The Hardware/Software Interface

CSE351 Spring 2013

Integers

Roadmap

- Data & addressing
- **Integers & floats**
- Machine code & C
- x86 assembly programming
- Procedures & stacks
- Arrays & structs
- Memory & caches
- Processes
- Virtual memory
- Memory allocation
- Java vs. C

C:

```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
    c.getMPG();
```

Assembly language:

```
get_mpg:
    pushq   %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret
```

Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

OS:

Windows 8 Mac

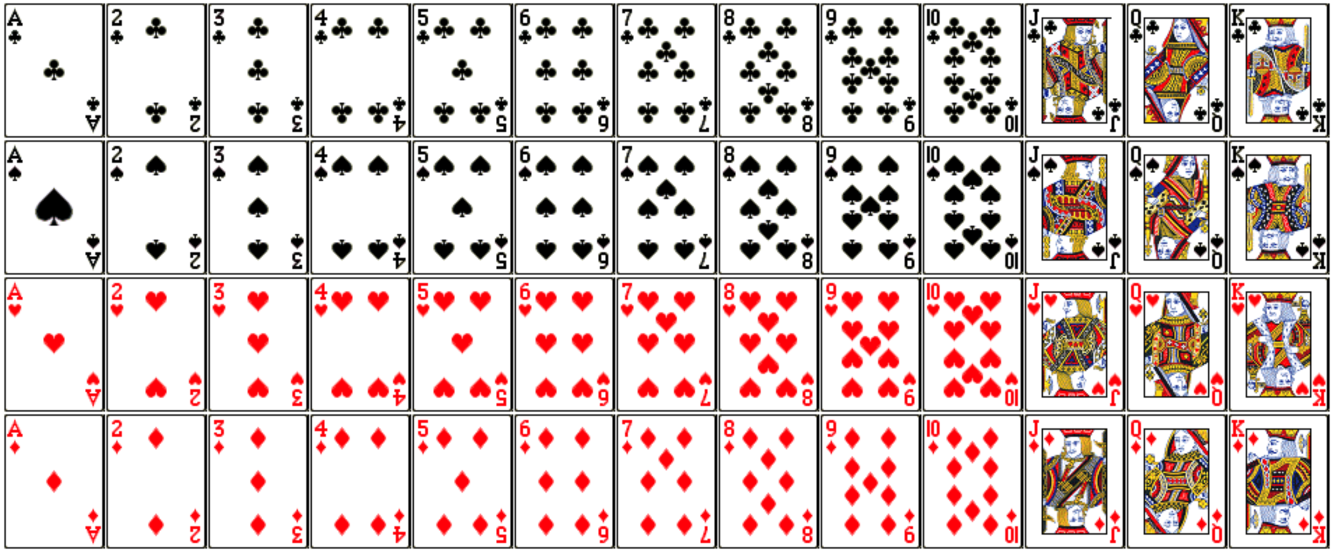
Computer system:

Today's Topics

- Representation of integers: unsigned and signed
- Casting
- Arithmetic and shifting
- Sign extension

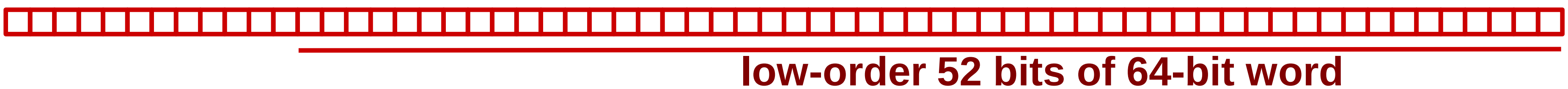
But before we get to integers….

- **How about encoding a standard deck of playing cards?**
- **52 cards in 4 suits**
  - How do we encode suits, face cards?
- **What operations do we want to make easy to implement?**
  - Which is the higher value card?
  - Are they the same suit?



Two possible representations

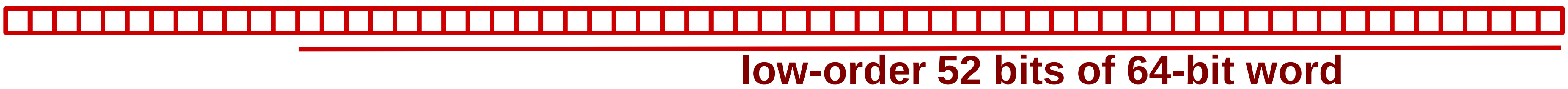
- **52 cards – 52 bits with bit corresponding to card set to 1**



  - "One-hot" encoding
  - Drawbacks:
    - Hard to compare values and suits
    - Large number of bits required

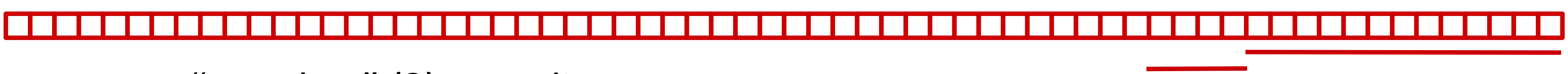
Two possible representations

- **52 cards – 52 bits with bit corresponding to card set to 1**



  - "One-hot" encoding
  - Drawbacks:
    - Hard to compare values and suits
    - Large number of bits required

- **4 bits for suit, 13 bits for card value – 17 bits with two set to 1**



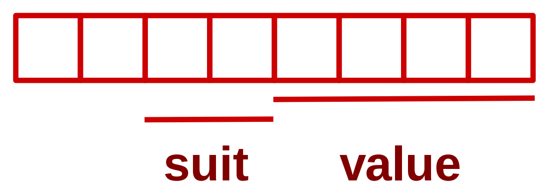
  - "Two-hot" (?) encoding
  - Easier to compare suits and values
    - Still an excessive number of bits

Two better representations

- **Binary encoding of all 52 cards – only 6 bits needed**

  - Fits in one byte
  - Smaller than one-hot or two-hot encoding, but how can we make value and suit comparisons easier?

- **Binary encoding of suit (2 bits) and value (4 bits) separately**



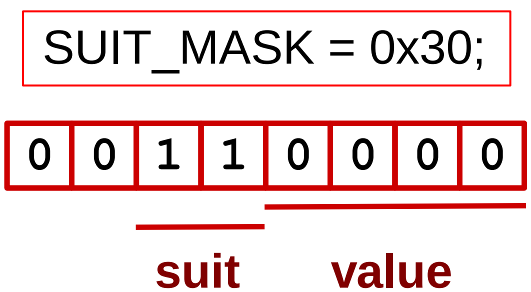
  - Also fits in one byte, and easy to do comparisons

Some basic operations

- **Checking if two cards have the same suit:**

```
#define SUIT_MASK  0x30
char array[5];        // represents a 5 card hand
char card1, card2;    // two cards to compare
card1 = array[0];
card2 = array[1];
...
if sameSuitP(card1, card2) {
...

bool sameSuitP(char card1, char card2) {
  return (! (card1 & SUIT_MASK) ^ (card2 & SUIT_MASK));
   //return (card1 & SUIT_MASK) == (card2 & SUIT_MASK);
}
```



Some basic operations

- **Comparing the values of two cards:**

```
#define SUIT_MASK  0x30
#define VALUE_MASK 0x0F
char array[5];      // represents a 5 card hand
char card1, card2;  // two cards to compare
card1 = array[0];
card2 = array[1];
...
if greaterValue(card1, card2) {
...
bool greaterValue(char card1, char card2) {
  return ((unsigned int)(card1 & VALUE_MASK) >
          (unsigned int)(card2 & VALUE_MASK));
}
```

VALUE_MASK = 0x0F;

0	0	0	0	1	1	1	1

suit value

Encoding Integers

- **The hardware (and C) supports two flavors of integers:**
  - unsigned – only the non-negatives
  - signed – both negatives and non-negatives
- **There are only 2^W distinct bit patterns of W bits, so...**
  - Can't represent all the integers
  - Unsigned values are 0 ... 2^W-1
  - Signed values are -2^{W-1} ... 2^{W-1}-1
- **Reminder: terminology for binary representations:**

"Most-significant" or "high-order" bit(s)

"Least-significant" or "low-order" bit(s)

0110010110101001

Unsigned Integers

- **Unsigned values are just what you expect**
  - $b_7b_6b_5b_4b_3b_2b_1b_0 = b_7 2^7 + b_6 2^6 + b_5 2^5 + \ldots + b_1 2^1 + b_0 2^0$
    - Useful formula: $1+2+4+8+\ldots+2^{N-1} = 2^N - 1$

- **You add/subtract them using the normal "carry/borrow" rules, just in binary**

```
  00111111
 +00000001
  01000000
```

```
   63
 +  8
   71
```

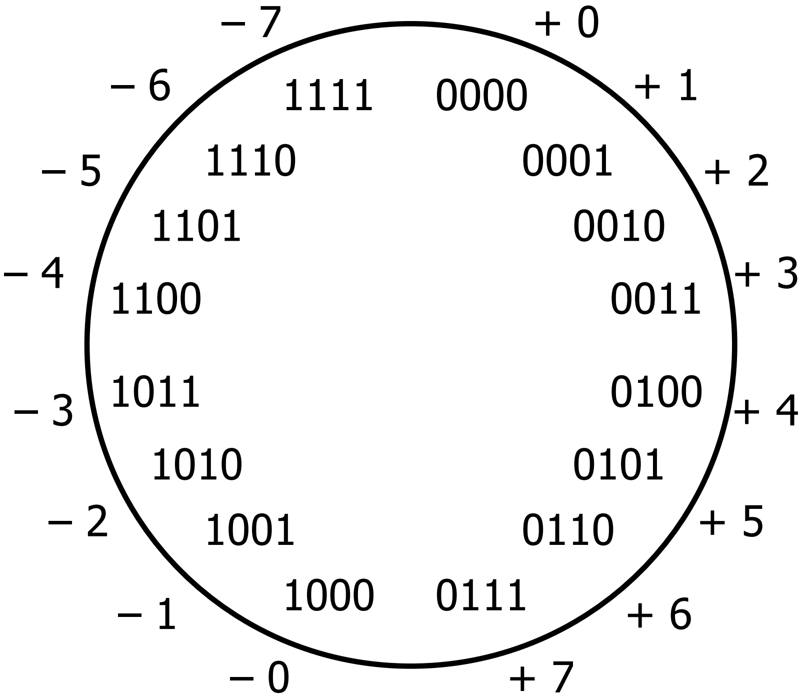
- **How would you make *signed* integers?**

Signed Integers

- **Let's do the natural thing for the positives**
  - They correspond to the unsigned integers of the same value
    - Example (8 bits): 0x00 = 0, 0x01 = 1, …, 0x7F = 127
- **But, we need to let about half of them be negative**
  - Use the high order bit to indicate *negative*: call it the “sign bit”
    - Call this a “sign-and-magnitude” representation
  - Examples (8 bits):
    - 0x00 = 00000000_2 is non-negative, because the sign bit is 0
    - 0x7F = 01111111_2 is non-negative
    - 0x85 = 10000101_2 is negative
    - 0x80 = 10000000_2 is negative…

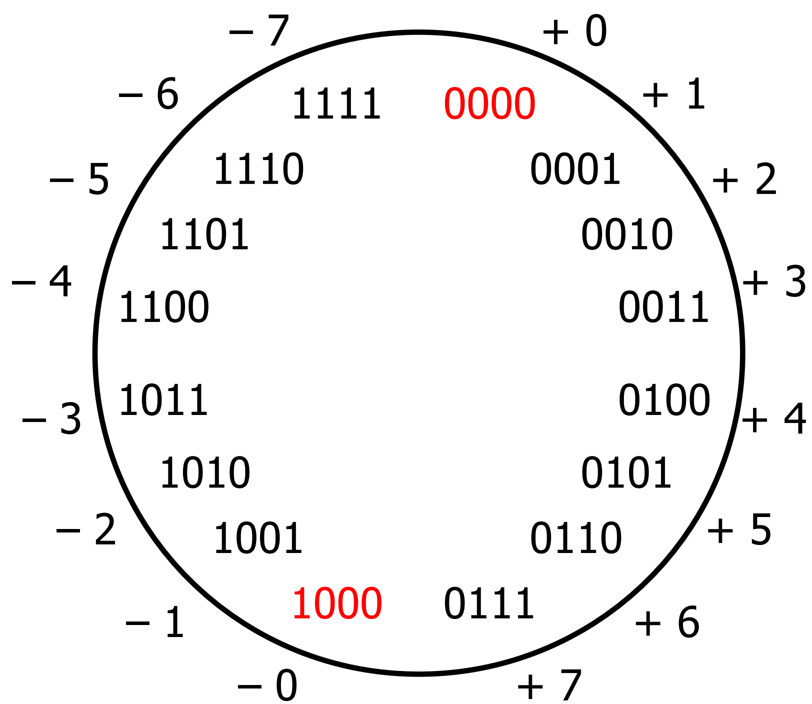
Sign-and-Magnitude Negatives

- **How should we represent -1 in binary?**
  - Sign-and-magnitude: 10000001_2
    Use the MSB for + or -, and the other bits to give magnitude



Sign-and-Magnitude Negatives

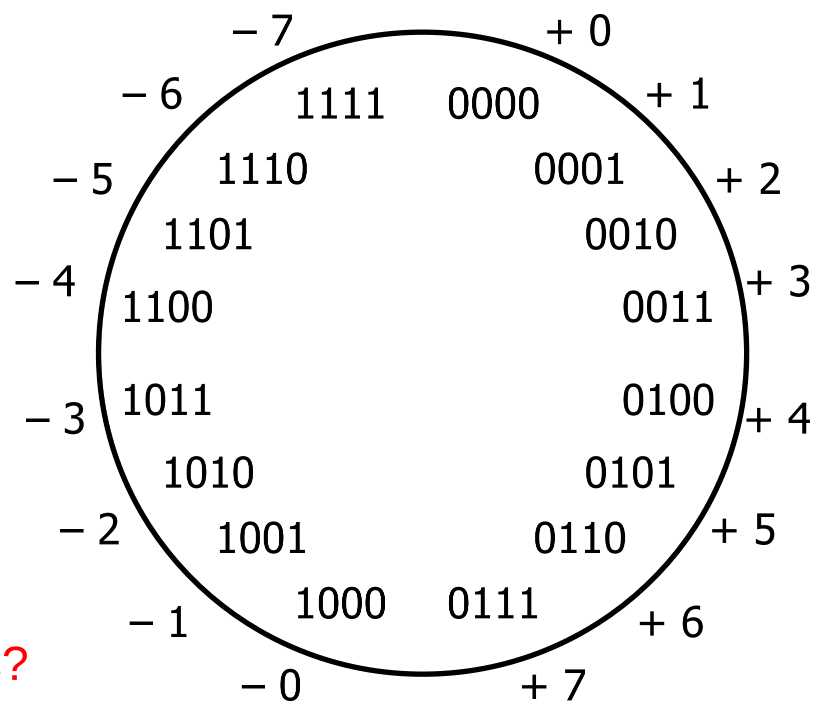
- **How should we represent -1 in binary?**
  - Sign-and-magnitude: 10000001_2
    Use the MSB for + or -, and the other bits to give magnitude
    (Unfortunate side effect: there are two representations of 0!)



Sign-and-Magnitude Negatives

- **How should we represent -1 in binary?**
  - Sign-and-magnitude: 10000001_2
    Use the MSB for + or -, and the other bits to give magnitude
    (Unfortunate side effect: there are two representations of 0!)
  - Another problem: math is cumbersome
    - Example:
      4 - 3 != 4 + (-3)

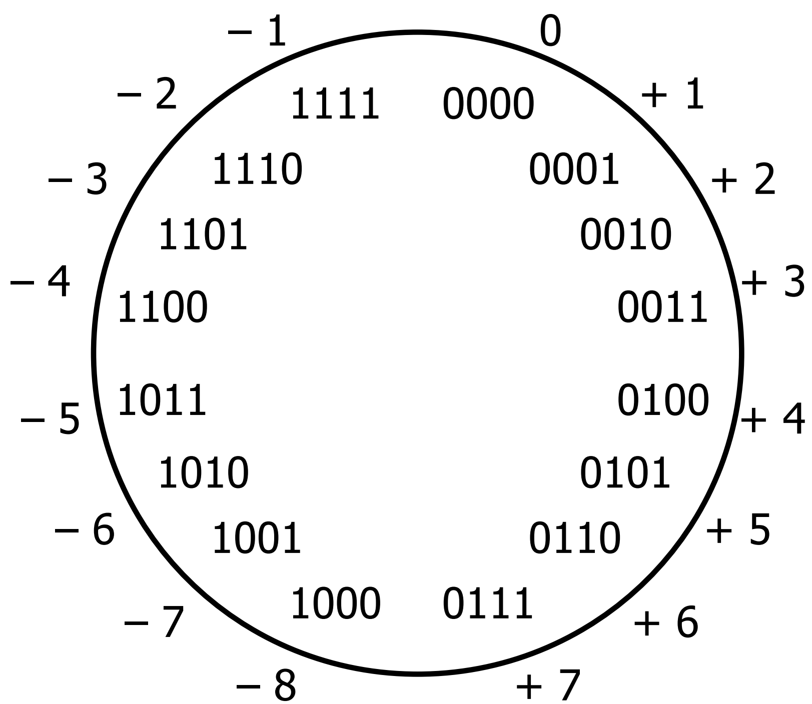
```
  0100
+ 1011
  ----
  1111
```



How do we solve these problems?

Two's Complement Negatives

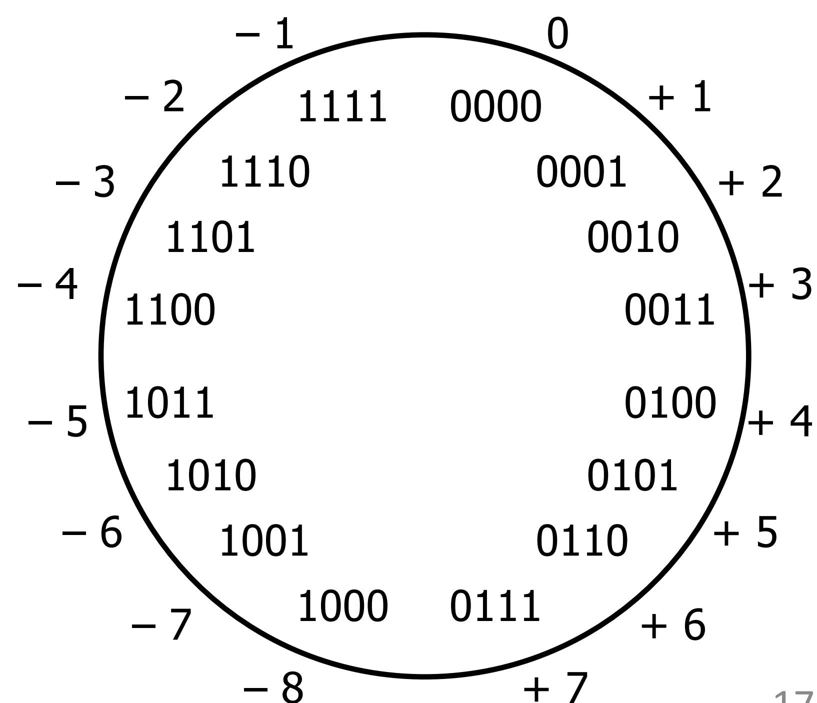
- **How should we represent -1 in binary?**



Two's Complement Negatives

- **How should we represent -1 in binary?**
  - Rather than a sign bit, let MSB have same value, but *negative* weight
    - W-bit word: Bits 0, 1, …, W-2 add 2^0, 2^1, …, 2^{W-2} to value of integer when set, but bit W-1 adds -2^{W-1} when set
    - e.g. unsigned 1010_2: $1*2^3 + 0*2^2 + 1*2^1 + 0*2^0 = 10_{10}$
      2's comp. 1010_2: $-1*2^3 + 0*2^2 + 1*2^1 + 0*2^0 = -6_{10}$
  - So -1 represented as 1111_2; all negative integers still have MSB = 1
  - Advantages of two's complement: only one zero, simple arithmetic
  - To get negative representation of any integer, take bitwise complement and then add one!

```
~x + 1 == -x
```



Two's Complement Arithmetic

- **The same addition procedure works for both unsigned and two's complement integers**
  - Simplifies hardware: only one adder needed
  - Algorithm: simple addition, discard the highest carry bit
    - Called "modular" addition: result is sum *modulo* 2^W
- **Examples:**

4	0100	4	0100	– 4	1100
+ 3	+ 0011	– 3	+ 1101	+ 3	+ 0011
= 7	= 0111	= 1	1 0001	– 1	1111
		drop carry	= 0001		

Two's Complement

- **Why does it work?**
  - Put another way: given the bit representation of a positive integer, we want the negative bit representation to always sum to 0 (ignoring the carry-out bit) when added to the positive representation
  - This turns out to be the *bitwise complement plus one*
    - What should the 8-bit representation of -1 be?

```
  00000001
+????????     (we want whichever bit string gives the right result)
----------
  00000000

  00000010      00000011
+????????     +????????
----------    ----------
  00000000      00000000
```

Two's Complement

- **Why does it work?**
  - Put another way: given the bit representation of a positive integer, we want the negative bit representation to always sum to 0 (ignoring the carry-out bit) when added to the positive representation
  - This turns out to be the *bitwise complement plus one*
    - What should the 8-bit representation of -1 be?

```
  00000001
 +11111111     (we want whichever bit string gives the right result)
 ---------
 100000000

  00000010        00000011
 +????????       +????????
 ---------       ---------
  00000000        00000000
```

Two's Complement

- **Why does it work?**
  - Put another way: given the bit representation of a positive integer, we want the negative bit representation to always sum to 0 (ignoring the carry-out bit) when added to the positive representation
  - This turns out to be the *bitwise complement plus one*
    - What should the 8-bit representation of -1 be?

```
  00000001
 +11111111     (we want whichever bit string gives the right result)
 ---------
 100000000

  00000010       00000011
 +11111110      +11111101
 ---------      ---------
 100000000      100000000
```

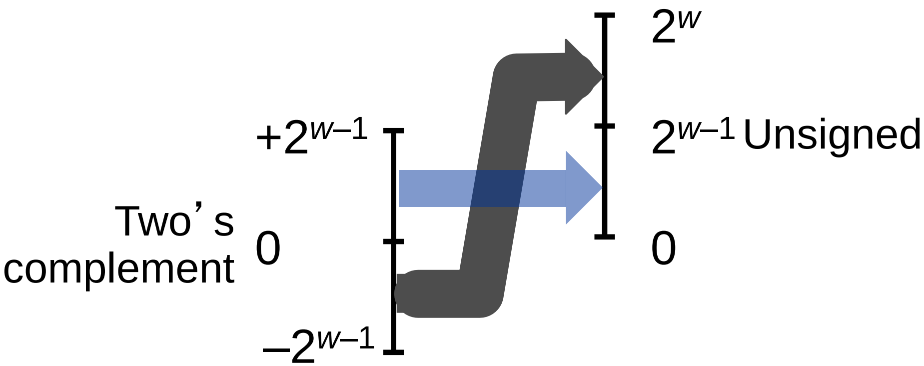
Unsigned & Signed Numeric Values

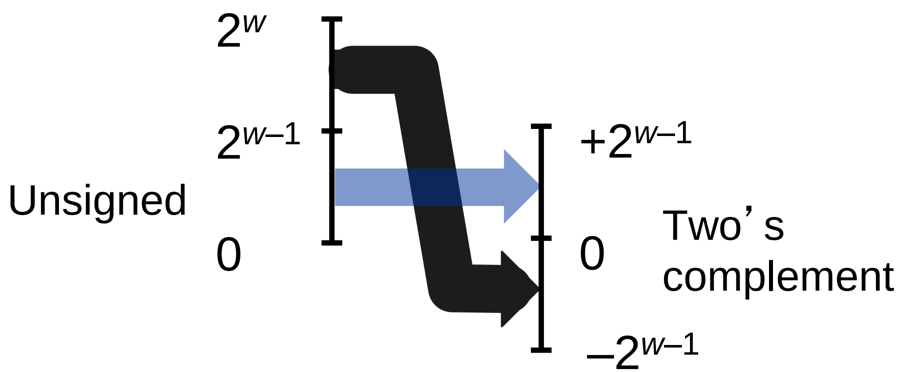
X	Unsigned	Signed
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	–8
1001	9	–7
1010	10	–6
1011	11	–5
1100	12	–4
1101	13	–3
1110	14	–2
1111	15	–1

- Both signed and unsigned integers have limits
  - If you compute a number that is too big, you wrap: 6 + 4 = ? 15U + 2U = ?
  - If you compute a number that is too small, you wrap: -7 - 3 = ? 0U - 2U = ?
  - Answers are only correct mod 2^b
- 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...

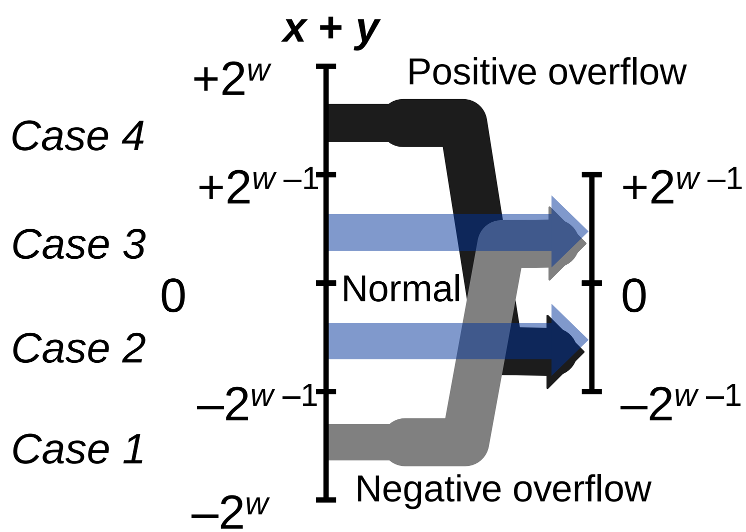
Visualizations

- Same W bits interpreted as signed vs. unsigned:





- Two's complement (signed) addition: x and y are W bits wide



Values To Remember

- **Unsigned Values**
  - UMin = 0
    - 000…0
  - UMax = $2^w - 1$
    - 111…1
- **Two's Complement Values**
  - TMin = -2^{w-1}
    - 100…0
  - TMax = $2^{w-1} - 1$
    - 011…1
  - Negative 1
    - 111…1 0xFFFFFFFF (32 bits)

Values for W = 16

	Decimal	Hex	Binary
`UMax`	65535	FF FF	11111111 11111111
`TMax`	32767	7F FF	01111111 11111111
`TMin`	-32768	80 00	10000000 00000000
`-1`	-1	FF FF	11111111 11111111
`0`	0	00 00	00000000 00000000

Values for Different Word Sizes

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

- **Observations**
  - $|TMin| = TMax + 1$
    - Asymmetric range
  - $UMax = 2 * TMax + 1$
- **C Programming**
  - #include <limits.h>
  - Declares constants, e.g.:
    - ULONG_MAX
    - LONG_MAX
    - LONG_MIN
  - Values are platform specific
  - See: /usr/include/limits.h on Linux

Signed vs. Unsigned in C

- **Constants**
  - By default are considered to be signed integers
  - Use "U" suffix to force unsigned:
    - `0U`, `4294967259U`

Signed vs. Unsigned in C

- **Casting**
  - `int tx, ty;`
  - `unsigned ux, uy;`
- Explicit casting between signed & unsigned:
  - `tx = (int) ux;`
  - `uy = (unsigned) ty;`
- Implicit casting also occurs via assignments and function calls:
  - `tx = ux;`
  - `uy = ty;`
  - The gcc flag *-Wsign-conversion* produces warnings for implicit casts, but *-Wall* does not!
- How does casting between signed and unsigned work – what values are going to be produced?
  - *Bits are unchanged*, just interpreted differently!

Casting Surprises

- **Expression Evaluation**
  - If you mix unsigned and signed in a single expression, then ***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

Shift Operations

- **Left shift: x << y**
  - Shift bit-vector x left by y positions
    - Throw away extra bits on left
    - Fill with 0s on right
  - Equivalent to multiplying by 2^y (if no bits lost)
- **Right shift: x >> y**
  - Shift bit-vector x right by y positions
    - Throw away extra bits on right
  - Logical shift (for unsigned values)
    - Fill with 0s on left
  - Arithmetic shift (for signed values)
    - Replicate most significant bit on left
    - Maintains sign of x
  - Equivalent to dividing by 2^y
    - Correct rounding (towards 0) requires some care with signed numbers

Argument x	01100010
<< 3	00010*000*
Logical >> 2	*00*011000
Arithmetic >> 2	*00*011000

Argument x	10100010
<< 3	00010*000*
Logical >> 2	*00*101000
Arithmetic >> 2	*11*101000

Undefined behavior when $y < 0$ or $y \geq$ word_size

Using Shifts and Masks

- **Extract the 2nd most significant byte of an integer:**
  - First shift, then mask: (x >> 16) & 0xFF

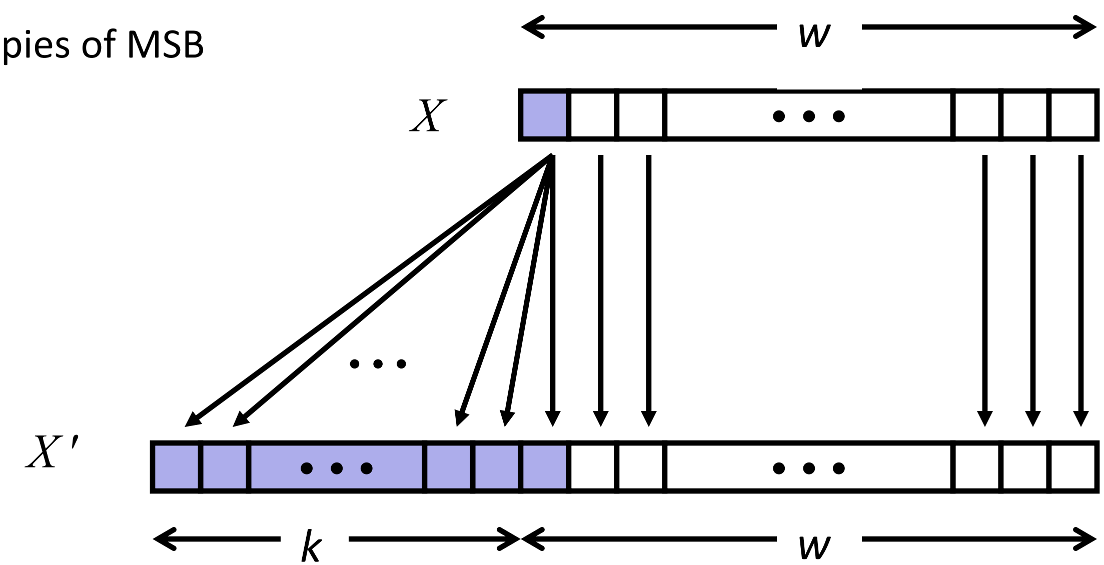
x	01100001 $\boxed{\text{01100010}}$ 01100011 01100100
x >> 16	00000000 00000000 01100001 $\boxed{\text{01100010}}$
(x >> 16) & 0xFF	*00000000 00000000 00000000 11111111* 00000000 00000000 00000000 01100010

- **Extract the sign bit of a signed integer:**
  - (x >> 31) & 1   - need the "& 1" to clear out all other bits except LSB
- **Conditionals as Boolean expressions (assuming x is 0 or 1)**
  - if (x) a=y else a=z;   which is the same as   a = x ? y : z;
  - Can be re-written (assuming arithmetic right shift) as:

    a = ((x << 31) >> 31) & y + ((!x) << 31) >> 31) & z;

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 ⍰ = $\underbrace{x_{w-1},\ldots,x_{w-1}}_{k \text{ copies of MSB}}, x_{w-1}, x_{w-2},\ldots, x_0$



Sign Extension Example

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

```
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