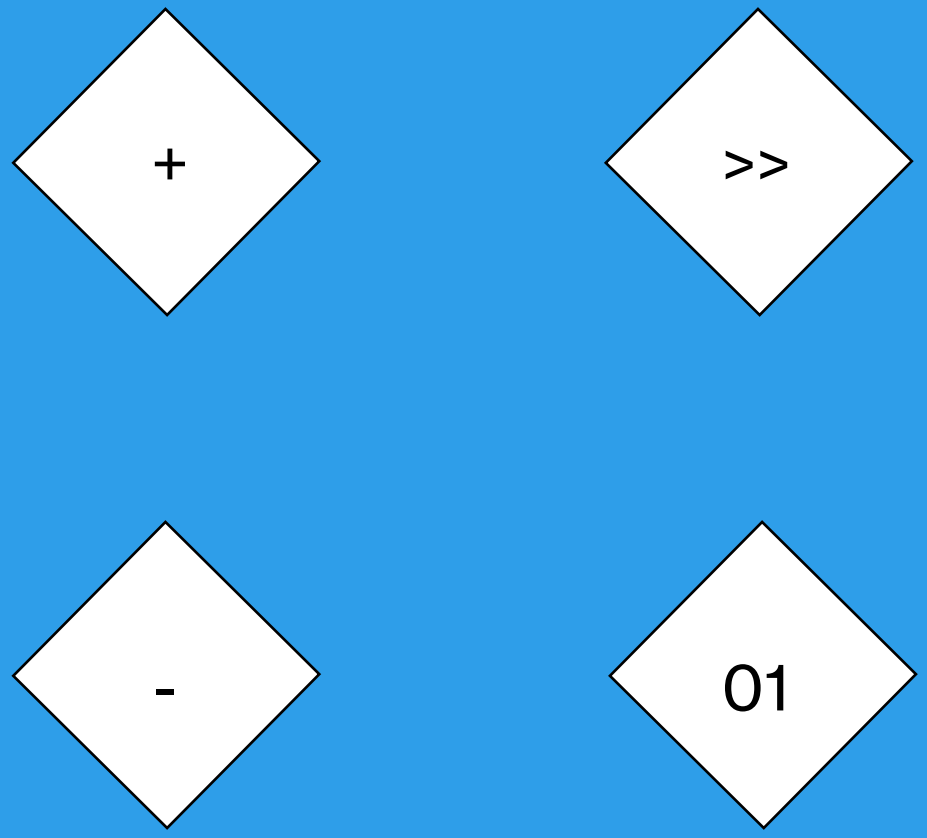


Computer Systems

Binary Arithmetic

CS51 – Fall 2026

Addition

Think

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

Think

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

0 1 1 1 ← carry

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

Think

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

Think

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

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

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?*

Think

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

1 0 1 ← carry

2 2 3_5 = 2*25 + 2*5 + 3*1 = 63

\+ 4 1 4_5 = 4*25 + 1*5 + 4*1 = 109

1 1 4 2_5 = 1* 125 + 1*25 + 4*5 + 2*1 = 172

0 0 1 ← carry

6 3_{10}

\+ 1 0 9_{10}

1 7 2_{10}

Think

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

Think

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

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

Think

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

  0 0 1 ← carry

  1_2

  $+ \quad 1\,0\,1_2$

  $1\,1\,0_2$

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

Think

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

$$\begin{array}{r} 0\ 0\ 1 \leftarrow \text{carry} \\ 1_2 = 1_{10} \\ +\ 1\ 0\ 1_2 = 4+1=5_{10} \\ \hline 1\ 1\ 0_2 = 4+2=6_{10} \end{array} \qquad \begin{array}{r} 0 \leftarrow \text{carry} \\ 1_{10} \\ +\ 5_{10} \\ \hline 6_{10} \end{array}$$

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

Think

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

0 0 1 1 ← carry

$$\begin{array}{r} 11\,F\,5_{16} \\ +\quad 1\,4\,E_{16} \\ \hline 1\,3\,4\,3_{16} \end{array}$$

Think

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

0 0 1 1 ← carry

$$\begin{array}{r} 11\,F\,5_{16} \\ +\quad 1\,4\,E_{16} \\ \hline 13\,4\,3_{16} \end{array}$$

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

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?*

PRACTICE TIME - Addition

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

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}

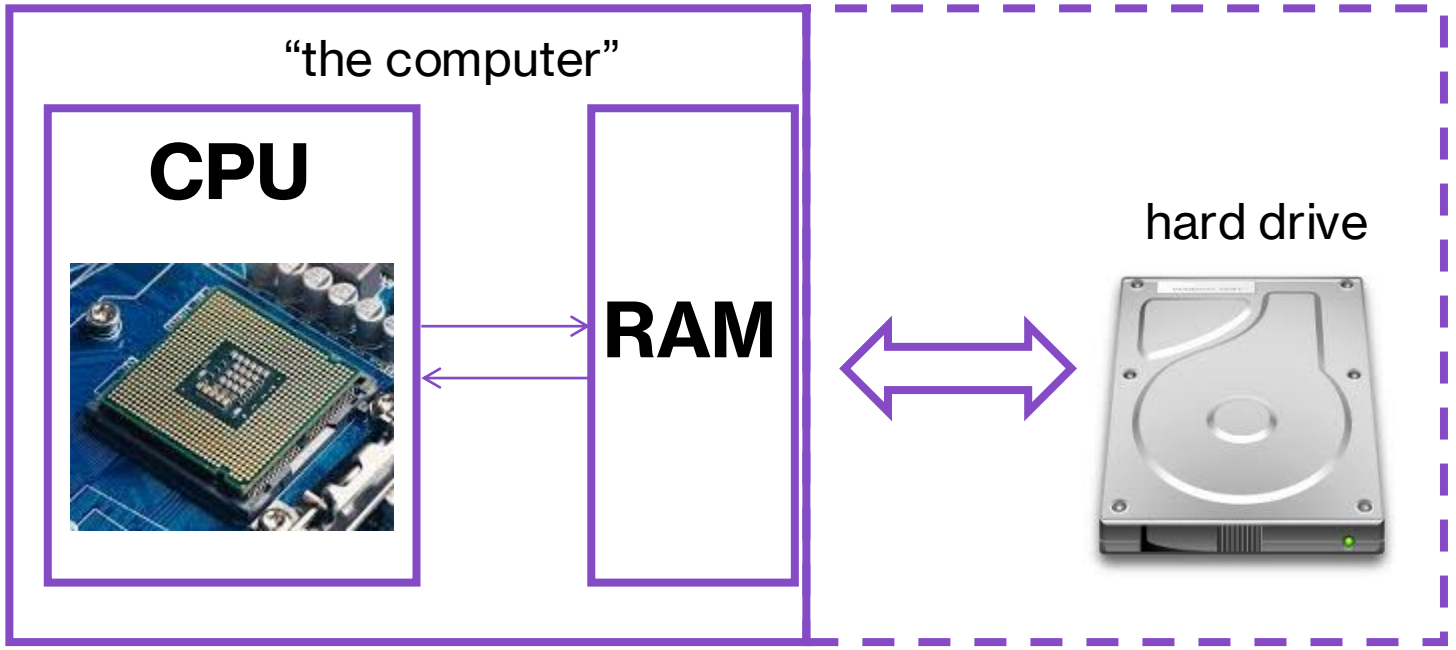
\+ 3 8_{16}

---

9 E B_{16}

Overflow

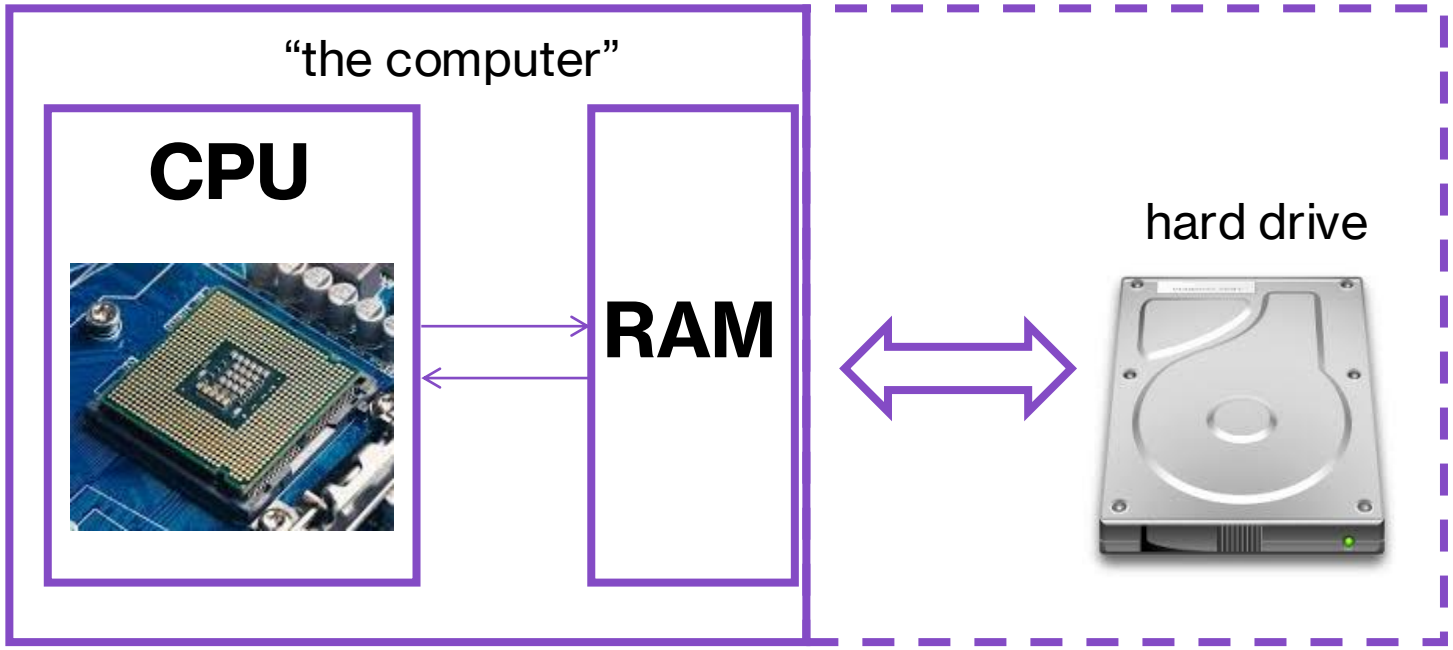
Simplified view of a computer



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

0100100100110100100101111000100010 10

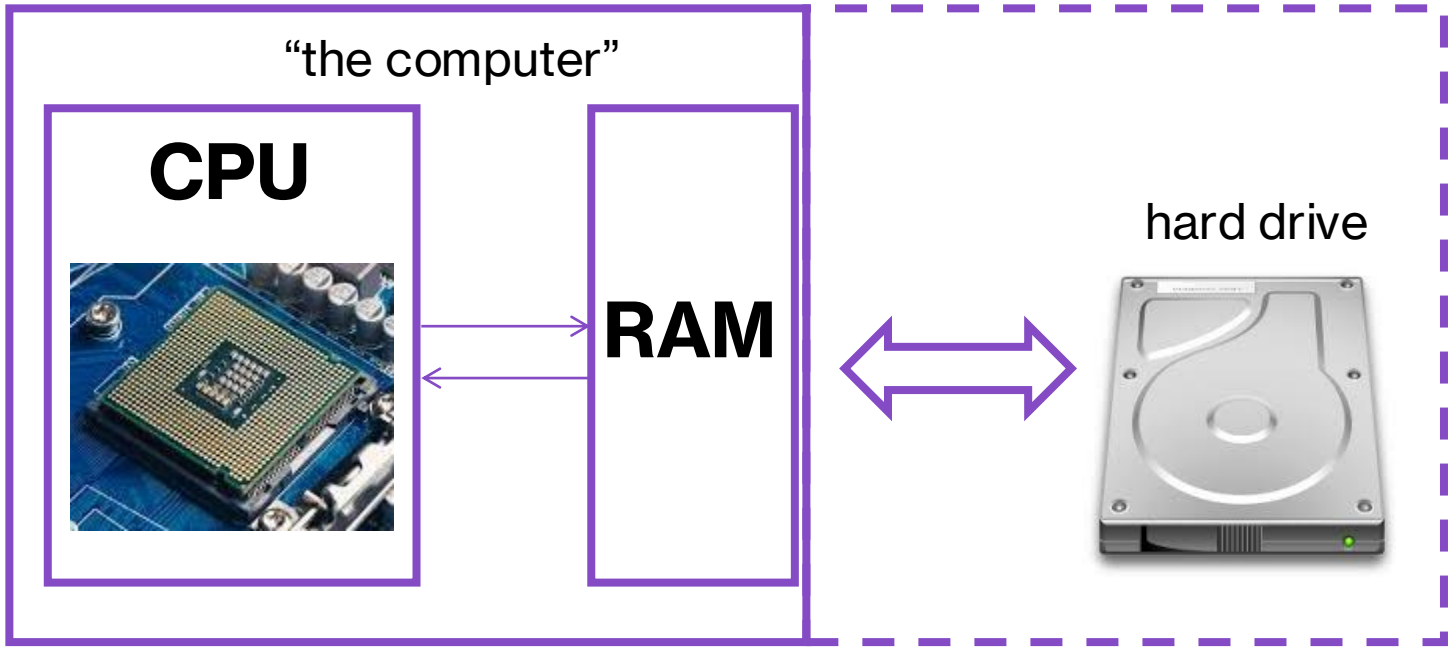
Word length



- 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

Computers today



- Most computers today work with 64-bit words.

Think

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

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

$1\ 0\ 0\ 1_2$

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

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

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.

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

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?*

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.

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

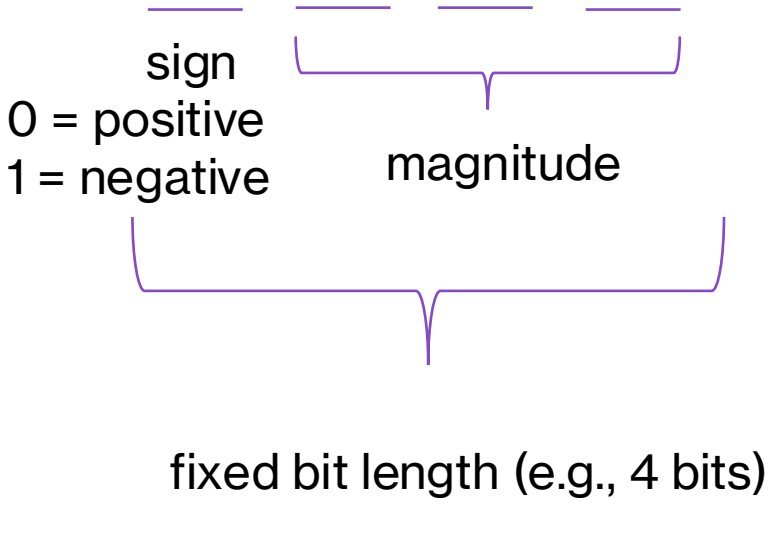
- *How can we detect overflow?*
  - If the carry bit for the MSB is 1, then we have overflow

Signed numbers

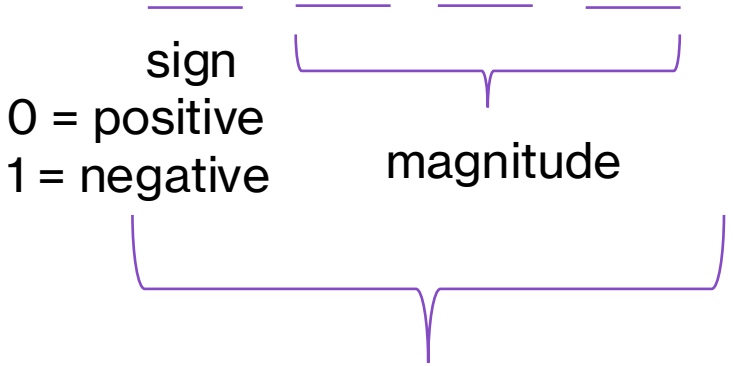
Binary numbers revisited

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

One option: sign/magnitude

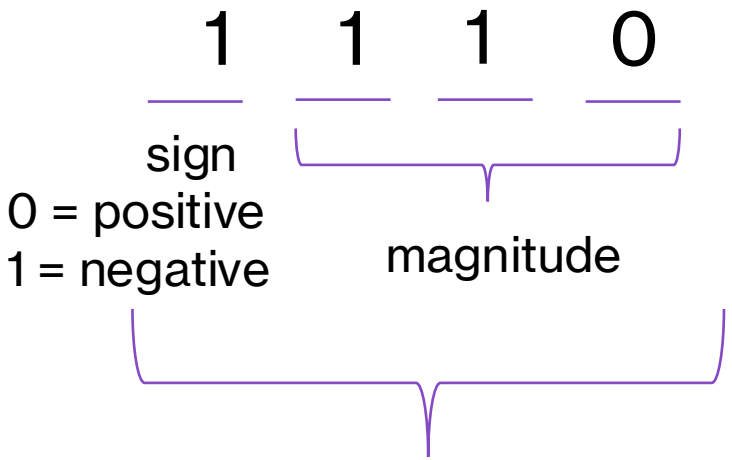


One option: sign/magnitude



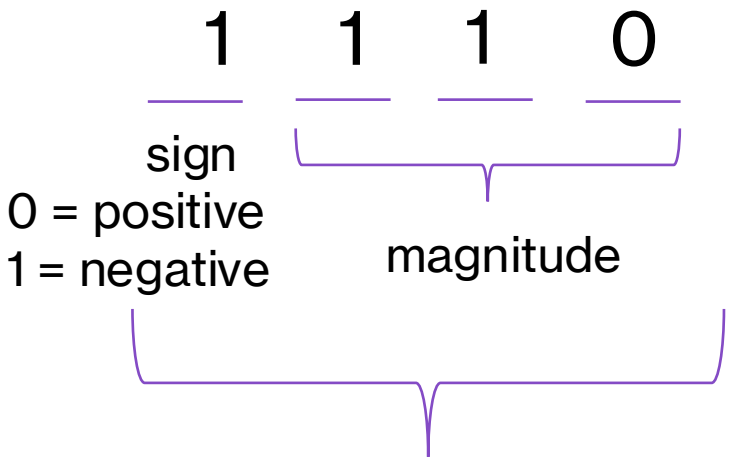
What is -6_{10} in sign/magnitude?

One option: sign/magnitude



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

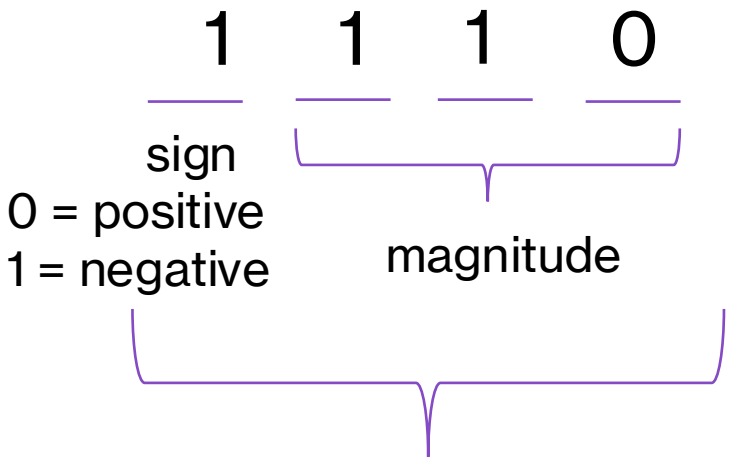
One option: sign/magnitude



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

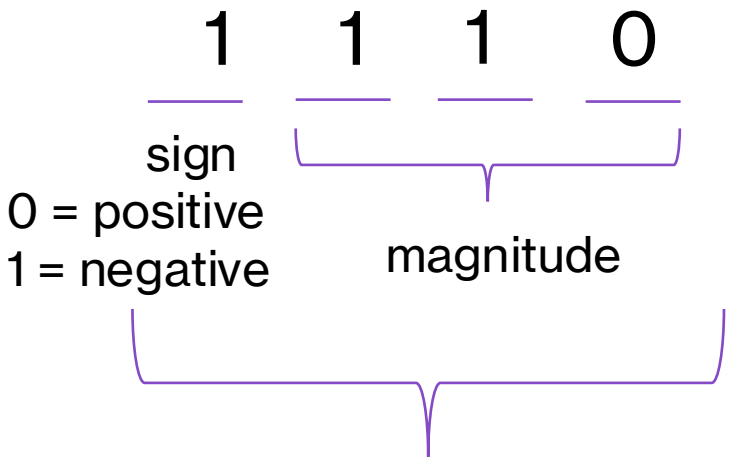
1111 = -7 to 0111 = 7

Think



Any problems?

Think



Any problems?

We have two zeros, 0000 and 1000

Think

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

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!

A better option: two's complement representation

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

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

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

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	

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

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	

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

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	

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

Think

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

unsigned

2^3	2^2	2^1	2^0

signed
(two's complement)

-2^3	2^2	2^1	2^0

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

Think

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

unsigned

2^3	2^2	2^1	2^0

signed
(two's complement)

-2^3	2^2	2^1	2^0

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

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

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	

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	

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	

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

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?

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

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

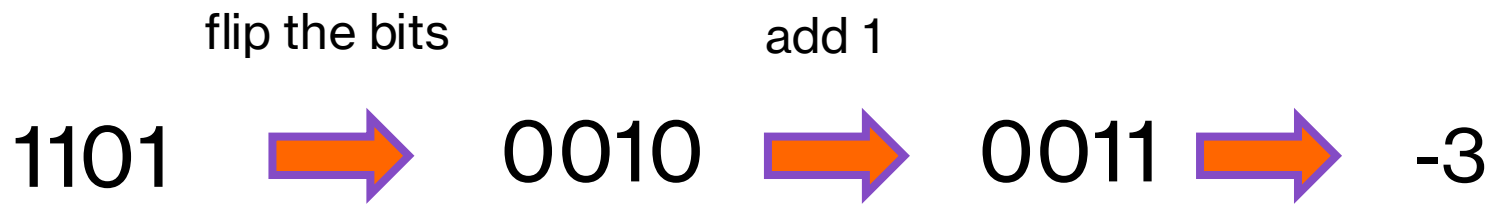
binary representation	**unsigned**	**two's complement**
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	-8
1001	9	-7
1010	10	-6
1011	11	-5
1100	12	-4
1101	13	-3
1110	14	-2
1111	15	-1

-1 is always all 1s

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



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

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

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

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

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

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

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

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

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

$$
\begin{array}{rr}
 & {\scriptstyle 1\ 0\ 0} \\
 & 0110 \\
+ & 0101 \\
\hline
 & 1011?
\end{array}
$$

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)

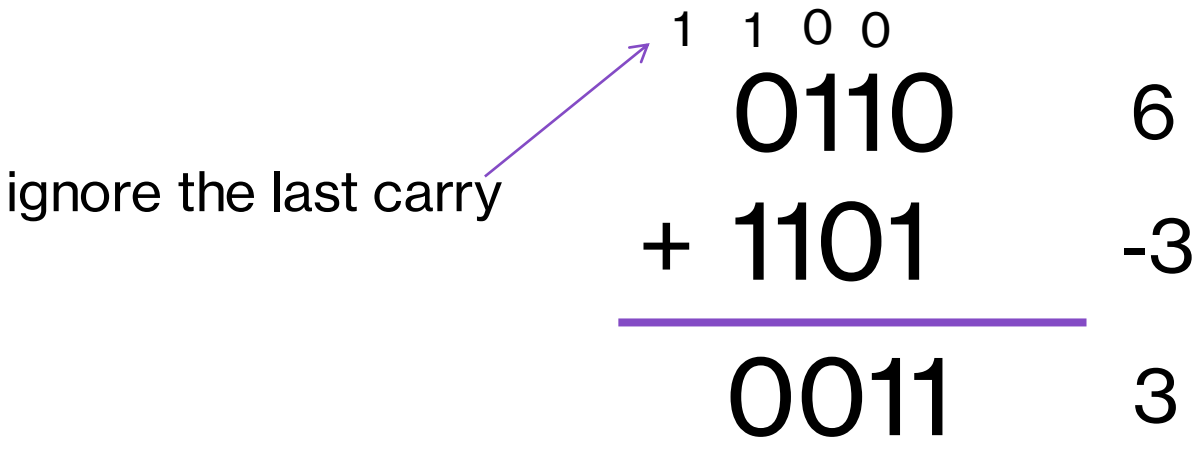
Addition with 4-bit *two's complement* numbers

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

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

$$
\begin{array}{r}
\ 1\ \ 1\ \ 0\ \ 0 \\
\ 0110 \\
+\ 1101 \\
\hline
\ 0011
\end{array}
$$

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



Think

- *How do you know that overflow has occurred with 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

Subtraction

Subtraction

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

PRACTICE TIME – Subtracting two’s complements

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

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 $1110_2 = -2_{10}$

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.

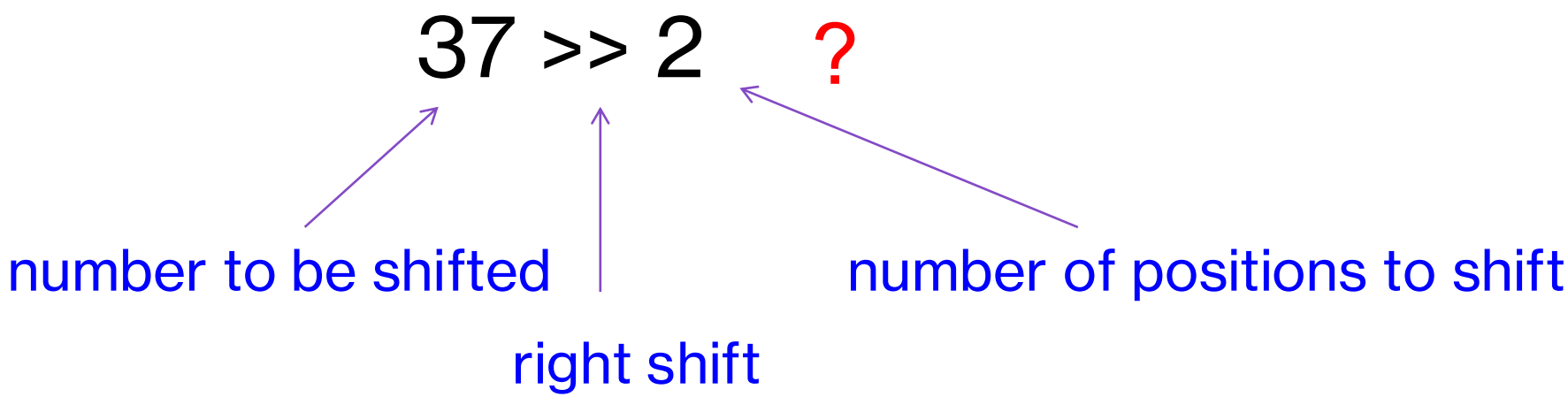
Why number representation errors matter

- 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.

Shifting

Shifting: variable length numbers

Shifting shifts the binary representation of the number right or left



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

Think

Shifting shifts the binary representation of the number right or left

37 >> 3 ?

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)

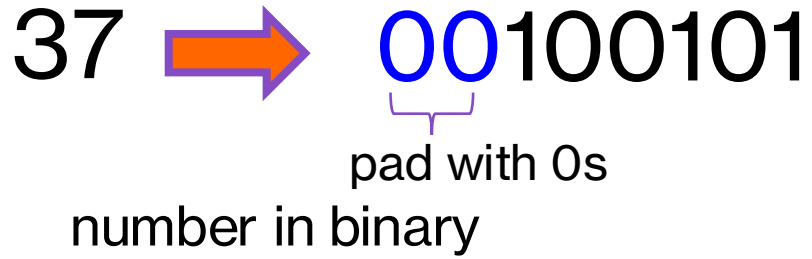
decimal form

Shifting 8-bit numbers

Shifting shifts the binary representation of the number right or left

37 >> 2

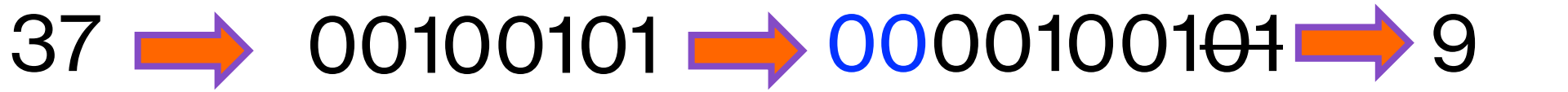
What is 37 as an 8-bit binary number?



Shifting 8-bit numbers

Shifting shifts the binary representation of the number right or left

37 >> 2



number in binary

shift right two positions
(discard away bits shifting off and pad)

decimal form

Think

Shifting shifts the binary representation of the number right or left

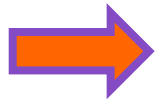
15 << 2

?

Think

Shifting shifts the binary representation of the number right or left

15 << 2

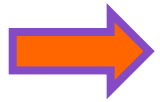
15  ?

number in binary

Shifting 8-bit numbers

Shifting shifts the binary representation of the number right or left

15 << 2

15  00001111

number in binary

Think

Shifting shifts the binary representation of the number right or left

15 << 2

number in binary

shift left two positions
(discard bits shifting off)

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)

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)

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)

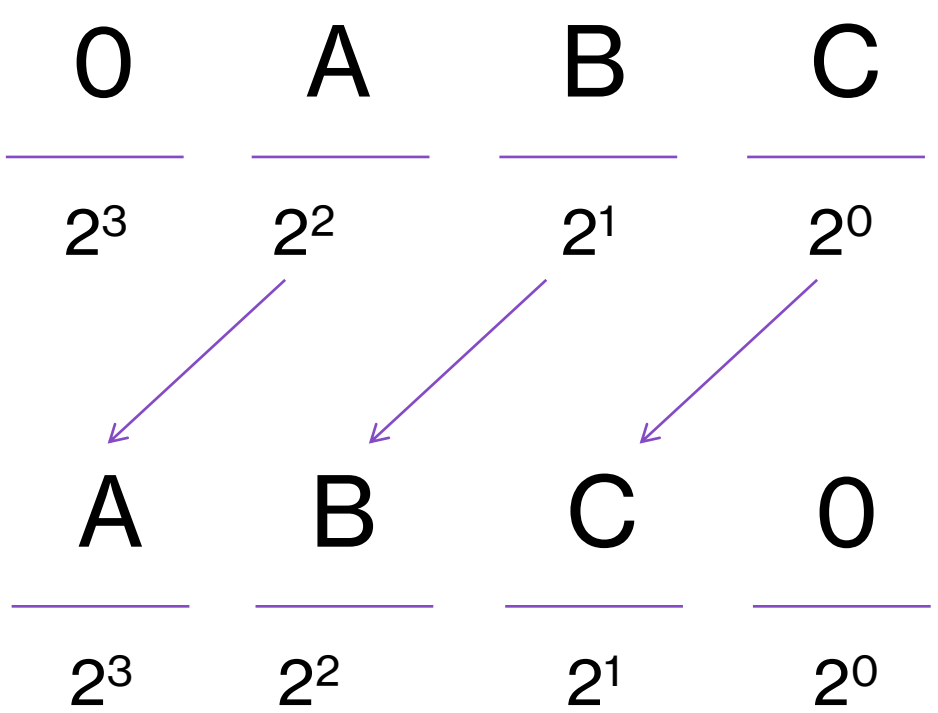
decimal form

Think

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

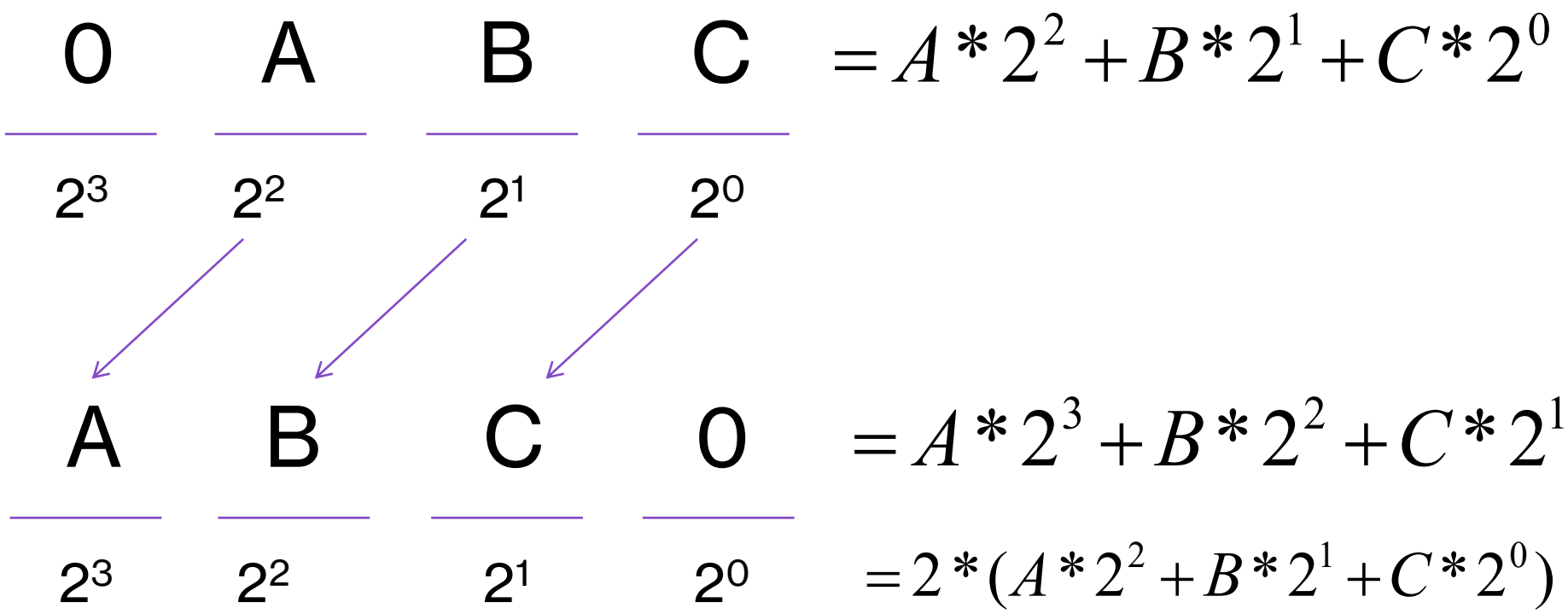
Shifting mathematically: unsigned

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



Shifting mathematically: unsigned

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



Shifting mathematically: unsigned

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

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

$$= A*2^2 + B*2^1 + C*2^0$$

Doubles the number!

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

$$= A*2^3 + B*2^2 + C*2^1$$

$$= 2*(A*2^2 + B*2^1 + C*2^0)$$

Think

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

Think

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

Multiply by 2^n (doubles it n times)

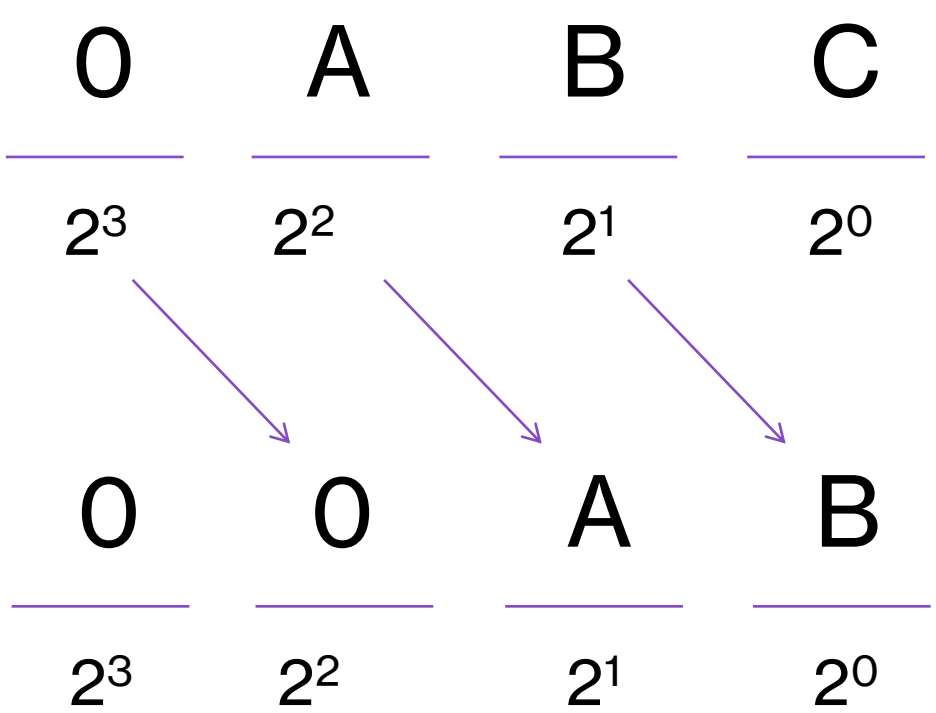
Shifting mathematically: unsigned

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

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

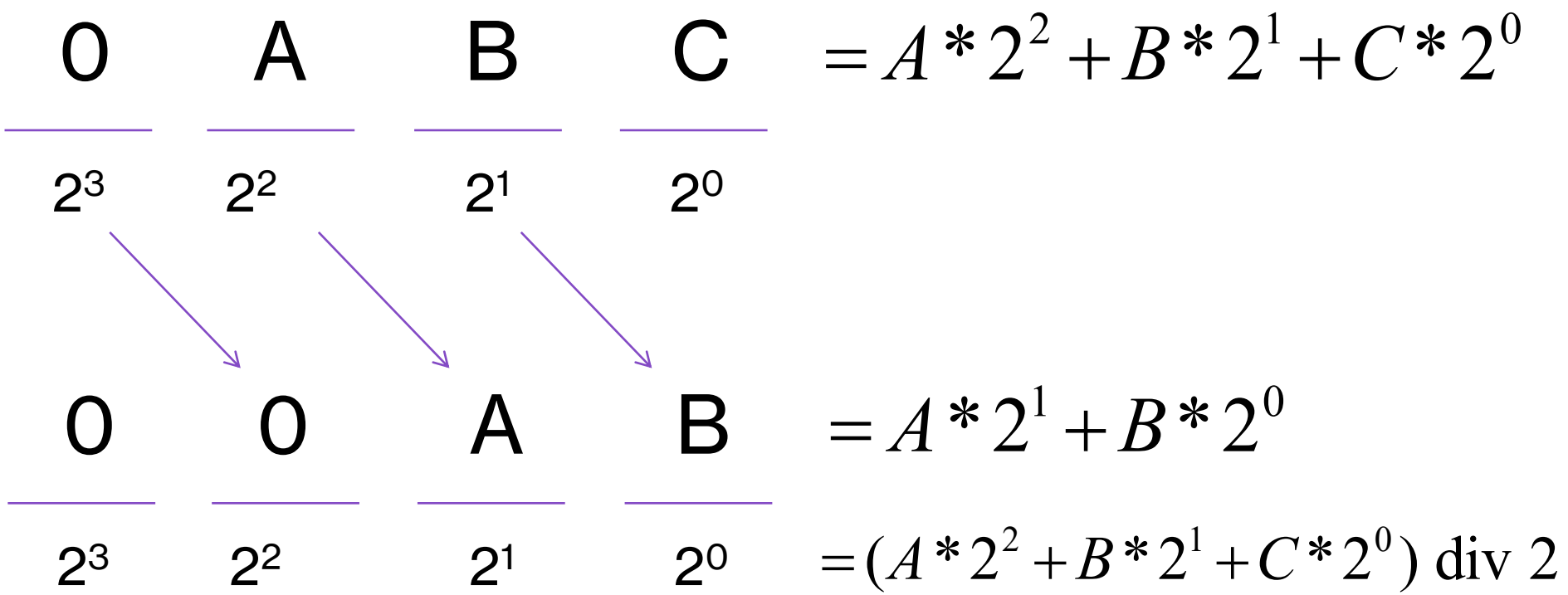
Shifting mathematically: unsigned

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



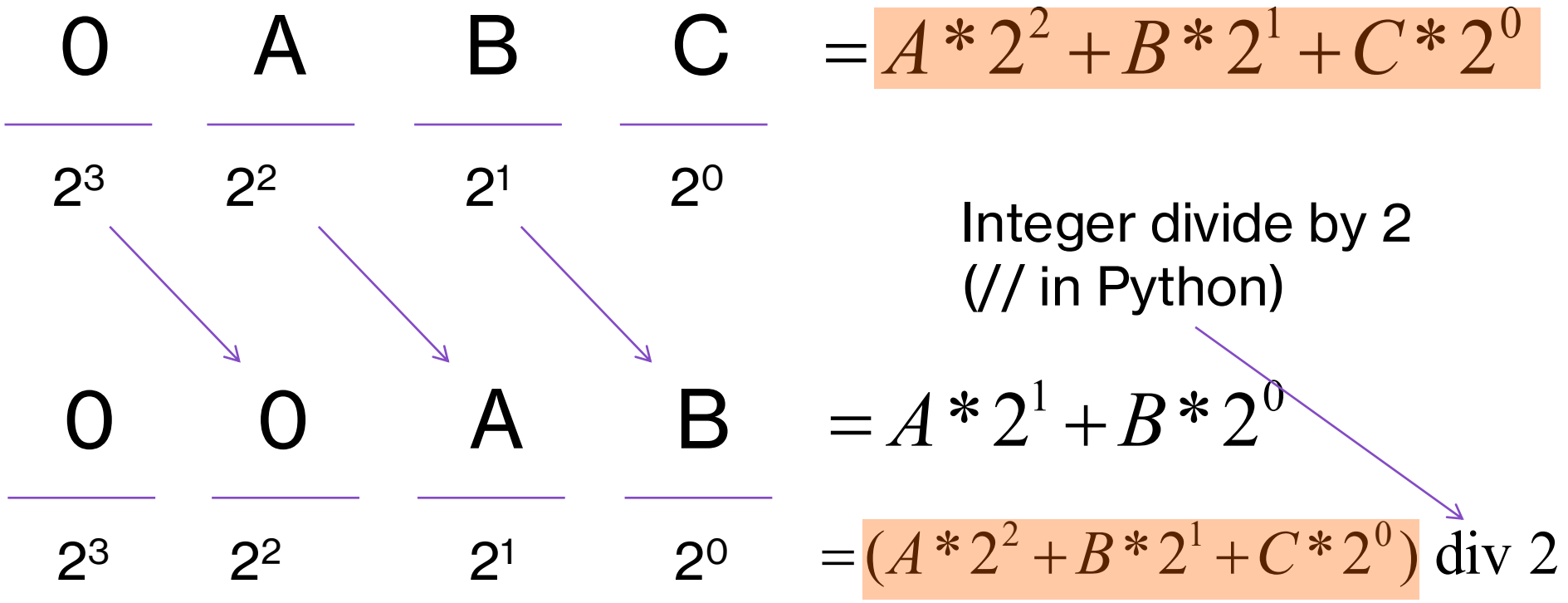
Shifting mathematically: unsigned

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



Shifting mathematically: unsigned

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



Think

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

Think

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

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

Think

Shifting shifts the binary representation of the number right or left

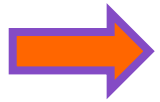
$$-4 >> 1$$

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

Shifting 4-bit numbers

Shifting shifts the binary representation of the number right or left

-4 >> 1

-4  1100

number in binary

Think

Shifting shifts the binary representation of the number right or left

-4 >> 1

How do we fill in the leftmost bit?

number in binary

shift right one position
(discard bits shifting off)

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)

Think

- 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)

Think

- 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)

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

number in binary

shift right one position
(discard bits shifting off)

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**

-4 >> 1

number in binary

shift right one position
(discard bits shifting off)

decimal form

Think

Shifting shifts the binary representation of the number right or left

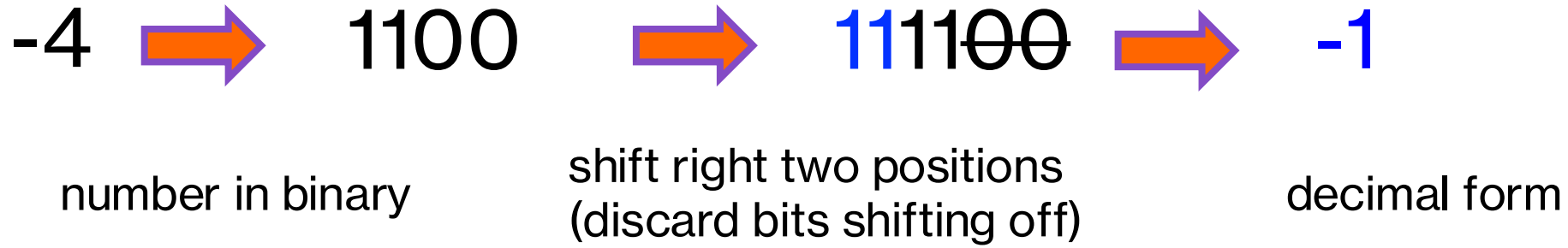
$$-4 >> 2$$

?

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



Think

*What does **right** arithmetic shifting by **n** positions do mathematically for **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!!

Think

- 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)

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)

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

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

number in binary

shift right one position
(discard bits shifting off)

decimal form

Think

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

arithmetic

-3 << 1 ?

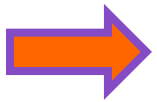
logical

-3 <<< 1 ?

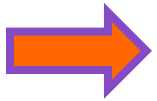
Left shifts

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

arithmetic

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

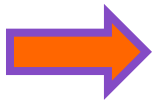
logical

-3 <<< 1 1101  11010

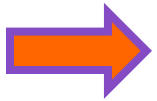
Left shifts

- 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

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)

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}.

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:
  - 111010, 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.

PRACTICE TIME – Left shift

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

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 $-10 \times 2^3 = -80$.

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.

Practice Problems

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?

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