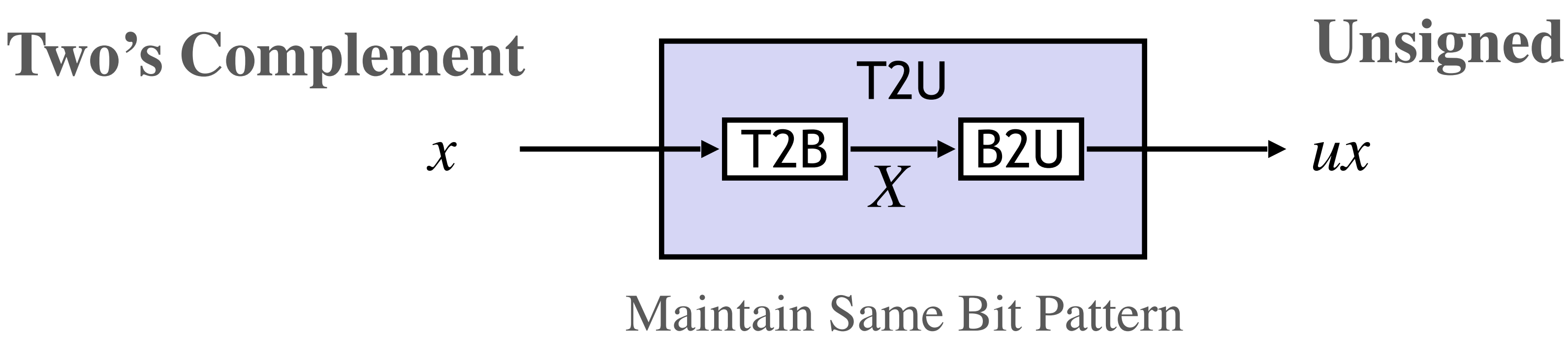
Interpretation of Bit Vectors

Mapping signed ↔ unsigned

- The computer itself has no idea if a given bit pattern at a particular location in memory "signed" or "unsigned".
- The program interprets some given bit pattern according to the *type* that value has been assigned.
- Moreover, mappings between unsigned and two's complement numbers keep the same bit representations but are interpreted differently depending on type, *which may yield a different value in your program*.



Mapping signed ↔ unsigned *con't*

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

+/- 2^w

Insights into overflow

- Lets say you have a signed char with the bit pattern…

```
01111111
```

- What is its value in two’s complement in decimal? How about unsigned?

Insights into overflow

- Lets say you have a signed char with the bit pattern…

```
01111111
```

- What is its value in two's complement in decimal? How about unsigned?

```
t: 127
```

```
u: 127
```

- Lets say 1 is added to 127. What is the bit pattern for 128?

Insights into overflow

- Lets say you have a signed char with the bit pattern…

```
01111111
```

- What is its value in two's complement in decimal? How about unsigned?

```
t: 127
```

```
u: 127
```

- Lets say 1 is added to 127. What is the bit pattern for 128?

```
10000000
```

- What is this bit pattern's value in two's complement in decimal? How about unsigned?

Insights into overflow

- Lets say you have a signed char with the bit pattern…

```
01111111
```

- What is its value in two's complement in decimal? How about unsigned?

```
t: 127
```

```
u: 127
```

- Lets say 1 is added to 127. What is the bit pattern for 128?

```
10000000
```

- What is this bit pattern's value in two's complement in decimal? How about unsigned?

```
t: -128
```

```
u: 128
```

- See *overflow.c*

It’s all a matter of interpretation

- The key idea so far here is that a bit pattern is just a bit pattern!!
  - It has no intrinsic value or semantics.
- How that bit pattern is ‘*interpreted*’ determines its value in your program.
- Ok, so how are bit patterns interpreted in programs?

It’s all a matter of interpretation

- The key idea so far here is that a bit pattern is just a bit pattern!!
  - It has no intrinsic value or semantics.
- How that bit pattern is ‘*interpreted*’ determines its value in your program.
- Ok, so how are bit patterns interpreted in programs?

Datatypes!

Conversion & Casting with Integers

Signed vs. unsigned in C

- **Constants**
  - By default are considered to be signed integers
  - If you want unsigned you must add a “U” suffix

```
unsigned int x = 0U;
unsigned int y = 4294967259U;
```

- **Casting**
  - *Explicit* casting between signed & unsigned

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

  - *Implicit* casting also occurs during assignments and function calls

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

Casting surprises

- If there is a mix of unsigned and signed in single expression, signed values are *implicitly cast to unsigned*
  - Includes expressions with comparison operators: <, >, ==, <=, >=
  - See *casting_surprise.c*
- There can also be unexpected results when working with array indices
  - See *array_surprise.c* and *array_surprise2.c*

Casting signed ↔ unsigned: summary

- When the coercion takes place the bit pattern is *maintained*
  - However the program will *reinterpret* its value!
  - Can have unexpected effects if not careful, as we just observed.
- Again, expressions containing signed and unsigned int…
  - signed integral is coerced to an unsigned integral!!

Signed 'extension'

- When we do a 'widening conversion' of a value via casting, what happens?

- In other words, given *w-bit* signed typed integer value x, convert it to *w+k-bit* typed integer with same value.
  - *w* is the number of bits in the type of x
    - ex. short = 16
  - *k* is the number of bits difference between the two types
    - ex. k of short vs int = 16

- Moreover, what happens in cases like this?

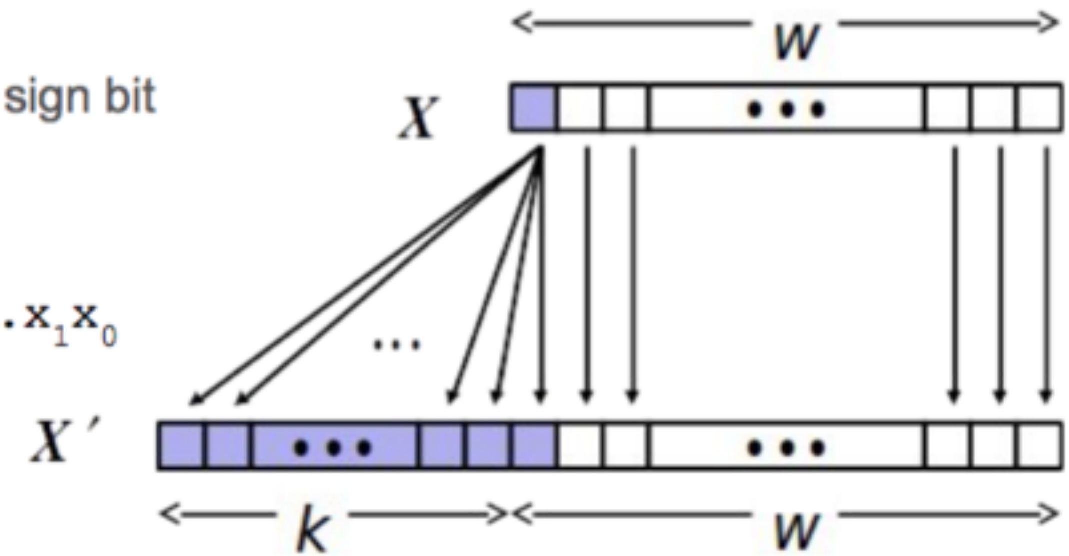
```
short x   =  15213;
int   ix  = (int) x;
short y   = -15213;
int   iy  = (int) y;
```

Signed ‘extension’ *con’t*

Solution: make k copies of the sign bit

$X = x_{w-1}\ x_{w-2} \ldots x_1 x_0$

$X' = \underbrace{x_{w-1} \ldots x_{w-1}}_{\leftarrow\ k\ \text{times}\ \rightarrow} x_{w-1}\ x_{w-2} \ldots x_1 x_0$



- Unsigned: zeros added
- Signed: sign bit extension
- Both yield intuitive and expected result

Signed ‘extension’ *con’t*

- Therefore, converting from smaller to larger integer data type C *automatically* performs sign extension

- Therefore, this code…

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

- …has the values….

	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

Truncation

- When we do a 'narrowing conversion' of a value via coercion or casting, what happens? (i.e. from 32-bit int to 16-bit short)

- Higher-order bits are *truncated*. Value is altered, will be reinterpreted.

- Might yield reasonable result if value is 'small enough' to fit in smaller type…

```
int i = 1;
short s = (short) i;
```

- But what about something like this?

```
short s = 256;
char c = (char) s;
```

- This non-intuitive behavior can lead to buggy code!

- See *coercion.c*

+ Summary

- **Extension** (e.g. short to int)
  - Unsigned: zeroes added
  - Signed: sign extension
  - Both yield expected results
- **Truncation** (e.g. unsigned short to unsigned int)
  - Unsigned/signed: Higher weighted bits are lopped off
  - Result must be reinterpreted
  - For ‘small numbers’ (e.g. int w/ value 16 into short), ok
  - For ‘large numbers’ (e.g. int w/ value 2^{20} into short), problematic.

+

Negation & Addition

Negation

- **Task**: given a bit-vector X compute -X
- **Solution**: -X = ~X + 1
  - Negating a value is done by computing its complement and adding 1
- **Example:**

$$X = 011001_2 = 25_{10}$$
$$\sim X = 100110_2 = -26_{10}$$
$$\sim X+1 = 100111_2 = -25_{10}$$

- Notice, therefore, that for any signed integral type x, *~x + x = -1*
  - See *negation.c*

Addition in base 2

- Very simple, works same as base 10, just remember..
  - 0 + 0 = 0
  - 0 + 1 = 1
  - 1 + 0 = 1
  - 1 + 1 = 10
- Examples:

```
  101        1011
 +101       +1011
  ——        ———
 1010       10110
```

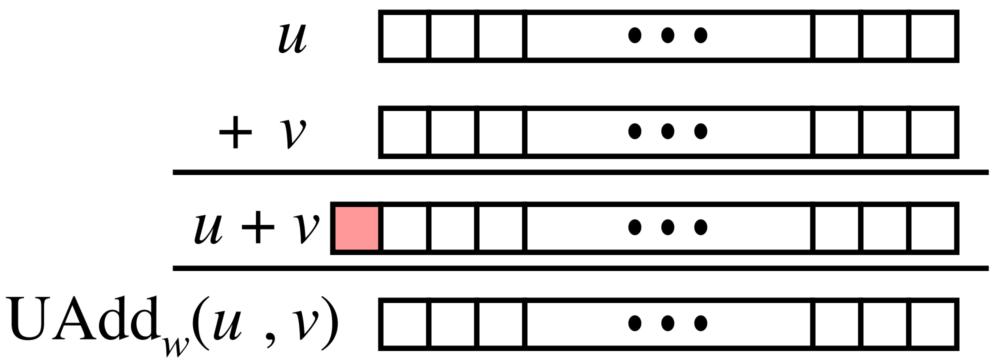
- Note in the second example that in the 2^1 column, we have 1 + (1 + 1), where the first 1 is "carried" from the 2^0 column.

Unsigned addition

Operands: w bits

True Sum: w+1 bits

Discard Carry: w bits



- However since types have a limited number of bits, any carry bits after the MSB simply get truncated.

$$
\begin{array}{rcl}
10010_2 & = & 18_{10} \\
+\ 11011_2 & = & 27_{10} \\
\hline
101101_2 & = & 45_{10} \\
\hline
01101_2 & = & 13_{10}
\end{array}
$$

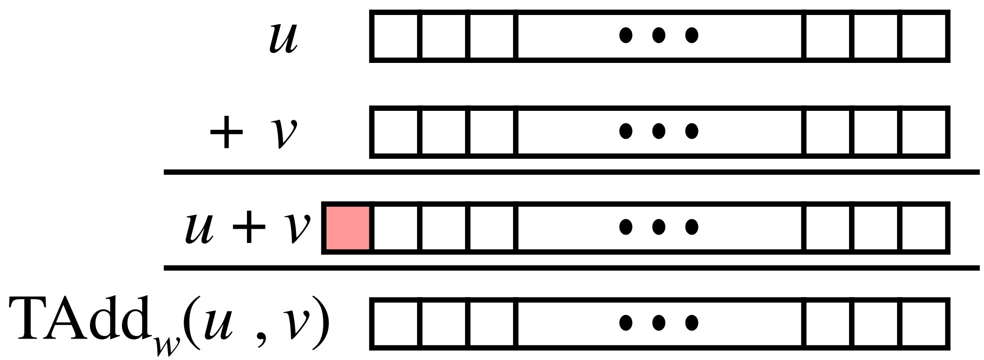
- See *unsigned_addition_overflow.c*

Signed addition

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits

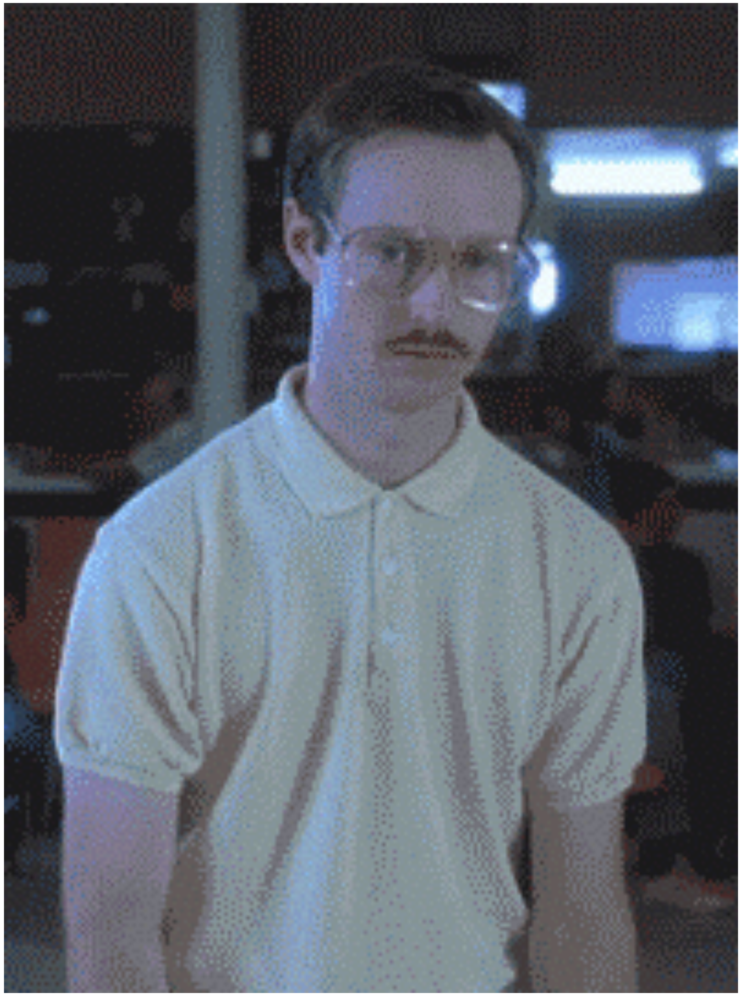


- **TAdd** and **UAdd** have identical bit-level behavior. If true sum requires w+1 bits, any carry bits after the MSB simply get truncated.

$$
\begin{array}{rcr}
10010_2 & = & -14_{10} \\
+\ 11011_2 & = & -5_{10} \\
\hline
101101_2 & = & -19_{10} \\
\hline
01101_2 & = & 13_{10}
\end{array}
$$

Signed addition *con't*

- One important notable difference!
  - If sum $\geq 2^{w-1}$, value becomes negative (overflow)
  - If sum $< -2^{w-1}$, value becomes positive (underflow)
- An now you can explain integer overflow to all your friends!!
- See *signed_addition_overflow.c*



+

Multiplication & Division

Multiplication

- **Task**: Computing exact product of w-bit numbers x, y (either signed or unsigned)

- **Range of Results:**

  - Unsigned multiplication requires *up to* 2*w bits to store result

$$0 \le x * y \le (2^{w} - 1)^{2} = 2^{2w} - 2^{w+1} + 1$$

  - Two's complement **min** possible value requires up to 2*w-1 bits

$$x * y \ge (-2^{w-1}) * (2^{w-1}-1) = -2^{2w-2} + 2^{w-1}$$

  - Two's complement **max** possible value requires up to 2*w bits

$$x * y \le (-2^{w-1})^{2} = 2^{2w-2}$$

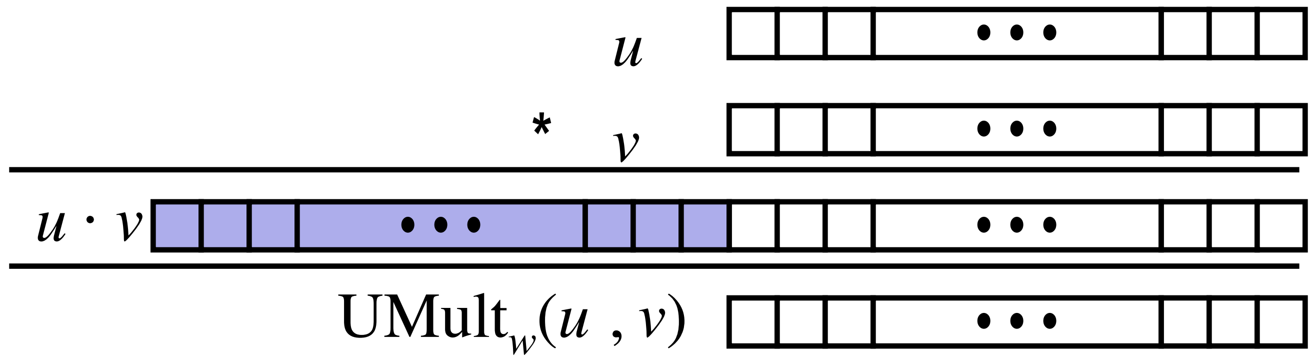
- Therefore, maintaining exact results would need to keep expanding size with each product computed.

Multiplication *con't*

Operands: *w* bits

True product: 2**w* bits

Discard *w* bits: *w* bits



- Similar to overflow in addition as standard multiplication function drops higher order bits
- Yields same non-intuitive results as overflow in addition.

Multiplication *con't*

- Maintaining exact results would require…
  - Keep expanding memory footprint with each product computed
  - This is expensive!
- Therefore, must be done in software
  - e.g., by "arbitrary precision" arithmetic packages
  - ex. Java's BigDecimal

Power-of-2 multiply with shifting

- Multiplication by a power of two is equivalent to the left shift operation.

  u << k == u * 2^k

- Examples

  u << 3 == u * 8

  (u << 5) – (u << 3) == u * 24

  (u + (u << 1)) << 2 == u * 12

- Most machines shift and add faster than multiply.

Generated multiplication code

- Compiler will convert some multiplication to shift operations during compilation process.Example, for something like this…

```
int multi_by_12(int x)
{
  return x * 12;
}
```

- The compiler generates something like this…

```
int multi_by_12(int x)
{
  int t = x + x*2
  return t << 2;
}
```

Unsigned power-of-2 division with shifting

- Unsigned integer division by a power of two is equivalent to right shift

$$\text{-u} \;>>\; \text{k} == \lfloor \text{u} / 2^{\text{k}} \rfloor$$

- Uses logical shift.
- With signed integers, when u is negative the results are rounded incorrectly.

	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`

Generated division code

- Again, a C compiler automatically generates shift/add code when dividing by constant.

- Example, for something like this…

```
unsigned int udiv(unsigned x)
{
  return x/8;
}
```

- The compiler generates something like this…

```
unsigned int udiv(unsigned x)
{
  return x >> 3;
}
```

Summary

- **Addition**:
  - Unsigned/signed: Normal addition followed by truncate, same operation on bit level
  - Unsigned: addition mod 2^w
    - Mathematical addition + possible subtraction of 2^w
  - Signed: modified addition mod 2^w (result in proper range)
    - Mathematical addition + possible addition or subtraction of 2^w
- **Multiplication**:
  - Unsigned/signed: Normal multiplication followed by truncate, same operation on bit level