Bitwise Operations

CHAPTER 15

Objectives

- Integer Representations
  - Binary Integers
  - Integer Types
- Bit Operations
- Applying Bit Operations
  - Count the Bits
  - Reverse the Bit Order
  - Calculate the Parity of the Bits

Binary Integers
Input Conversion
Signed and Unsigned Integers
Integer Notation
Writing Numbers in Hex
Formats for Unsigned Numbers

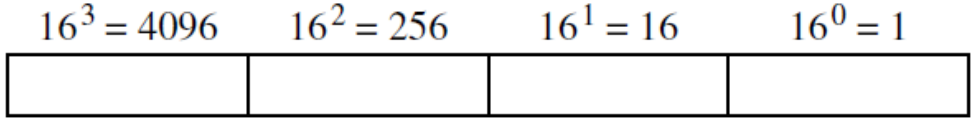
INTEGER REPRESENTATIONS

Representation - a Review

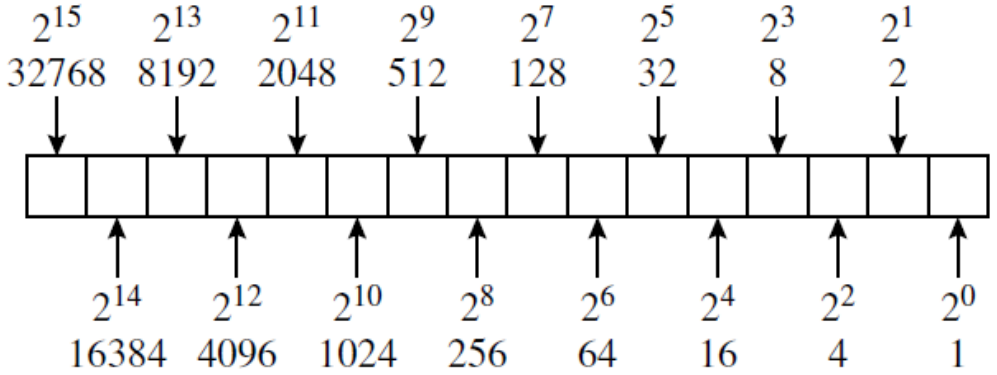
Everything in a computer is represented by bits. Bits stands for **Binary Digits.** A bit can store values of **zero or one**.

All integers are stored in binary, even when we think of them or print them to the console as base 8 or 10 or 16.

One hexadecimal digit occupies the same memory space as **four binary bits** because 24 = 16.



Place value for **hexadecimal**

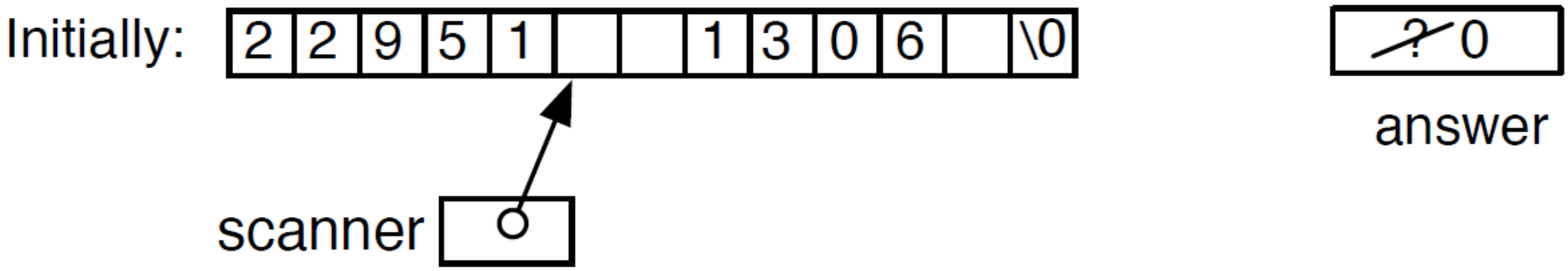


Place value for **binary**

Input Conversion in C: 1

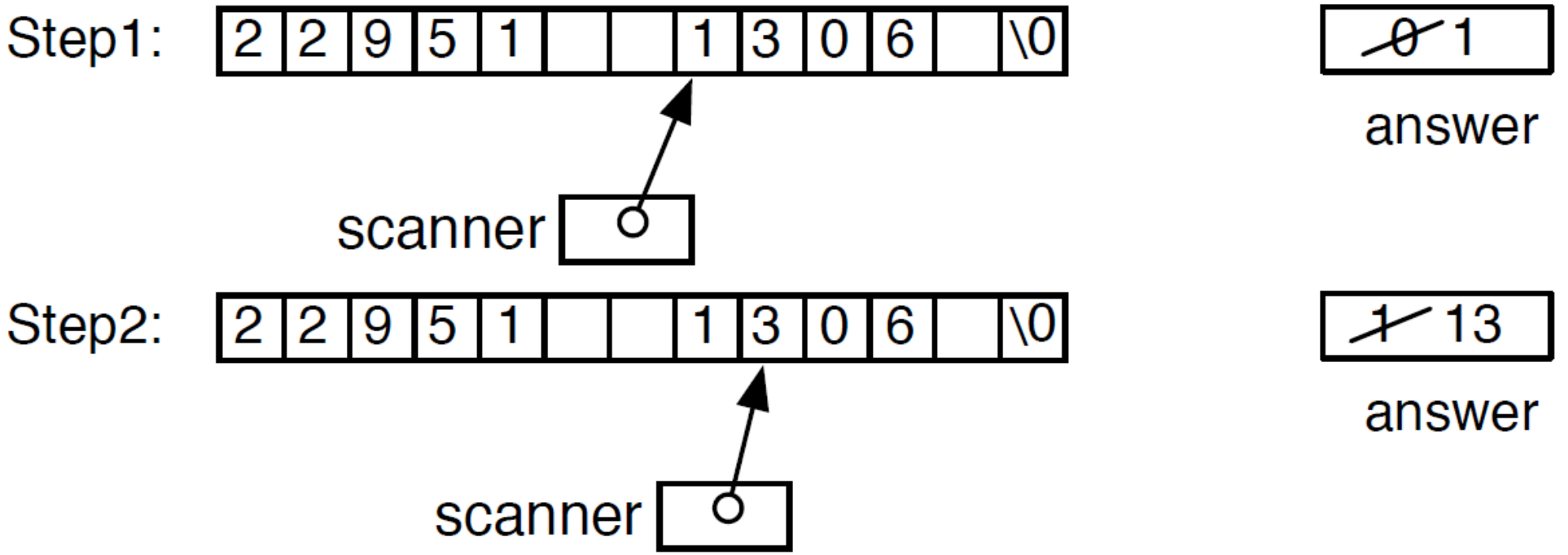
When scanf() reads a number, the ASCII representation of that number must be converted to binary for internal storage. Suppose we have a file buffer that contains two numbers. The first one has been read already and we are ready to read in the second number with %i format. Here is what happens:

1. Skip whitespace, initialize answer to 0.
2. Loop until scanner is NOT pointing at a digit.
   - 2.1 Multiply the previous answer by 10
   - 2.2 Calculate the integer corresponding to the ascii character by subtracting value of the character ‘0’.
   - 2.3 Add the value of the digit to the answer.
   - 2.4 Increment the scanner.



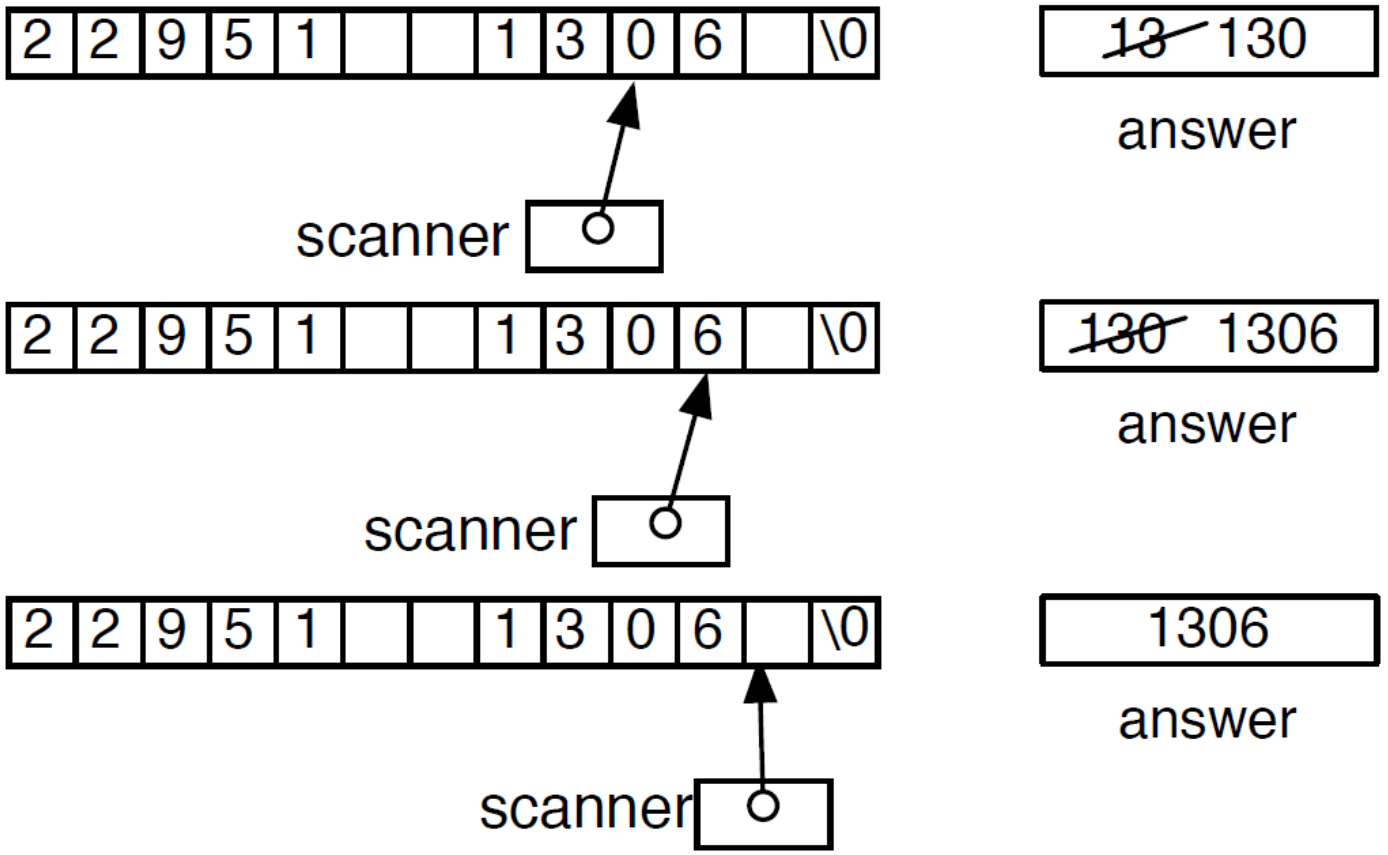
Input Conversion: 2

```
while (isdigit (*scanner)) {
answer *= 10; //Shifts values over one place when there is another
answer += (*scanner++ - '0');  //ascii value of character – ascii value of zero
}
```



Input Conversion: 3

```
while (isdigit (*scanner)) {
answer *= 10; //Shifts values over one place when there is another
answer += (*scanner++ - '0');  //ascii value of character – ascii value of zero
}
```



Input Conversion: 4

```
while (isdigit (*scanner)) {
answer *= 10; //Shifts values over one place when there is another
answer += (*scanner++ - '0');  //ascii value of character – ascii value of zero
}
```

Long, Short, and Int

- The c99 standard (no need for updates) defines **integers** in **four lengths**.
- Int can be the same as **short or long** (usually long today).
- Int and char can be either **signed or unsigned**.

Common Name	Full Name	Normal Length
`char`	`signed char` `unsigned char`	1 byte
`short` `unsigned short`	`signed short int` `unsigned short int`	2 bytes
`int` `unsigned`	`signed int` `unsigned int`	2 or 4 bytes
`long` `unsigned long`	`signed long int` `unsigned long int`	4 bytes
`long long` `unsigned short`	`signed long long int` `unsigned long long int`	4 or 8 bytes

Signed and Unsigned Integers

In a signed integer, **the leftmost bit is the sign bit**. A 1 in this position has a large negative value. The other positions have positive values, as shown.

The binary representations of several signed and unsigned integers follow. Several of these values turn up frequently during debugging, so it is useful to be able to recognize them.

$2^{15} = 32768$	$2^{14} = 16384$	$2^{13} = 8192$	$2^{12} = 4096$	$2^{11} = 2048$	$2^{10} = 1024$	$2^{9} = 512$	$2^{8} = 256$	$2^{7} = 128$	$2^{6} = 64$	$2^{5} = 32$	$2^{4} = 16$	$2^{3} = 8$	$2^{2} = 4$	$2^{1} = 2$	$2^{0} = 1$	Interpreted as a signed `short int` high-order bit = −32768	Interpreted as an unsigned `short int` high-order bit = 32768
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	32767	32767
0	0	1	0	0	1	1	1	0	0	0	1	0	0	0	0	10000	10000
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	−32768 + 32767 = −1	+32768 + 32767 = 65535
1	1	0	1	1	0	0	0	1	1	1	1	0	0	0	0	−32768 + 22768 = −10000	+32768 + 22768 = 55536
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	−32768 + 0 = −32768	+32768 + 0 = 32768
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	−32768 + 1 = −32767	+32768 + 1 = 32769

Using Unsigned Types

Sometimes a program must deal directly with hardware

components or the actual pattern of bits stored in the memory.

Applications include

- Cryptography, where you work with bits, not numbers.
- Random number generation
- Programs that control hardware switches
- Sometimes the data is naturally unsigned, as in digital photography.

Integer Notation

C provides more than one way to write an integer number because:

- Base-10 notation is **very inconvenient** for many applications that rely on unsigned numbers.
- **Base 10 does not correspond** in an easy way to the internal representation of numbers or to their length in bytes.
- It **takes calculation** starting with a base-10 number, to arrive at its binary representation.

Integer Literals

There are four ways to write a number in your program (a literal):

- **Decimal integer**: Use digits 0 - 9 but **DO NOT start with 0**.
- **Octal integer**: Use digits 0 - 7 and **always start with 0**. *Very useful for Electrical and computer Engineering. Think of trios of switches.*
- **Hexadecimal integer**: Has 16 unique values, use digits 0 - 9 and A - F (or a – f) and **always start with 0x**.
- **Hexadecimal character constant**: Start with 'x, followed by one or two digits 0 - 9 and/or A - F (or a - f) and end with ‘. E.g. ‘x1F’

Remember: there are two hex digits per byte in an integer value.

Writing Numbers in Hex

- These numbers are signed short ints.
- Leading zeros may be either written or omitted.

Decimal	Hex	Decimal	Hex	Decimal	Hex
0	0x0	17	0x11	65	0x0041
7	0x7	18	0x12	91	0x005b
8	0x8	30	0x1E	127	0x007f
10	0xA	32	0x20	128	0x0080
11	0xB	33	0x21	255	0x00FF
15	0xF	63	0x3F	256	0x0100
16	0x10	64	0x40	32767	0x7FFF
-1	0xFFFF	-2	0xFFFE	-32768	0x8000

Conversion Specifiers for Unsigned Values

	int	long	short	char
Character I/O				%c
Decimal I/O	%u	%lu	%hu	
Hex input	%i or %x	%li or %lx	%hi or %hx	
Hex output	%04x or %08x	%08lx	%04hx	%x

- If the **input stream contains** numbers in the form of **hexadecimal literals: 0x35**, they can be read as hex values using **%i**.
- **If the '0x' is not part of the input**, then **%x must be used** to read a number **as hexadecimal**.

I. Opposite Operations

Opposites – three types

Arithmetic negation, logical not, and bitwise complement are three ways to get the “opposite" of a number. If the binary value of x = 0000 0000

- **Bitwise compliment: ~x** means take the compliment of x.
  - Complement inverts all the bits in the number. i.e. flip the switches.
  - ~x = 1111 1111
- **Arithmetic negation: -x** means negative x or negate x.
  - To negate a number, take its complement (~x) and add 1. i.e. Flip the switches and add 1 to the result.
  - -x = 1111 1111 + 0000 0001 = 0000 0000 (high 1 carries off the values that can be represented, so you flip back to zero).
- **Logical not: !x** means if x is true, !x is false and if x is false, !x is true.
  - 0000 0000 means false, anything else is true.
  - !x = 0000 0001 which is true

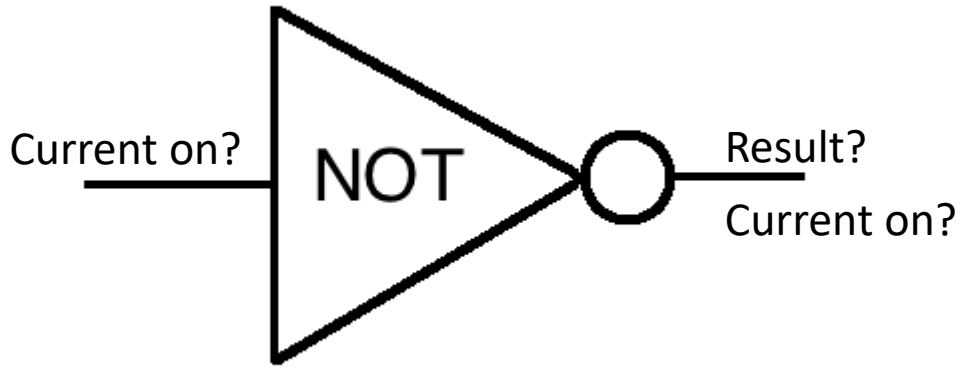
Logical NOT (it’s not bitwise)

To avoid confusion, remember, logical NOT acts on one value (the whole number, not bitwise) and returns the opposite. This isn’t a bitwise operator.

Logical NOT **is true** (one) if the input is **false** (zero). Any **input value that is not a zero** is interpreted as **true**.

Truth Table for Logical NOT

A	!A
True (not 0)	False (0)
False (0)	True (1)



NOT gate or inverter for circuit design

In Class Exercise: Think-Pair-Share

1. With your partner, calculate the answer for your assigned value and operation.
2. Come up and **share your understanding with the class**.
   a. What operation are you doing? Bitwise compliment ~x, arithmetic negation –x, or logical not !x
   b. What are the steps to do this operation?
   c. Show x and all the work on the paper and your answer in binary.
   d. Show the hex and decimal values if you can.

	Decimal	Hex	Binary		Decimal	Hex	Binary
x	2	0x02	0000 0010	x	1	0x01	0000 0001
~x				~x			
-x				-x			
!x				!x			
x	15	0x0F	0000 1111	x	−10	0xF6	1111 0110
~x				~x			
-x				-x			
!x				!x			

II. Bit Operations

Arity	Symbol	Meaning	Precedence
Unary	~	Bitwise complement	15
Binary	<<	Left shift	11
	>>	Right shift	11
	&	Bitwise AND	8
	^	Bitwise exclusive OR (XOR)	7
	|	Bitwise OR	6

Bitwise Operators

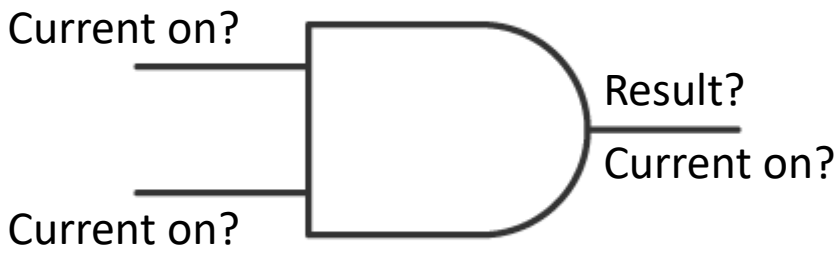
Bitwise AND, &

Binary AND acts on two values and applies the AND truth table at the bit level between the two bits. Work rightmost bit to leftmost bit from each number.

Binary AND **is only true** (one) if **both bits are true** (one).

Truth Table for binary &

A	B	A & B
0	0	0
0	1	0
1	0	0
1	1	1



AND gate for circuit design

Bitwise AND

Use AND with a mask to **isolate selected bits (where the ones are in the mask)** from a number. This mask checks the values of bits in the 1s, 4s and 32s places, all others **will always be zero** because the **mask is zero at those places**.

Syntax: `result = a&mask;`

Variable	Binary	Number in Hex
a =	0110 1110	0x6E
mask =	& 0010 0101	0x25
a & mask =	0010 0100	0x24

You will need a mask in the steganography program, to isolate specific of bits from each pixel of the modified image.

In Class Exercise: Think, Pair Share with Bitwise AND

1. With your partner, calculate the answer for your assigned problem.
2. Come up and **share your understanding with the class**.

   a. What are the steps to do bitwise AND with a mask?

   b. Show all variables and all the work on the paper and your answer in binary.

   c. Show the hexadecimal values if you can. Be ready to walk us though your solution and both of you must speak.

	Binary	Hex		Binary	Hex
x	1111 1111	0xFF	y	0111 1010	0x7A
mask	0010 0000	0x20	mask	1111 0000	0xF0
x & mask			y & mask		
z	0010 1010	0x7A	a	0111 1010	0x7A
mask	1100 0000	0xC0	mask	0000 0011	0x03
z & mask			a & mask		

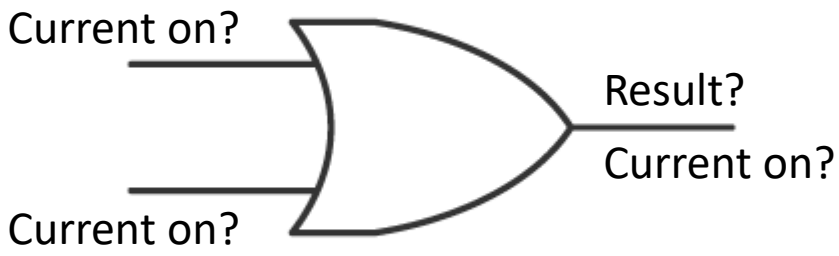
Bitwise OR, |

Binary OR acts on two values and applies the OR truth table at the bit level between the two integers one pair at a time.

Binary OR **is true** (one) if **either one or both bits are true** (one).

Truth Table for binary |

A	B	A | B
0	0	0
0	1	1
1	0	1
1	1	1



OR gate for circuit design

Bitwise OR

Use OR with a mask to Use or to **turn on bits (where the ones are in the mask)** or to **combine parts into a whole**. This mask would **ensure that the 1s, 4s and 32s place bits are on (one),** all others **will remain the same** because the **mask is zero at those places**.

Syntax: `result = a|mask;`

Variable	Binary	Number in Hex
a =	0110 1110	0x6E
mask =	| 0010 0101	0x25
a | mask =	0110 1111	0x6F

For example we use this in the encryption program to combine bits that were moved around back into one variable.

In Class Exercise: Think, Pair Share with Bitwise OR

1. With your partner, calculate the answer for your assigned problem.
2. Come up and **share your understanding with the class**.

   a. What are the steps to do bitwise OR with a mask?

   b. Show all variables and all the work on the paper and your answer in binary.

   c. Show the hexadecimal values if you can. Be ready to walk us though your solution and both of you must speak.

	Binary	Hex		Binary	Hex
x	0000 0000	0x00	y	0000 1100	0x0C
mask	0010 0000	0x20	mask	1111 0000	0xF0
x | mask			y | mask		
z	0011 1110	0x3E	a	1001 1100	0x9C
mask	1010 0111	0xA7	bits	0000 0001	0x01
z | mask			a | bits		

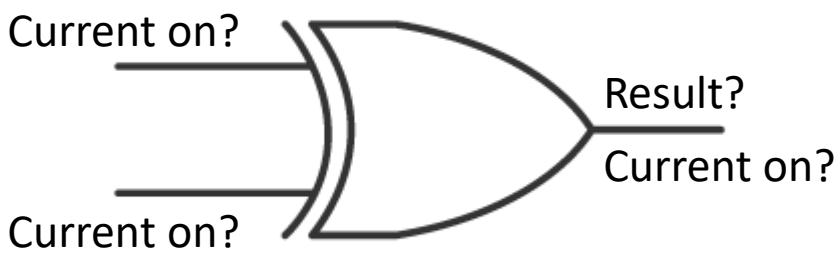
Bitwise “Exclusive OR” aka XOR, ^

Binary exclusive OR acts on two values and applies the XOR truth table at the bit level between the two bits.

Binary exclusive OR (XOR) **is true** (one) if **one or the other bit is one** (one) but not when neither one is true or both are true. This is useful to **toggle** a bit aka flip the value of a bit to its opposite.

Truth Table for binary ^

A	B	A ^ B
0	0	0
0	1	1
1	0	1
1	**1**	**0**



XOR gate for circuit design

Bitwise XOR

Use XOR with a mask **to toggle some of the bits (where the ones are in the mask)** from a number or to test for inequality. This mask toggles (turns on zeros and turns off ones) the bits in the 1s, 4s and 32s places. All others **will remain the same** because the **mask is zero at those places**.

Syntax: `result = a^mask;`

Variable	Binary	Number in Hex
a =	0110 1110	0x6E
mask =	& 0010 0101	0x25
a ^ mask =	0100 1011	0x8B

In Class Exercise: Think, Pair Share with Bitwise XOR

1. With your partner, calculate the answer for your assigned problem.
2. Come up and **share your understanding with the class**.

   a. What are the steps to do bitwise XOR with a mask?

   b. Show all variables and all the work on the paper and your answer in binary.

   c. Show the hexadecimal values if you can. Be ready to walk us though your solution and both of you must speak.

	Binary	Hex		Binary	Hex
x	1111 1111	0xFF	y	1100 1100	0xCC
mask	0011 0000	0x30	mask	1111 0000	0xF0
x ^ mask			y ^ mask		
z	0111 1010	0x7A	a	0111 1010	0x7A
bits	1010 0011	0xA3	bits	0111 1010	0x7A
z ^ bits			a ^ bits		

Shifts

SHIFT LEFT AND RIGHT

Left Shift

- Left shift **moves the bits of the number to the lef**t; bits that **move off the left end are forgotten.**
- **0 bits are pulled in to fill the right end**.
- This is a very fast operation for hardware. Just one CPU operation per shift.
- **Shifting left by one position** has the same effect on a number as **multiplying by 2** but it is **a lot faster** for the machine.

The table on the next slide shows the results of several shift operations on these variables:

```
signed char s; // A one-byte signed integer.
unsigned char u; // A one-byte unsigned integer.
```

Note that **the result of n << 2 is four times the value of n, as long as no significant bits fall off the left end**.

Signed	Decimal	Hex	Binary
s	15	0x0F	0000 1111
s << 2	60	0x3C	0011 1100
s	−10	0xF6	1111 0110
s << 2	−40	0xD8	1101 1000
Unsigned	**Decimal**	**Hex**	**Binary**
u	10	0x0A	0000 1010
u << 2	40	0x28	0010 1000
u	255	0xFF	1111 1111
u << 2	252	0xFC	1111 1100

Left Shifts with signed and unsigned integers

Right Shift

The table on the next slide shows the results of several right shift operations.

- Right shift moves the bits of the number to the right; **bits that move off the right end are forgotten**.
- In a **signed right shift**, **copies of the sign bit are pulled in to fill the left end.**
- For **an unsigned right shift, 0 bits are used to fill the left end.**
- **Shifting right by one position** has the same effect on a number as **dividing by 2**. To enable this, **a signed right shift keeps the sign bit unchanged.**
- Syntax: `result = n>>2;` or `result = n>>4;`
- Note that **the result of n >> 2 is one fourth of the value of n.** $2^2 = 4$

Signed	Decimal	Hex	Binary
s	15	0x0F	0000 1111
s >> 2	3	0x03	0000 0011
s	−10	0xF6	1111 0110
s >> 2	−3	0xFD	1111 1101
Unsigned	**Decimal**	**Hex**	**Binary**
u	10	0x0A	0000 1010
u >> 2	2	0x02	0000 0010
u	255	0xFF	1111 1111
u >> 2	63	0x3F	0011 1111

Right Shifts with signed and unsigned integers

In Class Exercise: Think, Pair Share with Shift Left and Right

1. With your partner, calculate the answer for your assigned problem.
2. Come up and **share your understanding with the class**.
   a. What are the steps to do the assigned shift? Pay attention to the type of integer – are the rules the same for signed and unsigned?
   b. Show the original variable and all the work on the paper and your answer in binary.
   c. Show the hexadecimal values if you can. Be ready to walk us though your solution and both of you must speak.

	Binary	Hex		Binary	Hex
x unsigned x >>3	1111 1111	0xFF	y signed y>>3	1100 1100	0xCC
z unsigned z << 1	0111 1010	0x7A	a signed a << 3	0111 1010	0x7A

II. Applying Bit Operations:

1. Count the Bits
2. Reverse the Bit Order
3. Calculate the Parity of the Bits

Count the Bits

Count Bits - Algorithm

To count the bits we need to isolate one bit at a time and add one to the count if the bit is one.

What are the steps to do this?

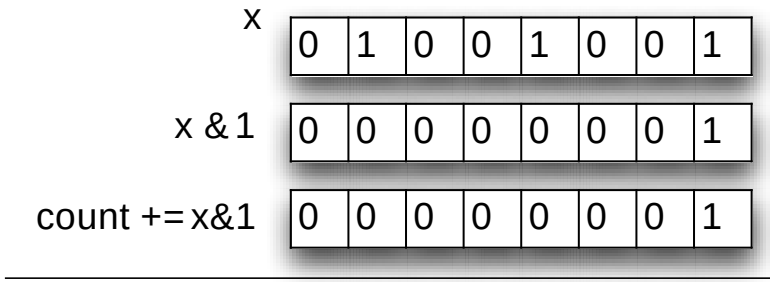
- What operation do we use to select bit(s)?
- How do we specify which bit(s) to choose?
- Then what?

Count the Bits – Steps 1 and 2

These drawings go with the `popcount3()` function in the `bitops` program. The goal is to count the number of 1 bits in the byte.

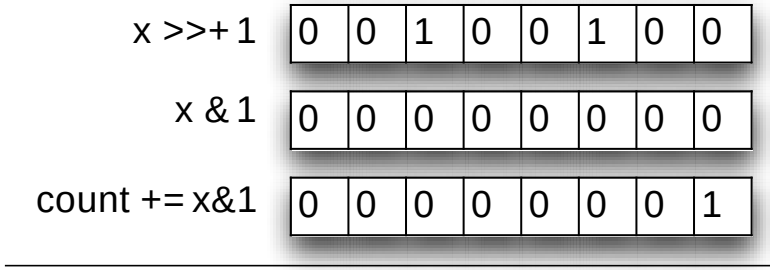
We execute the function popcount3(0x49), which is the bitstring 0100 1001. Initially, count=0.

1st time



The result of an AND is 1 if both bits are 1. Otherwise it is 0.

2nd time



Using 1 as a mask keeps the rightmost bit and zeros out the other seven.

Count the Bits – Steps 3 through 5

3rd time

x >>+ 1	0	0	0	1	0	0	1	0	If x ends in a 0 bit, the count is not changed. If it ends in a 1 bit, we add 1 to the bit count.
x & 1	0	0	0	0	0	0	0	0	
count += x&1	0	0	0	0	0	0	0	1	

4th time

x	0	0	0	0	1	0	0	1	Now the second 1 bit has been shifted to the end of the word, so the count grows.
x & 1	0	0	0	0	0	0	0	1	
count += x&1	0	0	0	0	0	0	0	2	

5th time

x >>+ 1	0	0	0	0	0	1	0	0	Using 1 as a mask keeps the rightmost bit and zeros out the other seven.
x & 1	0	0	0	0	0	0	0	0	
count += x&1	0	0	0	0	0	0	0	2	

Count the Bits – Steps 6 through finish

6th time

x >>+ 1	0	0	0	0	0	0	1	0	If x ends in a 0 bit, the count is not changed. If it ends in a 1 bit, we add 1 to the bit count.
x & 1	0	0	0	0	0	0	0	0	
count += x&1	0	0	0	0	0	0	0	2	

7th time

x	0	0	0	0	0	0	0	1	Now the second 1 bit has been shifted to the end of the word, so the count grows.
x & 1	0	0	0	0	0	0	0	1	
count += x&1	0	0	0	0	0	0	0	3	

End of loop

x >>+ 1	0	0	0	0	0	0	0	0	X has become 0 so we leave the loop and return the count.
count	0	0	0	0	0	0	0	3	

Reverse the Order of the Bits

Reverse the Order - Algorithm

To reverse the order of the bits we need to isolate one bit at a time and move it to the reversed number.

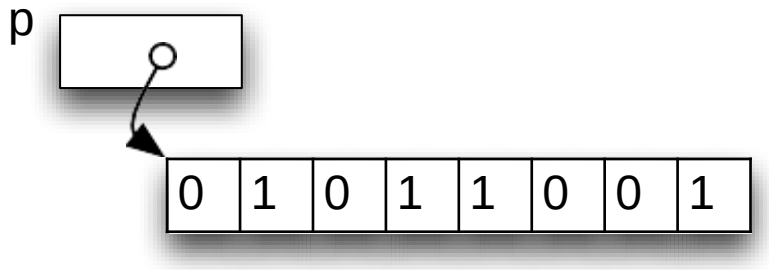
What are the steps to do this?

- What operation do we use to select one bit?
- How do we specify which bit(s) to choose?
- Then what?

Reverse the Bit Order

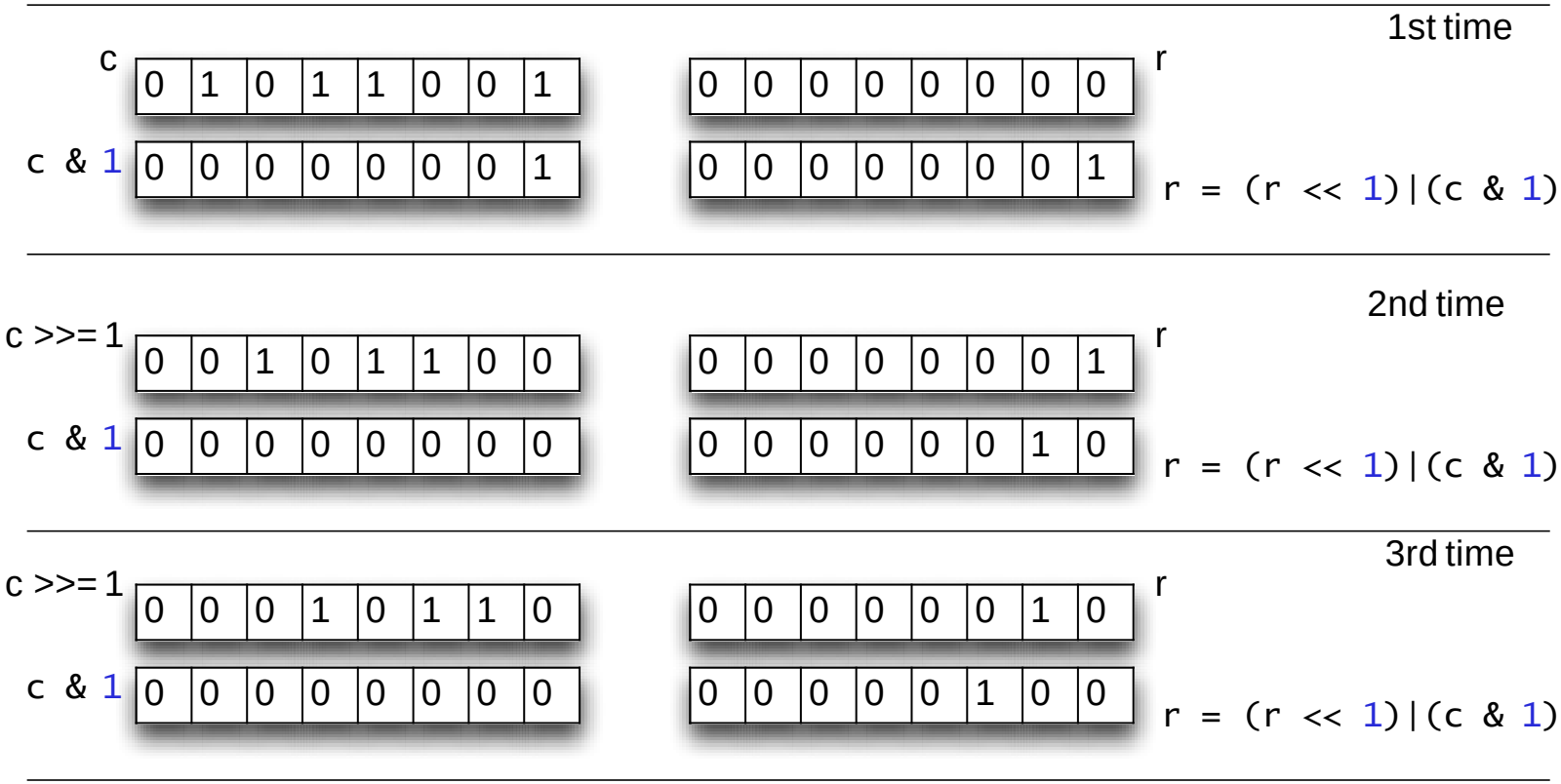
These drawings go with the Reverse program. The goal is to take a byte and reverse the order of its bits so that the last bit becomes the first bit, etc.

The code was written to reverse the bits in all the bytes in a pixel image. The variable p points at the current pixel.

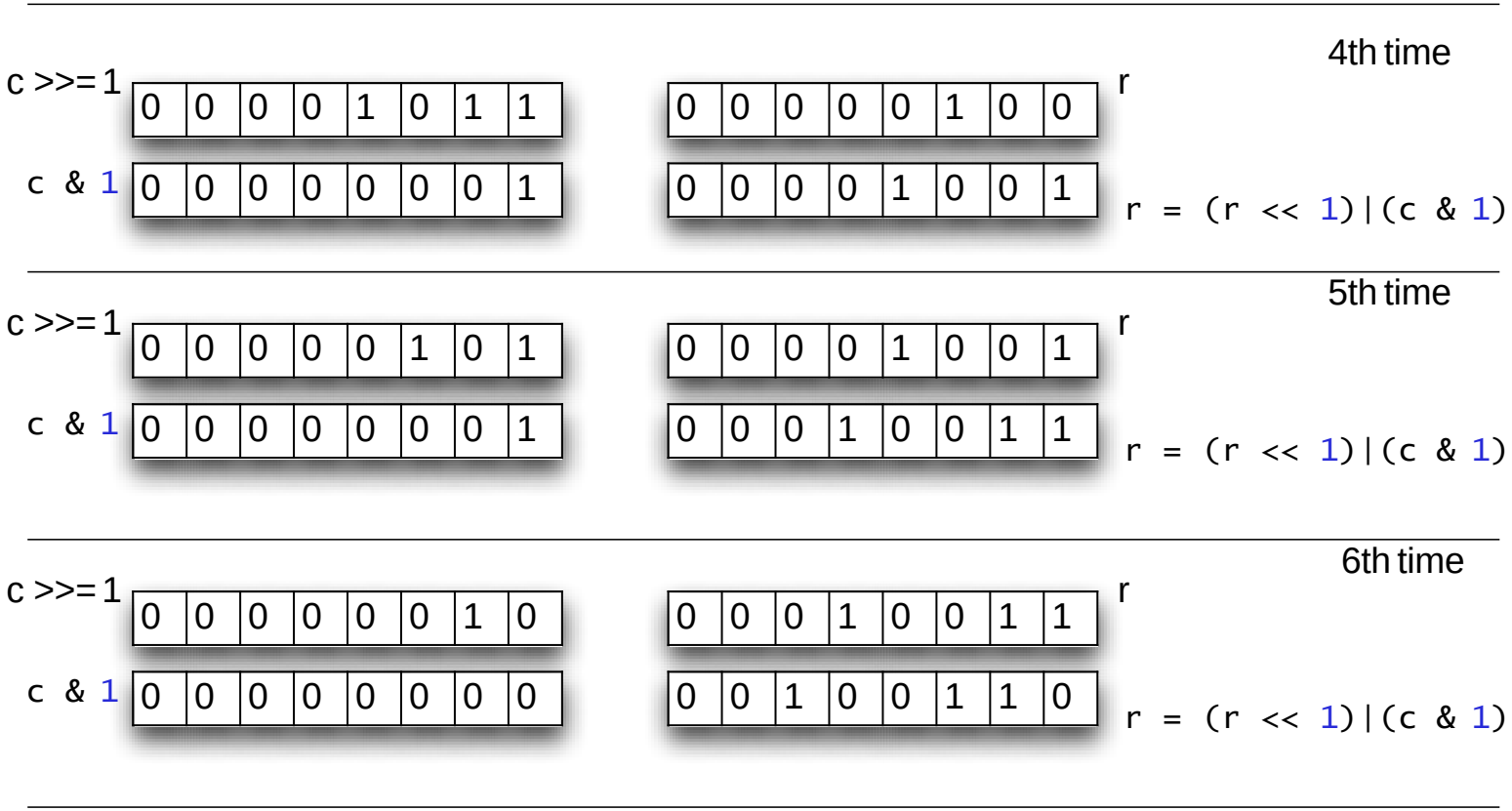


We will trace the reversal process on this pixel, 0x59.

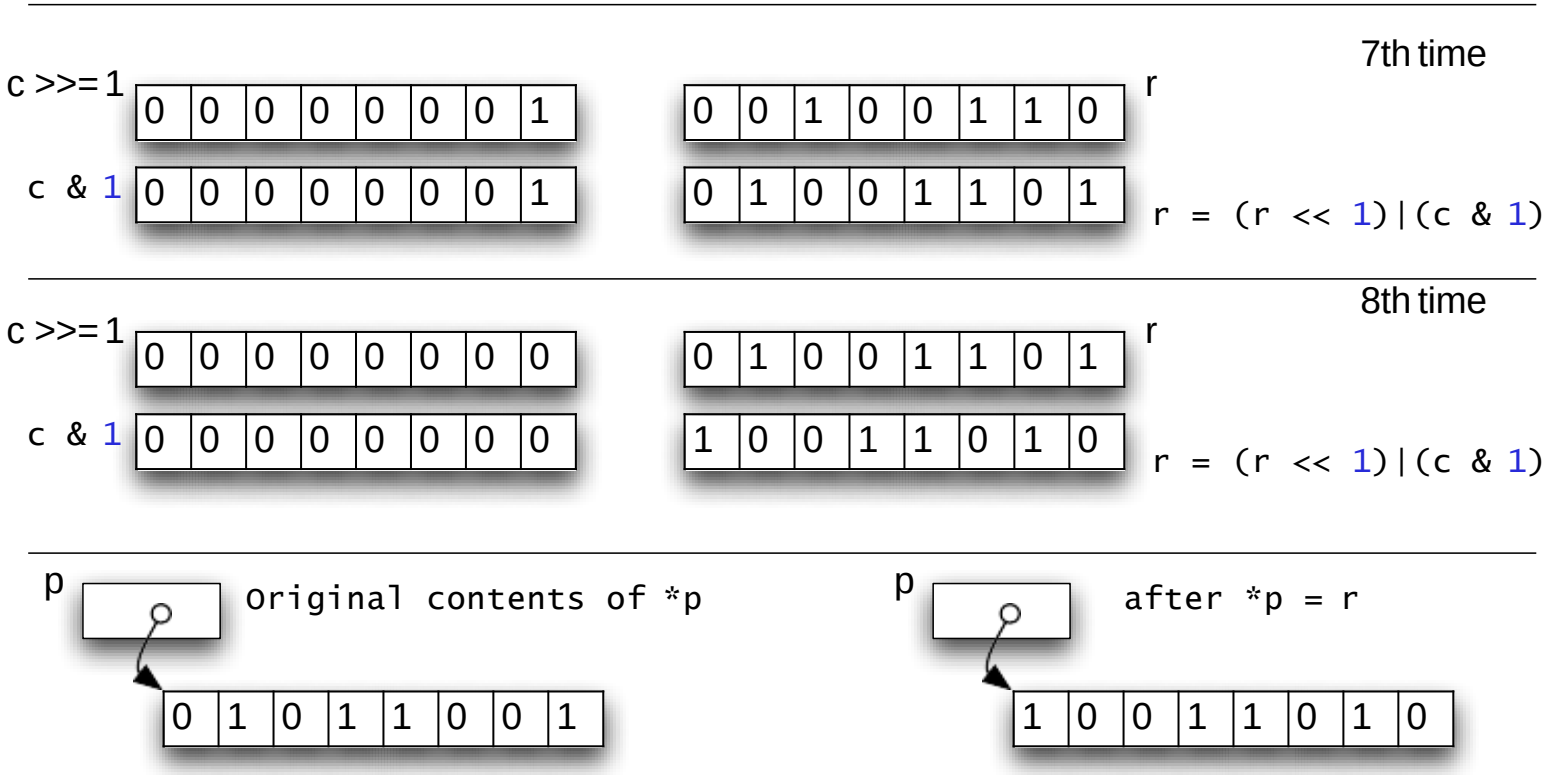
First Three Steps.



Steps Four through Six.



Ending the Reversal Loop.



Find Parity of a Number

Parity – A few ideas

Parity check how many one bits are in a set of bits. This can be the entire number of a subset of the bits.

If there are an **even number of 1 bits**, the value has **even parity**.

If there are an **odd number of 1 bits**, the value has **odd parity**.

If you count the bits, all you must **check is the rightmost bit of the count** to see the parity – a one means its an odd number. Why?

You can also count the bits and then **use % (modulus, integer remainder)** to find out if it is an odd number. How would you use this?

Parity with XOR

You can also use XOR.

What does XOR do? What is its TRUTH TABLE?

This algorithm basically pairs off the ones. Every other one cancels the one before it. If you end up with one in the parity bit at the end, the number has odd parity. Otherwise it is even.

THE ONLY IMPORTANT BIT in this algorithm is the rightmost.

Is this more efficient than counting the bits? By how much?

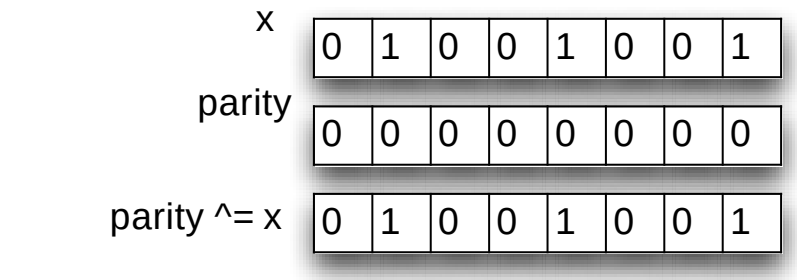
Is this easy to understand? Would the potential maintainer get it?

Parity – Steps 1 and 2

These drawings go with the `parity3()` function in the `bitops` program. The goal is to calculate whether a byte has odd or even bit parity.

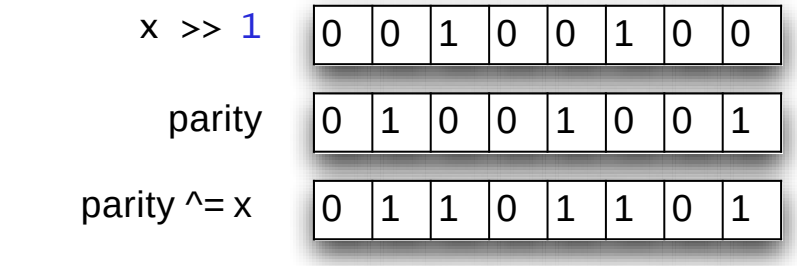
We execute the function parity3(0x49), which is the bitstring 0100 1001.

1st time



The result of an XOR is 0 if the bits are the same, 1 if they are different.

2nd time



The rightmost bit of parity will change every time x is shifted and the last bit of x is a 1.

Parity – Steps 3 through 5

3rd time

x >> 1

0	0	0	1	0	0	1	0

parity

0	1	1	0	1	1	0	1

parity ^= x

0	1	1	1	1	1	1	1

If there are an odd number of 1 bits, we call it "odd parity".

4th time

x >> 1

0	0	0	0	1	0	0	1

parity

0	1	1	1	1	1	1	1

parity ^= x

0	1	1	1	0	1	1	0

If there are an even number of 1 bits, we call it "even parity".

5th time

x >> 1

0	0	0	0	0	1	0	0

parity

0	1	1	1	0	1	1	0

parity ^= x

0	1	1	1	0	0	1	0

If the rightmost bit of x is a 0, parity does not change.

Parity – Steps 6 through finish

6th time

x >> 1	0	0	0	0	0	0	1	0	
parity	0	1	1	1	0	0	1	0	When we finish, only the rightmost bit will matter; the others will be masked off.
parity ^= x	0	1	1	1	0	0	0	0	

7th time

x >> 1	0	0	0	0	0	0	0	1	
parity	0	1	1	1	0	0	0	0	We are done when the last 1 bit is shifted off the end of x.
parity ^= x	0	1	1	1	0	0	0	1	

Finish

x >> 1	0	0	0	0	0	0	0	0	
parity	0	1	1	1	0	0	0	1	The value of x is now zero, so we leave the while loop. Then we mask off all the bits of parity except the final one and return the answer. In this case, we have odd parity.
parity & 1	0	0	0	0	0	0	0	1	

Summary of Objectives

- Integer Representations
  - Binary Integers
  - Integer Types
- Bit Operations
- Applying Bit Operations
  - Count the Bits
  - Reverse the Bit Order
  - Calculate the Parity of the Bits