Number Representation

Prof. Harry H. Porter
Portland State University

Decimal Number Representation

Example:

4037

$= 4000 + 30 + 7$

$= \ldots + 0 \cdot 10000 + 4 \cdot 1000 + 0 \cdot 100 + 3 \cdot 10 + 7 \cdot 1$

$= \ldots + 0 \cdot 10^4 + 4 \cdot 10^3 + 0 \cdot 10^2 + 3 \cdot 10^1 + 7 \cdot 10^0$

Base 10:

$\ldots + X \cdot 10^4 + X \cdot 10^3 + X \cdot 10^2 + X \cdot 10^1 + X \cdot 10^0$

Set of numerals (the “digits”):

{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }

Hexadecimal Number Representation

Base 16:

$$\ldots + X \cdot 16^4 + X \cdot 16^3 + X \cdot 16^2 + X \cdot 16^1 + X \cdot 16^0$$

$$\ldots + X \cdot 65536 + X \cdot 4096 + X \cdot 256 + X \cdot 16 + X \cdot 1$$

Set of numerals:

{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F }

Example:

3A0F

$$= \ldots + 0 \cdot 16^4 + 3 \cdot 16^3 + A \cdot 16^2 + 0 \cdot 16^1 + F \cdot 16^0$$

$$= \ldots + 0 \cdot 65536 + 3 \cdot 4096 + A \cdot 256 + 0 \cdot 16 + F \cdot 1$$

$$= \ldots + 0 \cdot 65536 + 3 \cdot 4096 + 10 \cdot 256 + 0 \cdot 16 + 15 \cdot 1$$

$$= 12{,}288 + 2{,}560 + 15 = 14{,}863 \text{ (in decimal)}$$

Decimal	Hex
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	A
11	B
12	C
13	D
14	E
15	F

Hexadecimal Number Representation

Base 16:

$$\ldots + X \cdot 16^4 + X \cdot 16^3 + X \cdot 16^2 + X \cdot 16^1 + X \cdot 16^0$$

$$\ldots + X \cdot 65536 + X \cdot 4096 + X \cdot 256 + X \cdot 16 + X \cdot 1$$

Set of numerals:

{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F }

Example:

2CB

$$= \ldots + 0 \cdot 16^4 + 0 \cdot 16^3 + 2 \cdot 16^2 + C \cdot 16^1 + B \cdot 16^0$$

Hexadecimal Number Representation

Base 16:

$$\ldots + X \cdot 16^4 + X \cdot 16^3 + X \cdot 16^2 + X \cdot 16^1 + X \cdot 16^0$$

$$\ldots + X \cdot 65536 + X \cdot 4096 + X \cdot 256 + X \cdot 16 + X \cdot 1$$

Set of numerals:

{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F }

Example:

2CB

$$= \ldots + 0 \cdot 16^4 + 0 \cdot 16^3 + 2 \cdot 16^2 + C \cdot 16^1 + B \cdot 16^0$$

$$= \ldots + 0 \cdot 65536 + 0 \cdot 4096 + 2 \cdot 256 + C \cdot 16 + B \cdot 1$$

$$= \ldots + 0 \cdot 65536 + 0 \cdot 4096 + 2 \cdot 256 + 12 \cdot 16 + 11 \cdot 1$$

$$= 512 + 192 + 11 = 715 \text{ (in decimal)}$$

Binary Number Representation

Base 2:

$$\ldots + X \cdot 2^5 + X \cdot 2^4 + X \cdot 2^3 + X \cdot 2^2 + X \cdot 2^1 + X \cdot 2^0$$

$$\ldots + X \cdot 32 + X \cdot 16 + X \cdot 8 + X \cdot 4 + X \cdot 2 + X \cdot 1$$

Set of numerals:

{ 0, 1}

Example:

110101

$$= \ldots + 1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$$

$$= \ldots + 1 \cdot 32 + 1 \cdot 16 + 0 \cdot 8 + 1 \cdot 4 + 0 \cdot 2 + 1 \cdot 1$$

$$= 32 + 16 + 4 + 1$$

= 53 (in decimal)

Decimal Number Representation

10,000	1,000	100	10	1
4	**8**	**2**	**9**	**3**

Binary Number Representation

64	32	16	8	4	2	1
1	**0**	**1**	**1**	**0**	**1**	**1**

Hex Number Representation

16,777,216	1,048,576	65,536	4,096	256	16	1
4	**E**	**7**	**D**	**F**	**2**	**0**

“C” Notation

Decimal

```
48293
```

Binary

(not standard)

Hex

```
0x4E7DF20
0x4e7df20
```

Practice

Convert the following

10110111_2 to Base 10 =
11011001_2 to Base 16 =
0x2ae to Base 2 =
0x13e to Base 10 =
150_{10} to Base 2 =

301_{10} to Base 16 =

Binary Number Representation

128	64	32	16	8	4	2	1

Hex Number Representation

16,777,216	1,048,576	65,536	4,096	256	16	1

Practice

Convert the following

10110111_2 to Base 10 = $128 + 32 + 16 + 4 + 2 + 1$ = 183

11011001_2 to Base 16 = 0xd9

0x2ae to Base 2 = $0010\ 1010\ 1110_2$

0x13e to Base 10 = $1 \cdot 256 + 3 \cdot 16 + 14 = 318_{10}$

150_{10} to Base 2 =

$1 \cdot 128 + 0 \cdot 64 + 0 \cdot 32 + 1 \cdot 16 + 0 \cdot 8 + 1 \cdot 4 + 1 \cdot 2 + 0 \cdot 1 = 010010110_2$

301_{10} to Base 16 = $1 \cdot 256 + 3 \cdot 16 + 13$ = 0x12d

Binary Number Representation

128	64	32	16	8	4	2	1

Hex Number Representation

16,777,216	1,048,576	65,536	4,096	256	16	1

One-to-one correspondence between hex and binary;

```
  3    A    0    F
0011 1010 0000 1111
```

Decimal	Binary	Hex
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Byte (8 bits)

Hex: `3A`

Binary: `0011 1010`

Halfword (16 bits)

Hex: `3A0F`

Binary: `0011 1010 0000 1111`

Word (32 bits)

Hex: `3AOF 12D8`

Binary: `0011 1010 0000 1111 0001 0010 1101 1000`

Octal Notation

Bad match with byte alignment

```
 1     5     2     2     6     1
 1   1 0 1   0 1 0   0 1 0   1 1 0   0 0 1
 1 1 0 1 0 1 0 0   1 0 1 1 0 0 0 1
      byte             byte
```

Decimal	Binary	Octal
0	000	0
1	001	1
2	010	2
3	011	3
4	100	4
5	101	5
6	110	6
7	111	7

The numbers get too long.

Word (32 bits)

Octal: **12305570426**

Hex: **3A0F 12D8**

Every octal looks like a decimal number (and often they get confused).

$263_{8} = 179_{10}$

$263_{10} = 263_{10}$

$263_{16} = 611_{10}$

C Notation for octals (leading zero is significant!)

```
0263
```

Data Representations in C

(Size in bytes)

C Data Type	IA32
char	1
short	2
int	4
long	4
long long	8
float	4
double	8
long double	16
char *	4
(…or any other pointer)	

short = short int
long = long int
long long = long long int

Data Representations in C

(Size in bytes)

C Data Type	IA32	x86-64
char	1	1
short	2	2
int	4	4
long	4	8
long long	8	8
float	4	4
double	8	8
long double	16	16
char *	4	8
(…or any other pointer)		

short = short int
long = long int
long long = long long int

Unsigned Number Representation

Example: 8-bits

Always non-negative

0,1,2, ... 255

0,1,2, ... 2^8-1

Value (in decimal)	Binary	Hex
0	0000 0000	00
1	0000 0001	01
2	0000 0010	02
3	0000 0011	03
4	0000 0100	04
5	0000 0101	05
6	0000 0110	06
7	0000 0111	07
...	...	...
252	1111 1100	FC
253	1111 1101	FD
254	1111 1110	FE
255	1111 1111	FF

Unsigned Number Representation

Example: 32-bits

Always non-negative

0,1,2, ... 4,294,967,295

0,1,2, ... 2^{32}-1

Value (in decimal)	Binary	Hex
0	0000 0000 0000 0000 0000 0000 0000 0000	0000 0000
1	0000 0000 0000 0000 0000 0000 0000 0001	0000 0001
2	0000 0000 0000 0000 0000 0000 0000 0010	0000 0002
3	0000 0000 0000 0000 0000 0000 0000 0011	0000 0003
4	0000 0000 0000 0000 0000 0000 0000 0100	0000 0004
5	0000 0000 0000 0000 0000 0000 0000 0101	0000 0005
6	0000 0000 0000 0000 0000 0000 0000 0110	0000 0006
7	0000 0000 0000 0000 0000 0000 0000 0111	0000 0007
...	...	...
4,294,967,292	1111 1111 1111 1111 1111 1111 1111 1100	FFFF FFFC
4,294,967,293	1111 1111 1111 1111 1111 1111 1111 1101	FFFF FFFD
4,294,967,294	1111 1111 1111 1111 1111 1111 1111 1110	FFFF FFFE
4,294,967,295	1111 1111 1111 1111 1111 1111 1111 1111	FFFF FFFF

Unsigned Number Representation

Largest Number Representable

Byte (8-bits)

2^8-1

= 255

= FF (in hex)

Halfword (16-bits)

2^{16}-1

= 65,535

= 64K - 1

= FFFF (in hex)

Word (32-bits)

2^{32}-1

= 4,294,967,295

= 4G -1

= FFFF FFFF (in hex)

"Two's complement" number representation

Signed Number Representation

Example: 8-bits

Binary	Hex	Unsigned Value	Signed Value
0000 0000	00	0	
0000 0001	01	1	
0000 0010	02	2	
...	...	...	
0111 1101	7D	125	
0111 1110	7E	126	
0111 1111	7F	127	
1000 0000	80	128	
1000 0001	81	129	
1000 0010	82	130	
...	...	...	
1111 1101	FD	253	
1111 1110	FE	254	
1111 1111	FF	255	

“Two’s complement” number representation

Signed Number Representation

Example: 8-bits

Binary	Hex	Unsigned Value	Signed Value	
0000 0000	00	0	0	
0000 0001	01	1	1	
0000 0010	02	2	2	
...	...	...	...	
0111 1101	7D	125	125	2^7-3
0111 1110	7E	126	126	2^7-2
0111 1111	7F	127	127	2^7-1
1000 0000	80	128		
1000 0001	81	129		
1000 0010	82	130		
...	...	...		
1111 1101	FD	253		
1111 1110	FE	254		
1111 1111	FF	255		

"Two's complement" number representation

Signed Number Representation

Example: 8-bits

Binary	Hex	Unsigned Value	Signed Value	
0000 0000	00	0	0	
0000 0001	01	1	1	
0000 0010	02	2	2	
...	...	...	...	
0111 1101	7D	125	125	2^7-3
0111 1110	7E	126	126	2^7-2
0111 1111	7F	127	127	2^7-1
1000 0000	80	128	-128	$-(2^7)$
1000 0001	81	129	-127	$-(2^7-1)$
1000 0010	82	130	-126	$-(2^7-2)$
...	...	...	...	
1111 1101	FD	253	-3	
1111 1110	FE	254	-2	
1111 1111	FF	255	-1	

"Two's complement" number representation

Signed Number Representation

Example: 8-bits

Most significant bit
0 means ≥ zero (in hex: 0..7)
1 means < zero (in hex: 8..F)

Binary	Hex	Unsigned Value	Signed Value	
0000 0000	00	0	0	
0000 0001	01	1	1	
0000 0010	02	2	2	
...	...	...	...	
0111 1101	7D	125	125	2^7-3
0111 1110	7E	126	126	2^7-2
0111 1111	7F	127	127	2^7-1
1000 0000	80	128	-128	$-(2^7)$
1000 0001	81	129	-127	$-(2^7-1)$
1000 0010	82	130	-126	$-(2^7-2)$
...	...	...	...	
1111 1101	FD	253	-3	
1111 1110	FE	254	-2	
1111 1111	FF	255	-1	

"Two's complement" number representation

Signed Number Representation

Example: 8-bits

Most significant bit
0 means ≥ zero (in hex: 0..7)
1 means < zero (in hex: 8..F)

Binary	Hex	Unsigned Value	Signed Value	
0000 0000	00	0	0	
0000 0001	01	1	1	
0000 0010	02	2	2	
...	...	...	...	
0111 1101	7D	125	125	2^7-3
0111 1110	7E	126	126	2^7-2
0111 1111	7F	127	127	2^7-1
1000 0000	80	128	-128	$-(2^7)$
1000 0001	81	129	-127	$-(2^7-1)$
1000 0010	82	130	-126	$-(2^7-2)$
...	...	...	...	
1111 1101	FD	253	-3	
1111 1110	FE	254	-2	
1111 1111	FF	255	-1	

Always one more negative number than positive numbers:

$$\underbrace{-128,\ \ldots\ ,\ -1,}_{2^7 = 128 \text{ values}}\ \underbrace{0,\ 1,\ \ldots\ +127}_{2^7 = 128 \text{ values}}$$

$2^7 = 128$ values $+$ $2^7 = 128$ values $=$ $2^8 = 256$ values

“Two’s complement” number representation

Signed Number Representation

Example: 32-bits

Binary	Hex	Unsigned Value	Signed Value
0000...0000	0000 0000	0	
0000...0001	0000 0001	1	
0000...0010	0000 0002	2	
...	...	...	
0111...1101	7FFF FFFD	2,147,483,645	
0111...1110	7FFF FFFE	2,147,483,646	
0111...1111	7FFF FFFF	2,147,483,647	
1000...0000	8000 0000	2,147,483,648	
1000...0001	8000 0001	2,147,483,649	
1000...0010	8000 0002	2,147,483,650	
...	...	...	
1111...1101	FFFF FFFD	4,294,967,294	
1111...1110	FFFF FFFE	4,294,967,295	
1111...1111	FFFF FFFF	4,294,967,296	

"Two's complement" number representation

Signed Number Representation

Example: 32-bits

Binary	Hex	Unsigned Value	Signed Value	
0000...0000	0000 0000	0	0	
0000...0001	0000 0001	1	1	
0000...0010	0000 0002	2	2	
...	...	...	...	
0111...1101	7FFF FFFD	2,147,483,645	2,147,483,645	$2^{31}-3$
0111...1110	7FFF FFFE	2,147,483,646	2,147,483,646	$2^{31}-2$
0111...1111	7FFF FFFF	2,147,483,647	2,147,483,647	$2^{31}-1$
1000...0000	8000 0000	2,147,483,648	-2,147,483,648	$-(2^{31})$
1000...0001	8000 0001	2,147,483,649	-2,147,483,647	$-(2^{31}-1)$
1000...0010	8000 0002	2,147,483,650	-2,147,483,646	$-(2^{31}-1)$
...	...	...	...	
1111...1101	FFFF FFFD	4,294,967,294	-3	
1111...1110	FFFF FFFE	4,294,967,295	-2	
1111...1111	FFFF FFFF	4,294,967,296	-1	

"Two's complement" number representation

Signed Number Representation

Example: 32-bits

Binary	Hex	Unsigned Value	Signed Value	
0000...0000	0000 0000	0	0	
0000...0001	0000 0001	1	1	
0000...0010	0000 0002	2	2	
...	...	...	...	
0111...1101	7FFF FFFD	2,147,483,645	2,147,483,645	$2^{31}-3$
0111...1110	7FFF FFFE	2,147,483,646	2,147,483,646	$2^{31}-2$
0111...1111	7FFF FFFF	2,147,483,647	2,147,483,647	$2^{31}-1$
1000...0000	8000 0000	2,147,483,648	-2,147,483,648	$-(2^{31})$
1000...0001	8000 0001	2,147,483,649	-2,147,483,647	$-(2^{31}-1)$
1000...0010	8000 0002	2,147,483,650	-2,147,483,646	$-(2^{31}-1)$
...	...	...	...	
1111...1101	FFFF FFFD	4,294,967,294	-3	
1111...1110	FFFF FFFE	4,294,967,295	-2	
1111...1111	FFFF FFFF	4,294,967,296	-1	

Always one more negative number than positive numbers:

$$\underbrace{-2{,}147{,}483{,}648,\ \ldots\ ,\ -1,}_{2^{31}\text{ values}}\ \underbrace{0,\ 1,\ \ldots\ +2{,}147{,}483{,}647}_{2^{31}\text{ values}}$$

2^{31} values + 2^{31} values = 2^{32} values

Ranges of Numbers Using "Signed" Values

...in the "two's complement" system of number representation:

	Total Number of Values		
Byte (8-bits)	2^8 256		
Halfword (16-bits)	2^{16} 64K 65,536		
Word (32-bits)	2^{32} 4G 4,294,967,296		

Ranges of Numbers Using "Signed" Values

...in the "two's complement" system of number representation:

	Total Number of Values	Largest Positive Number	Most Negative Number
Byte (8-bits)	2^8 256	2^7-1 127	-(2^7) -128
Halfword (16-bits)	2^{16} 64K 65,536		
Word (32-bits)	2^{32} 4G 4,294,967,296		

Ranges of Numbers Using "Signed" Values

...in the "two's complement" system of number representation:

	Total Number of Values	Largest Positive Number	Most Negative Number
Byte (8-bits)	2^8 256	2^7-1 127	-(2^7) -128
Halfword (16-bits)	2^{16} 64K 65,536	2^{15}-1 32K-1 32,767	-(2^{15}) -32K -32,768
Word (32-bits)	2^{32} 4G 4,294,967,296		

Ranges of Numbers Using "Signed" Values

...in the "two's complement" system of number representation:

	Total Number of Values	Largest Positive Number	Most Negative Number
Byte (8-bits)	2^8 256	2^7-1 127	-(2^7) -128
Halfword (16-bits)	2^{16} 64K 65,536	2^{15}-1 32K-1 32,767	-(2^{15}) -32K -32,768
Word (32-bits)	2^{32} 4G 4,294,967,296	2^{31}-1 2G-1 2,147,483,647	-(2^{31}) -2G -2,147,483,648

Leading Zeros Can Be Ignored

Decimal

```
1,234
… 000,000,000,000,001,234
```

Hex

```
3A0F1C
… 0000 0000 0000 0000 003A 0F1C
```

Binary

```
101011010
… 0000 0000 0000 0001 0101 1010
```

Two's Complement Numbers: Leading Ones Can Be Ignored

Decimal

–9,099

Hex

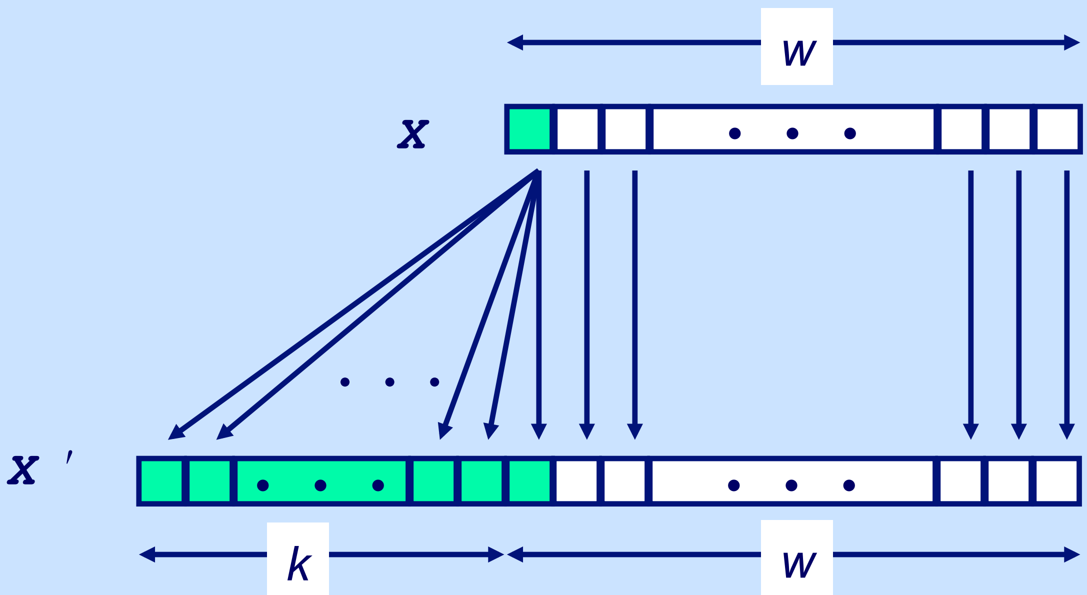
```
DC75
… FFFF FFFF FFFF FFFF FFFF DC75
```

Binary

```
1101 1100 0111 0101
… 1111 1111 1111 1111 1111 1101 1100 0111 0101
```

Sign Extension

Increase the size of a number by copying the "Sign Bit"



Sign Extension

Increase the size of a number by copying the "Sign Bit"

Binary - Positive

```
                          0110 1010   = 106₁₀
                0000 0000 0110 1010
0000 0000 0000 0000 0000 0000 0110 1010
```

$0110\ 1010 = 106_{10}$

Binary - Negative

```
                          1100 0101
                1111 1111 1100 0101
1111 1111 1111 1111 1111 1111 1100 0101
```

$1100\ 0101 = -59_{10}$

Hex - Positive

```
     6A
   006A
0000 006A
```

$6A = 106_{10}$

0-7: positive or zero

Hex- Negative

```
     C5
   FFC5
FFFF FFC5
```

$C5 = -59_{10}$

8-F: negative

Reducing the Size

Eliminate the leading bits.
Must not change the sign bit!!!
Must not eliminate significant bits!!!

Binary

```
0000 0000 0000 0000 0000 0000 0110 1010
                              0110 1010
```
$= 106_{10}$

```
1111 1111 1111 1111 1111 1111 1100 0101
                              1100 0101
```
$= -59_{10}$

Hex

```
0000 006A
       6A
```
$= 106_{10}$

```
FFFF FFC5
       C5
```
$= -59_{10}$

Reducing the Size

Eliminate the leading bits.
Must not change the sign bit!!!
Must not eliminate significant bits!!!

Binary

0000 0000 0000 0000 0000 0000 1100 0101 = 197_{10}

1111 1111 1111 1111 1111 1111 0110 1010 = -150_{10}

Hex

0000 00C5 = 197_{10}

FFFF FF6A = -150_{10}

Reducing the Size

Eliminate the leading bits.
Must not change the sign bit!!!
Must not eliminate significant bits!!!

Binary

```
0000 0000 0000 0000 0000 0000 1100 0101
                              1100 0101
```

$= 197_{10}$ / $= -59_{10}$

```
1111 1111 1111 1111 1111 1111 0110 1010
                              0110 1010
```

$= -150_{10}$ / $= 106_{10}$

Hex

`0000 00C5` $= 197_{10}$
`C5` $= -59_{10}$

`FFFF FF6A` $= -150_{10}$
`6A` $= 106_{10}$

Reducing the Size

Eliminate the leading bits.

Must not change the sign bit!!!

Must not eliminate significant bits!!!

Binary

```
0000 0000 0000 0000 0000 0000 1100 0101 = 197₁₀
                              1100 0101 = –59₁₀

1111 1111 1111 1111 1111 1111 0110 1010 = –150₁₀
                              0110 1010 = 106₁₀
```

Hex

```
0000 00C5   = 197₁₀
       C5   = –59₁₀

FFFF FF6A   = –150₁₀
       6A   = 106₁₀
```

OOPS!

What does C do?

```
char c;
short s;
int i;

c = -123;
s = c;
i = c;

printf (”i = %d\n", i);
printf ("s = %d\n", s);
printf (”c = %d\n", c);
```

Output:

```
i = -123
s = -123
c = -123
```

What does C do?

```
char c;
short s;
int i;

c = -123;
s = c;
i = c;

printf ("i = %d\n", i);
printf ("s = %d\n", s);
printf ("c = %d\n", c);
```

Sign extended; no problem

Output:

```
i = -123
s = -123
c = -123
```

$85_{16} = -123_{10}$

$FF85_{16} = -123_{10}$

$FFFF\ FF85_{16} = -123_{10}$

$1000\ 0101 = -123_{10}$

$1111\ 1111\ 1000\ 0101 = -123_{10}$

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1000\ 0101 = -123_{10}$

What does C do?

```
char c;
short s;
int i;

i = -59;
s = i;
c = i;

printf ("i = %d\n", i);
printf ("s = %d\n", s);
printf ("c = %d\n", c);
```

Output:

```
i = -59
s = -59
c = -59
```

Sometimes, truncation does not change the value…

What does C do?

```
char c;
short s;
int i;

i = 100000;
s = i;
c = i;

printf ("i = %d\n", i);
printf ("s = %d\n", s);
printf ("c = %d\n", c);
```

Sometimes it is a problem!

Output:

```
i = 100000
s = -31072
c = -96
```

What does C do?

```
char c;
short s;
int i;

i = 100000;
s = i;
c = i;

printf ("i = %d\n", i);
printf ("s = %d\n", s);
printf ("c = %d\n", c);
```

Sometimes it is a problem!

Output:

```
i = 100000
s = -31072
c = -96
```

$\texttt{0001 86A0}_{16} = 100{,}000_{10}$

$\texttt{86A0}_{16} = -31{,}072_{10}$

$\texttt{A0}_{16} = -96_{10}$

$\texttt{0000 0000 0000 0001 1000 0110 1010 0000} = 100{,}000_{10}$

$\texttt{1000 0110 1010 0000} = -31{,}072_{10}$

$\texttt{1010 0000} = -96_{10}$

Casting in C

The default is "signed"
Use **`unsigned`** keyword if you want it.

Casting from signed to unsigned:

```
short int              x =  15213;
unsigned short int ux = (unsigned short) x;
short int              y  = -15213;
unsigned short int uy = (unsigned short) y;
```

Copies bits without processing.
    Does not change bit values.

Non-negative values will be unchanged.
Negative values change into large positive values
    Why? Sign bit becomes the most significant bit.

Casting in C

The default is "signed"
Use **`unsigned`** keyword if you want it.

Casting from signed to unsigned:

```
short int                x =  15213;
unsigned short int ux = (unsigned short) x;
short int                y  = -15213;
unsigned short int uy = (unsigned short) y;
```

Copies bits without processing.
Does not change bit values.

Non-negative values will be unchanged.
Negative values change into large positive values
Why? Sign bit becomes the most significant bit.

Signed vs. Unsigned

Constants are assumed to be signed.
If you want a signed constant…

```
0U
123456U
```

You can cast between signed and unsigned.

```
int tx, ty;
unsigned ux, uy;
tx = (int) ux;
uy = (unsigned) ty;
```

Implicit casting
    Occurs in assignment statements
    Occurs when arguments are passed to functions
Compilers often warn about possible issues
    Always pay attention to compiler warnings and fix your code!

Casting Surprises

What relation do you expect these to have?
When both are mixed, the signed value is cast to unsigned.

`0`	`0U`	`==`	unsigned
`-1`	`0`	`<`	signed
`-1`	`0U`	`>`	unsigned
`2147483647`	`-2147483648`	`>`	signed
`2147483647U`	`-2147483648`	`<`	unsigned
`-1`	`-2`	`>`	signed
`(unsigned) -1`	`-2`	`>`	unsigned
`2147483647`	`2147483648U`	`<`	unsigned
`2147483647`	`(int) 2147483648U`	`>`	signed

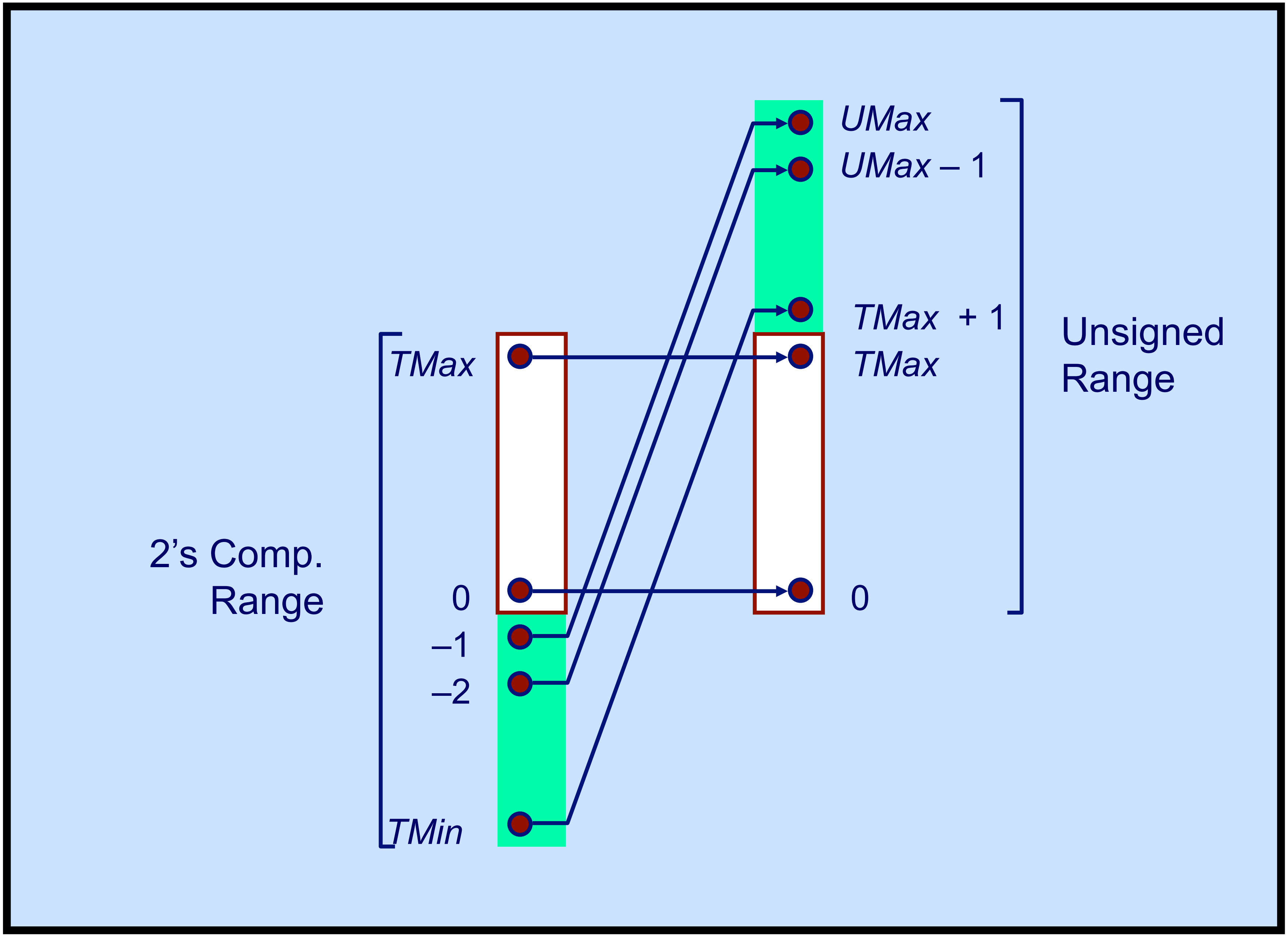
An Example Bug

Code for determining which string is longer.
What goes wrong here?

```
size_t strlen(const char*);

int strlonger(char *s, char *t) {
    return (strlen(s) - strlen(t)) > 0;
};
```

Imagine we use this to test to see if something will fit in a buffer.
It returns true.
We copy the string in to the buffer.
Buffer overrun → system security violation


UMax
UMax – 1
TMax + 1
TMax
0
Unsigned
Range
2’s Comp.
Range
TMax
0
–1
–2
TMin

Addition

Decimal:

```
    1 1 1
      3 8 5 3
 +    9 3 7 4
 ------------
    1 3 2 2 7
```

Binary:

```
  1 1 1   1
    1 1 1 0 1 1 0 0
 +  1 0 1 0 1 0 1 0
 ------------------
  1 1 0 0 1 0 1 1 0
```

Addition

Decimal:

```
    1 1 1
      3 8 5 3
  +   9 3 7 4
  -----------
    1 3 2 2 7
```

	0	1	2	3	4	5	6	7	8	9	10
0	0	1	2	3	4	5	6	7	8	9	10
1	1	2	3	4	5	6	7	8	9		
2	2	3	4	5	6	7	8				
3	3	4	5	6	7	8					
4	4	5	6	7	8						
5	5	6	7	8				etc.			
6	6	7									
7	7										
8	8										
9											
10											

Binary:

```
    1 1 1   1
      1 1 1 0 1 1 0 0
  +   1 0 1 0 1 0 1 0
  -------------------
    1 1 0 0 1 0 1 1 0
```

0 + 0 = 0

1 + 0 = 1

1 + 1 = 10

1 + 1 + 1 = 11

Addition:

The algorithm is the same for SIGNED and UNSIGNED.

8-bit Unsigned:

```
     1110 1100   = 236
+    1010 1010   = 170
---------------------
   1 1001 0110   = 406
```

8-bit Signed:

```
     1110 1100   = -20
+    1010 1010   = -86
---------------------
   1 1001 0110   = -106
```

Addition:

The algorithm is the same for SIGNED and UNSIGNED.
Overflow detection is slightly different.

8-bit Unsigned:

```
     1110 1100   = 236
+    1010 1010   = 170
   1 1001 0110   = 406
```

Overflow! (max value = 255)

8-bit Signed:

```
     1110 1100   = -20
+    1010 1010   = -86
   1 1001 0110   = -106
```

Addition:

The algorithm is the same for SIGNED and UNSIGNED.
Overflow detection is slightly different.

8-bit Unsigned:

```
     1110 1100   = 236
+    1010 1010   = 170
   1 1001 0110   = 406
```

Overflow! (max value = 255)

8-bit Signed:

```
     1110 1100   = -20
+    1010 1010   = -86
   1 1001 0110   = -106
```

Subtraction:

The algorithm is the same for SIGNED and UNSIGNED.
Overflow detection is slightly different.

Addition:

The algorithm is the same for SIGNED and UNSIGNED.
Overflow detection is slightly different.

8-bit Unsigned:

```
     1110 1100   = 236
+    1010 1010   = 170
   1 1001 0110   = 406
```

Overflow! (max value = 255)

8-bit Signed:

```
     1110 1100   = -20
+    1010 1010   = -86
   1 1001 0110   = -106
```

Subtraction:

The algorithm is the same for SIGNED and UNSIGNED.
Overflow detection is slightly different.

Multiplication:

Two algorithms.

8-bit Signed:

```
               1111 1110   = -2
      x        1111 1110   = -2
 0000 0000 0000 0100       = +4
```

8-bit Unsigned:

```
               1111 1110   = 254
      x        1111 1110   = 254
 1111 1100 0000 0100       = 64,516
```

(NOTE: Result may be twice as long as operands.)

Addition:

The algorithm is the same for SIGNED and UNSIGNED.
Overflow detection is slightly different.

8-bit Unsigned:

```
      1110 1100   = 236
+     1010 1010   = 170
    1 1001 0110   = 406
```

8-bit Signed:

```
      1110 1100   = -20
+     1010 1010   = -86
    1 1001 0110   = -106
```

Overflow! (max value = 255)

Subtraction:

The algorithm is the same for SIGNED and UNSIGNED.
Overflow detection is slightly different.

Multiplication:

Two algorithms.

8-bit Signed:

```
               1111 1110   = -2
         x     1111 1110   = -2
0000 0000 0000 0100   = +4
```

8-bit Unsigned:

```
               1111 1110   = 254
         x     1111 1110   = 254
1111 1100 0000 0100   = 64,516
```

(NOTE: Result may be twice as long as operands.)

Division:

Two algorithms.

Arithmetic Negation

The Algorithm to Negate a Signed Number:

Bitwise complement (i.e., logical NOT)
Followed by "add 1"

Example:

```
                     0000 0010   =  2
complementing:
add 1:
```

Arithmetic Negation

The Algorithm to Negate a Signed Number:

Bitwise complement (i.e., logical NOT)
Followed by "add 1"

Example:

```
                       0000 0010   =  2
complementing:        1111 1101
add 1:
```

Arithmetic Negation

The Algorithm to Negate a Signed Number:

Bitwise complement (i.e., logical NOT)
Followed by "add 1"

Example:

```
                   0000 0010   =  2
complementing:     1111 1101
add 1:            +0000 0001
```

Arithmetic Negation

The Algorithm to Negate a Signed Number:

Bitwise complement (i.e., logical NOT)
Followed by "add 1"

Example:

```
                    0000 0010   =  2
complementing:      1111 1101
add 1:             +0000 0001
                    1111 1110   = -2
```

Arithmetic Negation

The Algorithm to Negate a Signed Number:

Bitwise complement (i.e., logical NOT)
Followed by "add 1"

Example:

```
                     0000 0010   =  2
complementing:       1111 1101
add 1:              +0000 0001
                     1111 1110   = -2
```

Arithmetic negation can overflow!
Every signed number can be negated,
... except the most negative number.

Arithmetic Negation

The Algorithm to Negate a Signed Number:

Bitwise complement (i.e., logical NOT)
Followed by "add 1"

Example:

```
                     0000 0010    =  2
   complementing:    1111 1101
   add 1:           +0000 0001
                     1111 1110    = -2
```

Arithmetic negation can overflow!
Every signed number can be negated,
... except the most negative number.

8-Bit Example:

```
                     1000 0000    =  -128
   complementing:
   add 1:
```

Arithmetic Negation

The Algorithm to Negate a Signed Number:

Bitwise complement (i.e., logical NOT)
Followed by "add 1"

Example:

```
                   0000 0010   =  2
complementing:     1111 1101
add 1:            +0000 0001
                   1111 1110   = -2
```

Arithmetic negation can overflow!
Every signed number can be negated,
... except the most negative number.

8-Bit Example:

```
                   1000 0000   =  -128
complementing:     0111 1111
add 1:
```

Arithmetic Negation

The Algorithm to Negate a Signed Number:

Bitwise complement (i.e., logical NOT)
Followed by “add 1”

Example:

```
                     0000 0010   =  2
complementing:       1111 1101
add 1:              +0000 0001
                     1111 1110   = -2
```

Arithmetic negation can overflow!
Every signed number can be negated,
... except the most negative number.

8-Bit Example:

```
                     1000 0000   =  -128
complementing:       0111 1111
add 1:              +0000 0001
```

Arithmetic Negation

The Algorithm to Negate a Signed Number:

Bitwise complement (i.e., logical NOT)
Followed by "add 1"

Example:

```
                    0000 0010   =  2
complementing:      1111 1101
add 1:             +0000 0001
                    1111 1110   = -2
```

Arithmetic negation can overflow!
Every signed number can be negated,
... except the most negative number.

8-Bit Example:

```
                    1000 0000   =  -128
complementing:      0111 1111
add 1:             +0000 0001
                    1000 0000   =  -128
```

Arithmetic Negation

The Algorithm to Negate a Signed Number:

Bitwise complement (i.e., logical NOT)
Followed by "add 1"

Example:

```
                      0000 0010   =  2
complementing:        1111 1101
add 1:               +0000 0001
                      1111 1110   = -2
```

Arithmetic negation can overflow!
Every signed number can be negated,
... except the most negative number.

8-Bit Example:

```
                      1000 0000   =  -128
complementing:        0111 1111
add 1:               +0000 0001
                      1000 0000   =  -128
```

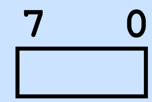
The most negative 32-bit number, 0x80000000

Hex: `8    0    0    0       0    0    0    0`
Binary: `1000 0000 0000 0000    0000 0000 0000 0000`
Decimal: `-2,147,483,648`

Storing Numbers In Memory

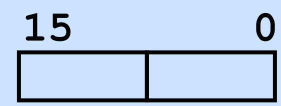
Byte

8 bits



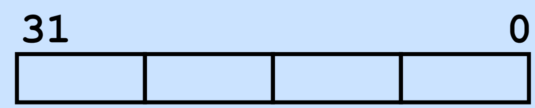
Halfword

16 bits = 2 bytes



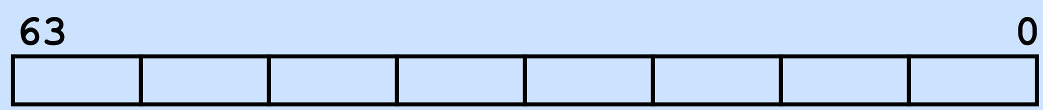
Word

32 bits = 4 bytes



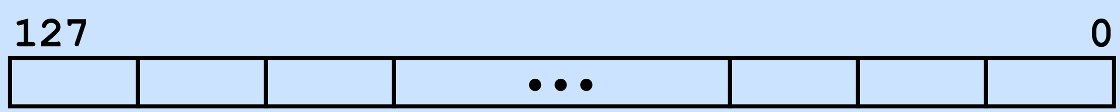
Doubleword

64 bits = 8 bytes



Quadword

128 bits = 16 bytes



Byte Ordering

Big Endian

Sparc (Sun), PowerPC

Little Endian

Intel (Macs, PCs)

Example: 32-bit Integer Value

`0x01234567`

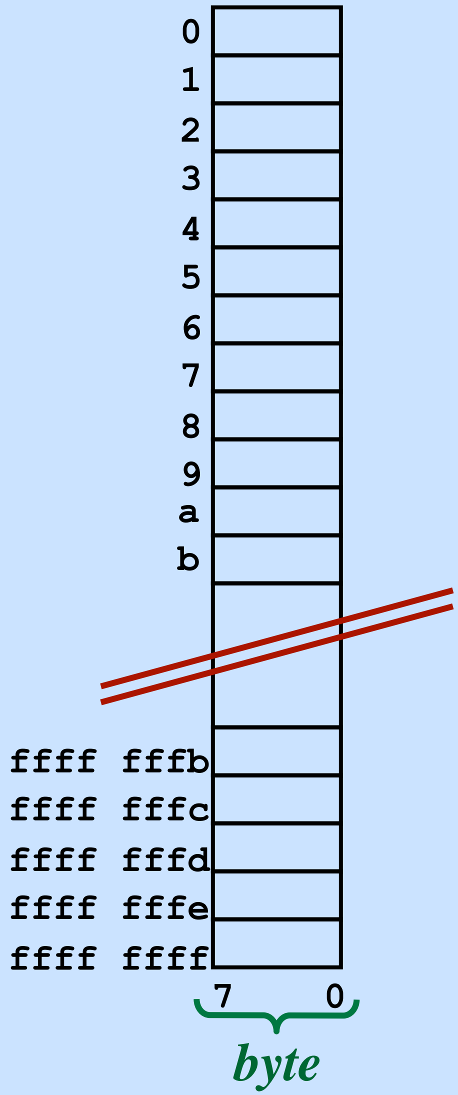
Big Endian:

	100	101	102	103	104	105	106	107	
• • •			01	23	45	67			• • •

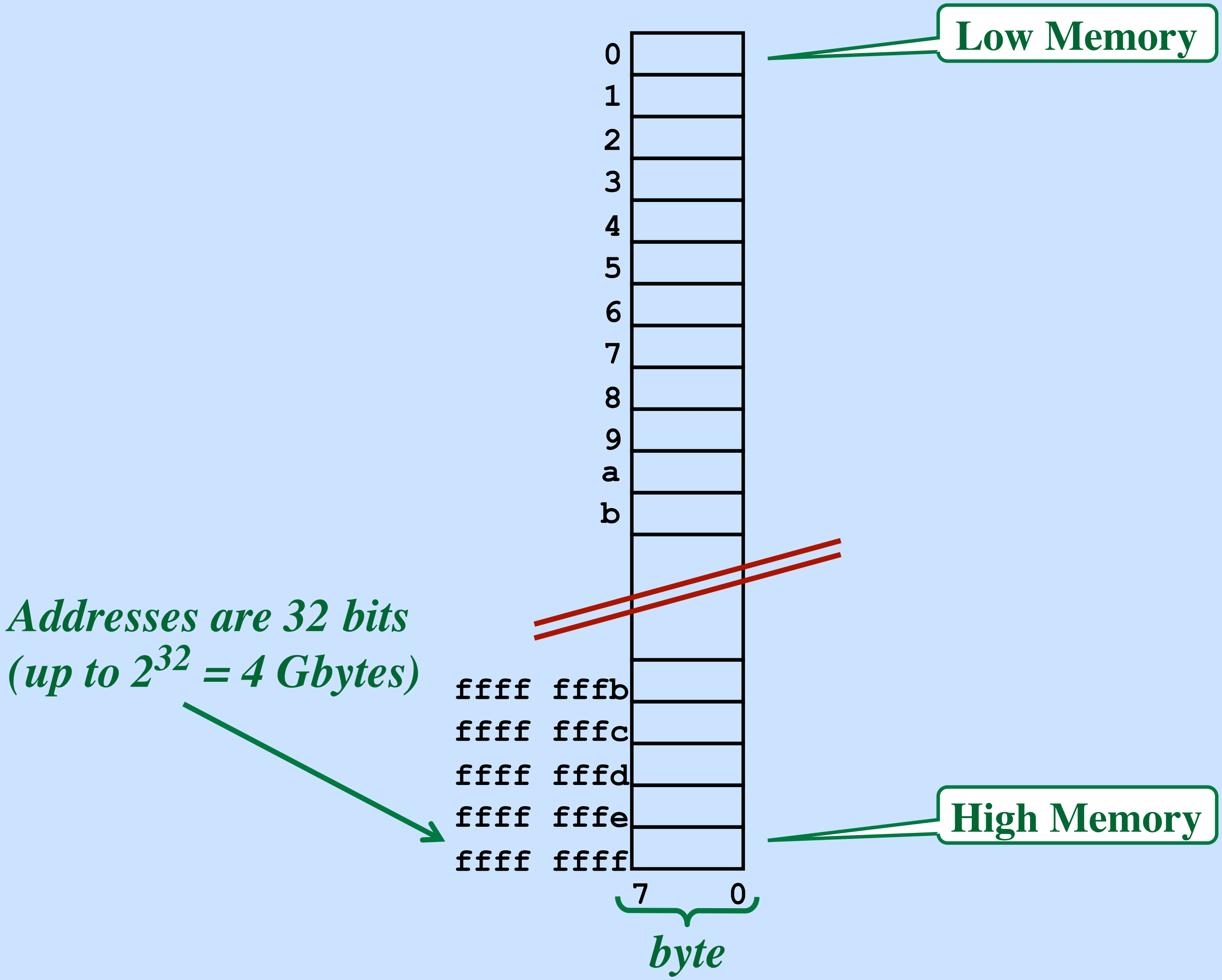
Little Endian:

	100	101	102	103	104	105	106	107	
• • •			67	45	23	01			• • •

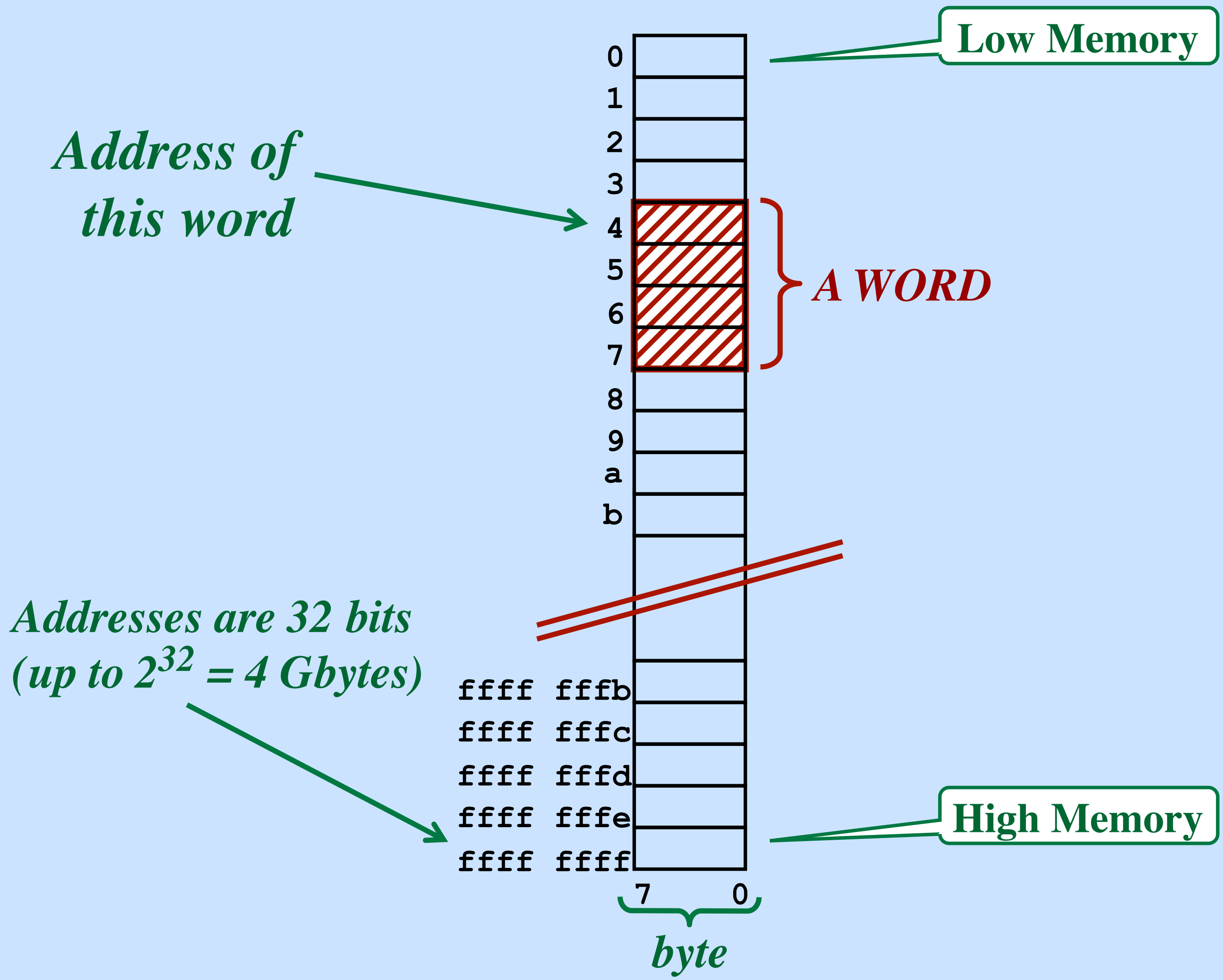
Main Memory Organization



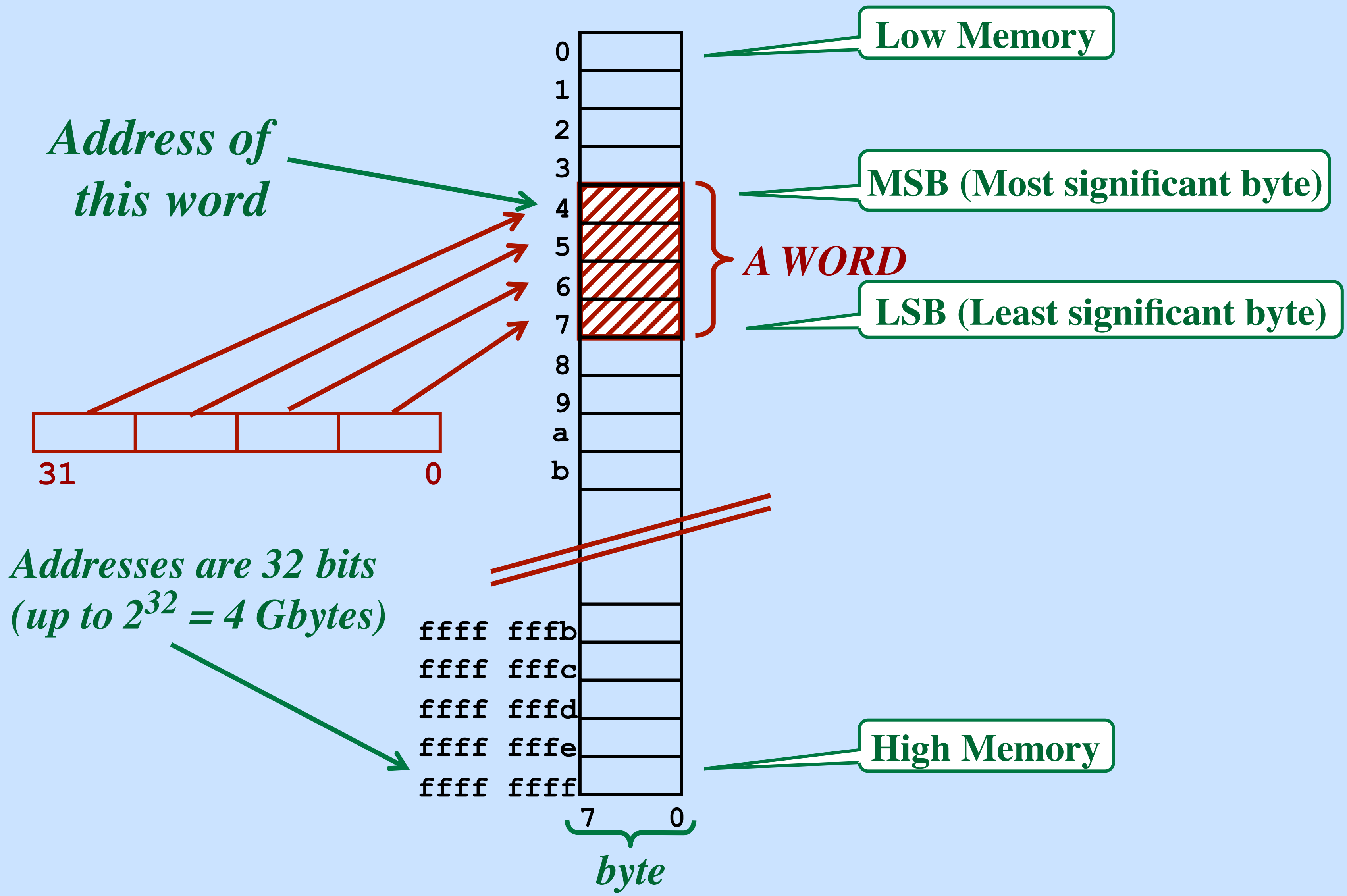
Main Memory Organization



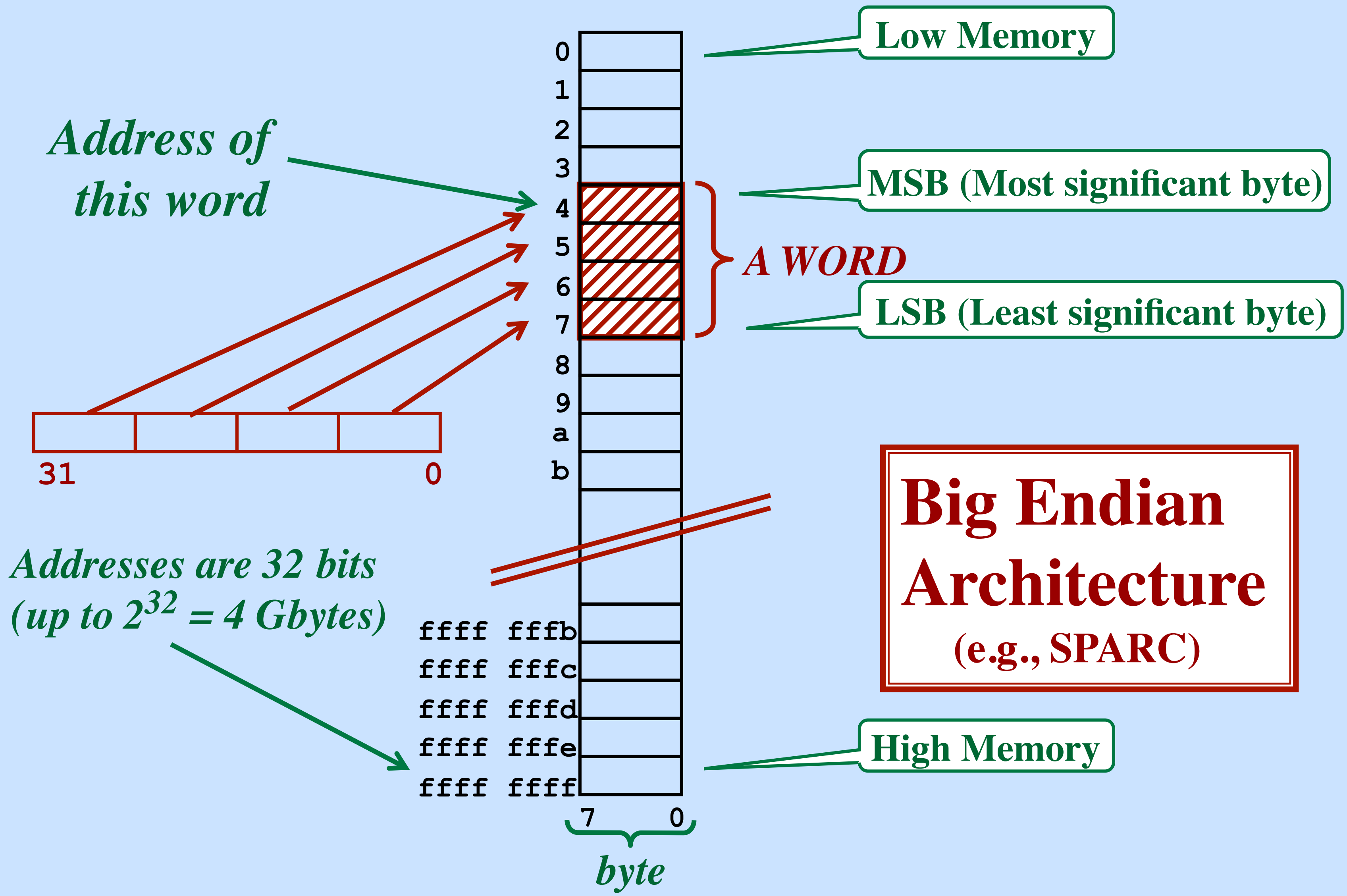
Main Memory Organization



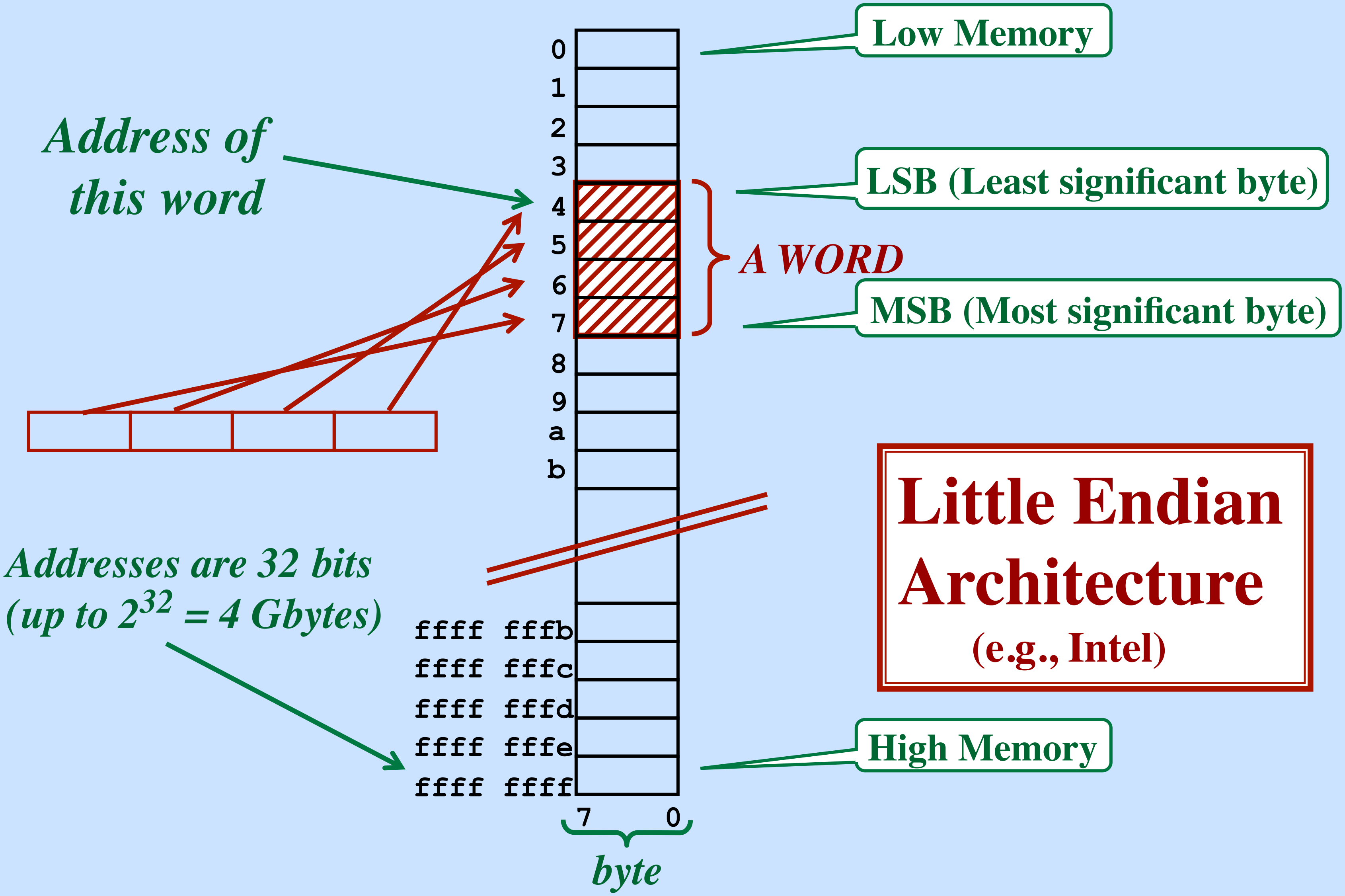
Main Memory Organization



Main Memory Organization



Main Memory Organization



Little Endian Architecture

(e.g., Intel)

Program to Show Byte Order

```
#include <stdio.h>
#include <string.h>
void show_bytes(unsigned char * start, int len) {
        int i;
        for (i = 0; i < len; i++)
                printf(" %2.2x", start[i]);
        printf("\n");
}

int main() {
        int i=0x01020304;
        float f=123.456;
        int *ip=&i;
        char *s = "ABCDEF";

        show_bytes ((char *) &i, sizeof(int));
        show_bytes ((char *) &f, sizeof(float));
        show_bytes ((char *) &ip, sizeof(char *));
        show_bytes (s, strlen(s));
}
```

Output (Sun, Big Endian):

```
% a.out
 01 02 03 04
 42 f6 e9 79
 ff bf f8 cc
 41 42 43 44 45 46
```

Program to Show Byte Order

```
#include <stdio.h>
#include <string.h>
void show_bytes(unsigned char * start, int len) {
        int i;
        for (i = 0; i < len; i++)
                printf(" %2.2x", start[i]);
        printf("\n");
}

int main() {
        int i=0x01020304;
        float f=123.456;
        int *ip=&i;
        char *s = "ABCDEF";

        show_bytes ((char *) &i, sizeof(int));
        show_bytes ((char *) &f, sizeof(float));
        show_bytes ((char *) &ip, sizeof(char *));
        show_bytes (s, strlen(s));
}
```

Output (Mac, Little Endian):

```
% a.out
 04 03 02 01
 79 e9 f6 42
 cc db 82 50 ff 7f 00 00
 41 42 43 44 45 46
```

Data Alignment

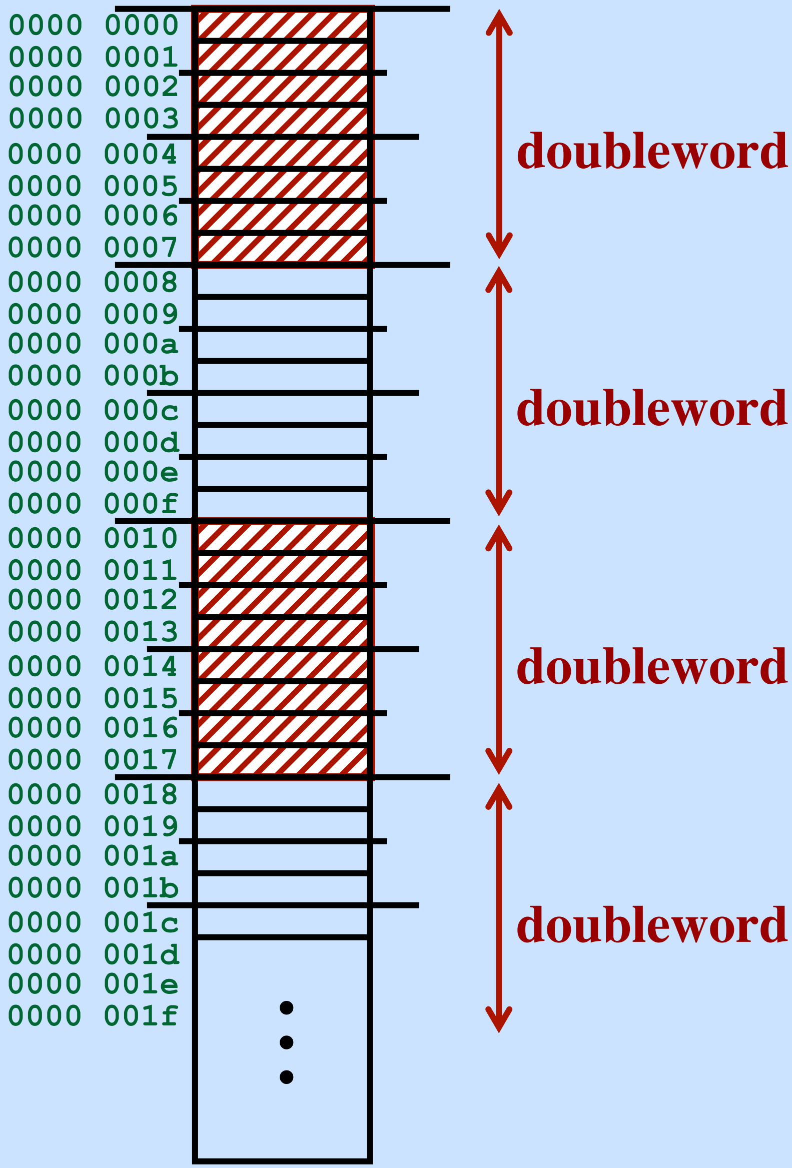
Data stored in memory must be “aligned”
according to the length of the data

Byte Data
can go at any address

Halfword Data
must be “halfword aligned”
addresses must be even numbers

Word Data
must be “word aligned”
addresses must be divisible by 4

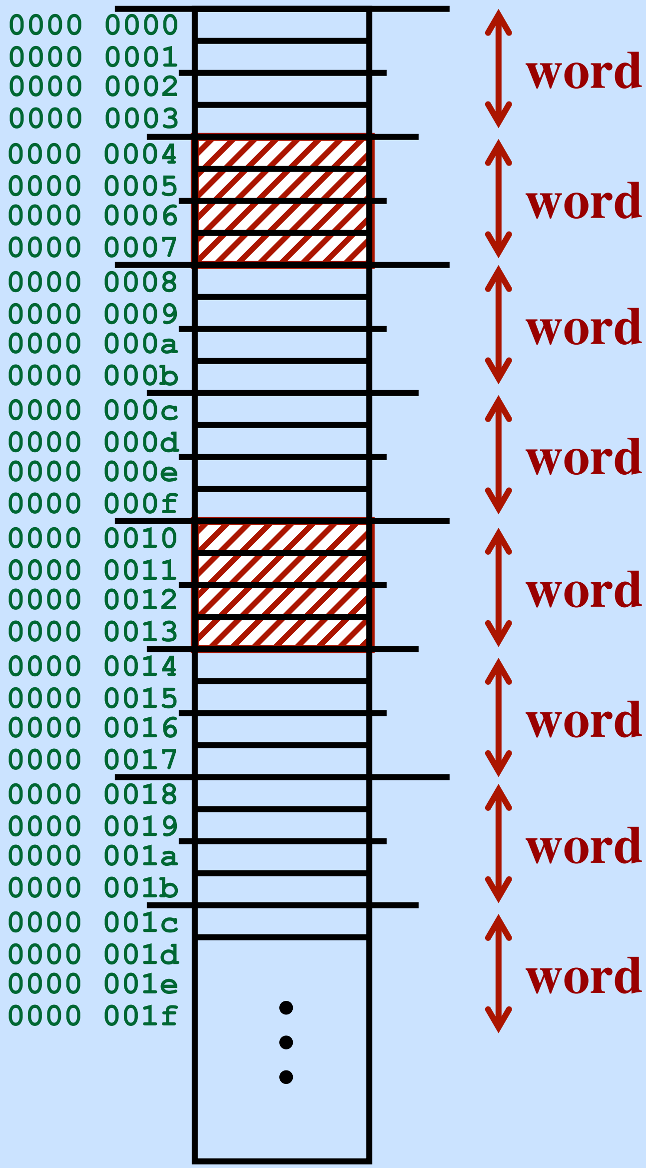
Doubleword Data
must be “doubleword aligned”
addresses must be divisible by 8



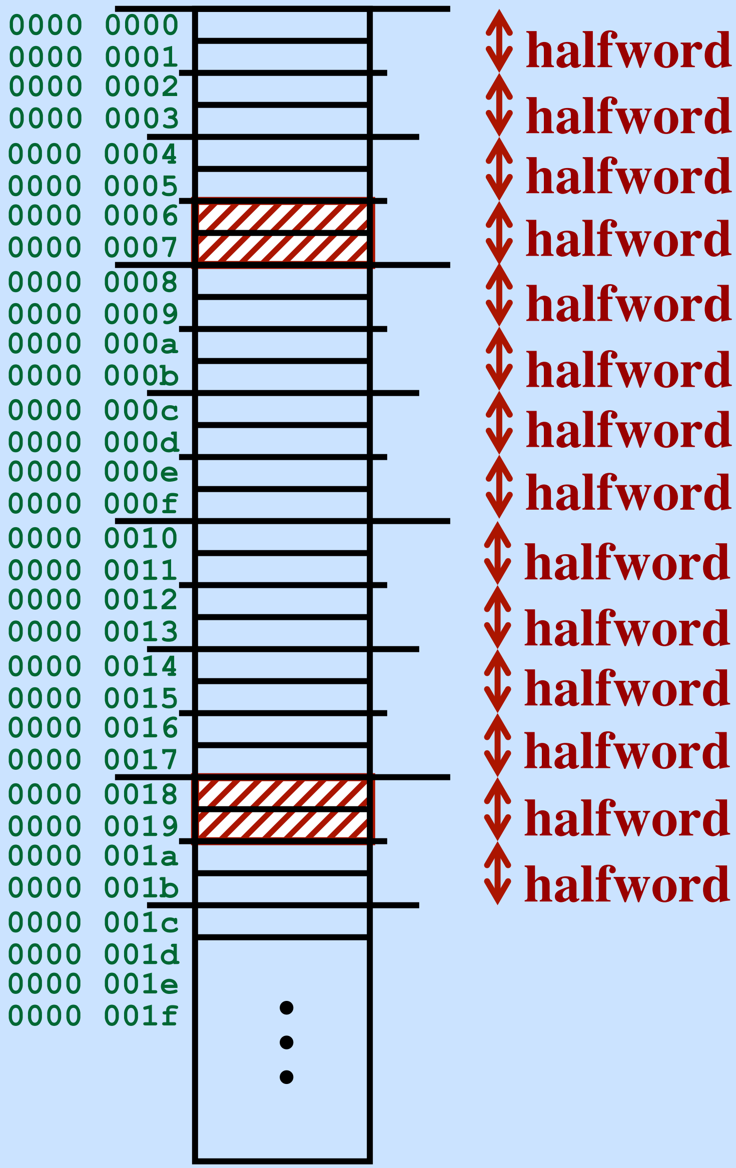
Doubleword addresses end in a number divisible by 8:
0, 8

In binary:

---- ---- ---- ---- ---- ---- ---- -000



Word addresses end in a number divisible by 4:
0, 4, 8, c
In binary:
---- ---- ---- ---- ---- ---- ---- --00



Halfword addresses end in an even number:
0, 2, 4, 6, 8, a, c, e
In binary:
---- ---- ---- ---- ---- ---- ---- ---0

Fixed-Point Numbers

Decimal

123.456

... 10^2 10^1 10^0 10^{-1} 10^{-2} 10^{-3} ...

... 100 10 1 $^1/10$ $^1/100$ $^1/1000$ ...

Binary

101.0101

... 2^2 2^1 2^0 2^{-1} 2^{-2} 2^{-3} 2^{-4} ...

... 4 2 1 $^1/_2$ $^1/_4$ $^1/_8$ $^1/_{16}$ ...

What is this number? $4 + 1 + {}^1/_4 + {}^1/_{16} = 5\ {}^5/_{16} = 5.3125$

“Every binary fraction can be represented exactly with a decimal fraction.”

$1001.01_{\mathbf{2}} = 9.25_{\mathbf{10}}$

(And the decimal representation will use no more decimal digits to the right of “.” than the binary number has bits.)

“Some decimal fractions cannot be represented exactly using binary fractions.”

$0.3_{\mathbf{10}} = 0.01\underbrace{0011}\underbrace{0011}\underbrace{0011}\underbrace{0011}\underbrace{0011}\ldots_{\mathbf{2}}$

$= 0.01\overline{0011}_{\mathbf{2}}$

(of finite length)

Floating Point Numbers

Decimal

```
  123.456
= 1.2345 × 10^2
  6.0225 × 10^23
```

$$123.456 = 1.2345 \times 10^{2}$$
$$6.0225 \times 10^{23}$$

Limited precision: Rounded to the nearest $\mathbf{10^{19}}$

The leading digit will always be 1,2,3, ..., 9 (never 0).

Floating Point Numbers

Decimal

```
  123.456
= 1.2345 × 10^2
  6.0225 × 10^23
```

$$123.456 = 1.2345 \times 10^{2}$$

$$6.0225 \times 10^{23}$$

Limited precision: Rounded to the nearest $\mathbf{10^{19}}$
The leading digit will always be 1,2,3, ..., 9 (never 0).

Binary

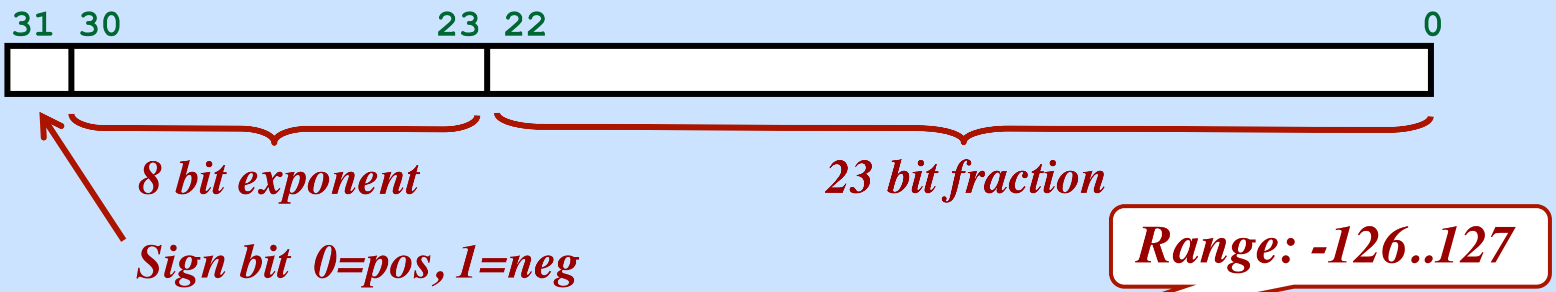
$$101.0101$$

$$= 1.010101 \times 2^{2}$$

$$= 1.328125 \times 4 = 5.3125$$

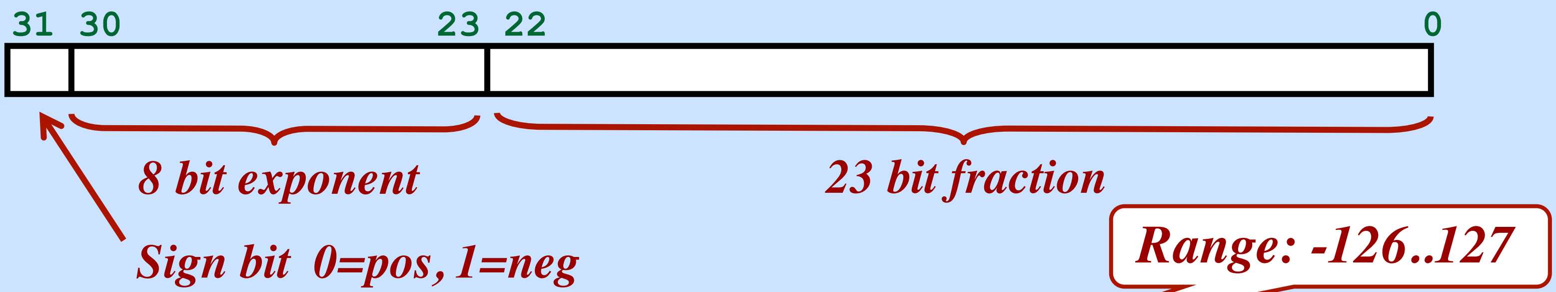
Note: The leading bit will always be "1" (never "0").
No need to store the first bit!

Single Precision Floating Point Number Representation



$$N = (-1)^{\text{sign}} \times 1.\text{fraction} \times 2^{\text{exp}}$$

Single Precision Floating Point Number Representation

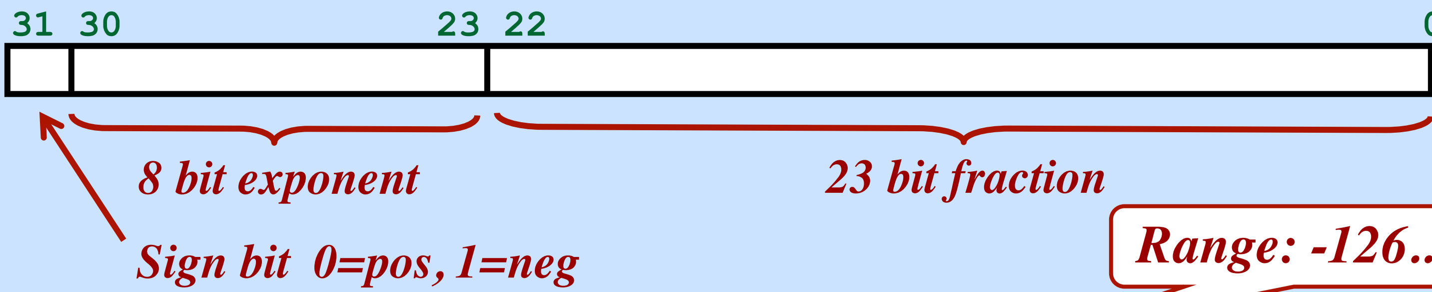


$$N = (-1)^{sign} \times 1.fraction \times 2^{exp}$$

The exponent is an 8 bit number interpreted as follows...

0000 0000	"sub-normal"
0000 0001	-126
...	...
0111 1110	-1
0111 1111	0
1000 0000	1
...	...
1111 1110	127
1111 1111	"not a number"

Single Precision Floating Point Number Representation



Range: -126..127

$$\mathbf{N} = (-1)^{\text{sign}} \times 1.\text{fraction} \times 2^{\text{exp}}$$

The exponent is an 8 bit number interpreted as follows...

0000 0000	"sub-normal"
0000 0001	-126
...	...
0111 1110	-1
0111 1111	0
1000 0000	1
...	...
1111 1110	127
1111 1111	"not a number"

Single-Precision

Smallest non-zero number:

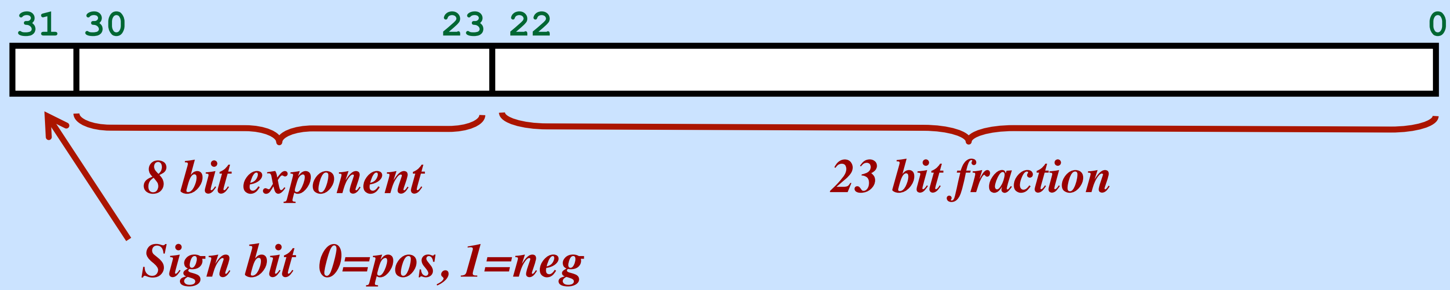
$1.17549435 \times 10^{-38}$

Largest number:

$3.40282347 \times 10^{+38}$

About 9 digits of accuracy!

Zero



Sign = `0` (positive)
`1` (negative)

Exp = `00000000`

Fraction = `00000000000000000000000`

`0x0000 00000` (= positive zero)
`0x8000 00000` (= negative zero)

Similar for double precision.

Other Special Cases

When

exp = `1111 1111`

a special meaning arises

Not A Number "**NaN**"

`0xFFFF FFFF`

(= -1 as a signed number)

Will cause an exception when used.

Positive Infinity "**+inf**"

$+\infty$

`0x7F80 0000`

Negative Infinity "**-inf**"

$-\infty$

`0xFF80 0000`

Divide $^{1}/_{0} \Rightarrow +\infty$

Divide $-^{1}/_{0} \Rightarrow -\infty$

You can compare other numbers to $+\infty$ and $-\infty$.

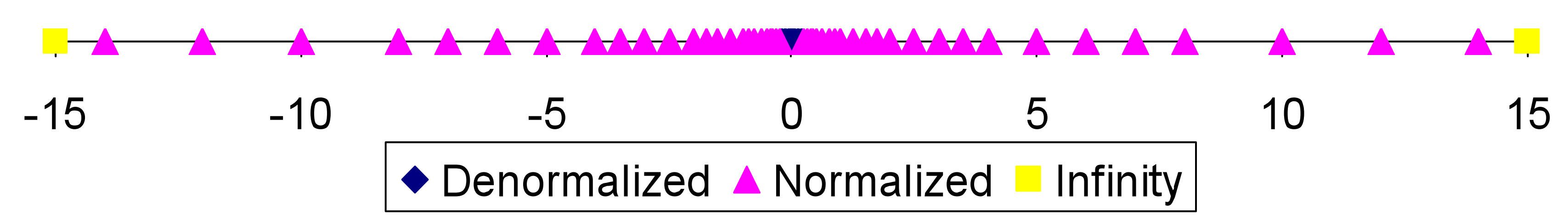
Distribution of Numbers

Example: 6-bit floating point numbers

Exponent: 3 bits

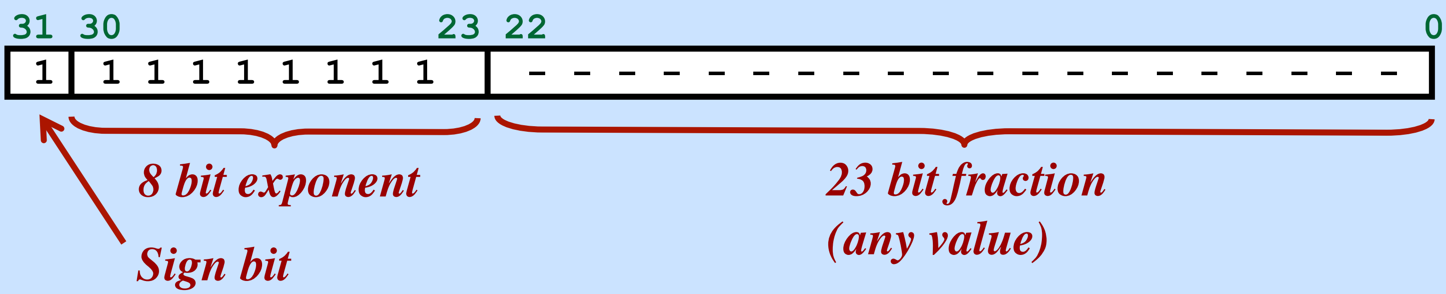
Fraction: 2 bits

Notice how the density is greater close to zero:



NaN Details

Representation:



The fraction is ignored.

Any value

from

`FF80 0000`

to

`FFFF FFFF`

Indicates an “invalid result.”

`0 ÷ 0  →  NaN`

Operands preserve Nan

`3.75 + NaN  →  NaN`

Denormalized Numbers

31	30 … 23	22 … 0
-	0 0 0 0 0 0 0 0	- - - - - - - - - - - - - - - - - - - -
Sign bit 0=pos, 1=neg	*8 bit exponent*	*23 bit fraction*

Sign bit   0=pos, 1=neg

Normalized:

$$N = (-1)^{\text{sign}} \times 1.\text{fraction} \times 2^{\text{exp}}$$

Range: -126..127

smallest value $1.000\ldots000 \times 2^{-126}$

Precision: 24 bits

Denormalized:

$$N = (-1)^{\text{sign}} \times 0.\text{fraction} \times 2^{-126}$$

largest value $0.111\ldots111 \times 2^{-126}$

Precision: 23 bits

smallest value $0.000\ldots001 \times 2^{-126}$

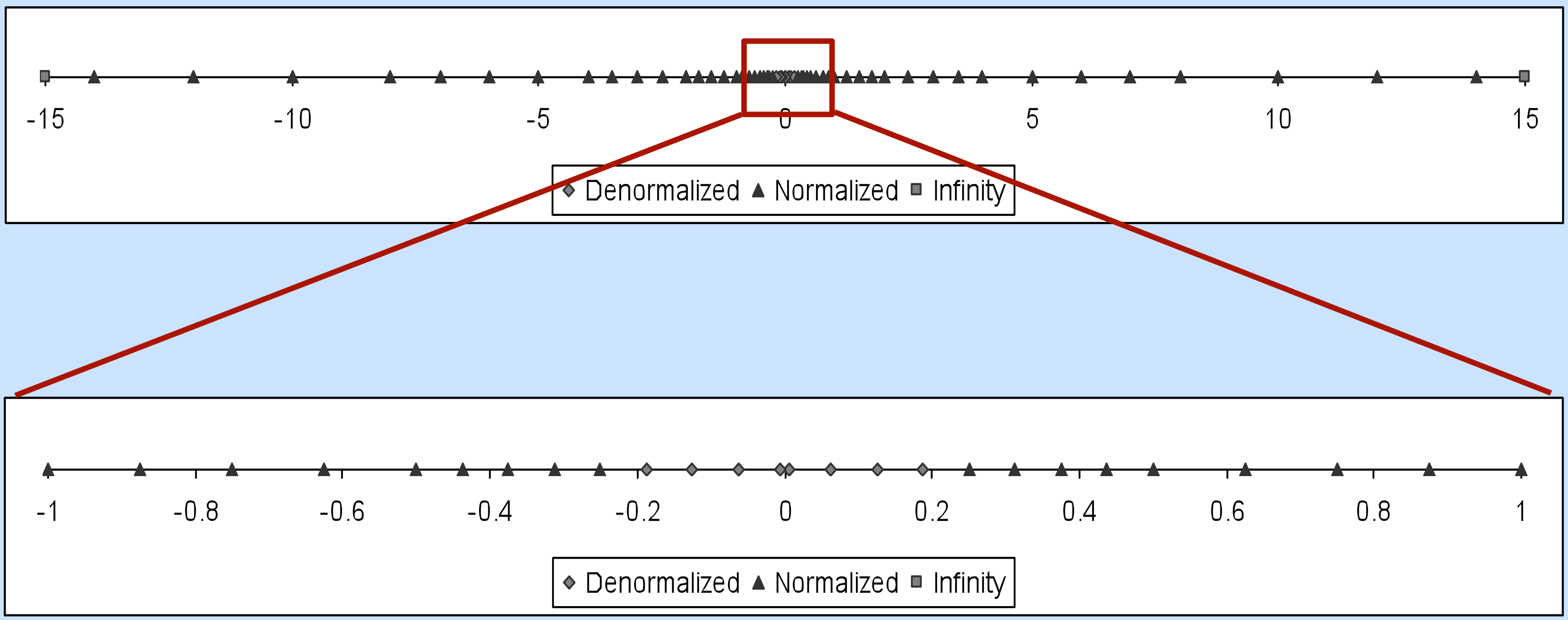
Precision: 1 bit

positive zero $0.000\ldots000 \times 2^{-126}$

Denormalized Numbers

Denormalized values are very close to zero.
They have reduced precision.
+0.0 and -0.0 are special cases of denormalized numbers.

Example (using floats with only 6 bits)



Floating Point Properties

Addition

Commutative: $x+y = y+x$

Not associative: $(x + y) + z \neq x + (y + z)$

due to rounding

Example:

$(3.14 + 1e10) - 1e10 = 0.0$, due to rounding

$3.14 + (1e10 - 1e10) = 3.14$

Multiplication

Not associative

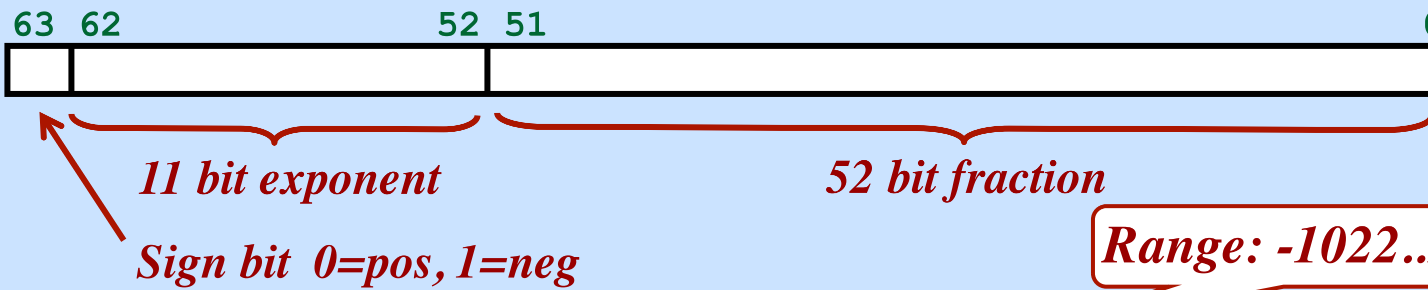
Multiplication does not distribute over addition

Example:

$1e20 \times (1e20 - 1e20) = 0.0$

$(1e20 \times 1e20) - (1e20 \times 1e20) = \text{NaN}$

Double Precision Floating Point Number Representation



Range: -1022..1023

$$N = (-1)^{sign} \times 1.fraction \times 2^{exp}$$

Double-Precision

Smallest non-zero number:

`2.225,073,858,507,201,4` $\times 10^{-308}$

Largest number:

`1.797,693,134,862,315,7` $\times 10^{+308}$

About 17 digits of accuracy!

Logical Functions

```
and
or
xor    =   (x ≠ y)
```

x	y	and	or	xor
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Instructions work on all 32 bits at once:

```
andl   %ecx,%eax

%eax ——> 0011 1100 ... 1010
%ecx ——> 1010 1101 ... 1001
         ------------------
%eax ——> 0010 1100 ... 1000
```

Logical Functions in C

Operate on integer data

char, int, short

Each operand is a bit vector

And

x = y & z;

```
1010 1100 0110 0010
0101 0111 0101 1010
-------------------
0000 0100 0100 0010
```

Or

x = y | z;

```
1010 1100 0110 0010
0101 0111 0101 1010
-------------------
1111 1111 0111 1010
```

Exclusive-Or

x = y ^ z;

```
1010 1100 0110 0010
0101 0111 0101 1010
-------------------
1111 1011 0011 1000
```

To turn on bits in a word...

Use the "**or**" instruction and a "mask" word

x or mask → result

Turn on bits in x wherever the mask has a 1 bit

Example: Turn on every other bit in 3A0F

```
0011 1010 0000 1111  <----  3A0F
0101 0101 0101 0101  <----  mask
-------------------
0111 1111 0101 1111  <----  result
```

To turn off bits in a word...

Use the "**and**" instruction and a mask

x and mask → result

Turn off bits in x wherever the mask has a 0 bit

To flip (or "toggle") bits in a word...

Use the "**xor**" instruction and a mask

x xor mask → result

Change the bits in x wherever the mask has a 1 bit

Shifting Instructions

shl

“Shift Left” <<

Number of bits 0..31

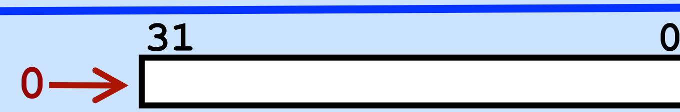
```
shl    int,reg
```

A fast way to multiply by 2^N...

Example: Multiply by 32 (= 2^5)

```
shl    $5,%eax
          0000 0000 0000 0011   = 3
          0000 0000 0110 0000   = 64+32 = 96
```

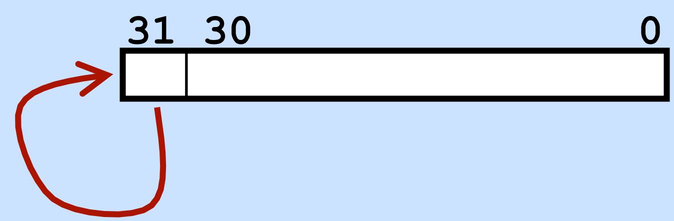
shr



“Shift Logical Right” >>

```
shr      int,reg
```

sar



“Shift Arithmetic Right” >>

```
sar      int,reg
```

A fast way to divide by 2^N, rounding toward $-\infty$...

```
sar      $3,%eax
```

Testing

cmp reg1,reg2

Compare operand1 to operand2

Set integer condition codes accordingly

The next instruction will normally be a conditional branch

Example:

```
cmp   %g3,73     ! if x <= 73 goto loop
ble   loop       ! .
```

Branch if the condition codes indicate $op1 \le op2$

Pointers

A pointer is a memory address.

Pointers are "typed"

…the type of the object at that address

Pointers are typed in order to determine
what gets returned when the pointer is dereferenced.

Use "*" to declare a pointer type

```
char* cp;   // cp points to a character in memory
int* ip;    // ip points to an integer in memory
```

"&" operator gives address of object

```
int x;
int* p;
```

What is the data type of p?
What is the data type of *p?
What is the data type of &x?

Pointers

Given the following code…

```
main() {
    int B = -15213;
    int* P = &B;
}
```

Suppose

The address of `B` is `0xbffff8d4`

The address of `P` is `0xbffff8d0`

What is the value of P?

What is the size of P?

Write the value of each byte of P in order as they appear in memory.

Pointers

Given the following code…

```
main() {
     int B = -15213;
     int* P = &B;
}
```

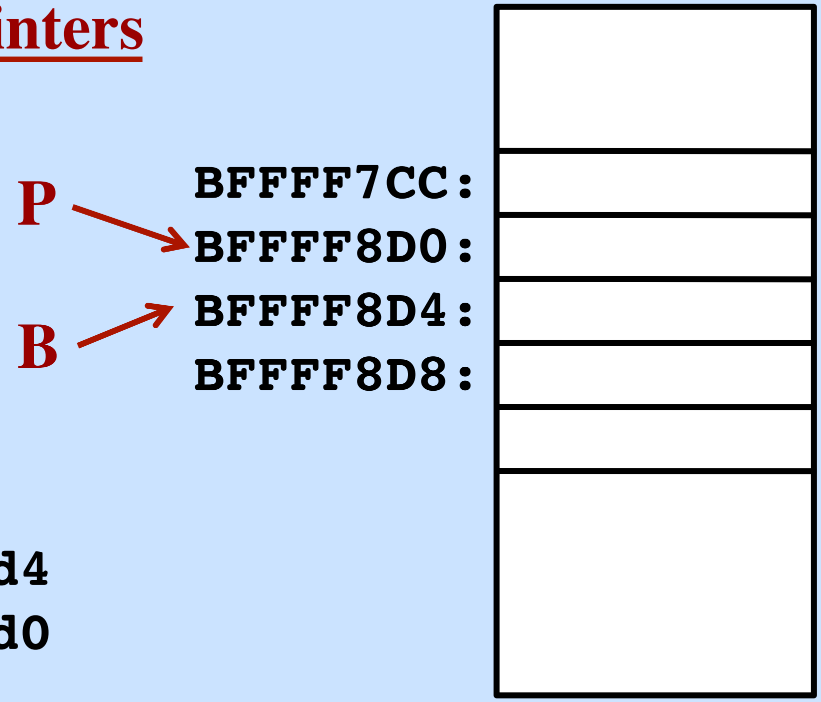
Suppose
- The address of `B` is `0xbffff8d4`
- The address of `P` is `0xbffff8d0`

What is the value of P?
What is the size of P?
Write the value of each byte of P in order as they appear in memory.

Pointers

Given the following code…

```
main() {
      int B = -15213;
      int* P = &B;
}
```



Suppose

The address of `B` is `0xbffff8d4`

The address of `P` is `0xbffff8d0`

What is the value of P?

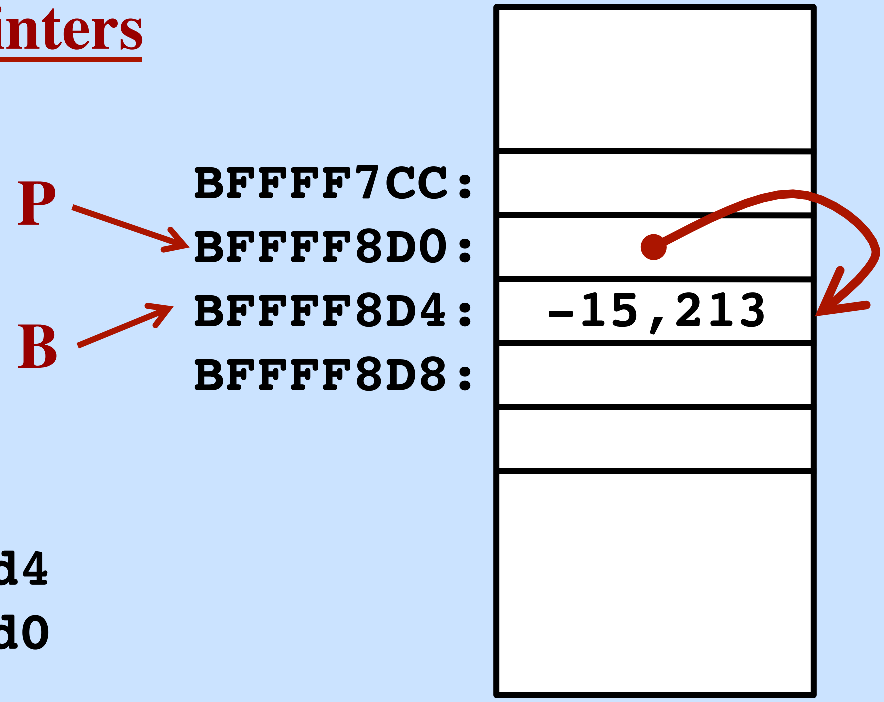
What is the size of P?

Write the value of each byte of P in order as they appear in memory.

Pointers

Given the following code…

```
main() {
     int B = -15213;
     int* P = &B;
}
```



Suppose

The address of `B` is `0xbffff8d4`

The address of `P` is `0xbffff8d0`

What is the value of P?

What is the size of P?

Write the value of each byte of P in order as they appear in memory.

Pointers

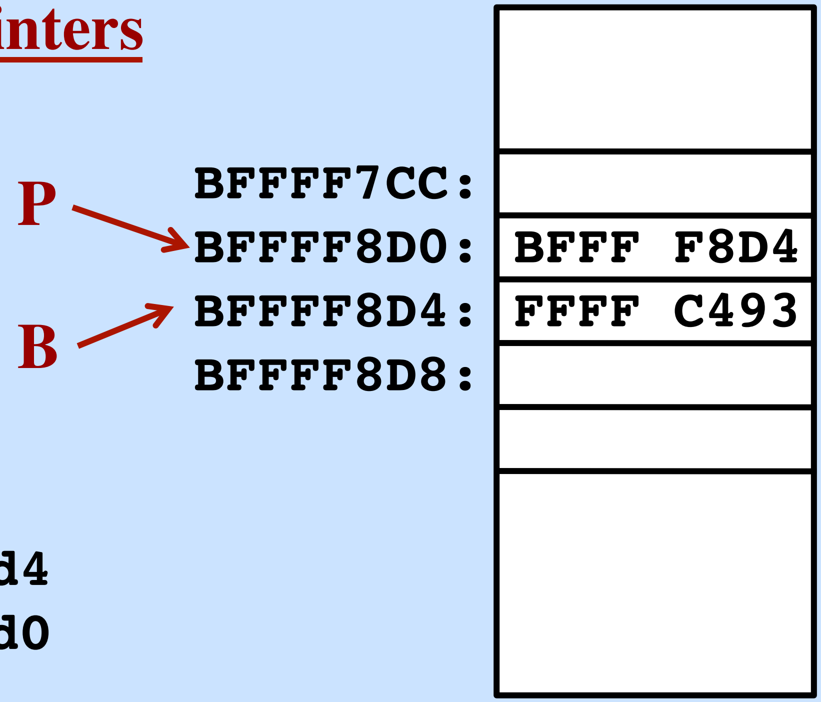
Given the following code…

```
main() {
    int B = -15213;
    int* P = &B;
}
```

Suppose

The address of `B` is `0xbffff8d4`

The address of `P` is `0xbffff8d0`



Big Endian

What is the value of P?

What is the size of P?

Write the value of each byte of P in order as they appear in memory.

Pointers

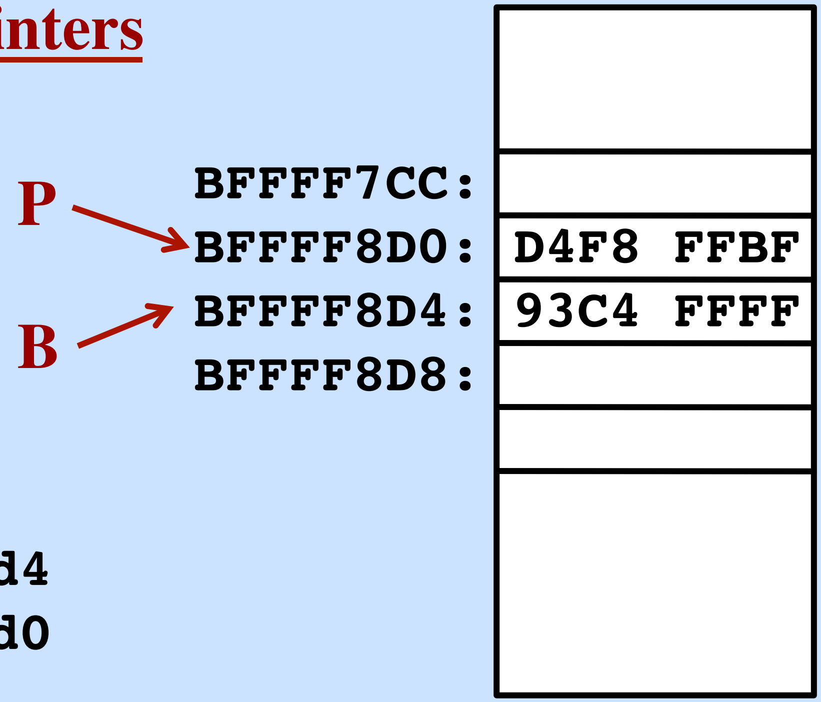
Given the following code…

```
main() {
     int B = -15213;
     int* P = &B;
}
```

Suppose

The address of `B` is `0xbffff8d4`

The address of `P` is `0xbffff8d0`



Little Endian

What is the value of P?

What is the size of P?

Write the value of each byte of P in order as they appear in memory.

Pointer Assignment / Dereference

Dereferencing pointers

Returns the data that is stored in the memory location

The unary operator *

Used to dereference a pointer variable

```
int x = 1, y = 2, z[10];
int* ip = &x;
y = *ip;       // y is now 1
*ip = 0;       // x is now 0
```

Dereferencing uninitialized pointers: What happens?

```
int* ip;
*ip = 3;
```

Segmentation fault   (or worse: nothing so obvious!)

Pointer Arithmetic

Type determines what is returned when "dereferenced"
Also: pointer arithmetic is based on the type of data referenced.

> Incrementing an `int *` adds 4 to the pointer.
> Incrementing a `char *` adds 1 to the pointer.
> Incrementing a `int * *` adds 4 or 8 to the pointer.

Example:

```
char*   cp=0x100;
int*    ip=0x200;
float*  fp=0x300;
double* dp=0x400;
int     i=0x500;
```

What are the hexadecimal values of each after each of these commands?

```
cp++;
ip++;
fp++;
dp++;
i++;
```

Pointers and Arrays

Arrays are stored in one contiguous block of memory.

An array is a collection of values

All the same type

Indexed (or “accessed”) by number

Example

```
int a[20];
```

The first element is

```
a[0]
```

The last element is

```
a[19]
```

The variable “a” is a pointer to int.

Similar to:

```
int * a;
```

```
i = *a;              i = a[0];
j = *(a+3);          j = a[3];
b = a+3;
```

Really adds 12

Example

```
#include <stdio.h>
main() {
  char* str="abcdefg\n";
  char* x;
  x = str;

  printf("str[0]: %c\n", str[0]);
  printf("str[1]: %c\n", str[1]);
  printf("str[2]: %c\n", str[2]);
  printf("str[3]: %c\n", str[3]);

  printf("x: %x  *x: %c\n", x, *x);   x++;
  printf("x: %x  *x: %c\n", x, *x);   x++;
  printf("x: %x  *x: %c\n", x, *x);   x++;
  printf("x: %x  *x: %c\n", x, *x);

}
```

```
str[0]: a
str[1]: b
str[2]: c
str[3]: d
x: 8054f4a  *x: a
x: 8054f4b  *x: b
x: 8054f4c  *x: c
x: 8054f4d  *x: d
```

```
#include <stdio.h>
main() {
  int numbers[10], *num, i;

  for (i=0; i < 10; i++)
        numbers[i]=i;

  num = (int *) numbers;
  printf("num: %x  *num: %d\n", num, *num);   num++;
  printf("num: %x  *num: %d\n", num, *num);   num++;
  printf("num: %x  *num: %d\n", num, *num);   num++;
  printf("num: %x  *num: %d\n", num, *num);

  num = (int *) numbers;
  printf("numbers=%x      num=%x\n", numbers, num);
  printf("&numbers[7]=%x  num+7=%x\n", &numbers[7], num+7);
  printf("numbers[7]=%d   *(num+7)=%d\n", numbers[7], *(num+7));
}
```

```
num: 5833fba0  *num: 0
num: 5833fba4  *num: 1
num: 5833fba8  *num: 2
num: 5833fbac  *num: 3
numbers=5833fba0           num=5833fba0
&numbers[7]=5833fbbc       num+7=5833fbbc
numbers[7]=7               *(num+7)=7
```

Representing Strings

In C:

- ASCII encoding of characters
- Each character takes 1 byte
  - (There are other encodings)
- Character “0” has code **`0x30`**
  - Digit i has code **`0x30+i`**
- The string should be “null terminated”
  - **`0x00`** = NUL = **`‘\0’`**
- Byte ordering is not an issue
  - Big Endian = Little Endian
- Text files are usually platform independent


```
char mystr[6] = "15213";
```

But line termination can be a problem.

`\n`	**`0x0A`**
`\r`	**`0x0D`**
`\n\r`	**`0x0A0D`**

ASCII

ASCII Chart

`cs.pdx.edu/~harry/compilers/AsciiChart.pdf`

Hex	Decimal	Character	
00	0	NUL	Control characters
...			
1F	31		
20	32	*(space)*	Punctuation
21	33	!	
...			
30	48	0	Digits
...			
39	57	9	
3A	58	:	Punctuation
...			
41	65	A	Uppercase
...			
5A	90	Z	

Hex	Decimal	Character	
5B	91	[	Punctuation
...			
61	97	a	Lowercase
...			
7A	122	z	
7B	123	{	Punctuation
...			
7F	127	DEL	Backspace
80	128		Not used
...			
FF	255		

ASCII Character Set

! " # $ % & ' ( ) * + , - . / 0 1 2 3 4

5 6 7 8 9 : ; < = > ? @ A B C D E F G H

I J K L M N O P Q R S T U V W X Y Z [\

] ^ _ ` a b c d e f g h i j k l m n o p

q r s t u v w x y z { | } ~

ASCII Character Set

All printable characters
with decimal codes

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

Unicode

ASCII is only suitable for Roman / Latin alphabet.

Unicode supports

Russian, Greek, Chinese, math symbols, etc.

Unicode is the default for newer software

Java

The C library contains some support.

Each characters is encoded with 32 bits per character.

4 bytes

Characters are called “code points”.

There are several “encodings”.

UTF-8

UTF-16

UTF-32

Unicode Examples

ASCII/Latin-1 U+0000 – U+007F (0–127)

! 5 A a k

Latin-1 supplement U+0080 – U+00FF (128–255)

¥ ¢ € ¼ Ñ ñ

ü × ÷ æ ©

Cyrillic U+0400 – U+04FF (1024–1279)

Љ Щ щ Ҕ Ж

Greek U+0370 – U+03FF (880–1023)

Θ Γ Δ Σ Ψ

Misc.

∀ ∰ ⊇ ⋧

☂ ☺ ♀ ♪

ㄆ ㄓ ㄝ ㄩ

ㄨ 儴 [illegible] [illegible]

Unicode 7.0 (June 2014) encodes 113,021 characters.

UTF-32

Every character is given a 32-bit number

"code point"

UTF-32 is very simple.

Each number is stored in 4 bytes.

C standard library type:

`32-bit wchar_t`

Not all combinations are allowed.

2^{21} = 2,097,152 possible characters

(Only 21 bits are acutally needed for the code points.)

There is a waste of memory

11 bits (out of 32) are never used.

Many of the characters are very rare.

A better (variable-length) encoding is desirable.

UTF-8 *Requires one byte for most common characters*

UTF-16 *Requires two bytes for most other characters*

UTF-8

A variable-length, byte-based encoding

Preserves ASCII transparency.

All of the ASCII characters (0..127) are unchanged.

ASCII text is also UTF-8 text.

All other characters are encoded with multibyte sequences.

See next slide…

The first byte indicates the number of bytes that follow.

The leading byte is in the range $\texttt{C0}_{16}$ to $\texttt{FD}_{16}$.

The trailing bytes are in the range $\texttt{80}_{16}$ to BF_{16}.

The byte values $\texttt{FE}_{16}$ and $\texttt{FF}_{16}$ are never used.

UTF-8 is relatively compact for encoding text in European scripts.

Uses 50% more space than UTF-16 for East Asian text.

Characters up to $\texttt{7F}_{16}$ take one byte
Characters up to $\texttt{7FF}_{16}$ take two bytes
Characters up to $\texttt{FFFF}_{16}$ take three bytes
Other characters take 4-6 bytes

UTF-8

How the bits of the "code point" are encoded
Uses between 1 and 6 bytes per character.

Bits of code point	First code point	Last code point	Bytes in sequence	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6
7	U+0000	U+007F	1	0xxxxxxx					
11	U+0080	U+07FF	2	110xxxxx	10xxxxxx				
16	U+0800	U+FFFF	3	1110xxxx	10xxxxxx	10xxxxxx			
21	U+10000	U+1FFFFF	4	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx		
26	U+200000	U+3FFFFFF	5	111110xx	10xxxxxx	10xxxxxx	10xxxxxx	10xxxxxx	
31	U+4000000	U+7FFFFFFF	6	1111110x	10xxxxxx	10xxxxxx	10xxxxxx	10xxxxxx	10xxxxxx

Credit: Wikipedia / Ken Thompson

ASCII characters are 0-127; start with a zero.
ASCII characters are encoded without any change

UTF-16

This is a commonly used encoding; good for all languages.

Can encode code points `00000000`$_{16}$ through `0010FFFF`$_{16}$

The first 1,114,112 code points.

Most common characters are in the range of `0` to `FFFF`$_{16}$.

These are encoded exactly as-is.

A text file is a sequence of 16-bits numbers.

'Ж'

U+0416

0 4	1 6
0000 0100	0001 0110

These characters are encoded with two 16-bit numbers:

`10000`$_{16}$ to `10FFFF`$_{16}$

Character values `D800`$_{16}$ to `DFFF`$_{16}$ are set aside for the encoding mechanism

(These values will never be assigned to actual characters)

Subtract `10000`$_{16}$ to get a number `00000`$_{16}$ to `FFFFF`$_{16}$ (20 bits)

The first 2 bytes must be in the range `D800`$_{16}$ to `DBFF`$_{16}$

The second 2 bytes must be in the range `DC00`$_{16}$ to `DFFF`$_{16}$.

D 8–B	
1101 10--	---- ----

D C–F	
1101 11--	---- ----

UTF-16

Uses 16-bit (2-byte) number units.
Endian-ness is now a problem!

Big Endian is assumed.

The text file may begin with this character:
`0xFEFF`
Called the “Byte Order Mark” (BOM)
This is invisible
A “zero-width, non-breaking space”

The character **`0xFFFE`** is invalid and reserved. It should never be used.

If the software reads **`0xFFFE`** as the first character…
It should flip the bytes for all remaining 16-bit numbers.

Glyphs vs. Characters

"We need to distinguish between characters and glyphs. A character is the smallest semantic unit in a writing system. It is an abstract concept such as the letter A or the exclamation point. A glyph is the visual presentation of one or more characters, and is often dependent on adjacent characters.

There is not always a one-to-one mapping between characters and glyphs.
In many languages (Arabic is an example), the way a character looks depends heavily on the surrounding characters. Standard printed Arabic has as many as four different printed representations (glyphs) for every letter of the alphabet. In many languages, two or more letters may combine together into a single glyph (called a ligature), or a single character might be displayed with more than one glyph."

http://userguide.icu-project.org/unicode#TOC-Overview-of-UTF-16