CSCI-UA.0201

Computer Systems Organization

Data Representation – Integers and Floating points

Thomas Wies
wies@cs.nyu.edu
https://cs.nyu.edu/wies

What happens if you change the type of a variable (aka type casting)?

Signed vs. Unsigned in C

- Constants
  - By default, signed integers
  - Unsigned with "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;
```

General Rule for Casting: signed <-> unsigned

Follow these two steps:

1. Keep the bit presentation
2. Re-interpret

Effect:

- Numerical value may change.
- Bit pattern stays the same.

Mapping Signed ↔ Unsigned

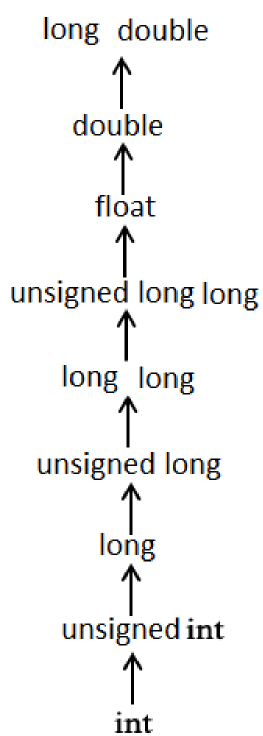
Bits	Signed	Unsigned
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	-8	8
1001	-7	9
1010	-6	10
1011	-5	11
1100	-4	12
1101	-3	13
1110	-2	14
1111	-1	15

=

+/- 16

Casting Surprises

- Expression Evaluation
  - If there is a mix of unsigned and signed in single expression, *signed values implicitly cast to unsigned*
  - Including comparison operations **<, >, ==, <=, >=**



If there is an expression that has many types, the compiler follows these rules.

Example

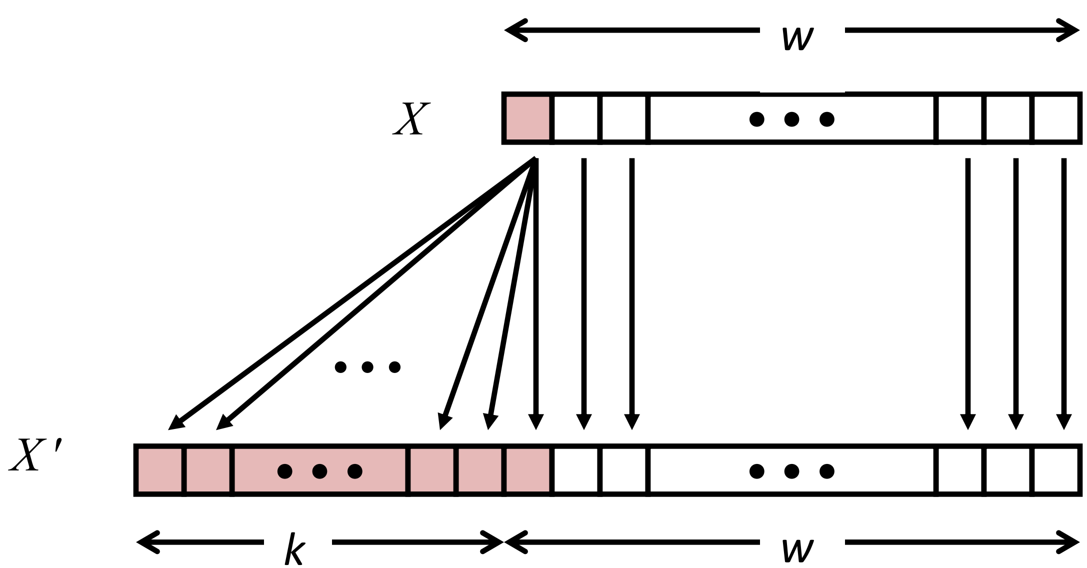
```
#include <stdio.h>


int main() {
  int  i = -7;
  unsigned j = 5;

  if(i > j)
    printf("Surprise!\n");
  return 0;
}
```

Condition is TRUE!

Expanding & Truncating a variable

Expanding

- Convert *w*-bit signed integer to *w*+*k*-bit with same value
- Convert unsigned: pad k 0 bits in front
- Convert signed: make *k* copies of sign bit



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

Truncating

- Example: from int to short (i.e. from 32-bit to 16-bit)
- High-order bits are truncated
- Value is altered → must reinterpret
- Can lead to buggy code! → So don't do it!

Addition, negation, multiplication, and shifting

Negation: Complement & Increment

- The two's complement of x satisfies

  `TC(x) + x = 0`

  where **`TC(x) = ~x + 1`**

- Proof sketch
  - Observation: **`~x + x = 1111…111 = -1`**

    → **`~x + x + 1 = 0`**

    → **`(~x + 1) + x = 0`**

    → **`TC(x) + x = 0`**

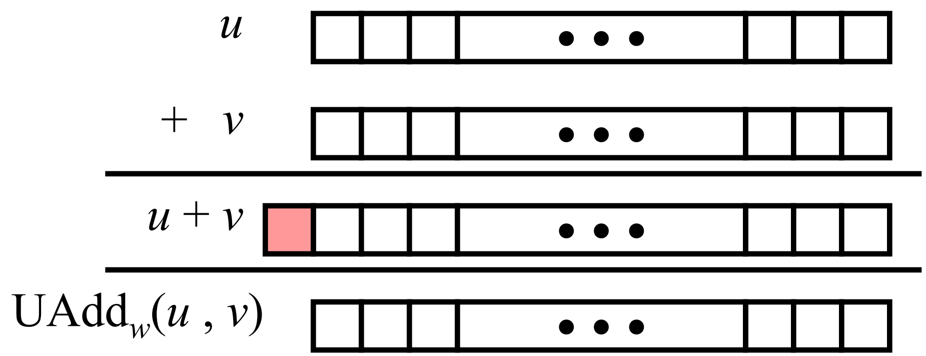
x	1	0	0	1	1	1	0	1
+ ~x	0	1	1	0	0	0	1	0
-1	1	1	1	1	1	1	1	1

Unsigned Addition

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits



Hardware Rules for addition/subtraction

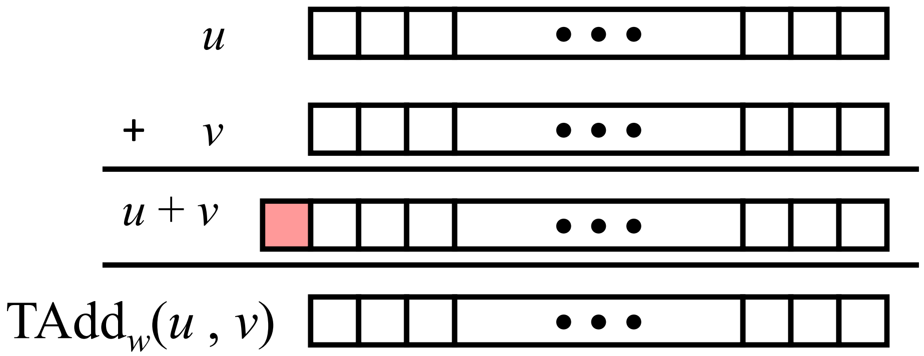
- The hardware must work with two operands of the same length.
- The hardware produces a result of the same length as the operands.
- The hardware does not differentiate between signed and unsigned.

Two's Complement Addition

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits



- If sum $\geq 2^{w-1}$, becomes negative (positive overflow)
- If sum $< -2^{w-1}$, becomes positive (negative overflow)

Signed Overflow in C

- **CAUTION**: signed overflow has undefined behavior in C!
- The compiler may assume that signed overflow never happens and exploit this in optimizations.
- Example:

```
int x = INT_MAX;
if (x + 1 < x) printf("Overflow!");
```

GCC assumes this is always FALSE!

Multiplication

- Exact Product of w-bit numbers x, y
  - Either signed or unsigned
- Ranges
  - Unsigned: $0 \le x * y \le (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
  - Two's complement min: $x * y \ge (-2^{w-1})*(2^{w-1}-1) = -2^{2w-2} + 2^{w-1}$
  - Two's complement max: $x * y \le (-2^{w-1})^2 = 2^{2w-2}$

Power-of-2 Multiply with Shift

- Operation
  - `u << k` gives `u *` 2^k
  - Both signed and unsigned

k

- Examples
  - `u << 3  ==  u * 8`
  - `(u << 5) - (u << 3)  ==  u * 24`
  - Most machines shift and add faster than multiply
    - Compiler generates this code automatically

Compiled Multiplication Code

C Function

```
int mul12(int x)
{
  return x*12;
}
```

Compiled Arithmetic Operations

```
  leal (%eax,%eax,2), %eax
  sall $2, %eax
```

Explanation

```
  t = x+x*2
  return t << 2;
```

- C compiler automatically generates shift/add code when multiplying by constant

Unsigned Power-of-2 Divide with Shift

- Quotient of Unsigned by Power of 2
  - `u >> k` gives $\lfloor$ `u / ` $2^k \rfloor$

Examples:

	Division	Computed	Hex	Binary
`x`	15213	15213	`3B 6D`	`00111011 01101101`
`x >> 1`	7606.5	7606	`1D B6`	`00011101 10110110`
`x >> 4`	950.8125	950	`03 B6`	`00000011 10110110`
`x >> 8`	59.4257813	59	`00 3B`	`00000000 00111011`

Compiled Unsigned Division Code

C Function

```
unsigned udiv8(unsigned x)
{
  return x/8;
}
```

Compiled Arithmetic Operations

```
  shrl $3, %eax
```

Explanation

```
  # Logical shift
  return x >> 3;
```

- Uses logical shift for unsigned
- For Java Users
  - Logical shift written as >>>

Signed Power-of-2 Divide with Shift

- Quotient of Signed by Power of 2
  - `x >> k` gives $\lfloor x / 2^k \rfloor$
  - Uses arithmetic shift

Examples

	Division	Computed	Hex	Binary
`y`	-15213	-15213	`C4 93`	`11000100 10010011`
`y >> 1`	-7606.5	-7607	`E2 49`	`11100010 01001001`
`y >> 4`	-950.8125	-951	`FC 49`	`11111100 01001001`
`y >> 8`	-59.4257813	-60	`FF C4`	`11111111 11000100`

Floating Points

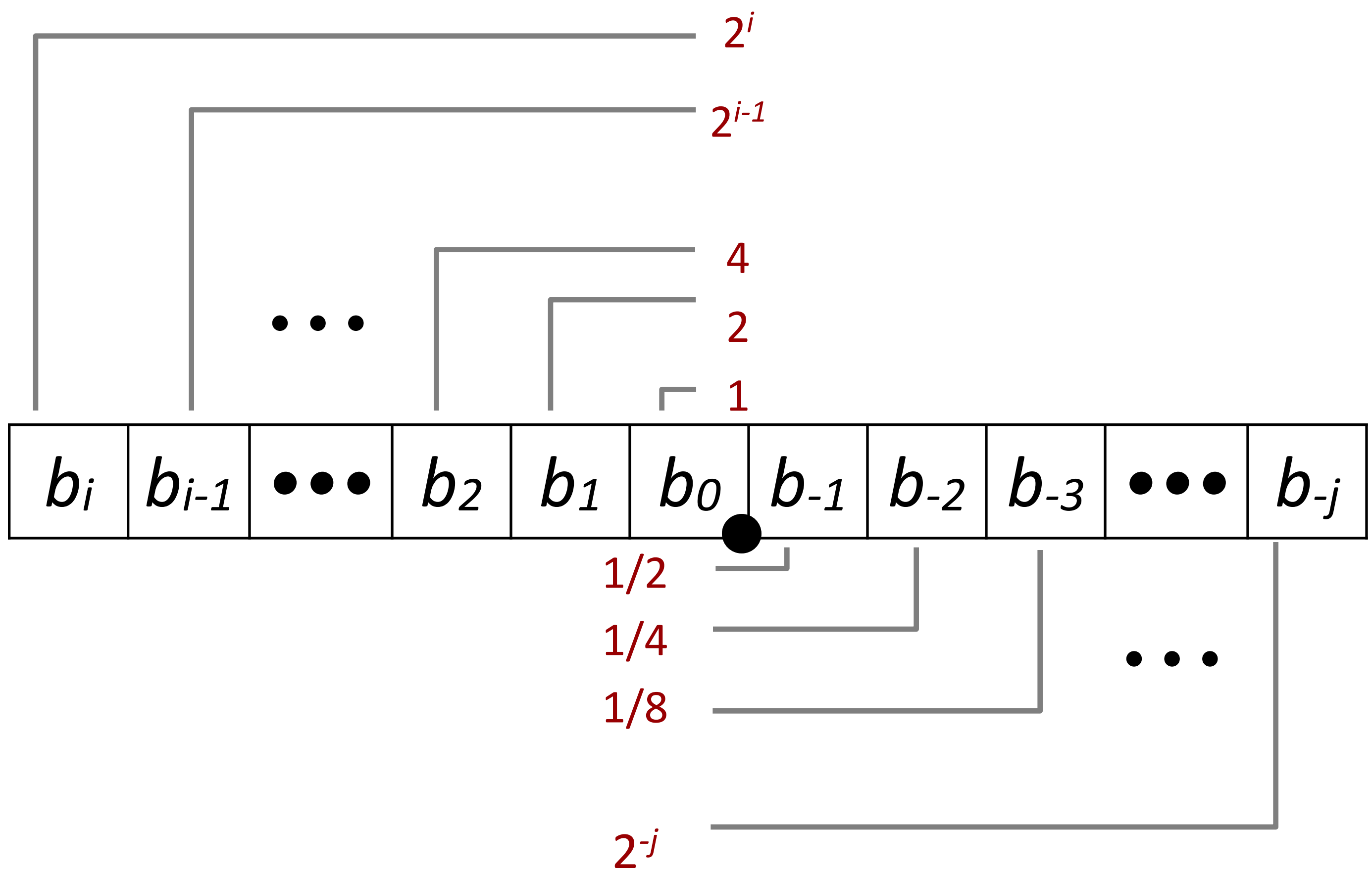
Some slides and information about FP are adopted from
Prof. Michael Overton book:
Numerical Computing with IEEE Floating Point Arithmetic

Turing Award 1989 to William Kahan for design of the IEEE Floating Point Standards 754 (binary) and 854 (decimal)

Background: Fractional binary numbers

- What is 1011.101_2?

Background: Fractional Binary Numbers



- Value: $$\sum_{k=-j}^{i} b_k \times 2^k$$

Fractional Binary Numbers: Examples

Value	Representation
5 3/4	101.11_2
2 7/8	10.111_2

Why not fractional binary numbers?

- Not efficient
  - $3 * 2^{100} \rightarrow$ 1010000000 0

    100 zeros

  - Given a finite length (e.g. 32-bits), cannot represent very large numbers nor numbers very close to 0