Integer Representation

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

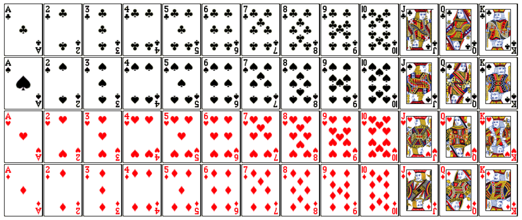
But first, encode deck of cards.

52 cards in 4 suits

- How do we encode suits, face cards?

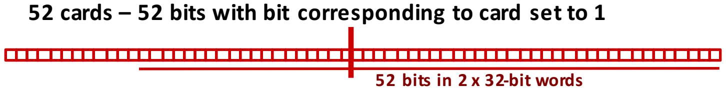
What operations should be easy to implement?

- Get and compare rank
- Get and compare suit



Two possible representations

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



"One-hot" encoding

- Two 32-bit words
- 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

Pair of one-hot encoded values

- Fits in one 32-bit word
- Easier to compare suits and values
- Still space-inefficient

Two better representations

Binary encoding of all 52 cards – only 6 bits needed

- Number each card
- Fits in one byte
- Smaller than one-hot encodings.
- How can we make value and suit comparisons easier?

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

- Number each suit
- Number each value
- Fits in one byte
- Easy suit, value comparisons

Compare Card **Suits**

mask: a bit vector that, when bitwise ANDed with another bit vector *v*, turns all *but* the bits of interest in *v* to 0

```
static final SUIT_MASK = 0x30;

boolean sameSuit(char card1, char card2) {
  return 0 == ((card1 & SUIT_MASK) ^ (card2 & SUIT_MASK));
  // return (card1 & SUIT_MASK) == (card2 & SUIT_MASK);
}
```

equivalent

SUIT_MASK = 0x30 = | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |

suit value

```
char hand[5];       // represents a 5-card hand
char card1, card2;  // two cards to compare
card1 = hand[0];
card2 = hand[1];
...
if ( sameSuit(card1, card2) ) { ... }
```

6

Compare Card **Values**

mask: a bit vector that, when bitwise ANDed with another bit vector *v*, turns all *but* the bits of interest in *v* to 0

```
static final VALUE_MASK = 0x0F;

boolean greaterValue(char card1, char card2) {
  return (card1 & VALUE_MASK) > (card2 & VALUE_MASK);
}
```

works even if value is stored in high bits

VALUE_MASK = 0x0F = | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |

suit value

```
char hand[5];       // represents a 5-card hand
char card1, card2;  // two cards to compare
card1 = hand[0];
card2 = hand[1];
...
if ( greaterValue(card1, card2) ) { ... }
```

7

Encoding Integers *in a fixed number of bits*

Two flavors:

- ***unsigned*** ($\subset \mathbb{N}$) – non-negatives only
- ***signed*** ($\subset \mathbb{Z}$) – both negatives and non-negatives
- **fixed-width** representations: **W bits** wide (W for word or width)

Positional representation, fixed # of positions.

???

Only 2^W distinct bit patterns...

- Cannot represent all the integers
- **Unsigned values: 0 ... 2^W-1**
- **Signed values: -2^{W-1} ... 2^{W-1}-1**

Terminology:

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

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

MSB 0110010110101001 **LSB**

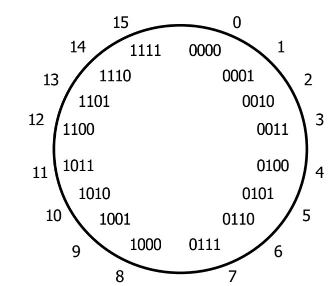
9

Unsigned modular arithmetic, overflow

Examples in 4-bit unsigned representation.

11 + 2 =

13 + 5 =



`x + y` **in N-bit unsigned arithmetic is** *(x + y) mod 2^N* **in math**

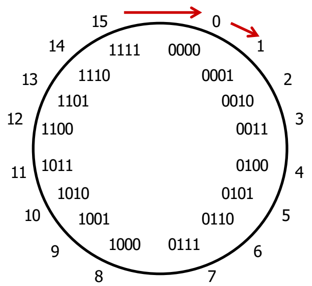
unsigned overflow ***= "wrong" answer = wrap-around***
= carry 1 out of MSB = math answer too big to fit

11

Overflow: Unsigned

Addition *overflows* if and only if a carry bit is dropped.

```
        1 1 1
 15      1111
+ 2    + 0010
 ~~17~~   10001
  1    Overflow.
```



Modular Arithmetic

12

Signed Integers: Sign-Magnitude?

!!!

Most-significant bit (MSB) is *sign bit*

0 means non-negative

1 means negative

Rest of bits are an unsigned magnitude

8-bit sign-and-magnitude:

0x00 = 00000000 represents ____________

0x7F = 01111111 represents ____________

0x85 = 10000101 represents ____________

0x80 = 10000000 represents ____________

Max and min for N-bit sign-magnitude?

What is weird about sign-magnitude representation?

13

Sign-Magnitude Negatives

!!!

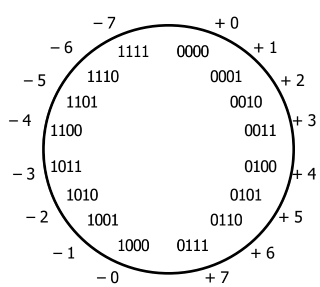
Another problem: cumbersome arithmetic.

Example:

4 - 3 != 4 + (-3)

```
 0100
+1011
```

What about zero?



Maybe sign-magnitude is not such a good idea...

14

Two's complement representation

for signed integers

w-bit representation

$-2^{(w-1)}$	...	2^i	...	2^3	2^2	2^1	2^0	← *weight*
w-1	...	i	...	3	2	1	0	← *position*

Positional representation, ***but***

most significant position has *negative weight*.

17

8-bit representations

0 0 0 0 1 0 0 1   1 0 0 0 0 0 0 1

1 1 1 1 1 1 1 1   0 0 1 0 0 1 1 1

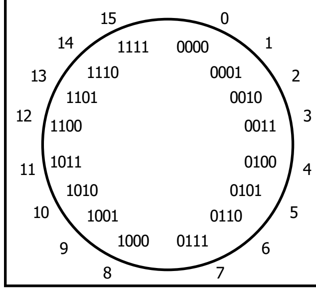
18

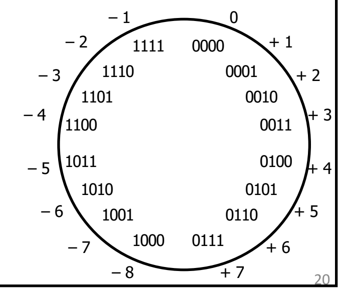
4-bit unsigned vs. 4-bit two's complement

1 0 1 1

$1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$ $1 \times -2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$

11 ← (math) difference = 16 = 2^4 → -5

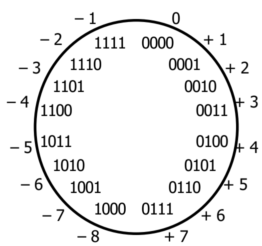




20

Two's complement: addition Just Works

2	0010	-2	1110
+ 3	+ 0011	+ -3	+ 1101
5	0101	-5	11011
-2	1110	2	0010
+ 3	+ 0011	+ -3	+ 1101
1	10001	-1	1111



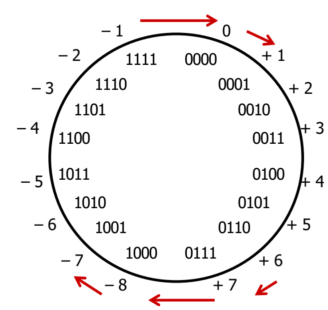
Modular Arithmetic

21

Overflow: Two's Complement

Addition *overflows* if and only if the inputs have the same sign but the output does not.
if and only if the carry in and out of the sign bit differ.

-1	1111
+ 2	+ 0010
1	10001
	No overflow.
6	0110
+ 3	+ 0011
~~9~~ -7	1001
	Overflow.



Modular Arithmetic

Some CPUs raise exceptions on overflow
C and Java cruise along silently… Oops?

22

A few reasons two's complement is awesome

Better be true!

`x + -x == 0`

Duh!

N-bit negative one is N ones.

Very useful!

Complement rules:

`x + ~x == -1`

`~x + 1 == -x`

Subtraction is just addition:

4 - 3 == 4 + (-3)

Great news for hardware.

Another view

How should we represent 8-bit negatives?

- For all positive integers ***x*** and widths ***n***, the n-bit representations of ***x*** and ***–x*** must sum to zero.
- Arithmetic should be "the same."

```
  00000001      00000010      00000011
+             +             +
----------    ----------    ----------
  00000000      00000000      00000000
```

- Find a rule to represent –x where that works…

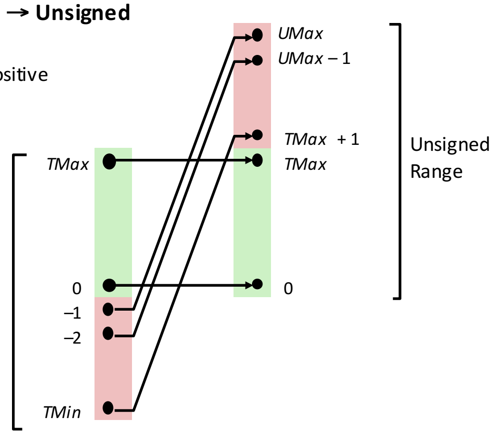
26

Conversion Visualized

Two's Complement → Unsigned

Ordering Inversion

Negative → Big Positive



28

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 one

111…1 0xF…F

Values for *W* = 32

	Decimal	Hex	Binary
UMax	4,294,967,296	FF FF FF FF	11111111 11111111 11111111 11111111
TMax	2,147,483,647	7F FF FF FF	01111111 11111111 11111111 11111111
TMin	-2,147,483,648	80 00 00 00	10000000 00000000 00000000 00000000
-1	-1	FF FF FF FF	11111111 11111111 11111111 11111111
0	0	00 00 00 00	00000000 00000000 00000000 00000000

29

Sign Extension

0 0 0 0 0 0 1 0	8-bit 2
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0	16-bit 2
1 1 1 1 1 1 0 0	8-bit -4
? ? ? ? ? ? ? ? 1 1 1 1 1 1 0 0	16-bit -4

Try some possibilities...

Sign Extension

Fill new bits with copies of the sign bit.

0 0 0 0 0 0 1 0	8-bit 2
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0	16-bit 2
1 1 1 1 1 1 0 0	8-bit -4
1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0	16-bit -4

Casting from smaller to larger signed type does sign extension.

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

Right shift: **x >> y**
- Shift bit vector x right by y positions
  - Throw away extra bits on right
  - Fill with ??? on left
- **Logical** shift
  - Fill with 0s on left
- **Arithmetic** shift
  - Replicate most significant bit on left
  - Why is this useful? Rain check!

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

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

!!! *in C: meaning of >> on signed types is compiler-defined! GCC: arithmetic shift*

in Java: >> is arithmetic, >>> is logical

Shift gotchas

!!!

For a type represented by n bits, shift by no more than n-1.

C: shift by <0 or >(*bits in type*) is undefined.

means anything could happen, including computer catching fire

Java: shift value is used modulo number of bits in shifted type

given int x: x << 34 == x << 2

Using Shifts and Masks

Extract 2nd most significant byte from a 32-bit integer:

x	01100001 $\boxed{01100010}$ 01100011 01100100

Extract the sign bit of a signed integer:

Shifting and Arithmetic: unsigned

unsigned
x = 27;
y = x << 2;
y == 108

0 0 0 1 1 0 1 1

0 0 0 1 1 0 1 1 0 0

logical shift left:
shift in zeros from the right

unsigned
x = 237;
y = x >> 2;
y == 59

1 1 1 0 1 1 0 1

0 0 1 1 1 0 1 1 0 1

logical shift right:
shift in zeros from the left

Shifting and Arithmetic: signed

signed
x = -101;
y = x << 2;
y == 108

1 0 0 1 1 0 1 1

1 0 0 1 1 0 1 1 0 0

logical shift left:
shift in zeros from the right

For x > 0:

signed
x = -19;
y = x >> 2;
y == -5

1 1 1 0 1 1 0 1

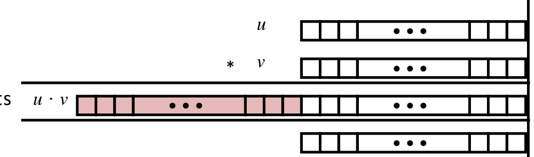
1 1 1 1 1 0 1 1 0 1

arithmetic shift right:
shift in copies of MSB from the left

For x < 0 it rounds the opposite direction!

Multiplication

!!!



Operands: w bits

True Product: $2*w$ bits

Discard w bits: w bits

unsigned: $u * v = (u \cdot v) \bmod 2^w$

overflow iff any bit $b_{y>=w} \mathrel{!}= 0$

signed: overflow iff any bit $b_{y>=w} \mathrel{!}= b_{w-1}$

Most programming languages

More generally true about loss of value when casting to smaller types.
High bits are just discarded.

Power-of-2 Multiply with Shift

Operation

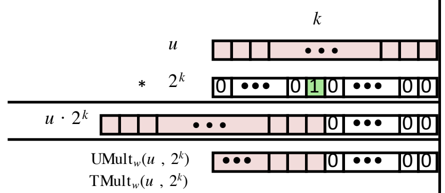
`u << k = u * 2^k`

Both signed and unsigned

Operands: w bits

True Product: $w+k$ bits

Discard k bits: w bits



Examples

```
          u << 3  ==  u * 8
(u << 5) - (u << 3)  ==  u * 24
```


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: ***bits unchanged, just interpreted differently.***

```
int tx, ty;
unsigned ux, uy;
```

Explicit casting:

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

Implicit casting via assignments and function calls:

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

gcc flag *-Wsign-conversion* warns about implicit casts; *-Wall* does not!

C Casting Surprises

!!!

Expression Evaluation

If you mix unsigned and signed in a single expression, then ***signed values are 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