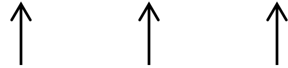
Data Representation

321 (in decimal)



100 10 1 ← How did we get these?

10^2 10^1 10^0 ← “Base-10”

3*100 + 2*10 + 1*1 ← Positions denote powers of 10

Digits 0-9 denote position values

101000001

101 million and one?

Actually, 321 in binary (Base-2)

Why should we care?

- Computers use binary (bits) to store all information

101000001

↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑

256 128 64 32 16 8 4 2 1 ← How did we get these?

↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑

2^8 2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0 ← "Base-2"

↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑

256 +0+ 64 + 0 + 0 + 0 + 0 + 0 + 1 = 321

↑

Positions denote powers of 2
Digits 0 and 1 denote position values

Binary group activity

Find decimal values of binary numbers below

```
      16 8 4 2 1
x =   1 0 0 1 1
y =   0 1 0 0 1
```

What are the decimal values of

x−y   x+y

What is the binary representation of

x−y   x+y

0x141

141 in decimal?

Actually, 321 in hexadecimal (Base-16)

Why hexadecimal?

Recall binary for 321

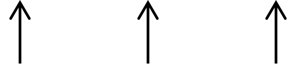
000101000001

0001 0100 0001

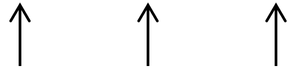
↓ ↓ ↓

1 4 1

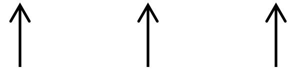
0x141



256 16 1 ← How did we get these?



16^2 16^1 16^0 ← “Base-16”



1*256 + 4*16 + 1*1 = 321

Positions denote powers of 16

But, requires 16 digit settings to denote values

- Binary 0,1
- Decimal 0,1,2,3,4,5,6,7,8,9
- Hexadecimal 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

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

Hexadecimal example

0x0CD

↑ ↑ ↑

256 16 1

↑ ↑ ↑

0*256 +12*16 + 13*1

192 + 13 = 205

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

Hexadecimal activity

0x1AF

↑ ↑ ↑

256 16 1

↑ ↑ ↑

1*256 +10*16 + 15*1

256 + 160 + 15 = 431

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

Converting bases

From Base 10 to other bases

- Find largest power `x` of base less than number `n`
- Find largest base digit `b` where `b*x < n`
- Recursively repeat using `n-(b*x)`

Example

- $15213_{10} = 1*10^4 + 5*10^3 + 2*10^2 + 1*10^1 + 3*10^0$

  Base 2 (binary)

x=2 16384 8192 4096 2048 1024 512 256 128 64 32 16 8 4 2 1

15213_{10} = 0 1 1 0 1 1 0 1 1 0 1 1 0 1_2

$11101101101101_2 = 1*2^{13} + 1*2^{12} + 1*2^{11} + 0*2^{10} + 1*2^9$ + etc…

Converting bases

Example

- $15213_{10} = 1*10^4 + 5*10^3 + 2*10^2 + 1*10^1 + 3*10^0$

  Base 16 (hexadecimal)

  » x=16 65536 4096 256 16 1

  » 15213_{10} = 0 3 B 6 D_{16}

  » $3B6D_{16} = 3*16^3 + 11*16^2 + 6*16^1 + 13*16^0$

  » Written in C as `0x3b6d`

From Base 2 binary to bases that are powers of two

- Done by grouping bits and assigning digits

  Base 2 (binary)

  » x=2 16384 8192 4096 2048 1024 512 256 128 64 32 16 8 4 2 1

  » 15213_{10} = 0 1 1 1 0 1 1 0 1 1 0 1 1 0 1_2

  » $11101101101101_2 = 1*2^{13} + 1*2^{12} + 1*2^{11} + 0*2^{10} + 1*2^9$ + etc…

- Example of binary to hex
  - 11 1011 0110 1101 = 3 B 6 D

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

Base 2
`128 64 32 16 8 4 2 1`

Base 16
`256 16 1`

Hex digits
a=10=1010_2
b=11=1011_2
c=12=1100_2
d=13=1101_2
e=14=1110_2
f=15=1111_2

0x333231

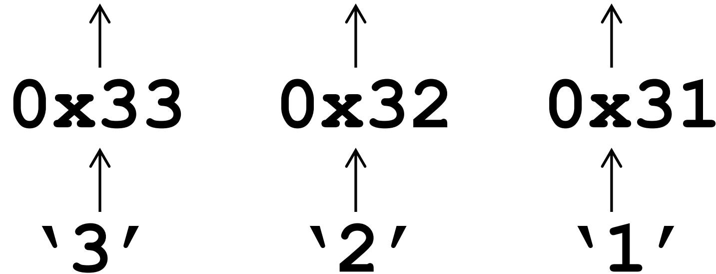
3,355,185?

Actually, “321” in ASCII

Humans encode characters in pairs of hexadecimal digits

- Each pair of hex digits is 8 bits or 1 byte
- Bytes are the smallest unit of data for computers

0x333231



ASCII maps byte values to characters

- Used in URLs, web pages, e-mail
- Other representations (Unicode, EBCDIC)

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

ASCII activity

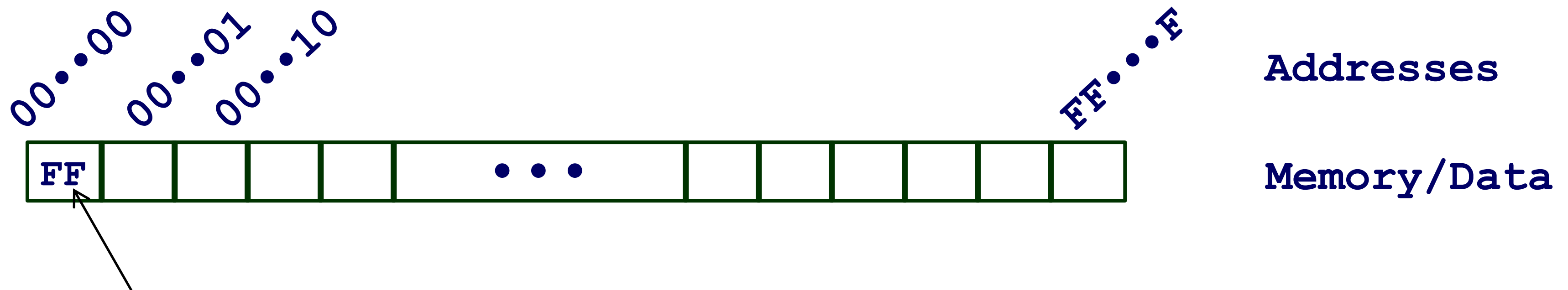
Snippet #1: 54 68 65 20 68 6F 6D 65 20 74 6F
Snippet #2: 65 76 65 72 79 6F 6E 65 20 69 73
Snippet #3: 74 6F 20 68 69 6D 20 68 69 73
Snippet #4: 63 61 73 74 6C 65 20 61 6E 64
Snippet #5: 66 6F 72 74 72 65 73 73 2E
Snippet #6: 45 64 77 61 72 64 20 43 6F 6B 65

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Byte-Oriented Memory Organization

Memory organized as an array of bytes



- Byte = 8 bits
  - Binary 00000000_2 to 11111111_2
  - Decimal: 0_{10} to 255_{10}
  - Hexadecimal 00_{16} to FF_{16}
- Address is an index into array
- Addressable unit of memory is a byte
- Recall system provides private address spaces to each "process"

Machine Words

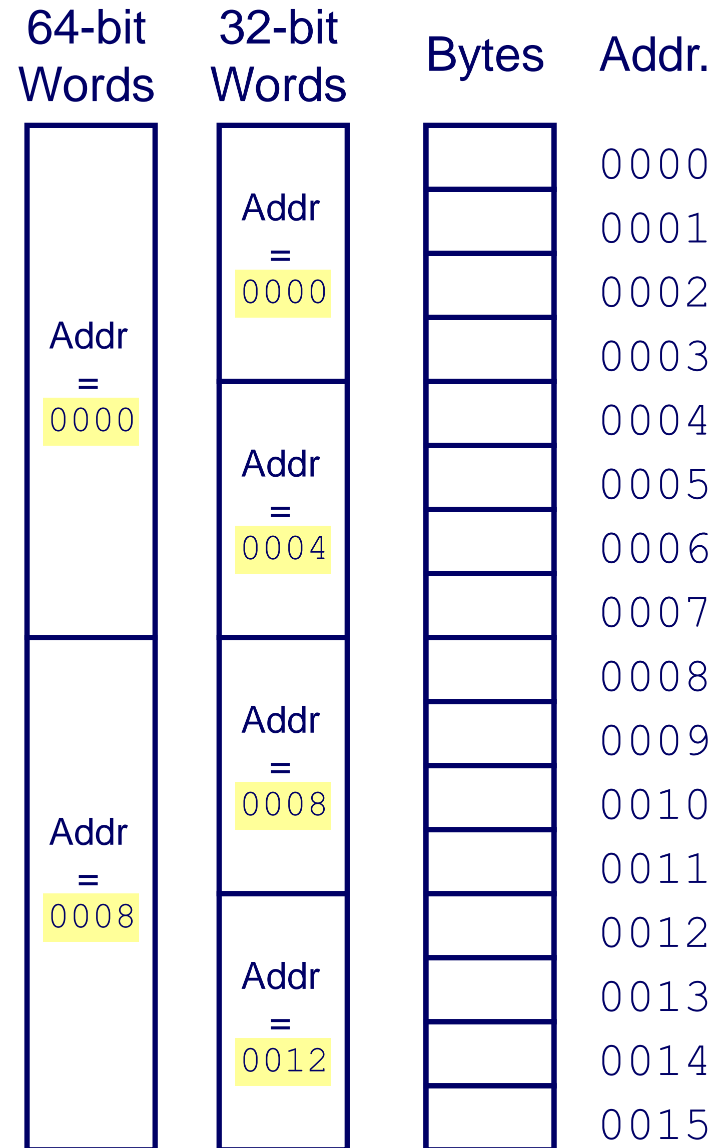
Any given computer has a "Word Size"

- Nominal size of integer-valued data
- Until recently, x86 word size was 32 bits (4 bytes)
  - Limits addresses to 4GB (2^{32} bytes)
- Now 64-bit word size
  - Potentially up to 18 PB (petabytes) of addressable memory
  - That's 18.4 X 10^{15} bytes of addressable memory

Word-Oriented Memory Organization

Addresses Specify Byte Locations

- Address of first byte in word
- Addresses of successive words differ by 4 (32-bit) or 8 (64-bit)



Byte Ordering

How should bytes within multi-byte word be ordered in memory?

Conventions

- Sun, PowerPC Macs, Internet protocols are "Big Endian"
  - Least significant byte has highest address
- x86 (PC/Mac), ARM (Android/iOS) are "Little Endian"
  - Least significant byte has lowest address

Byte Ordering Example

Big Endian

- Least significant byte has highest address

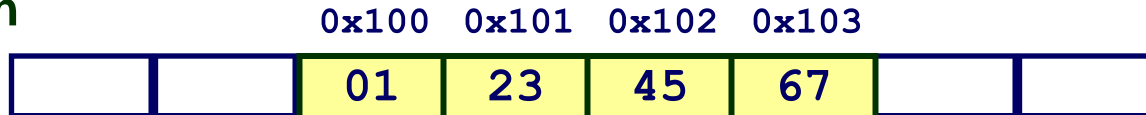
Little Endian

- Least significant byte has lowest address (LLL)

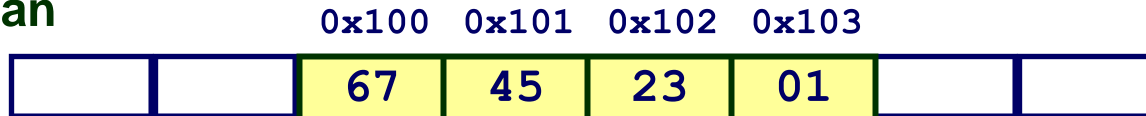
Example

- Variable `x` has 4-byte representation `0x01234567`
- Address given by `&x` is `0x100`

Big Endian



Little Endian



Endian

How do you test endian-ness?

- Direct inspection of memory via gdb
  - endian
    - gdb endian
    - break 5
    - run
    - p /x &i
    - x/b &i
    - x/b …

```
#include <stdio.h>
main()
{
        int i=0x01020304;
        printf("%d\n",i);
}
```

http://thefengs.com/wuchang/courses/cs201/class/04/endian.c

Endian

How do you test endian-ness?

- Simple program from book (`show_bytes`)

```
#include <stdio.h>
#include <string.h>
typedef unsigned char *byte_pointer;
void show_bytes(byte_pointer start, int len)
{
        int i;
        for (i = 0; i < len; i++)
                printf(" %.2x", start[i]);
        printf("\n");
}

void show_int(int x)
{       show_bytes((byte_pointer) &x, sizeof(int));}

void show_float(float x)
{       show_bytes((byte_pointer) &x, sizeof(float));}

void show_pointer(void *x)
{       show_bytes((byte_pointer) &x, sizeof(void*));}
```

```
int main()
{
        int i=0x01020304;
        float f=2345.6;
        int *ip=&i;
        char *s = "ABCDEF";

        show_int(i);
        show_float(f);
        show_pointer(ip);
        show_bytes(s,strlen(s));
}
```

```
Output:
 04 03 02 01
 9a 99 12 45
 28 61 61 63 fc 7f 00 00
 41 42 43 44 45 46
```

Representing pointers

Recall

A *pointer* is a variable containing a memory address of an object of a particular data type

- Contains a “reference” address for data

```
char* cp;   /* Declares cp to be a pointer to a character */
int* ip;    /* Declares ip to be a pointer to an integer */
```

- How many bytes is cp?
- How many bytes is ip?
- Both store address locations

Pointers in memory

Given the following code on x86-64…

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

Suppose the address of B is `0x007fff8d8` and the address of P is `0x007fff8d0`

What is the size of P?

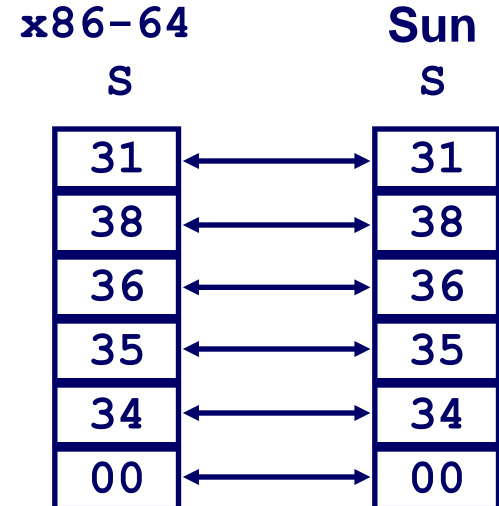
At the end of main, write the value of each byte of P in order as it appears in memory.

x86-64
D8
F8
FF
07
00
00
00
00

Representing strings

Strings in C

- Represented by array of characters
- Each character encoded in ASCII format
  - Standard 7-bit encoding of character set
  - Character "0" has code `0x30`
    - » Digit *i* has code `0x30`+*i*
- Must be null-terminated
  - Final character = 0



Compatibility

- Endian is not an issue
  - Data are single byte quantities
- Text files generally platform independent
  - Except for different conventions of line termination character(s)!

Representing strings

Alternate Unicode encoding

- 7-bit ASCII only suitable for English text
  - Can not cover characters in all languages
- 16-bit unicode character set
  - Supports Greek, Russian, Chinese, etc.
- Default encoding for strings in Java
- Support in C libraries for character set

Representing integers (signed)

Support for two types

- unsigned and signed
- Both are the same size (4 bytes or 32-bits on IA32)
- Differ based on how bits are interpreted

Unsigned integers

```
unsigned int i;
printf("%u\n",i)
```

- Encodes 0 to $(2^{32} – 1)$
- 0 to 4294967295
- Exactly as described in binary number slides

Signed integers in 2's complement format (default)

```
int i;
printf("%d\n",i)
```

- Encodes $–2^{31}$ to $(2^{31}-1)$
- -2,147,483,648 to 2,147,483,647

Encoding Integers

Unsigned

$$B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

$$= x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

Two's Complement

$$B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

16-bit example

- **Unsigned**

`32768 16384 8192 4096 2048 1024 512 256 128 64 32 16 8 4 2 1`

- **Signed**

`-32768 16384 8192 4096 2048 1024 512 256 128 64 32 16 8 4 2 1`

Encoding Integers

$$B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

$$B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

MSB
if 1, negative

```
short int x =  15213;
short int y = -15213;
```

C `short` 2 bytes long

	Decimal	Hex	Binary
x	15213	**3B 6D**	**00111011 01101101**
y	-15213	**C4 93**	**11000100 10010011**

Two-complement Encoding Example (Cont.)

```
x =      15213: 00111011 01101101
y =     -15213: 11000100 10010011
```

Weight	15213		-15213	
1	1	1	1	1
2	0	0	1	2
4	1	4	0	0
8	1	8	0	0
16	0	0	1	16
32	1	32	0	0
64	1	64	0	0
128	0	0	1	128
256	1	256	0	0
512	1	512	0	0
1024	0	0	1	1024
2048	1	2048	0	0
4096	1	4096	0	0
8192	1	8192	0	0
16384	0	0	1	16384
-32768	0	0	1	-32768
Sum		**15213**		**-15213**

Two's complement exercise

Exercise: Write -3 , -4, and –5 in two's complement format for w=4

−8	4	2	1
1	1	0	1
1	1	0	0
1	0	1	1

X	B2U(*X*)	B2T(*X*)
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

Exercise: Numeric ranges

For 16 bit signed numbers (w=16), write the greatest positive value and the most negative value, in hex and decimal. What does –1 look like?

- Greatest postive = 0x7FFF = 32767
- Least negative = 0x8000 = -32768
- Negative 1 = 0xFFFFF

Do the same for 32 bits.

Ranges for Different Word Sizes

	W			
	8	**16**	**32**	**64**
Unsigned Max	255	65,535	4,294,967,295	18,446,744,073,709,551,615
Signed Max	127	32,767	2,147,483,647	9,223,372,036,854,775,807
Signed Min	-128	-32,768	-2,147,483,648	-9,223,372,036,854,775,808

Casting Signed to Unsigned

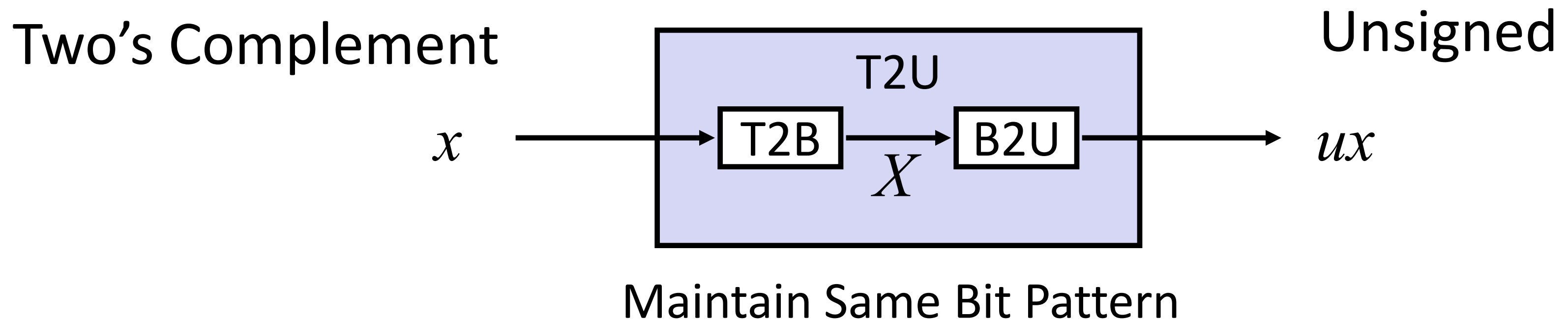
C allows conversions from signed to unsigned

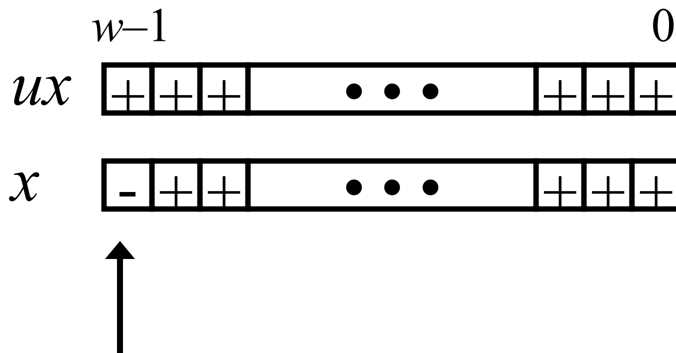
```
short int          x =  15213;
unsigned short int ux = (unsigned short) x;
short int          y  = -15213;
unsigned short int uy = (unsigned short) y;
```

Resulting Value

- No change in bit representation
- Non-negative values unchanged
  - *ux* = 15213
- Negative values change into (large) positive values
  - *uy* = 50323
  - Why? MSB treated as large positive number rather than large negative one.

Relation between Signed & Unsigned





Large negative weight
becomes
Large positive weight

Mapping Signed / 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

T2U →

← U2T

Mapping Signed / 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	+/- 16	11
1100	-4		12
1101	-3		13
1110	-2		14
1111	-1		15

Signed vs. unsigned example

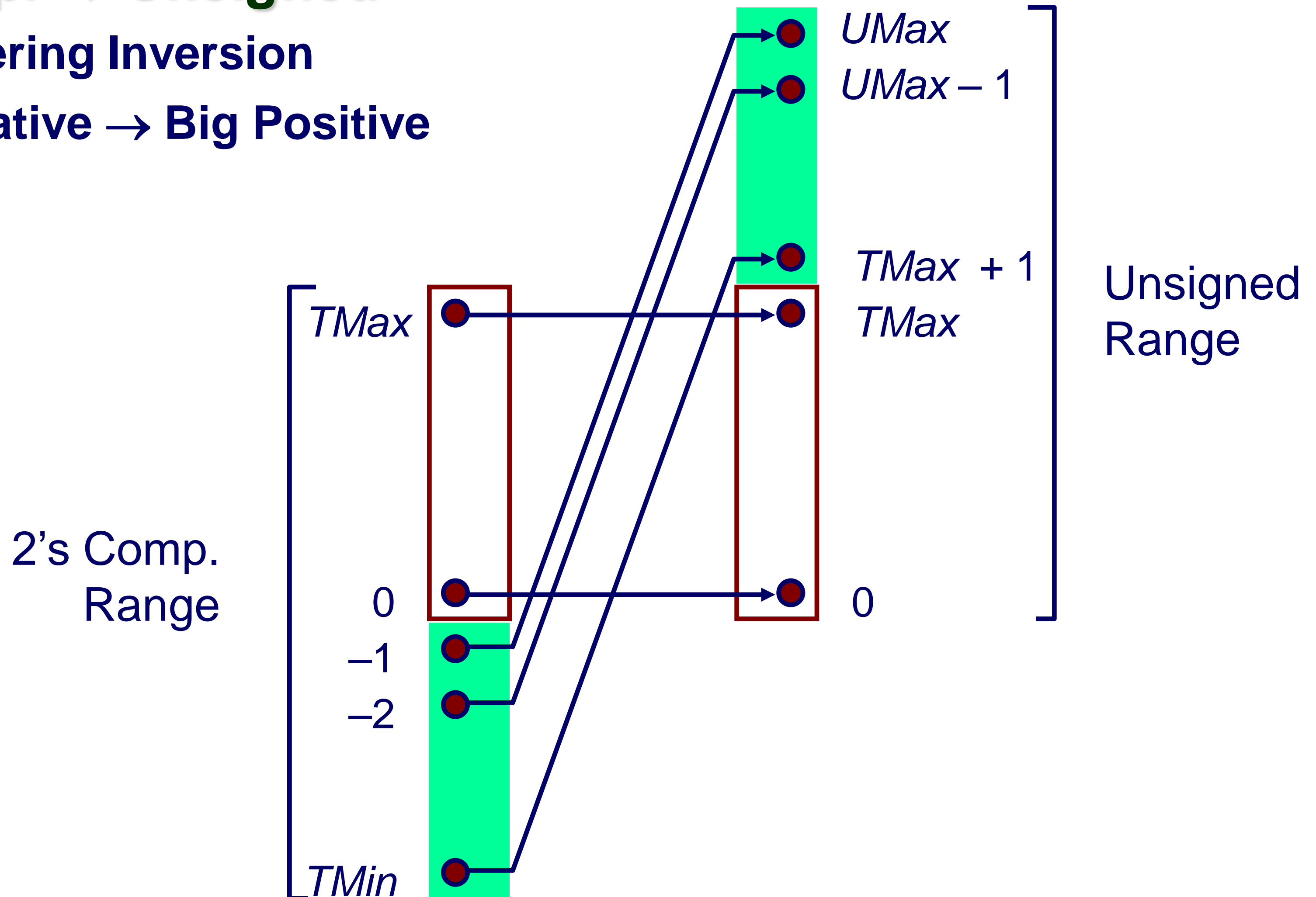
```
x = 11000100 10010011
```

Weight	x unsigned		x signed	
1	1	1	1	1
2	1	2	1	2
4	0	0	0	0
8	0	0	0	0
16	1	16	1	16
32	0	0	0	0
64	0	0	0	0
128	1	128	1	128
256	0	0	0	0
512	0	0	0	0
1024	1	1024	1	1024
2048	0	0	0	0
4096	0	0	0	0
8192	0	0	0	0
16384	1	16384	1	16384
±32768	1	32768	1	-32768
Sum		**50323**		**-15213**

Conversion visualized

2's Comp. → Unsigned

- Ordering Inversion
- Negative → Big Positive



Example

```
#include <stdio.h>
main()
{
  int i = 0xFFFFFFFF;
  unsigned int j = 0xFFFFFFFF;

  printf("Signed: 0x%x is %d , Unsigned: 0x%x is %u\n",i,i,j,j);

  i=0x80000000;
  j=0x80000000;
  printf("Signed: 0x%x is %d , Unsigned: 0x%x is %u\n",i,i,j,j);

  i=0x7FFFFFFF;
  j=0x7FFFFFFF;
  printf("Signed: 0x%x is %d , Unsigned: 0x%x is %u\n",i,i,j,j);

  short int x = 15213;
  short int y = -15213;

  unsigned short int ux = (unsigned short int) x;
  unsigned short int uy = (unsigned short int) y;
  printf("x:%d y:%d ux:%u uy:%u\n", x,y,ux,uy);

}
```

Output:

Signed: 0xffffffff is -1 , Unsigned: 0xffffffff is 4294967295

Signed: 0x80000000 is -2147483648 , Unsigned: 0x80000000 is 2147483648

Signed: 0x7fffffff is 2147483647 , Unsigned: 0x7fffffff is 2147483647

x:15213 y:-15213 ux:15213 uy:50323

http://thefengs.com/wuchang/courses/cs201/class/04/numbers.c

Signed vs. Unsigned in C

Constants

- By default are considered to be signed integers
- Unsigned if have "U" as suffix

```
0U, 4294967259U
```

Casting

- Explicit casting between signed & unsigned

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

- Implicit casting also occurs via assignments and procedure calls

```
tx = ux;
uy = ty
```

Casting Surprises

Expression Evaluation

- Mixing unsigned and signed in an expression, signed values implicitly cast to unsigned
- Including comparison operations <, >, ==, <=, >=
- Examples for $W = 32$ (TMIN = -2,147,483,648 , TMAX = 2,147,483,647)

Constant_1	Constant_2	Relation	Evaluation
`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

Errors from mixing signed and unsigned

Easy to make mistakes

```
unsigned int i;
int a[CNT];
for (i = CNT-2; i >= 0; i--)
  a[i] += a[i+1];
```

Is this ever false?

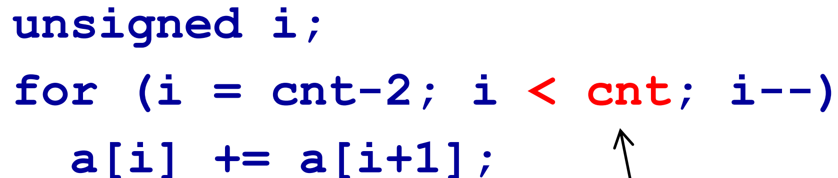
Can be very subtle. (Implicit casting of signed to unsigned)

```
#define DELTA sizeof(int)
int i;
for (i = CNT; i-DELTA >= 0; i-= DELTA)
  . . .
```

Is this ever false?

Counting Down with Unsigned

One potential fix…



```
unsigned i;
for (i = cnt-2; i < cnt; i--)
  a[i] += a[i+1];
```

Subtraction at 0 yields large positive number and exits loop

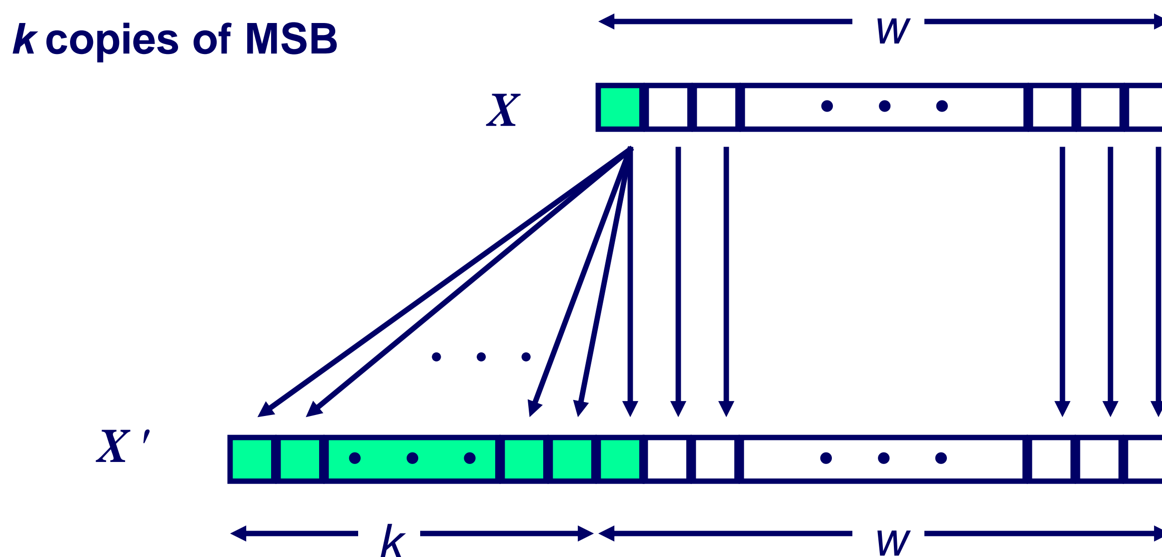
Casting with different integer sizes

Task:

- Given *w*-bit signed integer *x*
- Convert it to *w*+*k*-bit integer with same value

Rule:

- Make *k* copies of sign bit:
- $X' = x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_0$



Sign Extension Example

```
short int x =  15213;
int       ix = (int) x;
short int y = -15213;
int       iy = (int) y;
```

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
ix	15213	00 00 3B 6D	00000000 00000000 00111011 01101101
y	-15213	C4 93	11000100 10010011
iy	-15213	FF FF C4 93	11111111 11111111 11000100 10010011

- Converting from smaller to larger integer data type
- C automatically performs sign extension

Sign Extension Exercise

Calculate the hex value of -5 for w=4

Calculate the hex value of -5 for w=8

Calculate the hex value of -5 for w=16

Sign extension mayhem

Implicit casting and sign extension leads to errors

```
#include <stdio.h>
main() {
  char c=128;
  unsigned int uc;
  uc = (unsigned int) c;
  printf("%x %u\n",uc, uc);
}
Prints 0xffffff80 4294967168.
```

Sign extension mayhem

Spot the security bug

```
int snprintf(char *str, size_t size, const char *format, ...);

char dst[257];
char len;
len=get_len_field();                    /* read 8-bit length field */
snprintf(dst, len, "%s", src);          /* snprintf (size_t) len bytes */
```

- get_len_field reads 8-bit integer from network
- Can store field in a byte, but unfortunately a signed byte is used
- Any length > 128 converts to a negative number
- Casting from signed char to unsigned int sign extends!
- Input can overflow dst array

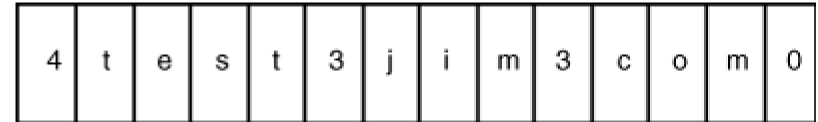
Sign extension mayhem

DNS parser vulnerability

- **Format being read: byte length followed by substring**

```
  char *indx;
  int count;
  char nameStr[MAX_LEN]; //256
...
  memset(nameStr, '\0', sizeof(nameStr));
...
  indx = (char *)(pkt + rr_offset);
  count = (char)*indx;
  while (count){
    (char *)indx++;
    strncat(nameStr, (char *)indx, count);
    indx += count;
    count = (char)*indx;
    strncat(nameStr, ".", sizeof(nameStr) - strlen(nameStr));
  }
  nameStr[strlen(nameStr)-1] = '\0';
```

test.jim.com



No length check to keep from overflowing nameStr

Count = 128? Negative count that is sign extended in strncat

http://www.informit.com/articles/article.aspx?p=686170&seqNum=6

Type mismatches and security

Comparison vulnerabilities

- Recall mixing casts signed to unsigned
- In `get_int`, what is returned when data is "-1"
- Will `n < 0` ever be true in either `get_int` or `main`?

```
int get_int(char *data) {
    unsigned int n = atoi(data);
    if(n < 0 || n > 1024)
        return -1;
    return n;
}
int main(int argc, char **argv) {
    unsigned long n;
    char buf[1024];
    if(argc < 2)
       exit(0);
    n = get_int(argv[1]);
    if(n < 0){
        fprintf(stderr, "illegal length specified\n");
        exit(-1);
    }
    memset(buf, 'A', n);
    return 0;
}
```

Passing -1 results in get_int returning -1 and a large memset

Type mismatches and security

2002 FreeBSD getpeername() bug

- Internal code implementing copy of hostname into user buffer used signed int (See B&O Ch. 2 Aside)
- `memcpy` call uses unsigned length
- What if adversary gives a length of “-1” for his buffer size?

```
#define KSIZE 1024
char kbuf[KSIZE]
void *memcpy(void *dest, void *src, size_t n);

int copy_from_kernel(void *user_dest, int maxlen){
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

- (`KSIZE < -1`) is false, so `len = -1`
- memcpy casts -1 to 2^{32}-1
- Unauthorized kernel memory copied out

Type mismatches and security

Truncation vulnerabilities

- What happens if `userstr` is 65,536 bytes long?

```
unsigned short int f;
char mybuf[1024];
char *userstr=getuserstr();

f=strlen(userstr);
if (f >= sizeof(mybuf))
  die("string too long!");
strcpy(mybuf, userstr);
```

- `strlen` returns `int`, but output truncated to 0
- `strcpy` overruns `mybuf` with entire `userstr` input

Move to 05Arithmetic

Pointer arithmetic

Arithmetic based on size of the type being pointed to

- Incrementing an (int *) adds 4 to pointer
- Incrementing a (char *) adds 1 to pointer

Pointer arithmetic exercise

Consider the following declaration on

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

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

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

C Data Type	Typical 32-bit	x86-64
`char`	1	1
`short`	2	2
`int`	4	4
`long`	4	8
`float`	4	4
`double`	8	8
`pointer`	4	8

Extras

Type mismatches and security

Signed comparison vulnerability example

```
int read_user_data(int sockfd){
    int length, sockfd, n;
    char buffer[1024];
    length = get_user_length(sockfd);
    if(length > 1024){
        error("illegal input, not enough room in buffer\n");
        return -1;
    }
    if(read(sockfd, buffer, length) < 0){
        error("read: %m");
        return -1;
    }
    return 0;
}
```

- get_user_length returns an unsigned 32-bit integer
- Since length is signed, what happens on a length > 2^{31}?
- length test passes since length negative
- read turns length into huge positive integer

Pointers and arrays

Arrays

- Stored contiguously in one block of memory
- Index specifies offset from start of array in memory

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

  - "a" used alone is a pointer containing address of the start of the integer array
- Elements can be accessed using index or via pointer increment and decrement
  - Pointer increments and decrements based on type of array

Example

```
#include <stdio.h>
main()
{
  char* str="abcdefg\n";
  char* x;
  x = str;
  printf("str[0]: %c str[1]: %c str[2]: %c str[3]: %c\n",
          str[0],str[1],str[2],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);

  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  &numbers[4]: %x  num+4: %x\n",
          numbers, num, &numbers[4],num+4);
  printf("%d %d\n",numbers[4],*(num+4));
}
```

```
Output:
str[0]: a str[1]: b str[2]: c str[3]: d

x: 8048690  *x: a
x: 8048691  *x: b
x: 8048692  *x: c
x: 8048693  *x: d

num: fffe0498  *num: 0
num: fffe049c  *num: 1
num: fffe04a0  *num: 2
num: fffe04a4  *num: 3

numbers: fffe0498  num: fffe0498
&numbers[4]: fffe04a8  num+4: fffe04a8
4 4
```

http://thefengs.com/wuchang/courses/cs201/class/04/p_arrays.c

Negating with Complement & Increment

For 2's complement, negation can be implemented as the bit-wise complement plus 1

- Claim: `~x + 1 == -x`

Complement

- Observation: `~x + x == 1111…11`$_2$ `== -1`

`x`	1	0	0	1	1	1	0	1
`+ ~x`	0	1	1	0	0	0	1	0
`-1`	1	1	1	1	1	1	1	1

Increment

- `~x + x + (-x + 1) == -1 + (-x + 1)`
- `~x + 1 == -x`

Negation with Complementation

4-bit examples

x		~x		incr(~x)	
0101	5	1010	-6	1011	-5
0111	7	1000	-8	1001	-7
1100	-4	0011	3	0100	4
0000	0	1111	-1	0000	0
1000	-8	0111	7	1000	-8

-8 | 4 | 2 | 1

Comp. & Incr. Examples

x = 15213

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
~x	-15214	C4 92	11000100 10010010
~x+1	-15213	C4 93	11000100 1001001**1**
y	-15213	C4 93	11000100 10010011

0

	Decimal	Hex	Binary
0	0	00 00	00000000 00000000
~0	-1	FF FF	11111111 11111111
~0+1	0	00 00	00000000 00000000