option: two's complement representation

For a number with n bits, the MSB represents -2^{n-1}



We will see another representation that addresses both of these problems. This representation is known as two's complement and the basic idea behind it is that the MSB (meaning the left most bit represents the number -2^(n-1). Every other bit works as usual.

Think

What numbers do we get?

unsigned	1	0	0	1
	2^3	2^2	2^1	2^0
signed (two's complement)	1	0	0	1
	-2^3	2^2	2^1	2^0

With that in mind, what numbers do we get if we have the bits 1001 in unsigned and signed (in two's complement) representation?

Think

What numbers do we get?

unsigned	1	0	0	1	9
	2^3	2^2	2^1	2^0	
signed (two's complement)	1	0	0	1	-7
	-2^3	2^2	2^1	2^0	

In the case of unsigned, we have 1x2^3+1x2^0=9. But in two's complement that is -2^3+2^0 = -8+1=-7.

Think

What numbers do we get now?

unsigned	1	1	1	1
	2^3	2^2	2^1	2^0
signed (two's complement)	1	1	1	1
	-2^3	2^2	2^1	2^0

What if we read the 4-bit number 1111 in each of the two representations?

Think

What numbers do we get now?

unsigned	1	1	1	1	15
	2^3	2^2	2^1	2^0	
signed (two's complement)	1	1	1	1	-1
	-2^3	2^2	2^1	2^0	

In the case of unsigned, we have 1x2^3+1x2^2+1x2^1+1x2^0=1. But in two's complement that is -2^3+2^2+2^1+2^0 = -8+4+2+1=-1

Think

What numbers do we get now?

unsigned	1	1	0	0
	2^3	2^2	2^1	2^0
signed (two's complement)	1	1	0	0
	-2^3	2^2	2^1	2^0

OK what about now?

Think

What numbers do we get now?

unsigned	1	1	0	0	12
	2^3	2^2	2^1	2^0	
signed (two's complement)	1	1	0	0	-4
	-2^3	2^2	2^1	2^0	

Did you get 12 and -4?

Think

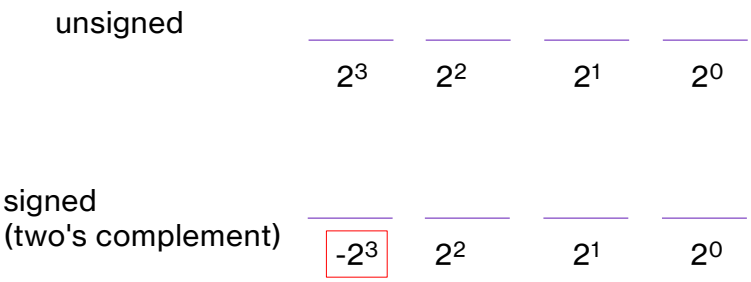
- *How many numbers can we represent with each representation if we have 4 bits?*

unsigned	2^3	2^2	2^1	2^0
signed (two's complement)	-2^3	2^2	2^1	2^0

If we have 4 bits, how many numbers can we represent in unsigned and how many in signed (with two's complement)

Think

- *How many numbers can we represent with each representation if we have 4 bits?*
  - 16 numbers: 0000, 0001, ..., 1111, regardless of representation



It's 16 (2^4), regardless of the representation

Think

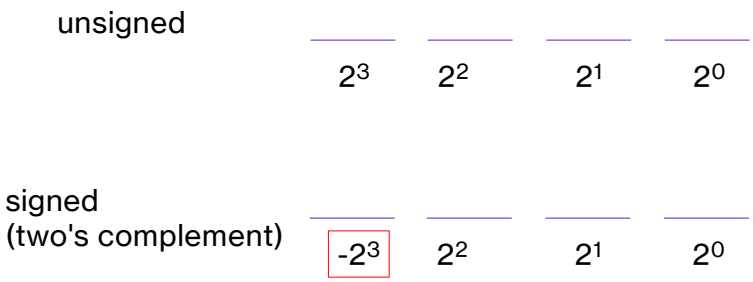
- *What is the range of numbers we can represent for each approach with 4 bits?*

unsigned	2^3	2^2	2^1	2^0
signed (two's complement)	-2^3	2^2	2^1	2^0

What is the range of numbers that each approach can support with 4 bits?

Think

- *What is the range of numbers we can represent for each approach with 4 bits?*
  - *unsigned: 0, 1, … 15*
  - *signed: -8, -7, …, 7*



For unsigned, since they are all positive it goes 0 to 15. But for signed it's -8 to 7

binary representation	unsigned	
0000	0	
0001	1	
0010	?	
0011		
0100		
0101		
0110		
0111		
1000		
1001		
1010		
1011		
1100		
1101		
1110		
1111		

If 0000 corresponds to 0 in unsigned and 0001 to 1, what does 0010 correspond to?

binary representation	unsigned	
0000	0	
0001	1	
0010	**2**	
0011		
0100		
0101		
0110		
0111		
1000		
1001		
1010		
1011		
1100		
1101		
1110		
1111		

2, right? What about the rest of the column

binary representation	unsigned	two's complement
0000	0	?
0001	1	
0010	2	
0011	3	
0100	4	
0101	5	
0110	6	
0111	7	
1000	8	
1001	9	
1010	10	
1011	11	
1100	12	
1101	13	
1110	14	
1111	15	

Well we just said, it would be 0,..., 15. How about two's complement representation of 0000?

binary representation	unsigned	two's complement
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	?
1001	9	
1010	10	
1011	11	
1100	12	
1101	13	
1110	14	
1111	15	

It's of course 0. And the same applies all the way to 0111. How about 1000?

binary representation	unsigned	two's complement
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	-8
1001	9	?
1010	10	
1011	11	
1100	12	
1101	13	
1110	14	
1111	15	

Did you get -8? How about the rest of the column?

binary representation	unsigned	two's complement
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	-8
1001	9	-7
1010	10	-6
1011	11	-5
1100	12	-4
1101	13	-3
1110	14	-2
1111	15	-1

Did you get this? Do you notice that 1xxxx is always the smallest number we can represent with the number of bits we have available and then the negative numbers keep increasing?

binary representation	unsigned	two's complement
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	-8
1001	9	-7
1010	10	-6
1011	11	-5
1100	12	-4
1101	13	-3
1110	14	-2
1111	15	-1

How can you tell if a number is negative?

Assuming two's complement representation, how can you tell if a number is negative?

binary representation	unsigned	two's complement
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	-8
1001	9	-7
1010	10	-6
1011	11	-5
1100	12	-4
1101	13	-3
1110	14	-2
1111	15	-1

How can you tell if a number is negative?

Look at the MSB

You just look at the most significant bit.

binary representation	unsigned	two's complement
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	-8
1001	9	-7
1010	10	-6
1011	11	-5
1100	12	-4
1101	13	-3
1110	14	-2
1111	15	-1

The smallest number is 1 followed by 0s for the rest of the bits

Here's another cool fact. If you have n bits, the smallest number you can represent will be 1000.000 where 1 is the MSB and 0s go to the rest of n-1 bits!

binary representation	unsigned	two's complement
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	-8
1001	9	-7
1010	10	-6
1011	11	-5
1100	12	-4
1101	13	-3
1110	14	-2
1111	15	-1

-1 is always all 1s

And one more: -1 is going to be all 1s

A two's complement trick

You can also calculate the value of a negative number represented as two's complement as follows:

- Flip all of the bits (0 → 1 and 1→ 0)
- Add 1
- The resulting number is the magnitude of the original negative number

There is a cool trick you can use to calculate the value of a negative number in two's complement. You flip the bits (so 0s turn to 1s and 1s to 0s). You add a 1 the usual way. And the resulting number is the magnitude of the negative number.

Think

You can also calculate the value of a negative number represented as two's complement as follows:

- Flip all of the bits (0 → 1 and 1→ 0)
- Add 1
- The resulting number is the magnitude of the original negative number

- Which number is 1110?

1110

Your turn. Which negative number is 1110?

Think

You can also calculate the value of a negative number represented as two's complement as follows:

- Flip all of the bits (0 → 1 and 1→ 0)
- Add 1
- The resulting number is the magnitude of the original negative number

You know that 1110 is going to be negative so follow the trick of flipping its bits, adding 1. You get 0010 which is 2. So then your original number was -2

Addition with 4-bit *two's complement* numbers

$$\begin{array}{r} 0001_2 \\ +\ 01\,01_2 \\ \hline ? \end{array}$$

OK, say we have these two 4-bit two's complement numbers, what's the result of this addition?

Addition with 4-bit *two's complement* numbers

$$\begin{array}{r} {\scriptstyle 0\ 0\ 1\ \ } \\ 0001_2 \\ +\ 0101_2 \\ \hline 0110_2 \end{array}$$

We add up as before, using the carry bit when needed

Addition with 4-bit *two's complement* numbers

$$\begin{array}{r} 0110 \\ +\ 0101 \\ \hline ? \end{array}$$

What about now? Assume that from now on all is binary numbers

Addition with 4-bit *two's complement* numbers

```
  1 0 0
  0110
+ 0101
------
  1011?
```

So I got 1011. Is this right? Remember, I added two positive numbers (how do I know this? they have 0 at the MSB) and ended up with a negative?

Addition with 4-bit *two's complement* numbers

```
  1 0 0
  0110      6
+ 0101      5
------
  1011?     -5? (11 unsigned)
```

Overflow! We cannot represent this number (it's too large)

We have overflow as we cannot represent this number. 6+5=11 which is out of the range of -8,..,7 we can represent with 4 bits.

Addition with 4-bit *two's complement* numbers

$$\begin{array}{r} 0110 \\ +\ 1101 \\ \hline ? \end{array}$$

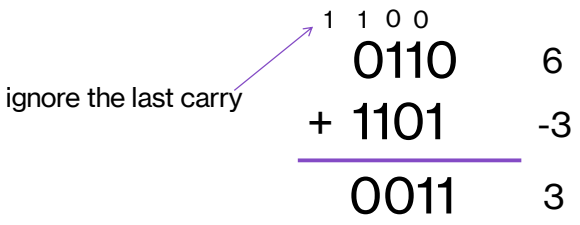
What about adding these two two's complement binary numbers?

Addition with 4-bit *two's complement* numbers

```
  1 1 0 0
    0110
+ 1 101
------
    0011
```

We get 0011 but also have a carry of 1 floating

Addition with 4-bit *two's complement* numbers



If we ignore the last carry, we actually see that what we got is correct!

Think

- *How do you know that overflow has occurred with two's complement numbers?*

So how do we know that overflow has occurred when adding two two's complement numbers?

Overflow in two's complement

- *How do you know that overflow has occurred with two's complement numbers?*
  - Can only happen when adding two numbers of the same sign
  - Add the numbers and discard the last carry bit
  - Check the sign of the resulting number: if the resulting differs than the sign of the two numbers added, there is overflow

Overflow can now only happen if we add two numbers with the same sign (i.e., two positive or two negative numbers). So we will add the numbers as usual and discard the carry bit for the MSB. Since overflow can happen only if two numbers have the same sign, we check the sign of the resulting number (i.e. the MSB) and if it's different than the two numbers we added, then we have overflow. That is, if you have two positive numbers (MSB is 0) and you get a result with MSB 1, you know you are in trouble. Similarly, adding two negative numbers (MSB 1) and getting MSB 0 in the result means overflow.

Subtraction

71

Any idea about how we can perform subtraction in binary?

Subtraction

- Negate the 2nd number (flip the bits and add 1)
- Add them!

The work is actually quite simple. We just have to negate the 2nd number, that is flip its bits from 0 to 1 and 1 to 0 and then 1, and add the first and negated number.

PRACTICE TIME – Subtracting two's complements

- Calculate $3_{10} - 5_{10}$ using 4-bit two's complement numbers.

73

Let's practice by calculating 3-5 in two's complement assuming 4 bits,

ANSWER – Subtracting two's complements

- Calculate $3_{10} - 5_{10}$ using 4-bit two's complement numbers.
- 3_{10} is 0011_2
- Take the two's complement of -$5_{10} = 1011_2$:
  - $5_{10} = 0101_2$
  - Invert 1010_2
  - Add 1: 1011_2
- Add them: $0011_2 + 1011_2 = 1110_2 = -2_{10}$
  - How do you know $1110_2 = -2_{10}$?
    - Invert and then 1: $0001 + 1 = 0010 = 2$
    - So $1110_2 = -2_{10}$

74

Did you get this?

Sign extension in two's complements

- **Sign extension**: when extending a two's complement number in more bits, we need to copy the sign bit.
- For example, if we have 4 bits the numbers -3_{10} and 3_{10} are written in binary as 1101_2 and 0011_2.
- If we want to sign extend them to 8 bits, we would need to copy the sign bit 4 times to 11111101_2 and 00000011_2, respectively.

75

When we want to extend a two's complement number to more bits, we are going to copy the sign bit (0 for positive, 1 for negative). E.g., if we have 4 bits and the binary representations for -3 and 3 (1101 and 0011, respectively), we would pad them with 1s and 0s, respectively.

Why number representation errors matter

- 1. Ariane 5 Rocket Explosion (1996)
  - Flight software converted a 64-bit floating point value into a 16-bit signed integer.
  - The value was too large which caused an overflow which led to system failure.
  - The rocket self-destructed 40 seconds after launch leading to a loss of $370 million.
- 2. Classic Video Games
  - Many classic video games stored scores/lives in 8-bit signed integers (−128, ... ,127)
  - If score exceeded 127 it wrapped to negative values.
  - Players suddenly saw “negative lives” or game crashes.

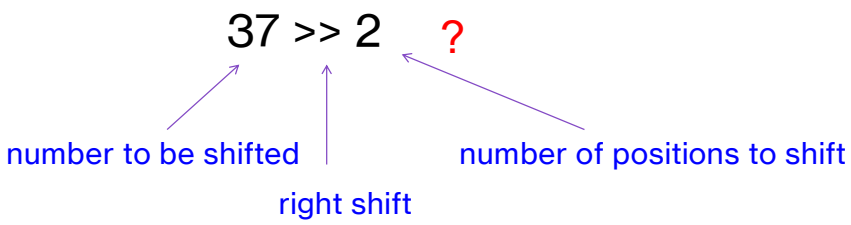
76

Why do representations of numbers matter? Here are two examples when things can go wrong. Ariane 5 rocket launched in 1996 and the flight software converted a 64-bit floating point number into a 16-bit integer. The value was too large, causing an overflow and system failure. The rocket self-destructed less than a minute pass its launch, taking with it more than $370 million. A less catastrophic example would be to look at classic video games, e.g., some used 8 bits to store scores or lives. If a score exceeded the max number, ie. 127, it wrapped to negative values and played suddenly saw negative lives or game crashes.

We will move next to shifting, a type of operation we can do on numbers.

Shifting: variable length numbers

Shifting shifts the binary representation of the number right or left

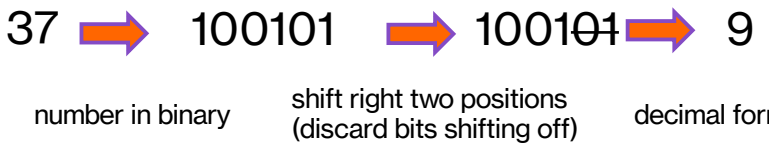


The next topic we will cover is that of shifting. Shifting shifts the binary representation of a number either right or left. For example, here I am saying I want to shift 37 2 positions to the right (note the direction of ?>>)

Shifting: variable length numbers

Shifting shifts the binary representation of the number right or left

37 >> 2



number in binary

shift right two positions (discard bits shifting off)

decimal form

So how do we shift 37 2 positions to the right? We convert it to its binary representation (assume unsigned numbers). Shift right two positions by discarding the 2 lowest bits. Convert 1001 to decimal and you get 9. So 37>>2=9!

Think

Shifting shifts the binary representation of the number right or left

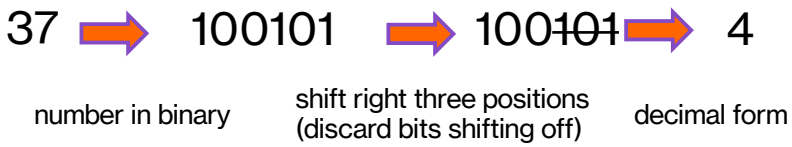
37 >> 3 ?

What if I asked you to shift 37 3 positions to the right?

Think

Shifting shifts the binary representation of the number right or left

37 >> 3



number in binary

shift right three positions
(discard bits shifting off)

decimal form

You would repeat the same process discarding the last 3 bits and end up with 100 which is equal to 4.

Shifting 8-bit numbers

Shifting shifts the binary representation of the number right or left

37 >> 2

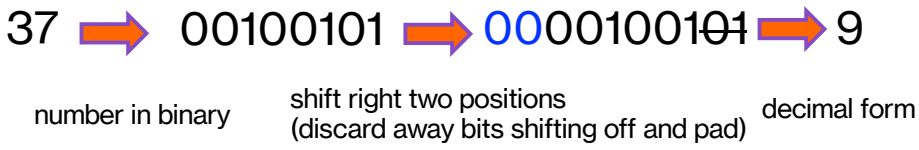
What is 37 as an 8-bit binary number?

What if instead of a variable number of bits, we have a fixed one, say 8. We will use sign extension to pad with 0s the front of the binary number.

Shifting 8-bit numbers

Shifting shifts the binary representation of the number right or left

37 >> 2



As we shift two bits to the right, we will continue padding the front with 0s since we have a fixed length representation.

Think

Shifting shifts the binary representation of the number right or left

15 << 2

?

What do you think happens if we wanted to shift 2 positions to the left?

Think

Shifting shifts the binary representation of the number right or left

15 << 2

15  ?

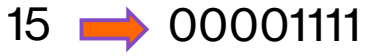
number in binary

Assuming an 8-bit representation, what is 15 in binary?

Shifting 8-bit numbers

Shifting shifts the binary representation of the number right or left

15 << 2



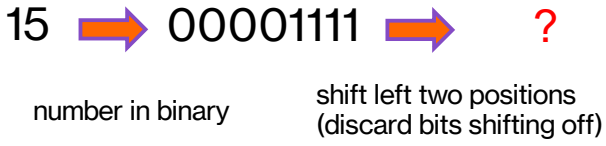
number in binary

00001111 (we need the four 0s as padding in the front)

Think

Shifting shifts the binary representation of the number right or left

15 << 2

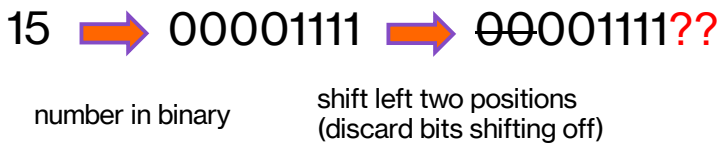


If we were to shift left two positions and discard bits we shift off, what would we get?

Shifting 8-bit numbers

Shifting shifts the binary representation of the number right or left

15 << 2



We discard the first two bits but what happens at the end so that we end up with 8 bits?

Shifting 8-bit numbers

Shifting shifts the binary representation of the number right or left

15 << 2

number in binary

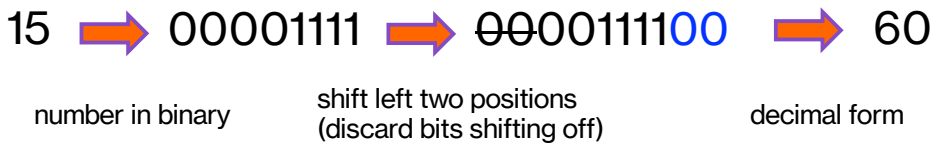
shift left two positions
(discard bits shifting off)

Just pad it with 0s! We always do so independently of sign

Shifting 8-bit numbers

Shifting shifts the binary representation of the number right or left

15 << 2



And then as usual, convert to its decimal form which gives you 60.

Think

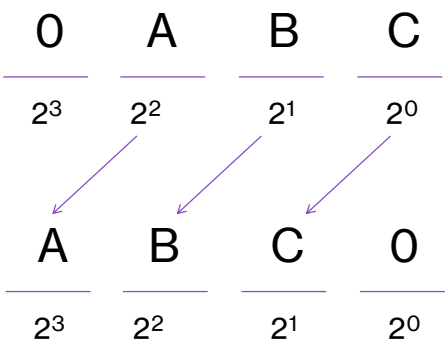
*What does **left** shifting by one position do mathematically?*

0	A	B	C
2^3	2^2	2^1	2^0

Let's think, what does left shifting by 1 position do mathematically?

Shifting mathematically: unsigned

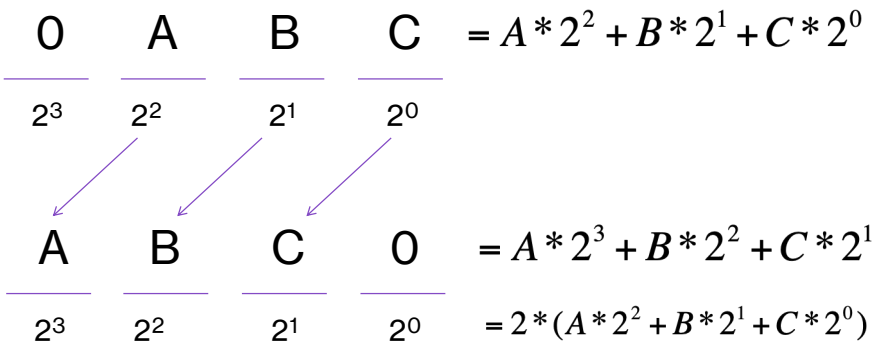
*What does **left** shifting by one position do mathematically?*



We had four bits (assume 0 at the MSB) and shifting discarded that 0 and padded 0 at the end. Try to calculate the decimal result

Shifting mathematically: unsigned

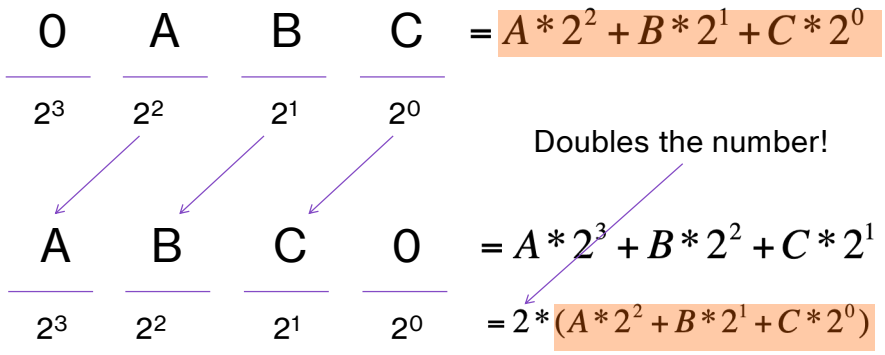
*What does **left** shifting by one position do mathematically?*



Is this what you got?

Shifting mathematically: unsigned

What does **left** shifting by one position do mathematically?



Do you see that the bottom is twice as big as the top?

Think

What does **left** shifting by ***n*** positions do mathematically?

So if we have left shifting by n positions, what do we end up with?

Think

What does **left** shifting by ***n*** positions do mathematically?

Multiply by 2^n (doubles it n times)

We double n times, or multiply by 2^n

Shifting mathematically: unsigned

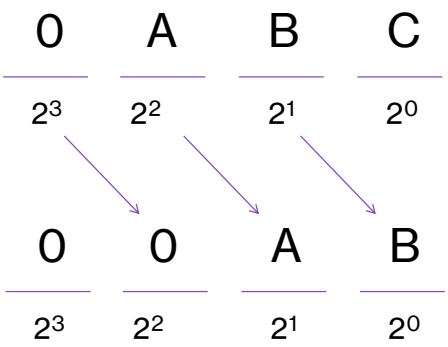
*What does **right** shifting by one position do mathematically?*

0	A	B	C
2^3	2^2	2^1	2^0

Lets' apply the same idea to right shifting

Shifting mathematically: unsigned

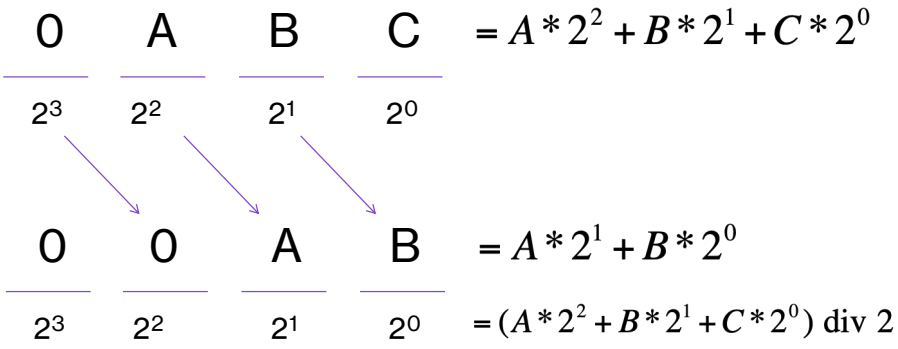
*What does **right** shifting by one position do mathematically?*



If we had four bits (assuming 0 for MSB), right shifting by 1 position would discard C and pad 0 in the front.

Shifting mathematically: unsigned

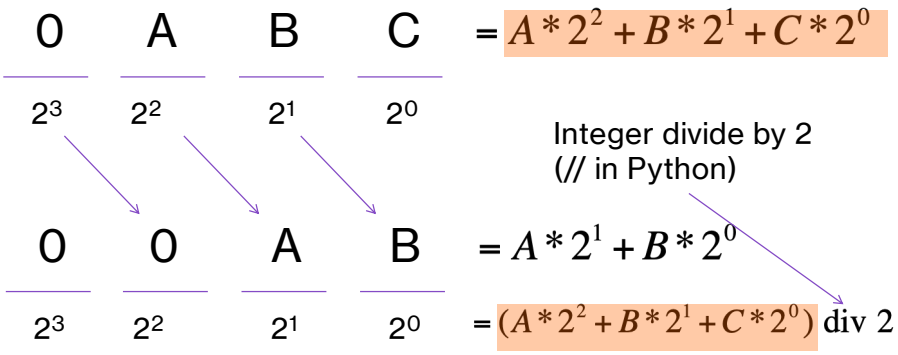
*What does **right** shifting by one position do mathematically?*



Try to convert this to decimal

Shifting mathematically: unsigned

*What does **right** shifting by one position do mathematically?*



That's like doing integer division by 2 which by rounding down to infinity.
Remember, integer division is // in Python

Think

*What does **right** shifting by **n** positions do mathematically?*

So what does right shifting an unsigned number by n positions do?

Think

*What does **right** shifting by **n** positions do mathematically?*

Integer division by 2^n (halves it n times)

It halves it n times or it performs integer division by 2^n

Think

Shifting shifts the binary representation of the number right or left

-4 >> 1

What is -4 as a 4-bit binary number?

Now what if we don't have only unsigned number. Say we work with 4-bit numbers and we want to shift -4 one position to the right.

Shifting 4-bit numbers

Shifting shifts the binary representation of the number right or left

-4 >> 1

-4  1100

number in binary

It's 1100 (assuming two's complement)

Think

Shifting shifts the binary representation of the number right or left

-4 >> 1

How do we fill in the leftmost bit?

If we are to shift one position to the right, how do we fill in the leftmost bit to end up with 4 bits?

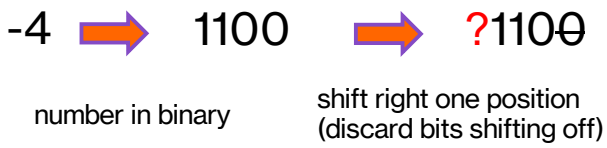
Shifting 4-bit numbers

- Two types of right shifts:
  - logical shift: always shift in 0s
  - arithmetic shift: shift in the same as the high-order bit

We have two options. One is called logical shift and it always adds 0s. In arithmetic shift, in contrast, we pad with the same bit as the MSB or high-order bit.

Think

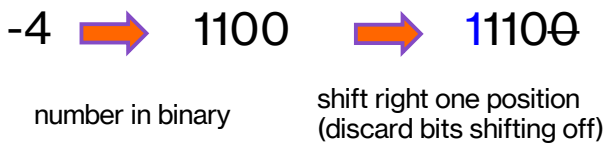
- Two types of right shifts:
  - logical shift: always shift in 0s
  - **arithmetic shift: shift in the same as the high-order bit**



What if we followed arithmetic shift next?

Think

- Two types of right shifts:
  - logical shift: always shift in 0s
  - **arithmetic shift: shift in the same as the high-order bit**

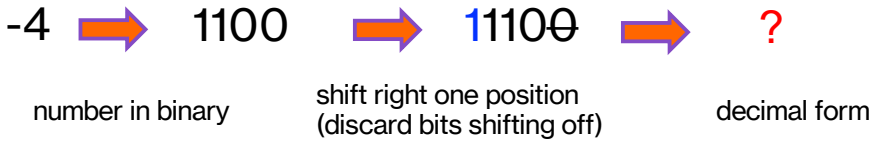


We would pad it with 1 as this was the MSB.

Shifting 4-bit numbers

- Two types of right shifts:
  - logical shift: always shift in 0s
  - **arithmetic shift: shift in the same as the high-order bit**

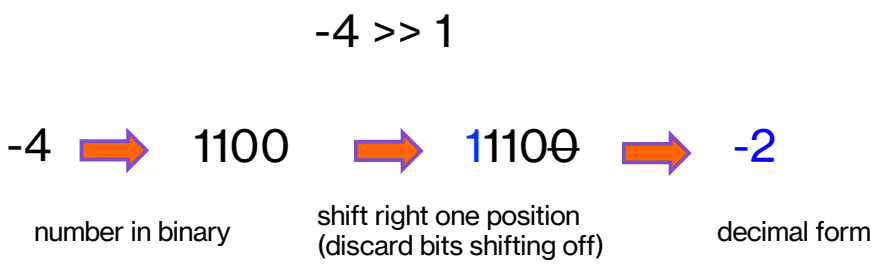
-4 >> 1



What is the decimal form?

Shifting 4-bit numbers

- Two types of right shifts:
  - logical shift: always shift in 0s
  - **arithmetic shift: shift in the same as the high-order bit**



It's -2

Think

Shifting shifts the binary representation of the number right or left

-4 >> 2

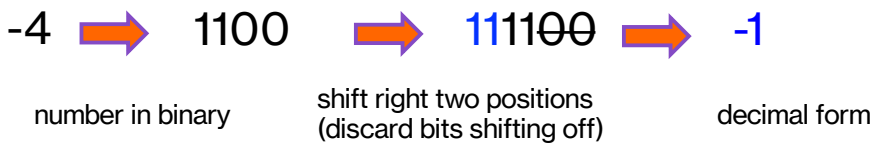
?

What if we wan to shift -4 2 positions to the right and we have a 4-bit representation?

Think

- Two types of right shifts:
  - logical shift: always shift in 0s
  - **arithmetic shift: shift in the same as the high-order bit**

-4 >> 2



Assuming again arithmetic shift, we would discard the last two 0s and add two 1s in the front which results in the number -1.

Think

*What does **right** arithmetic shifting by **n** positions do mathematically for **signed numbers**?*

So what does right arithmetic shifting by n positions do mathematically when we are working with signed numbers?

Arithmetic shifting mathematically

*What does **right** arithmetic shifting by **n** positions do mathematically for **signed numbers**?*

Integer division by 2^n (halve n times)

Same thing!!

Same thing with unsigned number, it divides them by 2^n.

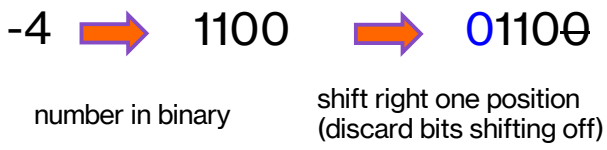
Think

- Two types of right shifts:
  - **logical shift: always shift in 0s**
  - arithmetic shift: shift in the same as the high-order bit

OK what if we did logical shifting?

Shifting 4-bit numbers

- Two types of right shifts:
  - **logical shift: always shift in 0s**
  - arithmetic shift: shift in the same as the high-order bit



number in binary

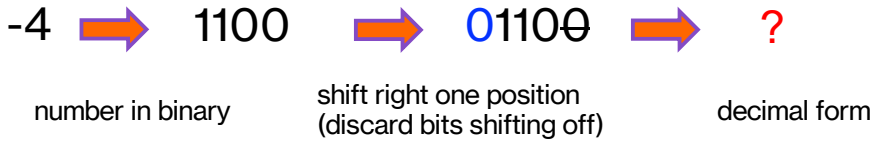
shift right one position
(discard bits shifting off)

We'd just pad with 0

Think

- Two types of right shifts:
  - **logical shift: always shift in 0s**
  - arithmetic shift: shift in the same as the high-order bit

-4 >>> 1



number in binary

shift right one position
(discard bits shifting off)

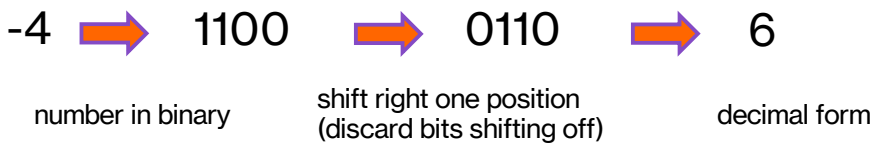
decimal form

Which would result to?

Shifting 4-bit numbers

- Two types of right shifts:
  - **logical shift: always shift in 0s**
  - arithmetic shift: shift in the same as the high-order bit

-4 >>> 1



6! So with logical shifts it doesn't seem like we are halving.

Think

- Are there two types of left shifts?
  - logical shift: always shift in 0s
  - arithmetic shift: ?

arithmetic

-3 << 1 ?

logical

-3 <<< 1 ?

So if we have logical and arithmetic right shift, do we have the same for left?

Left shifts

- Are there two types of left shifts?
  - logical shift: always shift in 0s
  - arithmetic shift: ?

arithmetic

-3 << 1 1101  1101? (double the number)

logical

-3 <<< 1 1101  11010

How can we double the number if we do an arithmetic left shift?

Left shifts

- Are there two types of left shifts?
  - logical shift: always shift in 0s
  - arithmetic shift: ?

arithmetic

-3 << 1 1101  11010 (double the number)

logical

-3 <<< 1 1101  11010

Only one type of left shift

We would need to add a 0. Which means that we only have one type of left shift

Shifting summarized

Arithmetic shift:

- Right shift n
  - shift n bits to the right
  - discard right n bits
  - left n bits match high-order bits of original number
  - Effect: Integer division by 2^n (halve n times)
- Left shift
  - shift n bits to the left
  - discard left n bits
  - right n bits are 0s
  - Effect: multiply by 2^n (double n times)

Logical shift right:

- left n bits are 0s (no mathematical guarantees for negative numbers)

So if we were to summarize shifting, arithmetic shifting is truly what happens both for left and right but logical shifting can only happen for right.

PRACTICE TIME - Arithmetic right shift

- Let's assume we work with 4 bits and we want to shift the binary representation of the number -6_{10} by 2 bits to the right, that is -6>>2.
- Using two's complement representation show what the effect of the shift will be on the binary representation of -6_{10}.

123

Your turn to apply arithmetic right shift by 2 bits to the binary -6 using two's complement and assuming 4 bits.

ANSWER - Arithmetic right shift

- We start by converting the decimal number to its binary representation in two's complement:
  - $-6_{10} = 1010_2$
- Remember, since we have arithmetic right shift, we shift two positions to the right and copy 2 times the MSB (1) to the left. This results in:
  - $1110\cancel{10}$, that is 1110_2.
- If we convert back to decimal, we get $1110_2 = -2_{10}$.
- We said that arithmetic right shift by k divides the original number by 2^k rounding down to minus infinity.
  - Indeed $-6/4 = -1.5$ which is rounded down to -2.

124

We will start by converting the decimal number to its binary representation in two's complement, i.e. 1010. We shift two to the right, i.e. 10 and pad it with two MSB signs, i.e. 1110. That would be the number -2. Let's check that the division by 4 holds. Yes, $-6/4 = -1.5$ which is rounded down to -2.

PRACTICE TIME – Left shift

- What happens when we shift -10<<3 assuming we have 8 bits?

125

Your turn, assume you have 8 bits, and you try to shift -10 by 3 bits to the left.

ANSWER – Left shift

- What happens when we shift -10<<3 assuming we have 8 bits?
- We start by converting the decimal number to its binary representation using two's complement and sign extension:
  - $-10_{10} = 11110110_2$
- Remember, since we have left shift, we shift three positions to the left and add three 0s to the right. This results in:
  - ~~111~~10110000, that is 10110000_2.
- If we convert back to decimal, we get $10110000_2 = -80_{10}$.
  - Invert to 01001111 and then adding 1 results to $01010000 = 80$ so overall $10110000_2 = -80_{10}$
- We said that left shift by k multiplies the original number by 2^k.
  - Indeed $-10 \times 2^3 = -80$.

126

Start by converting to the two's complement representation in binary I,e, 10110. Pad it with 1s to reach 8 bits. 11110110.
Shift three to the left and add three 0s to the right. 10110000 which is equal to -80 and it checks out in terms of multiplying -10 by 2^3!

Shifting in Python

- Python supports left and arithmetic right shifting. It denotes them as << and >>, respectively.
- Other programming languages, like Java, support both arithmetic and logical right shift and use the >> and >>> operators to distinguish them.

127

Python supports left shifting and right arithmetic shifting using the << and >> operators. Other languages, like Java, also support logical right shifting making a distinction by using >>>.

Practice Problems

128

Time to practice at home

Practice Problems – Problem 1

- Convert the following decimal numbers to their *unsigned* binary representation and add them:
  - $45_{10} + 27_{10}$
  - How many bits do you need to not have overflow?
- Assume 6-bit signed numbers in two's complement representation. Add them and state whether overflow occurs:
  - $101110_2 + 010101_2$
  - Convert these three numbers to decimal to double check your work.
- Add the hex numbers 0xA7+0x5C
  - Convert these three numbers to decimal to double check your work.

Practice Problems – Answer 1

- Convert the following decimal numbers to their *unsigned* binary representation and add them:
  - $45_{10} + 27_{10} = 101101_2 + 011011_2 = 1001000$
  - How many bits do you need to not have overflow? 7
- Assume 6-bit signed numbers in two's complement representation. Add them and state whether overflow occurs:
  - $101110_2 + 010101_2 = 000011_2$. No overflow as we drop the left most bit (1).
  - Convert these three numbers to decimal to double check your work. -18+21=-3
- Add the hex numbers $A7_{16} + 5C_{16}$
  - 103_{16}
  - Convert these three numbers to decimal to double check your work. $167_{10} + 92_{10} = 259_{10}$

Practice Problems – Problem 2

- What will the results of the following operations be in binary and decimal assuming 8 bits?

- $-28_{10} \gg 2$

Practice Problems – Answer 2

- What will the results of the following operations be in binary and decimal assuming 8 bits?

- $-28_{10} \gg 2 = 11100100_2 \gg 2 = 11111001_2 = -7_{10}$