COL106
Data Structures and Algorithms

Subodh Sharma and Rahul Garg

Representing Numbers

How do we Represent Numbers in Computers?

Binary Representations

- Every number is represented using binary digits
- Why Binary?
  - Reliability in storage and retrieval
  - Reliability in transmission over network

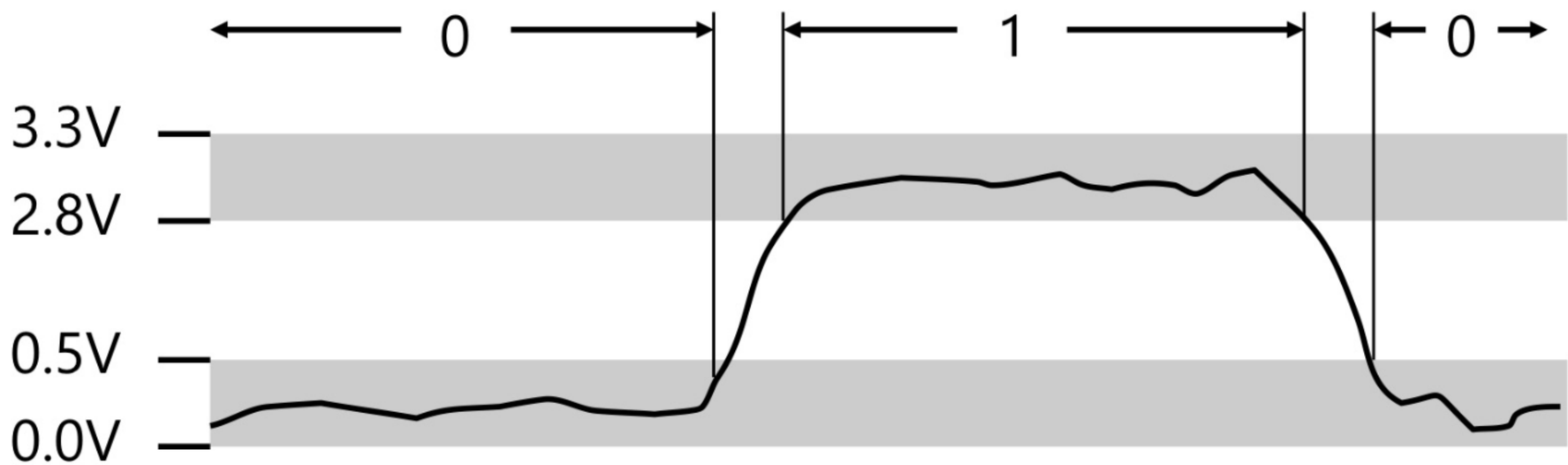
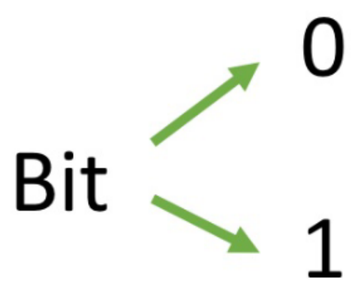


Image courtesy: Mark Oskin-University of Washington

Bits and Bytes

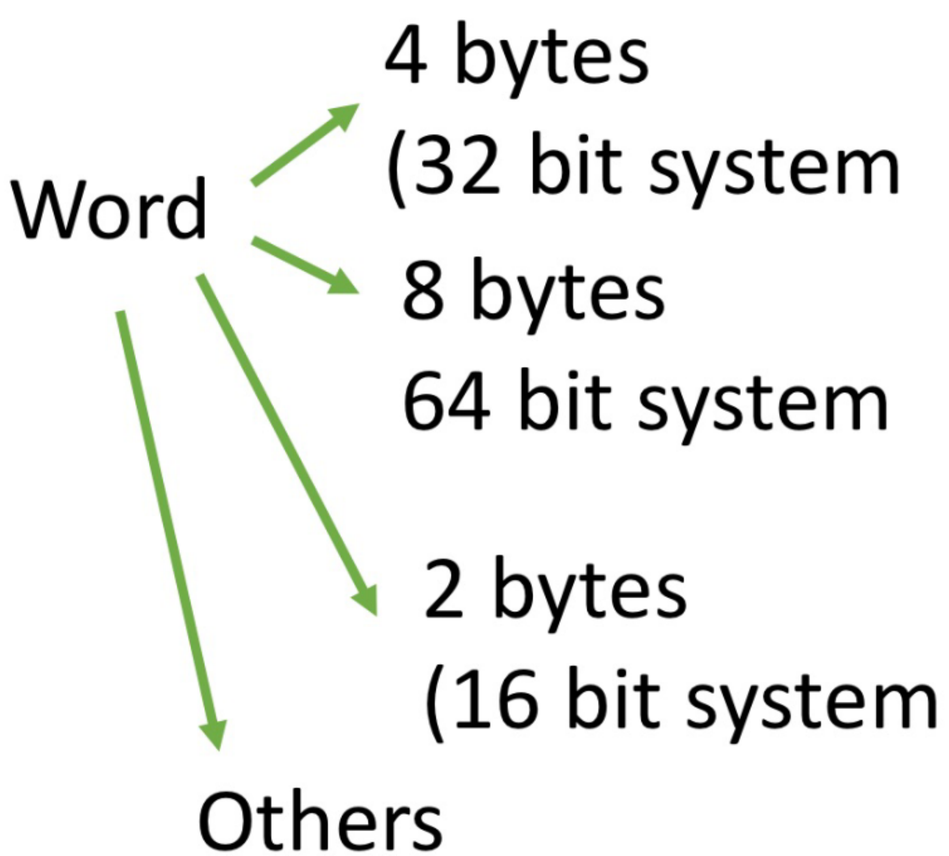
- Bit is a binary digit



- Byte is a sequence of 8 bits

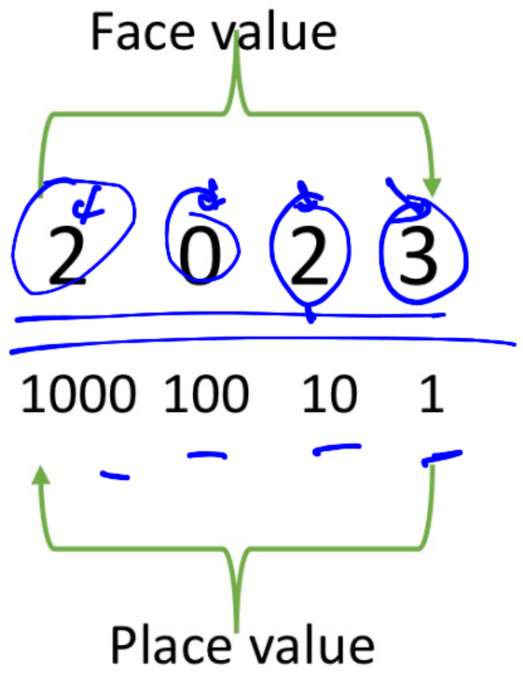
Byte 01001011

- Word is a sequence of bytes



Number Representation Using Bits and Bytes

- Basics of Decimal Notations
  - Place value
  - Face value



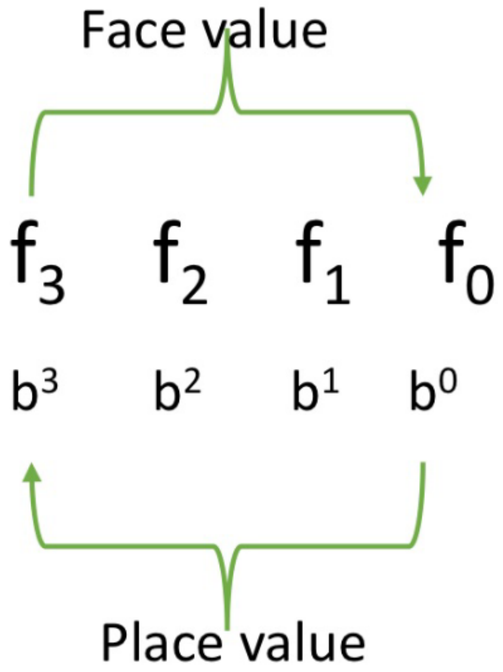
Place value

$$2023 = 2 * 10^3 + 0 * 10^2 + 2 * 10^1 + 3 * 10^0$$

Face value

Number Representation Using Bits and Bytes

- Basics of Decimal Notations
- Value = $f_3*b^3 + f_2*b^2 + f_1*b^1 + f_0*b^0$
- In general, a k-digit number in base b, consisting of digits f_i (where $0 \le f_i < b$) will have a value of:



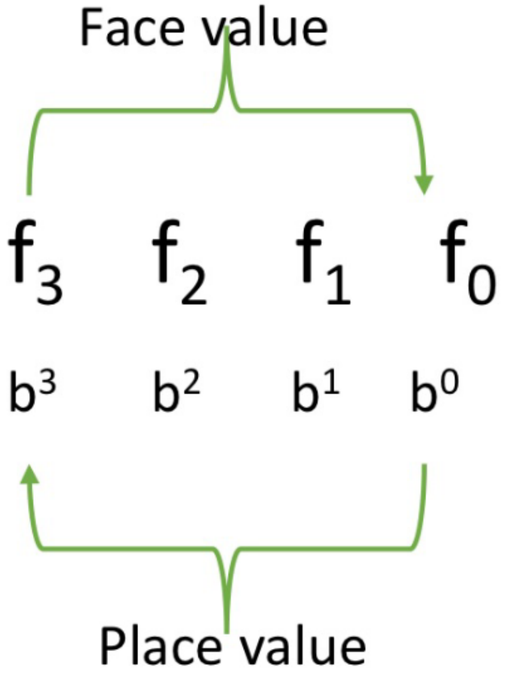
$$v = \sum_{i=0}^{k-1} f_i b^i$$

$b = 2$

f_0 f_1

Common Number Systems

- If b = 2 (binary), then f_i will be 0 or 1
- If b = 8 (octal) then, f_i will be 0, 1... 7
- If b = 10 (decimal) then, f_i will be 0, 1... 9
- If b = 16 (hex) then f_i will be 0,...9, A..F



Examples

Base 2

- 1011 0110
- 1111 1111

Base 8

- 2417
- 7070

Base 16

- FA91
- 0100

1 x 128 + 1 x 32 + 1 x 16 + 1 x 4 + 1 x 2

128 32 16 8 4 2 1

Base 10

182

1 + 2 + 4 + 8 ... 128 = 255 = $2^8 - 1$

2 x 512 + 4 x 64 + 8 x 1 + 7

1024 256 8 7

= 1295

4096 256 16 1

4096 * 15 + 256 x 10 + 16 * 9 + 1

256

Base 8 → Base 10

320

$f_3\ f_2\ f_1\ f_0$

$f_3 * b^3 + f_2 b^2 + f_1 b + f_0 = 320$ (divisible by b)

$320 \% 8 = 0$

LHS = $f_0 = 0$

$f_1 = \left[(320 - f_0)/8 \right] \% 8$ (Take mod b)

$40 \% 8 = 0$

$f_2 = 0$

$f_3 = 5$

Base 8

5 00 = 320

64 8 1

Examples

Base 2

•

•

Base 8

•

•

Base 16

•

•

Base 10

• 402

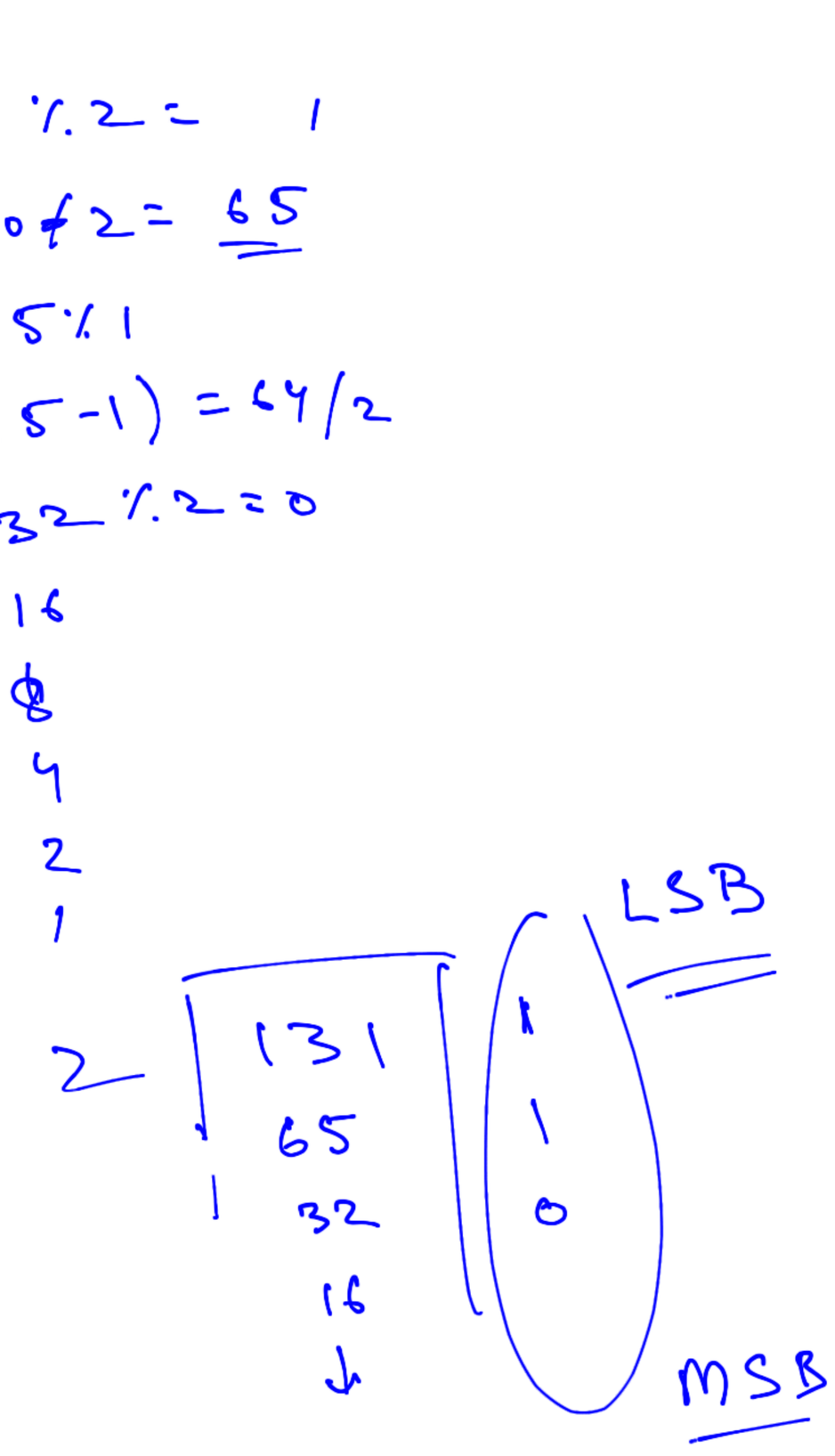
• 131

• 512

• 341

• 160

• 166



Examples

Base 2

- 1011 0110
- 1111 1111

Base 8

- 2417
- 7070

Base 16

- FA91
- 0100

Base 16

B 6

F F

Base 2

Negative Numbers

Representing Negative Numbers

- Idea 1: Add a leading sign digit representing the sign of the number

5 = 0101

4 = 0100

-5 = 1101

-4 = 1100

Sign digit

- 23

- 31

+ 32

- 42

+ 41

32

Sign digit

Number

Sign-and-Magnitude Negatives

- Use the most significant bit to represent sign
  - 0 ➔ +ve and 1 ➔ -ve
- Rest of the bits to represent the number
  - There are two representations of 0!)
  - Math is cumbersome
  - How to add?
    - +ve + -ve
    - -ve + -ve
- Need complex circuits

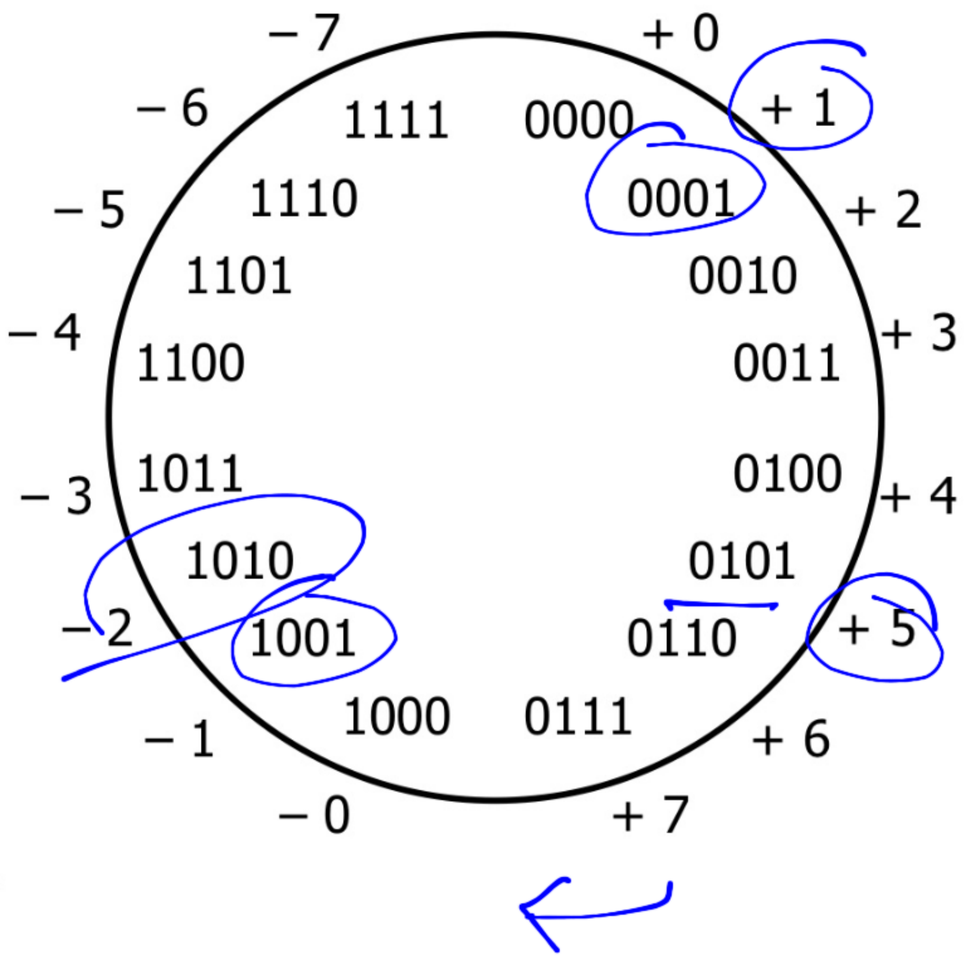
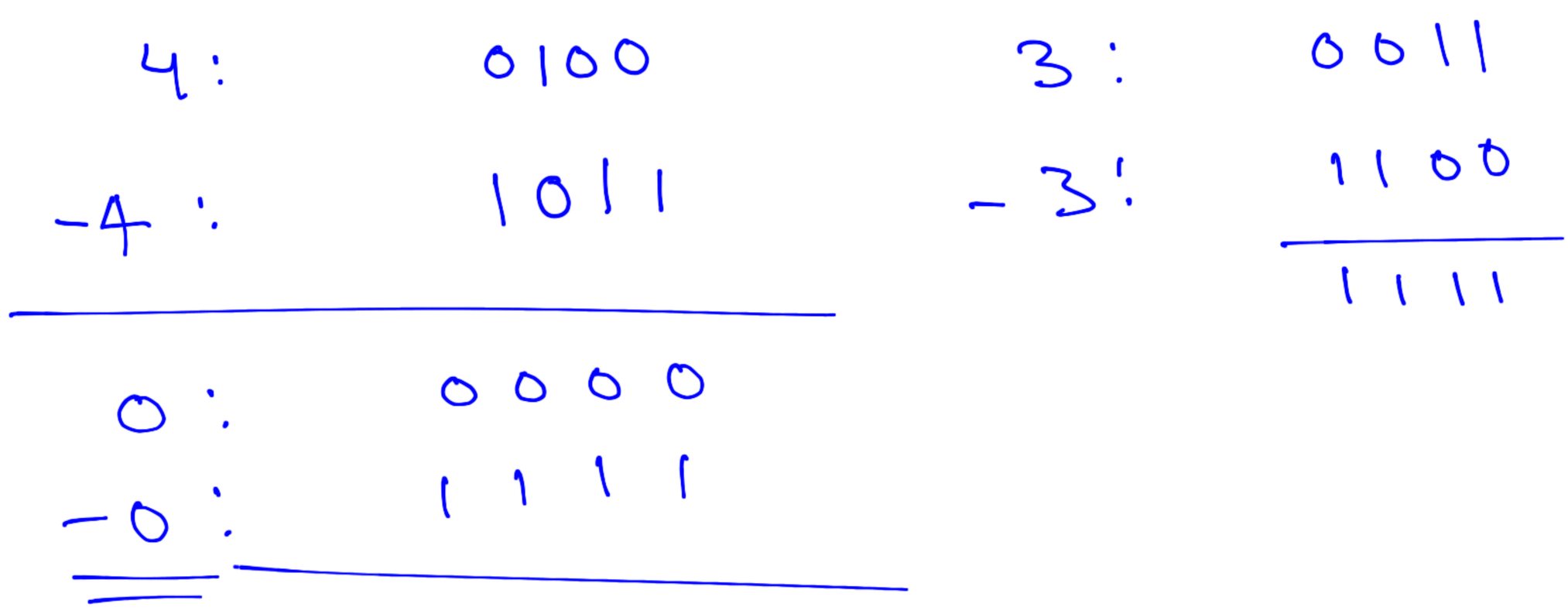


Image courtesy: Mark Oskin-University of Washington

How to Represent Negative Numbers

- Idea 2: Complement each digit of the number
- Complement means subtract from base
- Examples (base 10 and base 2)



Ones' Complement Negatives

- Again, two representations of zero
- How to add two numbers?
- +ve + +ve
- -ve + -ve
- +ve + -ve

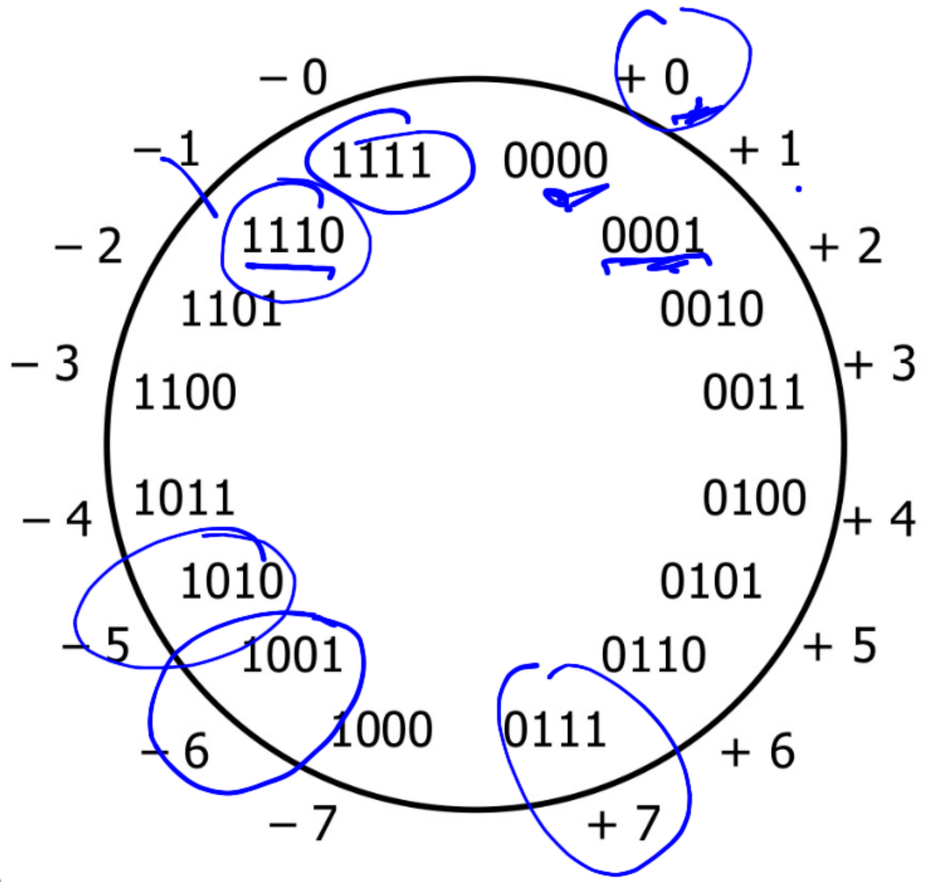


Image courtesy: Mark Oskin-University of Washington

-2
-3
-5

1101
1100
1 | 1001
1010

-3
+2
-1

1100
0010
1110

Ones' Complement Negatives

- Adding two numbers
- Case 1: +ve + +ve
- Case 2: -ve + -ve
- Case 3: +ve + -ve

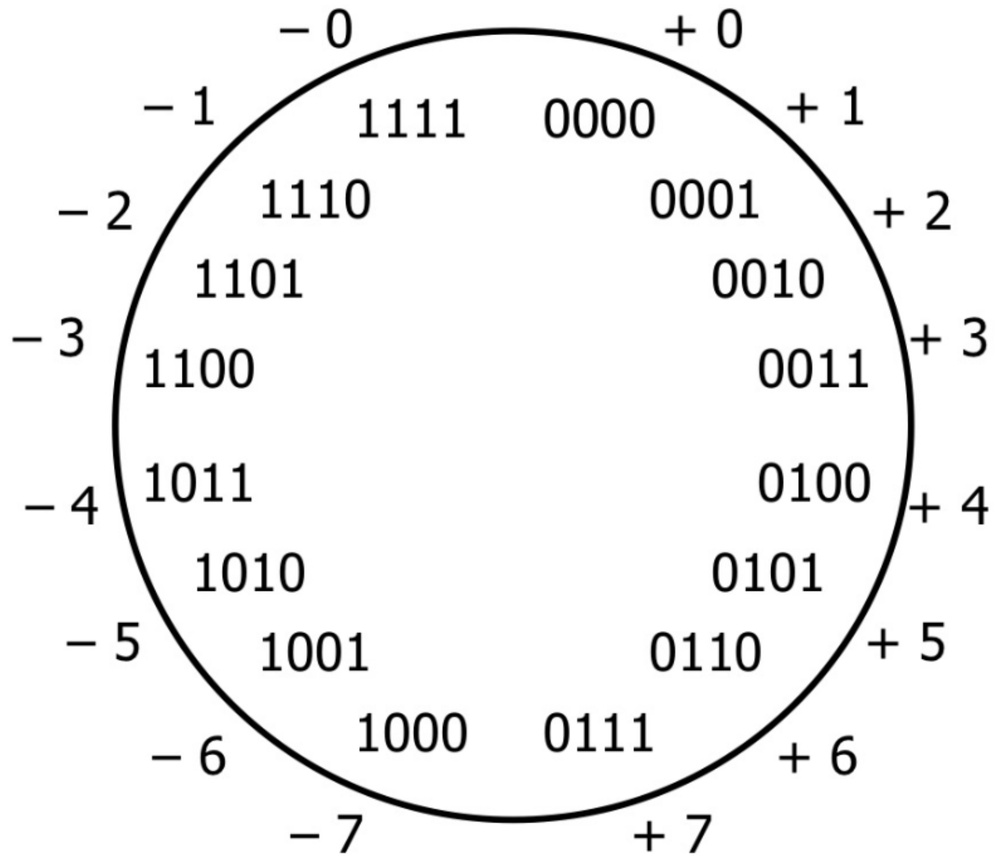


Image courtesy: Mark Oskin-University of Washington

Ones' Complement Negatives

- Negative numbers: bitwise complements of positive numbers
- Solves the arithmetic problem
- Need to add back the carry from MSB

Add		Invert, add, add carry		Invert and add	
4	0100	4	0100	– 4	1011
+ 3	+ 0011	– 3	+ 1100	+ 3	+ 0011
= 7	= 0111	= 1	1 0000	– 1	1110
		add carry:	+1		
			= 0001		

end-around carry

Slide adapted from: Mark Oskin-University of Washington

4 digit binary numbers

y : $b_3\ b_2\ b_1\ b_0$

$-y$: $(1-b_3)\ (1-b_2)\ (1-b_1)\ (1-b_0)$

$$-y = 2^4 - y - 1$$

$y = 1$

0 0 0 1

$-y$ 1 1 1 0 ✓

$$1 + y + (-y) = 1111$$

$10000 = 2^4$

$$\begin{array}{r} 10000 \\ -\ 0001 \\ \hline 1111 \end{array}$$

$(-y_1) + (-y_2)$

$(2^d - y_1 - 1) + (2^d - y_2 - 1)$

$2^{d+1} \boxed{- y_1 - y_2 - 1} - 1$

Homework Exercise

- Assume k-bits to represent numbers
- What is the range of numbers in one's complement representation?
- Overflow: When the result is out of range
- Suppose we use one's complement representation for negative numbers
- Prove that if you add back the MSB carry then you will always get the correct results (if there is no overflow)
  - Three cases: +ve + +ve, +ve + -ve, -ve + -ve

Number Representation: Idea 3

- Take one's complement and add 1
- Called two's complement

Two's Complement Negatives

- Bitwise complement plus one
- Only one zero
- Range: -2^{k-1} to $2^{k-1}-1$

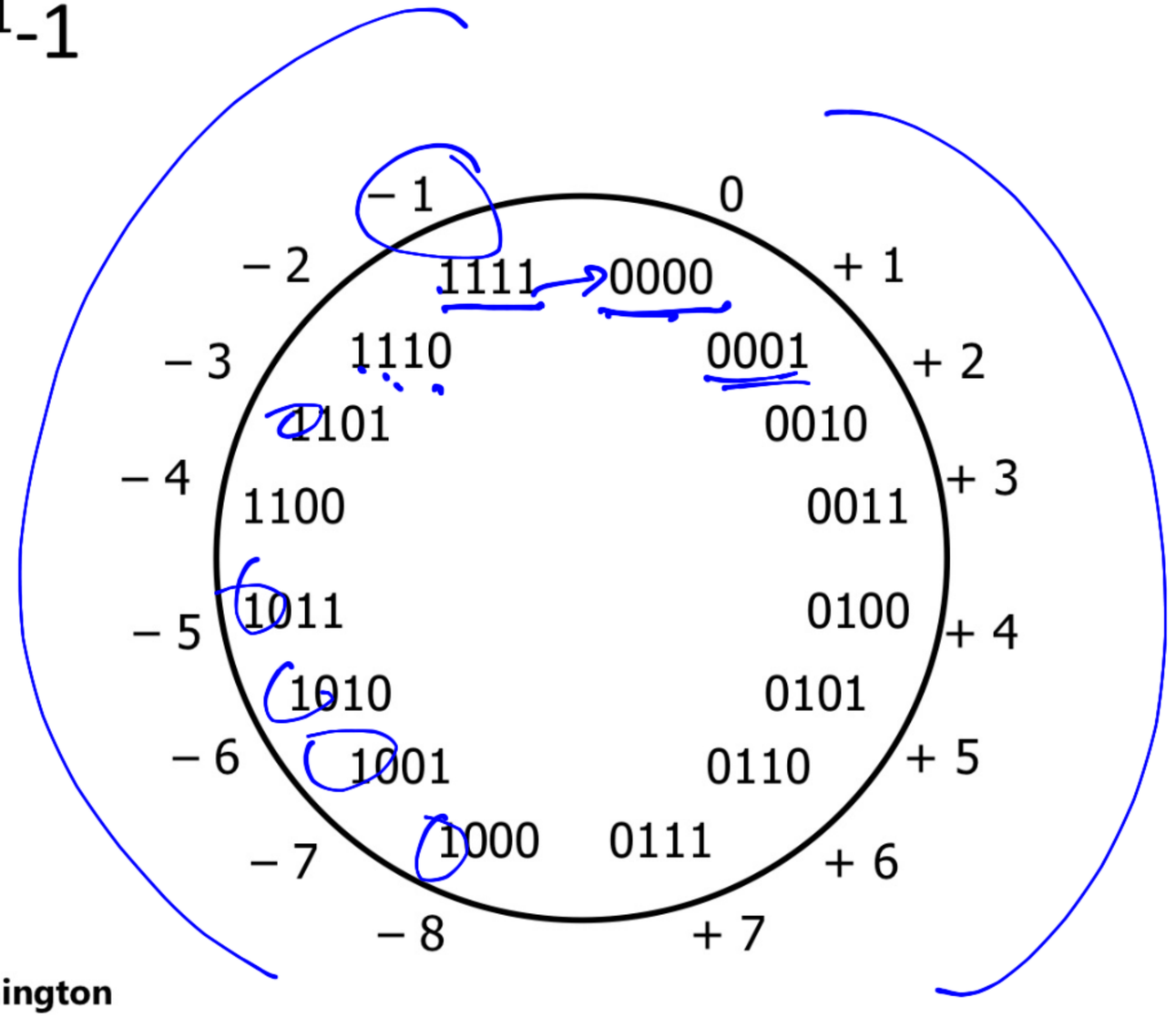
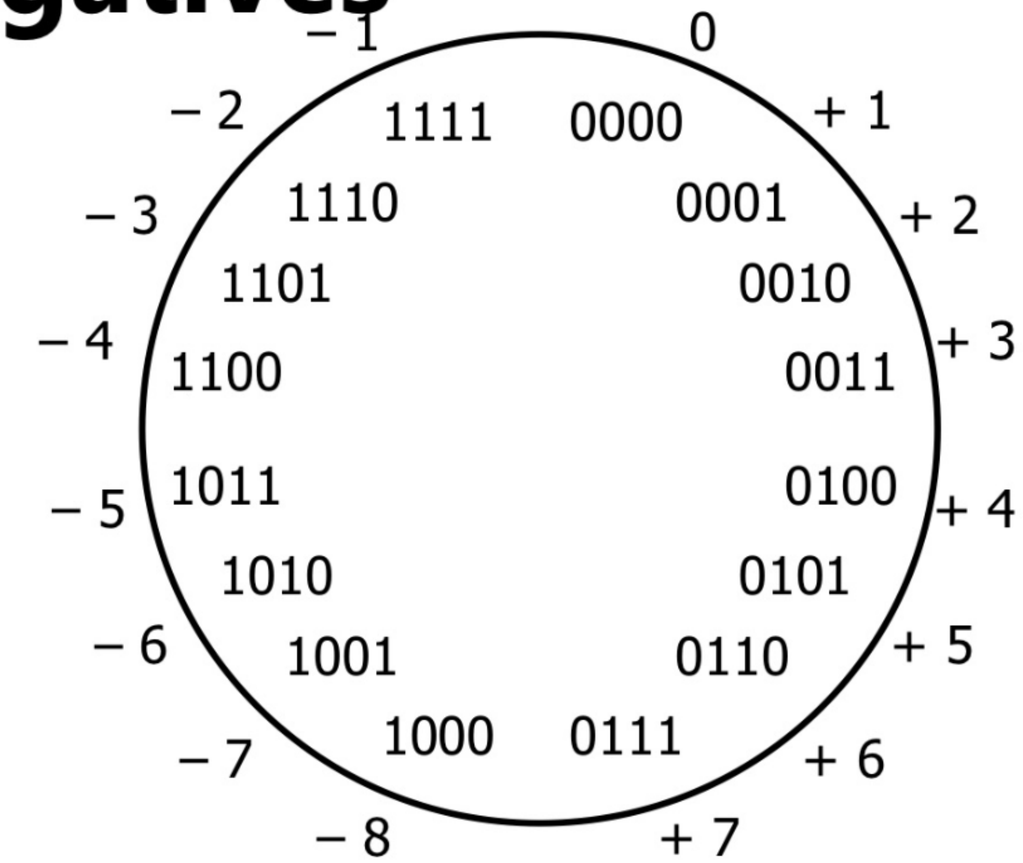


Image courtesy: Mark Oskin-University of Washington

Two's Complement Negatives

- Simplifies arithmetic
- Simple addition; discard the highest carry bit
- Same hardware to add signed and unsigned
- Used in most computers today

Two's Complement Negatives



Add		Invert and add		Invert and add	
4	0100	4	0100	− 4	1100
+ 3	+ 0011	− 3	+ 1101	+ 3	+ 0011
= 7	= 0111	= 1	1 0001	− 1	1111
		drop carry	= 0001		

Slide adapted from: Mark Oskin-University of Washington

Homework

- Prove that in the absence of overflows, if the numbers are added as unsigned, the two's complement representation will always give correct answer
  - Cases: +ve + +ve, +ve + -ve, -ve + -ve

Hint

- The ones' complement of a b-bit positive number y is $(2^b - 1) - y$
- Two's complement adds one to the bitwise complement, thus, -y is $2^b - y$
  - $-y$ and $2^b - y$ are equal mod 2^b

Dealing with overflows

X	Unsigned	Signed
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 CPU may be capable of “throwing an exception” for overflow on signed values
- It won't for unsigned
- But C and Java just cruise along silently when overflow occurs...
- Need to do something special for raising overflow exception

Slide adapted from: Mark Oskin-University of Washington

Mapping Signed and Unsigned

Bits	Signed	Unsigned
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	-7	9
1010	-6	10
1011	-5	11
1100	-4	12
1101	-3	13
1110	-2	14
1111	-1	15

=

+16

Slide adapted from: Mark Oskin-University of Washington

Bit-Wise Operations in C/C++

&: bitwise AND operation – AND of the bits

|: bitwise OR operation -- OR of the bits

<<: Bitwise left shift operation

- Throw away bits on the left
- Fill bits on the right with zeros

>>: Bitwise right shift operation

- Throw away bits on the right
- Fill bits on the left with zeros (for unsigned)
- Fills bits on the left with sign (for signed)

Bit-Wise Operations in C/C++

```
unsigned int x, y, z;
x = 0xF0;
y = 0x8D;
z = x & y;
z = x | y;
z = (x >> 2)
z = (y << 1)
z = (x >> 2) | (y << 1)
```

Shift Operations

- Left shift: **`x << y`**
  - Shift bit-vector `x` left by `y` positio
    - Throw away extra bits on left
    - Fill with 0s on right
  - Multiply by 2**y
- Right shift: **`x >> y`**
  - Shift bit-vector `x` right by `y` positions
    - Throw away extra bits on right
  - Logical shift (for unsigned)
    - Fill with 0s on left
  - Arithmetic shift (for signed)
    - Replicate most significant bit on righ
    - Maintain sign of x
  - Divide by 2**y
  - Correct truncation (towards 0) requires some care with signed numbers

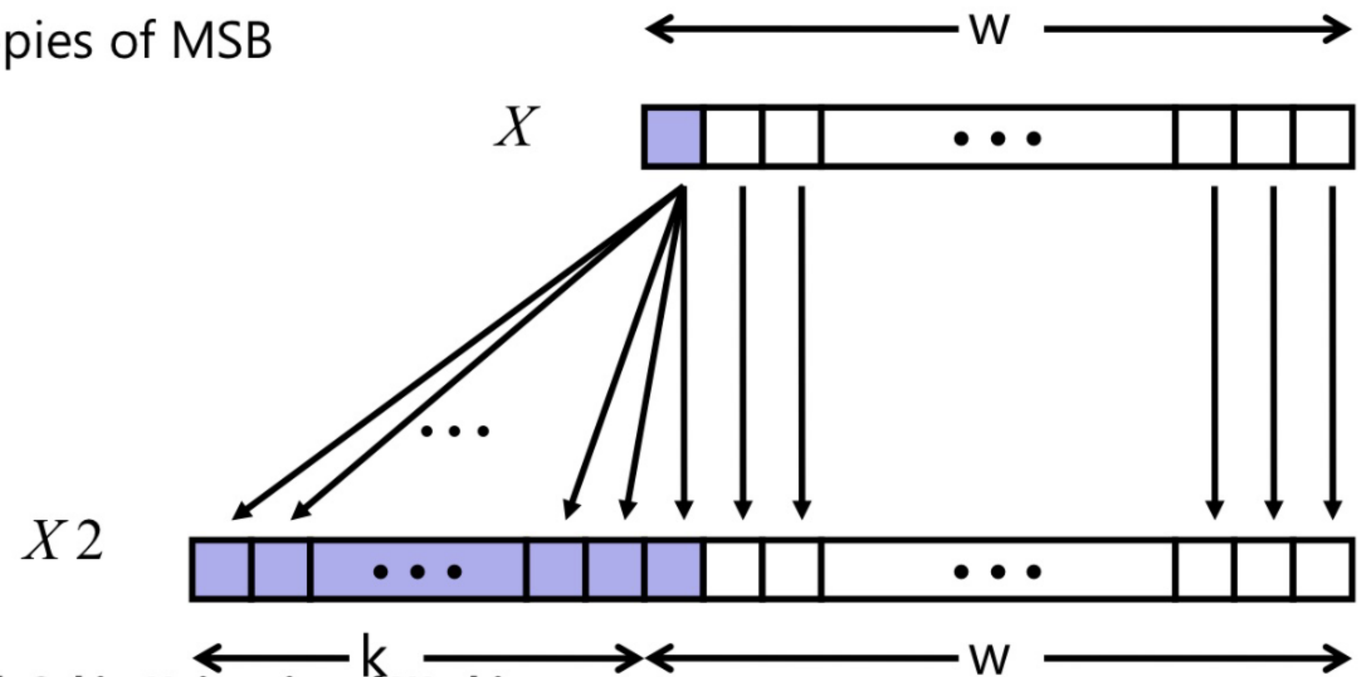
Argument x	01100010
<< 3	00010000
Logical >> 2	00011000
Arithmetic >> 2	00011000

Argument x	10100010
<< 3	00010000
Logical >> 2	00101000
Arithmetic >> 2	11101000

Slide adapted from: Mark Oskin-University of Washington

Sign Extension

- Task:
  - Given w-bit signed integer x
  - Convert it to w+k-bit integer with same value
- Rule:
  - Make k copies of sign bit:
  - $X\,2 = \underbrace{x_{w-1}, \ldots, x_{w-1}}_{\text{k copies of MSB}}, x_{w-1}, x_{w-2}, \ldots, x_0$



Slide adapted from: Mark Oskin-University of Washington

Sign Extension Example

```
short int x =  12345;
int       ix = (int) x;
short int y = -12345;
int       iy = (int) y;
```

	Decimal	Hex	Binary
x	12345	30 39	00110000 01101101
ix	12345	00 00 30 39	00000000 00000000 00110000 01101101
y	-12345	CF C7	11001111 11000111
iy	-12345	FF FF CF C7	11111111 11111111 11001111 11000111

- Converting from smaller to larger integer data type
- C automatically performs sign extension
- You might have to if converting a bizarre data type to a native one (e.g. PMC counters are sometimes 48 bits)

Representing Characters and Strings

Representing Characters

- Generally, characters are stored as 8-bit objects
- Two major representations
  - ASCII
    - American Standard Code for Information Interchange
  - EBCDIC
    - Extended Binary Coded Decimal Interchange Code
- Newer representation
  - Unicode supports major scripts of the world
  - One character is generally 2 bytes (up to 4 bytes)

ASCII Representation

```
char x = 'A';
int y = x;
cout << y;
```

32	space	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	”	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	’	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	I	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	del

One byte is used for every character. Most computers use ASCII. UNICODE is now displacing it.

Slide adapted from: Mark Oskin-University of Washington

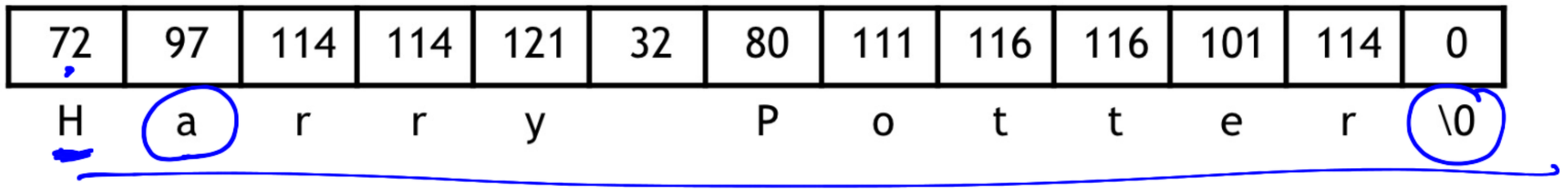
String Representation

- **A C-style string is represented by an array of bytes.**
  - Elements are one-byte ASCII codes for each character.
  - A 0 value marks the end of the array.

Slide adapted from: Mark Oskin-University of Washington

Null-terminated Strings

- **For example, "Harry Potter" can be stored as a 13-byte array.**



- **Why do we put a a 0, or null, at the end of the string?**

- **Computing string length?**

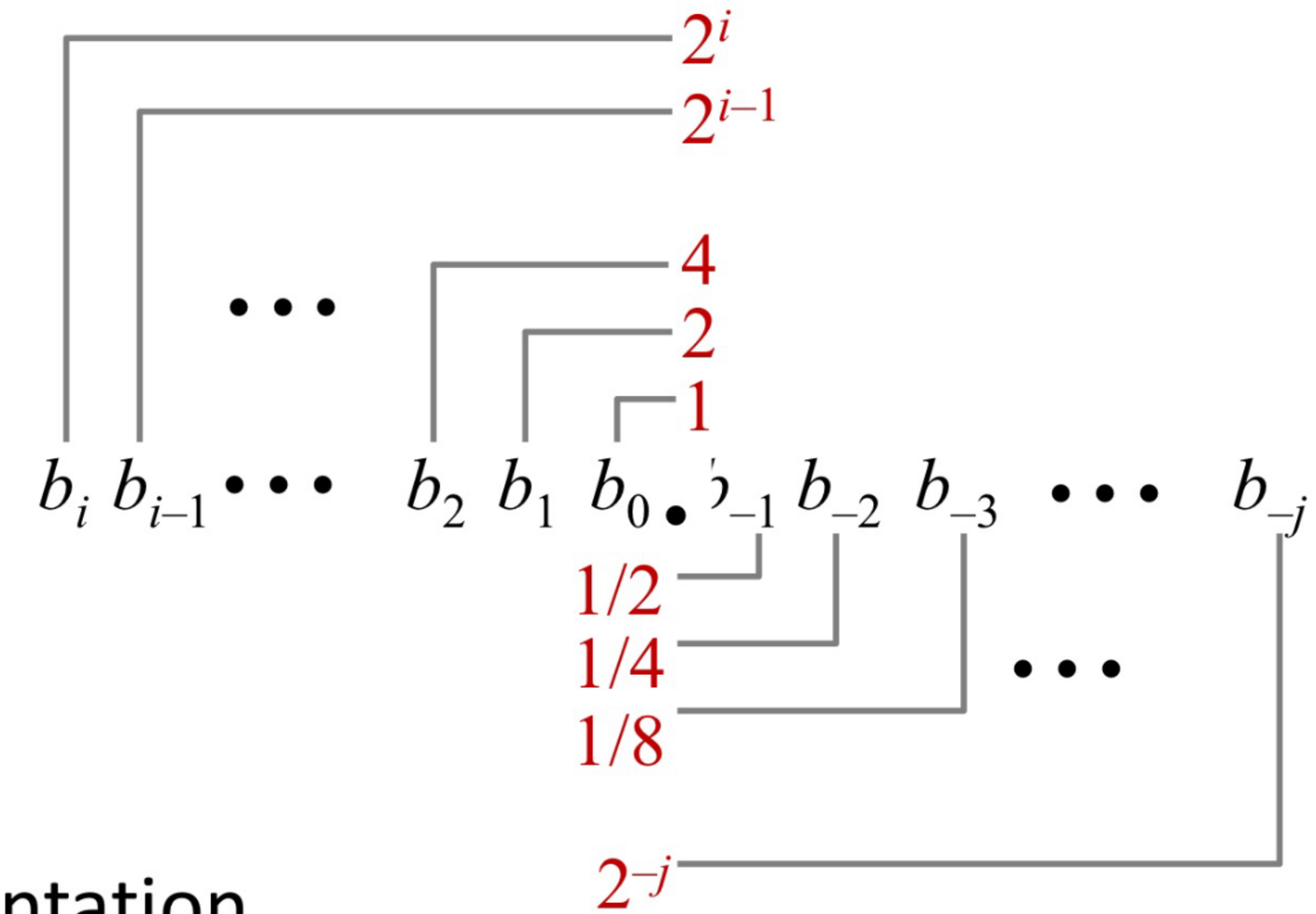
Slide adapted from: Mark Oskin-University of Washington

Representing Real Numbers

Fractional binary numbers

- What is 23.982?
- Similarly in base 2, what is 1011.101?

Fractional Binary Numbers



- Representation
  - Bits to right of “binary point” represent fractional powers of 2
  - Represents rational number: $\sum_{k=-j}^{i} b_k \times 2^k$

Slide adapted from: Mark Oskin-University of Washington

Fractional Binary Numbers: Examples

• Value	Representation
5 and 3/4	101.11_2
2 and 7/8	10.111_2
63/64	0.111111_2

- Observations
  - Divide by 2 by shifting right
  - Multiply by 2 by shifting left

Slide adapted from: Mark Oskin-University of Washington

Representable Numbers

- Limitation
  - Can only exactly represent numbers of the form $x/2^k$
  - Other rational numbers have repeating bit representations

Value	Representation
1/3	$0.0101010101[01]\ldots_2$
1/5	$0.001100110011[0011]\ldots_2$
1/10	$0.0001100110011[0011]\ldots_2$

Slide adapted from: Mark Oskin-University of Washington

Fixed Point Representation

- Represent the integer and fractional part of a number using fixed number of digits (bits)
- Example 1: 32 bits for the integral part and 32 bits for the fractional part
- Example 2: 20 bits for the integral part and 44 bits for the fractional part
- Precision: Smallest possible difference between two numbers
- Range: Difference between smallest and largest possible number
- Precision:
- Range:

Fixed Point Pros and Cons

- Pros
  - Simple
  - The same hardware that does integer arithmetic can do fixed point arithmetic
  - In fact, the programmer can use int with an implicit fixed point
- Cons
  - There is no good way to pick where the fixed point should be

    Sometimes you need range, sometimes you need precision. The more you have of one, the less of the other

Slide adapted from: Mark Oskin-University of Washington

IEEE Floating Point

- Floating point representation: Analogous to scientific notation
  - Not 12000000 but 1.2 x 10^{7}
  - Not 0.0000012 but 1.2 x 10^{-6}
- IEEE Standard 754
  - Established in 1985 as uniform standard for floating point arithmetic
    - Before that, many idiosyncratic formats
  - Supported by all major CPUs
- Driven by numerical concerns
  - Nice standards for rounding, overflow, underflow
  - Hard to make fast in hardware
    - Numerical analysts predominated over hardware designers in defining standard

Slide adapted from: Mark Oskin-University of Washington

Floating Point Representation

- Numerical Form:
  $$(-1)^s \; M \;\; 2^E$$
- Sign bit s determines whether number is negative or positive
- Significand (mantissa) M normally a fractional value in range [1.0,2.0).
- Exponent E weights value by power of two

- Encoding
- MSB **s** is sign bit s
- **frac** field encodes M (but is not equal to M)
- **exp** field encodes E (but is not equal to E)

Slide adapted from: Mark Oskin-University of Washington

Types of floating points

- Single precision: 32 bits (float)

- Double precision: 64 bits (double)

- Extended precision: 80 bits (Intel only)

Slide adapted from: Mark Oskin-University of Washington

Other Data Representations

- Sizes of objects (in bytes)

Java Data Type	C Data Type	Typical 32-bit	x86-64
boolean	bool	1	1
byte	char	1	1
char		2	2
short	short int	2	2
int	int	4	4
float	float	4	4
	long int	4	8
double	double	8	8
long	long long	8	8
	long double	8	16
(reference)	pointer *	4	8

Slide adapted from: Mark Oskin-University of Washington

Thank You

Normalization and Special Values

- **"Normalized" means mantissa has form 1.xxxxx**
- 0.011 x 2^5 and 1.1 x 2^3 represent the same number, but the latter makes better use of the available bits
- Since we know the mantissa starts with a 1, don't bother to store it

- **Special values:**
  - The float value 00…0 represents zero
  - If the exp == 11…1 and the mantissa == 00…0, it represents
  - E.g., 10.0 / 0.0 →
  - **If the exp == 11…1 and the mantissa != 00…0, it represents NaN**
  - "Not a Number"
  - Results from operations with undefined result

E.g., 0 *

How do we do operations?

- Is representation exact?
- How are the operations carried out?

Floating Point Operations: Basic Idea

- **$x +_f y = \text{Round}(x + y)$**
- **$x *_f y = \text{Round}(x * y)$**

- Basic idea
  - First compute exact result
  - Make it fit into desired precision
    - Possibly overflow if exponent too large
    - Possibly round to fit into **`frac`**

Floating Point Multiplication

$(-1)^{s1}\ M1\ \ 2^{E1}\ \ *\ \ (-1)^{s2}\ M2\ \ 2^{E2}$

- Exact Result: $(-1)^{s}\ M\ \ 2^{E}$
  - Sign s: s1 ^ s2
  - Significand M: M1 * M2
  - Exponent E: E1 + E2

- Fixing
  - If M ≥ 2, shift M right, increment E
  - If E out of range, overflow
  - Round M to fit **`frac`** precision

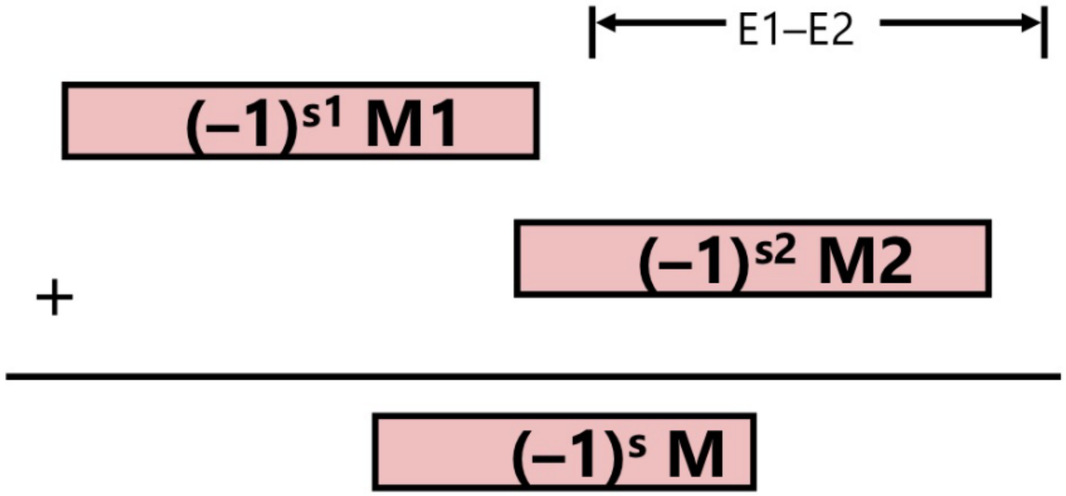
- Implementation
  - What is hardest?

Floating Point Addition

$(-1)^{s1}$ M1 2^{E1} + $(-1)^{s2}$ M2 2^{E2}

Assume E1 > E2

- Exact Result: $(-1)^{s}$ M 2^{E}
  - Sign s, significand M:
    - Result of signed align & add
  - Exponent E: E1



- Fixing
  - If M $\geq$ 2, shift M right, increment E
  - if M < 1, shift M left k positions, decrement E by k
  - Overflow if E out of range
  - Round M to fit **`frac`** precision

Mathematical Properties of FP Operations

- Not really associative or distributive due to rounding
- Infinities and NaNs cause issues
- Overflow and infinity

Floating Point in C

- C Guarantees Two Levels
  - **`float`** single precision
  - **`double`** double precision

- Conversions/Casting
  - Casting between **`int`**, **`float`**, and **`double`** changes bit representation
  - **`Double/float`** → **`int`**
    - Truncates fractional part
    - Like rounding toward zero
    - Not defined when out of range or NaN: Generally sets to TMin
  - **`int`** → **`double`**
    - Exact conversion, why?
  - **`int`** → **`float`**
    - Will round according to rounding mode

Floating Point and the Programmer

```
#include <stdio.h>

int main(int argc, char* argv[]) {

  float f1 = 1.0;
  float f2 = 0.0;
  int i;
  for ( i=0; i<10; i++ ) {
    f2 += 1.0/10.0;
  }
  printf("0x%08x  0x%08x\n", *(int*)&f1, *(int*)&f2);
  printf("f1 = %10.8f\n", f1);
  printf("f2 = %10.8f\n\n", f2);

  f1 = 1E30;
  f2 = 1E-30;
  float f3 = f1 + f2;
  printf ("f1 == f3? %s\n", f1 == f3 ? "yes" : "no" );

  return 0;
}
```

Floating Point and the Programmer

```
#include <stdio.h>

int main(int argc, char* argv[]) {

  float f1 = 1.0;
  float f2 = 0.0;
  int i;
  for ( i=0; i<10; i++ ) {
    f2 += 1.0/10.0;
  }

  printf("0x%08x  0x%08x\n", *(int*)&f1, *(int*)&f2);
  printf("f1 = %10.8f\n", f1);
  printf("f2 = %10.8f\n\n", f2);

  f1 = 1E30;
  f2 = 1E-30;
  float f3 = f1 + f2;
  printf ("f1 == f3? %s\n", f1 == f3 ? "yes" : "no" );

  return 0;
}
```

```
$ ./a.out
0x3f800000  0x3f800001
f1 = 1.000000000
f2 = 1.000000119

f1 == f3? yes
```

Summary

- **As with integers, floats suffer from the fixed number of bits available to represent them**
- **Can get overflow/underflow, just like ints**
- **Some "simple fractions" have no exact representation**
  - **E.g., 0.1**
- **Can also lose precision, unlike ints**
  - **"Every operation gets a slightly wrong result"**
- **Mathematically equivalent ways of writing an expression may compute differing results**
- **NEVER test floating point values for equality!**