Compiled on 30/08/2022 at 1:51pm

Data Representations

What does the sequence 1000011 represent?

The answer is: it depends. It depends on what data representation is used.

- If we are told that this is a **decimal** number, then this represents the number one million and eleven.
- If it is a **binary** number, then it **could** represent 67 or -3 or -61 since we still haven't been told what system was used (**unsigned** vs. **signed in sign-magnitude** vs. **signed in Two's complement**).
- It might be the representation of the character C (**ASCII** representation).
- It might actually be part of a longer sequence that represents a **machine instruction**.
- Or it might be that this is just garbage, as it was part of memory that has not been initialized.

Can you think of other things this sequence might represent?
[1]

In other words, what a sequence of bits represents is completely dependent on the data representation. In this module, we will learn about different number representations.

1 Binary Representation

Definition 1 *A **bit** is a **bi**nary digi**t**. That is, a 0 or 1 (on or off).*

A bit is the smallest unit of data that can be represented in a computer.

Definition 2 *A **nibble** is 4 bits.*

Example: 1001.

Definition 3 *A **byte** is 8 bits.*

Example: 10011101.

[1] An array of 7 booleans, the base 3 number 733 etc.

Definition 4 *A **word** is a machine-specific grouping of bytes. For us, a word will be 4 bytes (i.e., 32 bits, as we're using a 32-bit architecture). Note that 8-byte (for 64-bit architectures) words are more common in modern devices.*

Example: 10011101100111011001110110011101.
When writing longer sequences of bits, we will often add whitespace, either after every nibble or after every byte. This does not affect the meaning of the bit sequence and is just to make it easier to read.

Definition 5 *The **Most Significant Bit (MSB)** is the left-most bit.*

Definition 6 *The **Least Significant Bit (LSB)** is the right-most bit.*

We will discuss two types of binary representations for numbers:

- Unsigned (non-negative integers)
- Signed integers

However, there are many others (floating point, algebraic, etc.)

1.1 Unsigned Integers in Binary

This is a positional number system that works exactly how the unsigned (non-negative) decimal system works. The only difference is that positional values represent powers of 2 instead of 10. Therefore, the value of an integer stored using unsigned binary representation is the binary sum. For example, given an 8-bit unsigned binary number $b_7b_6b_5b_4b_3b_2b_1b_0$, its value in decimal is

$$b_7 2^7 + b_6 2^6 + b_5 2^5 + b_4 2^4 + b_3 2^3 + b_2 2^2 + b_1 2^1 + b_0$$

For example,

$$01010101_2 = 2^6 + 2^4 + 2^2 + 2^0 = 64 + 16 + 4 + 1 = 85_{10}$$

or

$$\begin{aligned} 11111111_2 &= 2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0 \\ &= 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 \\ &= 255_{10} \end{aligned}$$

1.1.1 Arithmetic with Unsigned Binary Integers

Arithmetic with unsigned binary integers works in the normal way:

$$\begin{array}{r} {\scriptstyle 1111\ 111} \\ 0100\,1001 \\ +\,0111\,1111 \\ \hline 1100\,1000 \end{array}$$

As always, we have to watch out for overflow errors! [2]

[2] Number representations in computers are often fixed in size. For example, an integer might be restricted to taking up exactly 32 bits. This means that if the result takes up less than 32 bits, it must be padded with zeroes. If the result takes up more than 32 bits—for example, if there is a carry when the last pair of bits are added—then this is called overflow. There are a variety of ways to handle overflow, but usually the overflowing bits are simply ignored.

1.1.2 Converting from Decimal to Binary

We will look at two ways of converting a decimal number to its binary representation.

Approach 1: Break the number into factors that are powers of 2. For example, the decimal value 38 when written as a sum of powers of 2 is $32 + 4 + 2$ which is $2^5 + 2^2 + 2^1$ or $(1 \times 2^5) + (0 \times 2^4) + (0 \times 2^3) + (1 \times 2^2) + (1 \times 2^1) + (0 \times 1) = 100110_2$.

One way of doing this is to take the largest power of 2 less than the number, subtract and repeat. For example, 32 is the largest power of two less than 38, subtracting gives 6. Next, 4 is the largest power of two less than 6 and subtracting gives 2. This is a power of two hence $38 = 32 + 4 + 2 = 100110_2$.

Try representing the decimal value 67 in binary using this approach (answer in footnote)
[3]

Approach 2: Repeatedly divide the number by 2.

Number	Quotient	Remainder
38	19	0
19	9	1
9	4	1
4	2	0
2	1	0
1	0	1

The binary representation of 38 is found by reading the remainders from the bottom to the top. To understand why this works, and why the answer is read bottom to top, consider:

$$N = b_0 + 2b_1 + 2^2 b_2 + ...$$

The remainder when dividing N by 2 gives the b_0 value. After doing $(N - b_0)/2$, we end up with

$$\frac{N - b_0}{2} = b_1 + 2b_2 + 2^2 b_3 + ...$$

and we can repeat the process.

What is the value in binary you get when you add 7 and 5 together in 1-byte unsigned integers? (Answer is footnote.)
[4]

Using fixed size 8-bit unsigned binary representation, write the decimal numbers 241 and 16 in binary, then add them. What is the result? (Answer in footnote.)
[5]

[3] $67 = 64 + 2 + 1 = 1000011_2$

[4] 00001100_2 (note the leading 0s are added to represent the answer using 8 bits.)

[5] 241 is 11110001 and 16 is 00010000 in 8-bit unsigned binary representation. Adding them gives 100000001, but this is 9 bits. This means that adding 241 and 16 using a fixed length of 8 bits causes overflow. One way overflow is handled is to discard the overflowing leftmost bit to obtain 00000001.

1.2 Signed Integers in Binary

A computer must obviously represent both positive and negative integers in binary (and zero). One approach is to use **Sign-Magnitude**: Use the first bit as an indicator of the sign (a sign bit); use 0 to indicate a positive integer, 1 to indicate negative. This approach has multiple problems and is not used in practice. Two problems are:

- Two representations for 0 (00000000 and 10000000 both represent 0 in a 1-byte representation). This is both wasteful and awkward.
- Arithmetic is tricky; will the sum of a positive or negative number be positive or negative? It depends! This leads to needing more complex hardware circuitry to do arithmetic.

The more common approach to representing signed integers, and the one we use in this course, is the **Two's complement form**:

- It is similar to sign-magnitude in spirit; first bit is 0 if non-negative, 1 if negative.
- To negate a value: Take the complement of all bits (flip the 0 bits to 1 and 1 bits to 0) and add 1. A slightly faster way is to locate the rightmost 1 bit and flip all the bits to the left of it.

  11011010 Negating 00100110

  Note: Flipping the bits and adding 1 is the same as subtracting 1 and flipping the bits for non-zero numbers.

1.2.1 Converting To and From Two's Complement

Let's compute -38_{10} using one-byte Two's complement. First, write 38 in binary: $38_{10} = 00100110_2$. Next, take the complement of all the bits 11011001_2. Finally, add 1: 11011010_2. This last value is -38_{10}.

To convert 11011010_2, a number in Two's complement representation, to decimal, one method is to flip the bits and add 1: $00100110_2 = 2^5 + 2^2 + 2^1 = 38$. Thus, the corresponding positive number is 38 and so the original number is -38.

Another way to do this computation is to treat the original number 11011010_2 as an unsigned number, convert to decimal and subtract 2^8 from it (since we have 8 bits, and the first bit is a 1 meaning it should be a negative value). This also gives -38:

$$\begin{aligned} 11011010_2 &= 2^7 + 2^6 + 2^4 + 2^3 + 2^1 - 2^8 \\ &= 128 + 64 + 16 + 8 + 2 - 256 \\ &= 218 - 256 = -38 \end{aligned}$$

The idea behind [one byte] Two's Complement notation is based on the following observations:

- The range for unsigned integers is 0 to 255. Recall that 255 is 11111111_2. If we add 1 to 255, then, after discarding overflow bits, we get the number 0.
- Thus, let's treat 2^8 as 0, i.e., let's work modulo $2^8 = 256$. In this vein, we set up a correspondence between the positive integer k and the unsigned integer $2^8 - k$. Since we are working modulo 2^8, subtracting a positive integer k from 0 is the same as subtracting it from 2^8.

- In this case, note that $2^8 - 1 = 2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0$ and in general:

$$2^n - 1 = \sum_{i=0}^{n-1} 2^i$$

Equivalently, consider overflow. If we take binary number 00000000 and subtract 00000010 (2), we'll get 11111110, ignoring the overflowing carry bit. 11111110 is the 8-bit Two's complement representation of -2, which is, of course, $0-2$. 11111110 is also the value we find using the approach above; in fact, taking the complement of the bits and adding 1 is equivalent to subtracting from 0. Two's complement arose from using the results of overflowing (unsigned) arithmetic to represent signed numbers.
As an explicit example (which can be generalized naturally) take a number, say $38_{10} = 00100110_2 = 2^5 + 2^2 + 2^1$. What should the corresponding negative number be? Well,

$$\begin{aligned} 2^8 - 1 &= 2^7 + 2^6 + \mathbf{2^5} + 2^4 + 2^3 + \mathbf{2^2} + \mathbf{2^1} + 2^0 \\ 2^8 - 1 &= \mathbf{38} + 2^7 + 2^6 + 2^4 + 2^3 + 2^0 \\ 2^8 - \mathbf{38} &= \underbrace{2^7 + 2^6 + 2^4 + 2^3 + 2^0}_{\text{flip the bits}} + \underbrace{1}_{add\ 1} \end{aligned}$$

We mentioned that another method of negating a two's complement number is to flip the bits to the left of the rightmost 1. Can you justify why this technique works?
[6]

1.2.2 Arithmetic of Signed Integers

The main difference between signed and unsigned binary arithmetic is that we are now working modulo 256 (or, more generally, 2^n in the case of n-bit Two's complement numbers). But, when you are adding or subtracting unsigned integers and discard overflowing bits, this is actually equivalent to working modulo a power of two. So, at least for addition and subtraction, there is no difference between signed and unsigned arithmetic! Multiplication and division do need different handling though to make sure the signs work out correctly.

$$\begin{array}{rl} {\scriptstyle 1\,1\,1\,1\ 1} & \\ 0000\,0100 & (+4) \\ +\,1111\,1101 & (-3) \\ \hline 0000\,0001 & \end{array} \qquad \begin{array}{rl} {\scriptstyle 1\,1\,1\,1\ 1} & \\ 1111\,1100 & (-4) \\ +\,1111\,1101 & (-3) \\ \hline 1111\,1001 & \end{array}$$

When working in Two's complement, overflow occurs when adding numbers if the original two numbers have the same sign, but the result has a different sign.

What is the decimal value for 11111111_2 in Two's complement notation?
[7]

[6] Every bit to the right of the rightmost 1 is a 0. When we "flip the bits", these become 1s. When we "add 1", the carry propagates up until the position of the "rightmost 1". Try negating the number 64 (01000000) to see this in action.

[7] Flip the bits: 00000000_2, add the one, 00000001_2. This number is 1 so the original number must be -1.

What is the decimal value for 10000000_2 in Two's Complement notation?
[8]

What is the range of numbers expressible in one-byte Two's Complement notation?
[9]

2 Hexadecimal Notation

Writing and reading values in binary can be tedious. A popular representation system is to use base-16 instead of binary.

Definition 7 *The base-16 representation system is called the* ***hexadecimal*** *system. It consists of the numbers from 0 to 9 and the letters A, B, C, D, E and F (which convert to the numbers from* 10 *to* 15 *in decimal notation).*

Similar to how values in decimal and binary work, given an unsigned number in hexadecimal notation $h_7h_6h_5h_4h_3h_2h_1h_0$, its value in decimal is

$$h_7 16^7 + h_6 16^6 + h_5 16^5 + h_4 16^4 + h_3 16^3 + h_2 16^2 + h_1 16^1 + h_0$$

For example, the hexadecimal number $F4A$ is $15 \times 16^2 + 4 \times 16 + 10 = 3914$. The conversion from decimal to hexadecimal works the same way as the conversion from decimal to binary; you can write the decimal number as a sum of powers of 16 or alternately repeatedly divide by 16.

Number	Quotient	Remainder
3914	244	10
244	15	4
15	0	15

The hexadecimal representation of 3914 is found by reading the remainders from the bottom to top, 15, 4 then 10, or $F4A_{16}$.

The reason why hexadecimal is popular is that it's straightforward to convert between binary and hexadecimal. Each hexadecimal digit is a nibble (4 bits). Therefore, one can convert from binary to hexadecimal by converting each nibble in a binary number into a corresponding hexadecimal digit, and vice-versa. For example, the binary number 1001 1101 (space added to show the nibbles) will convert to $9D_{16}$. A more common representation of hexadecimal values in computer science is to use the notation `0x9D` instead of $9D_{16}$ (this notation was introduced in the C programming language and is used in many modern programming languages).

What is the corresponding hexadecimal representation for this word (spaces included for convenience)? 1101 1110 1111 1010 0000 0001 0010 0011
[10]

What is the hexadecimal value you get when you add the decimal numbers -11 and 6, represented as a word in two's complement notation?

[8] Flip the bits: 01111111_2, add the one, 10000000_2. This number is -128.
[9] -128 to 127
[10] `0xDEFA0123`

11

3 ASCII Representation

Recall the sequence 1000011 from the start of this module. One possible answer for what this sequence represents was the uppercase Latin (and English) letter C. Although computers are computeers, it is useful for them to represent information other than integers. However, since computers store everything as 1s and 0s, a convention is needed so that each character is assigned a unique value. The ASCII (American Standard Code for Information Interchange) standard was developed in the 60s as such a standard based on the English alphabet. As ASCII is an American standard based on the English alphabet, it can't represent letters such as ç or glyphs from other languages. It used 7 bits (128 unique values) to represent the English alphabet (both lower and upper case), decimal digits and punctuation symbols. Ninety-five of the unique values are used to encode printable characters whereas the rest are non-printing control codes originating from Teletype machines. Most of the control codes, other than carriage return, line feed and tab, are now obsolete. Although ASCII uses 7 bits, it's far more common to use bytes (8 bits) to store ASCII characters, thus wasting one bit per byte (though this bit has found many other uses over the decades).

Take a look at the ASCII table below. One quick way to access this table from your Unix command line is to type the command: `man ascii`.

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	□

[11] Adding -11 and 6 gives us -5 in decimal which is 1111 1111 1111 1111 1111 1111 1111 1011 as a 32-bit word in Two's complement notation. This is 0xFFFFFFFB in hexadecimal.

Highlights:

- Characters 0-31 are control characters
- Characters 48-57 are the numbers 0 to 9
- Characters 65-90 are the letters A to Z
- Characters 97-122 are the letters a to z
- Note that 'A' and 'a' are 32 away from each other

What is the ASCII value for A? What about c?

[12]

What is the ASCII value for the digit 0? How many bits would be used to store the ASCII code for the digit 0?

[13]

As discussed in the answers to the question above, the ASCII for 0 is 48. This means that when you press 0 on your keyboard, while it appears on your screen as the symbol we recognize as 0, it is only because the display chose to interpret the value 48 as the symbol 0. The computer actually stores your keystroke as 48 (in binary of course). Whether this displays as the digit 0 or the decimal value 48 (or the binary value `00110000`), is dependent on how the information is interpreted. We discuss this in more detail at the end of this module.

The ASCII representation is restrictive and outdated. In particular, the ASCII encoding does not support letters from many other languages. Nowadays the common standard is Unicode, which itself contains multiple character encodings such as UTF-8, UTF-16 and UTF-32. The discussion of Unicode encodings is beyond the scope of this course, but an important observation is that Unicode, and UTF-8 in particular, is intentionally backwards compatible, and all ASCII characters when represented as bytes have the same representation in UTF-8.

4 Bit-Wise Operations

Most programming languages, including C, C++ and Racket, support operators to perform common bit-wise operations on binary numbers. It is worth learning these as they allow for manipulating data at the bit level. Very soon in the course, we will be generating data in binary, and these operations will come very handy.

Bitwise AND (&): Computes the value given by applying the logical AND operation to each pair of bits in the operands. Notice below that a 1 bit appears in the resulting value if and only if there is a 1 bit at the corresponding position in both of the operands.

[12] The ASCII value for upper-case A is 65, and lower-case c is 99.

[13] The digit 0 has the ASCII value of 48. Technically, the number of bits needed to store the ASCII code for the digit 0 is 6 bits (as the ASCII value is 48). But ASCII was designed as a 7-bit code, so 7 is also a valid answer. But modern computers typically use 8 bits per ASCII character, even though they technically only need 7 when "Extended ASCII" is not in use, so 8 is also a valid answer and likely the most accurate.

```
Truth Table:
a   b   a&b      Example:
0   0    0         0001 0010 0011 0100 0101 0110 0111 1000
0   1    0       & 1000 0111 0110 0101 0100 0011 0010 0001
1   0    0         -----------------------------------------
1   1    1         0000 0010 0010 0100 0100 0010 0010 0000
```

Suppose we have two binary sequences that represent sets of numbers. At position i in the sequence, a 1 bit means that the number i is in the set, and a 0 bit means that the number i is not in the set. What set does the bitwise AND of these two sequences represent?

[14]

Bitwise OR (|): Computes the value given by applying the logical OR operation to each pair of bits in the operands. Notice below that a 0 bit appears in the resulting value if and only if there is a 0 bit at the corresponding position in both of the operands.

```
Truth Table:
a   b   a|b      Example:
0   0    0         0001 0010 0011 0100 0101 0110 0111 1000
0   1    1       | 1000 0111 0110 0101 0100 0011 0010 0001
1   0    1         -----------------------------------------
1   1    1         1001 0111 0111 0101 0101 0111 0111 1001
```

Suppose we have two binary sequences that represent sets of numbers. At position i in the sequence, a 1 bit means that the number i is in the set, and a 0 bit means that the number i is not in the set. What set does the bitwise OR of these two sequences represent?

[15]

Left Bit Shift (<<): The left bit shift operator, $x << n$, is equivalent to $x \times 2^n$. It does so by shifting the bits in the given value left by padding additional 0 bits on the right; any bits that go beyond the fixed size representation are simply discarded.

```
x       n   x << n       Example: 0001 0010 0011 0100 0101 0110 0111 1000 << 16
0001    0    0001                 = 0101 0110 0111 1000 0000 0000 0000 0000
0001    1    0010
0001    2    0100
0001    3    1000
```

Right Bit Shift (>>): Our intended use of the Right Bit Shift operator is to perform arithmetic right shift, i.e., we expect $x >> n$ to be equivalent to $x/2^n$. If the most significant bit is 0, this can be achieved by padding 0s on the left. However, if the most significant bit is 1, it is important to know if we are dealing with signed or unsigned values. If the most significant bit is 1 for an unsigned value, 0s must be padded on the left. However, if the most significant bit is 1 for a signed value (Two's complement representation), then the 1 at the most significant bit position indicates that this is a negative number. For the number to continue to be negative, 1s must be padded on the left. It is best to test your programming language to confirm that the >> operator implements arithmetic right shift.

[14] The bitwise AND represents the intersection of the two sets.
[15] The bitwise OR represents the union of the two sets.

```
x       n   x >> n     Example: 0001 0010 0011 0100 0101 0110 0111 1000 >> 16
0100    0    0100                = 0000 0000 0000 0000 0001 0010 0011 0100
0100    1    0010
0100    2    0001
0100    3    0000

1000    0    1000      Example: 1101 0010 0011 0100 0101 0110 0111 1000 >> 16
1000    1    1100                = 1111 1111 1111 1111 1101 0010 0011 0100
1000    2    1110
1000    3    1111
```

The following C program illustrates some of these operations (C++ behaves similarly):

```c
#include <stdio.h>

int main(void){
  unsigned char a = 88;     // stored as unsigned binary 01011000
  unsigned char b = 55;     // stored as unsigned binary 00110111
  printf("%c\n", a);        // prints character X (ASCII value 88)
  printf("%d\n", a);        // prints number 88

  printf("%c\n", b);        // prints character 7 (ASCII value 55)
  printf("%d\n", b);        // prints number 55

  unsigned char c = ~a;     // bitwise not   01011000 (a)
  printf("%d\n",c);         // prints  167   10100111 (~a)

  unsigned char d = a & b; // bitwise and   01011000 (a)
  printf("%d\n",d);         // prints 16     00110111 (b)
                            //               00010000 (a&b)

  unsigned char e = a | b; // bitwise or    01011000 (a)
  printf("%d\n",e);         // prints 127    00110111 (b)
                            //               01111111 (a | b)

  unsigned char f = a ^ b; // bitwise xor   01011000 (a)
  printf("%d\n",f);         // prints 111    00110111 (b)
                            //               01101111 (a ^ b)

  unsigned char g = a >> 3;// right bitshift 01011000 (a)
  printf("%d\n",g);         // prints 11      00001011 (a >> 3)

  unsigned char h = a << 1;// left bitshift  01011000 (a)
  printf("%d\n",h);         // prints 176     10110000 (a << 1)
}
```

We have created a short video Module 1 Representations that discusses the code above and talks about interpretation of data.

Suppose you have a binary sequence that represents an array of boolean values. Each bit is either 1 ("true") or 0 ("false"). The least significant bit is index 0 of the array, the bit to the left is index 1, and so on. Write an expression using bitwise operations that sets index i of the array A to the boolean value 1 ("true").
16

Suppose you have a binary sequence that represents an array of boolean values. Each bit is either 1 ("true") or 0 ("false"). The least significant bit is index 0 of the array, the bit to the left is index 1, and so on. Write an expression using bitwise operations that sets index i of the array A to the boolean value b. (Hint: this is a harder problem than the one above. In particular, consider the case where A[i] is currently 1 but b is 0. Also, recall that the bitwise NOT operation can be used to flip the bits in a binary sequence.).
17

What is printed to the screen?

```
#include <stdio.h>
int main(void) {
  unsigned char c = 1;
  unsigned char d = 25;
  c <<= 3;
  c |= 1;
  c = c & d;
  printf("%d",c);
  return 0;
 }
```

18

[16] `A = A | (1 << i)`

[17] `A = (A & ~(1 << i)) | (b << i)`

We first create a binary string with a 1 in the i^{th} position. Then we use bitwise NOT to flip all the bits, i.e., we now have a string with all 1's except there is a 0 at the i^{th} position. We bitwise AND this string with A which results in A[i] to be set to 0 (all other bits are unchanged). Finally, we use bitwise OR similar to the previous example.

[18] Answer is 9. Line 5 updates the value for `c` to be the bit string 1000. After line 6, `c` has the value 1001. Since `d` is 11001, after line 7, `c` is 1001. Line 8 prints this value.