Integers II

CS 105: Computer Systems
Lecture 03

Prof Melissa O'Neill

January 28, 2026

Learning Goals

- **Continue to reason about signed vs. unsigned ints**
- **Understand bit extension and truncation when casting integral data types**
- **Explain the effects of addition *overflow* for unsigned and signed ints**

Signed vs. unsigned `int` types in C

```
  /* signed, two's complement */

char                s8;
short               s16;
int                 s32;
long                s64;
long long           s64;
```

Bit sizes vary, these are for LP64. Use <stdint.h> if you want specific sizes.

```
  /* unsigned */

unsigned char        u8;
unsigned short      u16;
unsigned /* int */  u32;
unsigned long       u64;
unsigned long long  u64;
```


Rules: Signed vs. Unsigned in C

- **Decimal-Base Constants**
  - By default are considered to be signed "big enough" integers (int, long, long long)
  - Explicitly specify unsigned with "U" as suffix, e.g., **`0U, 4294967259u`**
- **Casting**
  - Explicit casting between signed & unsigned (Does not change the underlying bit representation!)
  - Implicit casting also occurs via assignments and function calls

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

```
int tx, ty;
unsigned ux, uy;
tx = ux; /* cast as signed */
uy = ty; /* cast as unsigned */
```

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

Your Notes…

Casting Surprises

If there is a mix of unsigned and signed in an expression, *signed values are implicitly cast to unsigned before evaluating*

- **Examples for *W* = 32:** **TMIN = -2,147,483,648 , TMAX = 2,147,483,647**
- **For each pair of constants**
  - Will the evaluation between them be **signed** or **unsigned**?
  - Given evaluation type, how do the constants relate to each other? **<, >, or ==**

	$Constant_1$	$Constant_2$	Evaluation	Relation
1.	0	0u	unsigned	==
2.	-1	0		
3.	-1	0u		
4.	2147483647	-2147483648		
5.	2147483647u	-2147483648		
6.	-1	-2		
7.	(unsigned) -1	-2		

Casting between integral data types
unsigned extension

Extension: casting from smaller to larger unsigned data type

```
unsigned char  ucx =  105;                /* 1 byte */
unsigned short usx = (unsigned short) ucx; /* 2 bytes */
unsigned int   uix = (unsigned) ucx;       /* 4 bytes */
```

- **What would you expect the value of `usx` to be? How about `uix`?**

- ***When going from smaller to larger unsigned data type:***
  ***Zero extension*: padding in front with zeroes**
  0110 1001 ➔ 0000 0000 0110 1001
  0110 1001 ➔ 0000 0000 0000 0000 0000 0000 0110 1001

Casting between integral data types
signed extension

Extension: casting from smaller to larger signed data type

```
char   cx =  105;           /* 1 bytes */
short  sx =  (short) cx;    /* 2 bytes */
int    ix = (int) cx;       /* 4 bytes */
```

- **What would you expect the values of `sx` and `ix` to be?**
- **Does zero extension work?**

- **Another example... Does zero extension work?**

```
char  cx = -1                  /* 1 byte */
short sx =  (short cx);        /* 2 bytes */
int   ix =  (int) cx;          /* 4 bytes */
```

Casting between integral data types
signed extension

Extension: casting from smaller to larger signed data type

```
char  cx = -1                    /* 1 byte */
short sx =  (short cx);          /* 2 bytes */
int   ix =  (int) cx;            /* 4 bytes */
```

- ***When going from smaller to larger signed data type:***
  Sign extension: **padding in front with msb**

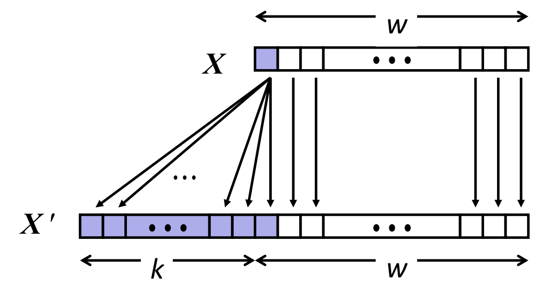
0110 1001 → 0000 0000 0000 0000 0000 0000 0110 1001

Vs.

1111 1111 → 1111 1111 1111 1111 1111 1111 1111 1111

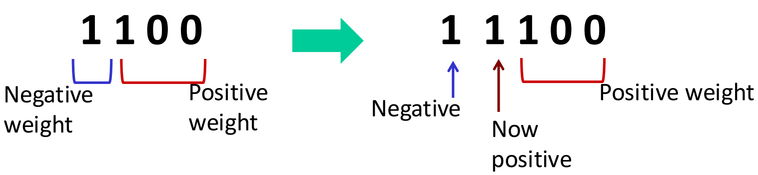
Sign Extension for signed integers

- **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: Intuition (4 bits → 5 bits)

- **Nonnegative**
  - Additional 0s in front do not add more weight!
  - Example: 0111 → 0 0111
- **Negative (recall** $B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$ **)**



- **New negative bit's value *two times* the new positive bit's value**
  - End up with (new negative) + (new positive) = (old negative)
  - In other words: $-2^w + 2^{w-1} = -2^{w-1}$

Exercise: An Extension Gotcha

- **What will this code print…**

```
unsigned char c = 0xFF;
unsigned int i = c << 24;
printf("Value is %x\n", i);
```

Casting between integral data types
unsigned truncation

Truncation: casting from larger to smaller unsigned data type

```
unsigned int   uix = U_MAX;                    /* 4 bytes U_max */
unsigned short usx = (unsigned short) uix;     /* 2 bytes */
unsigned char ucx = (unsigned char) uix        /* 1 byte */
```

- What would you expect the value of `usx` or `ucx` to be?

- ***When going from larger unsigned data type to smaller use truncation***
  - 1111 1111 1111 1111 1111 1111 1111 1111 → 1111 1111 1111 1111
    (4,294,967,295 $_{10}$) (65,535$_{10}$)
  - 1111 1111 1111 1111 1111 1111 1111 1111 → 1111 1111
    (4,294,967,295 $_{10}$) (255$_{10}$)

Truncation

- **Rule: to go from *w* bits to *k* bits, drop the top *w-k* bits**
  - Result is equivalent to zeroing out top bits (so they have no weight)

0 0 0 0 … 0 0 _ _ … _ _ _ _ _ _ _ _

w-k **k**

- **Same rule for unsigned and signed**
  - Interpret new bit pattern as either unsigned or signed

- **Impact on unsigned:**
  - Truncating integer A to *k* bits yields $A \bmod 2^k$

Exercise

- **Recall the rule: to go from *w* bits to *k* bits, drop the top *w-k* bits**

1. **First convert each of the <u>signed</u> integers into binary using <u>5 bits</u>:**

   a) 15

   b) -15

   c) 0

   d) 7

   e) -7

Exercise (cntd)

2. **For each of the integers in question 1, determine the decimal value when truncated to <u>4 bits</u> (again interpreting as signed integers)**

   a)

   b)

   c)

   d)

   e)

Truncation Impact: signed

- **Intuition for 5 bits → 4 bits**
  - Losing the MSB could either have no impact on original value (reverse of sign extension)
  - Or could yield integer with value +/- 2^4
- **Signed truncation**
  - In general, first treat bit pattern as an unsigned integer to yield **u mod 2^k**, then interpret result as signed

Summary:
Expanding, Truncating: Basic Rules

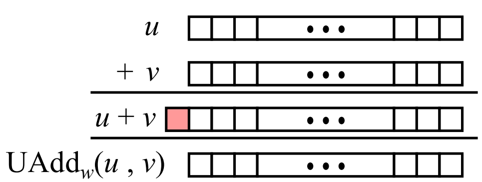
- **Expanding (e.g., short int to int)**
  - Unsigned: zeros added
  - Signed: sign extension
  - Both yield same value as original
- **Truncating (e.g., unsigned to unsigned short)**
  - Unsigned/signed: bits are truncated
    - Result reinterpreted
  - For small numbers yields expected behavior
  - For large magnitude unsigned performs modulo arithmetic
  - For large magnitude signed can change value substantially — **UB**

Unsigned Addition

Operands: w bits

True Sum: w+1 bits

Discard Carry: w bits



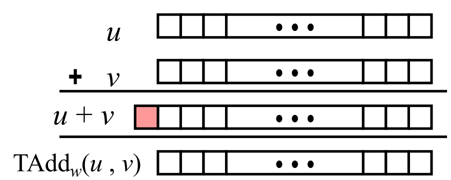
- **Standard Addition Function**
  - Ignores carry output

Two's Complement Addition

Operands: w bits

True Sum: w+1 bits

Discard Carry: w bits



- **TAdd and UAdd have Identical Bit-Level Behavior**
  - Signed vs. unsigned addition in C:

```
int s, t, u, v;
s = (int) ((unsigned) u + (unsigned) v);
t = u + v
```

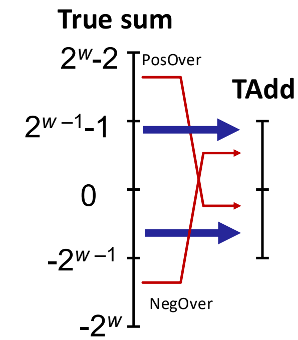
Will give `s == t`

Exercise

- **Assume you are using a 4-bit word (signed, two's complement)**
  1. Add 7 and 1
  2. Add -8 and -8
  3. Add -5 and 3
- **Which ones didn't "work"? Is carry out information enough to detect issues?**

Two's-Complement Overflow (intuition)

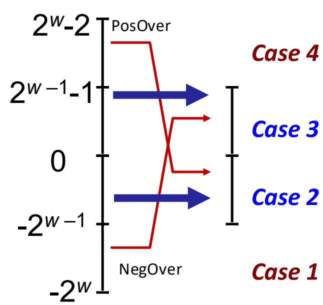
- **The true sum of two w-bit 2's complement numbers, u and v, may require w+1 bits**
  - Can we have an overflow if $u<0$ and $v\geq 0$?
- ***PosOver*: true sum of *u* and *v* is > $2^{W-1} - 1$**
- ***NegOver*: true sum of *u* and *v* is < 2^{W-1}**



Detecting Two's-Complement Overflow

- **Detecting overflow**
  - Given:

```
int s, u, v;
s = u + v;
```



  - Overflow iff either:

$u, v < 0, s \geq 0$ (Case 1: NegOver) → $u + v + 2^w$

$u, v \geq 0, s < 0$ (Case 4: PosOver) → $u + v - 2^w$

Multiplication and Division

- **Multiplication and division are slower than +/-, bit-ops**
  - Multiplication is a bit slower (e.g., 3 cycles latency, 1 cycle throughput)
  - Division is a ***lot*** slower (e.g., 25 cycles latency, 25 cycles throughput)
- **Compare with shifting for powers of 2**
  - `u << k` gives `u *` 2^k
    - both signed and unsigned
  - `u >> k` gives $\lfloor$ `u /` $2^k \rfloor$
    - For unsigned; special consideration for signed (for negative values)
- **Impact**
  - Multiplication: truncate high order bits
  - Division: integer division should round toward zero… implications for *signed* division?