CS429: Computer Organization and Architecture

Integers

Dr. Bill Young
Department of Computer Science
University of Texas at Austin

Last updated: June 10, 2019 at 14:22

Topics of this Slideset

- Numeric Encodings: Unsigned and two's complement
- Programming Implications: C promotion rules
- Basic operations:
  - addition, negation, multiplication
  - Consequences of overflow
  - Using shifts to perform power-of-2 multiply/divide

C Puzzles

```
int x = foo();
int y = bar();
unsigned ux = x;
unsigned uy = y;
```

- Assume a machine with 32-bit, two's complement integers.
- For each of the following, either:
  - Argue that is true for all argument values;
  - Give an example where it's not true.

```
x < 0              → ((x*2) < 0
ux >= 0
(x & 7) == 7       → (x<<30) < 0
ux > -1
x > y              → -x < -y
x * x >= 0
x > 0 && y > 0     → x + y > 0
x >= 0             → -x <= 0
x <= 0             → -x >= 0
```


Encoding Integers: Unsigned

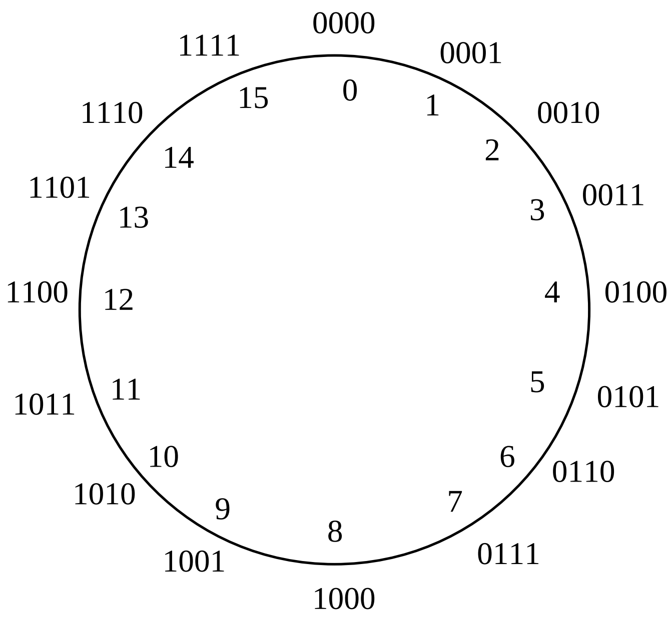
For unsigned integers, we treat all values as non-negative and use *positional notation* as with non-negative decimal numbers.

Positional Value	128	64	32	16	8	4	2	1
Binary Number	1	1	1	0	1	0	0	1
Calculate	1x128	1x64	1x32	0x16	1x8	0x4	0x2	1x1
Add them	128	+64	+32	+0	+8	+0	+0	+1
Result	233							

Assume we have a w length bit string X.

$$\textbf{Unsigned:}\ \mathrm{B2U}_w(X) = \sum_{i=0}^{w-1} X_i \times 2^i$$

Unsigned Integers: 4-bit System



Encoding Integers: Two's Complement

Two's complement is a way of encoding integers, including some positive and negative values. It's exactly like unsigned except *the high order bit is given negative weight.*

Two's complement: $\mathrm{B2T}_w(X) = -X_{w-1} \times 2^{w-1} + \sum_{i=0}^{w-2} X_i \times 2^i$

Decimal	Hex	Binary
15213	3B 6D	00111011 01101101
-15213	C4 93	11000100 10010011

Sign Bit:
For 2's complement, the most significant bit indicates the sign.

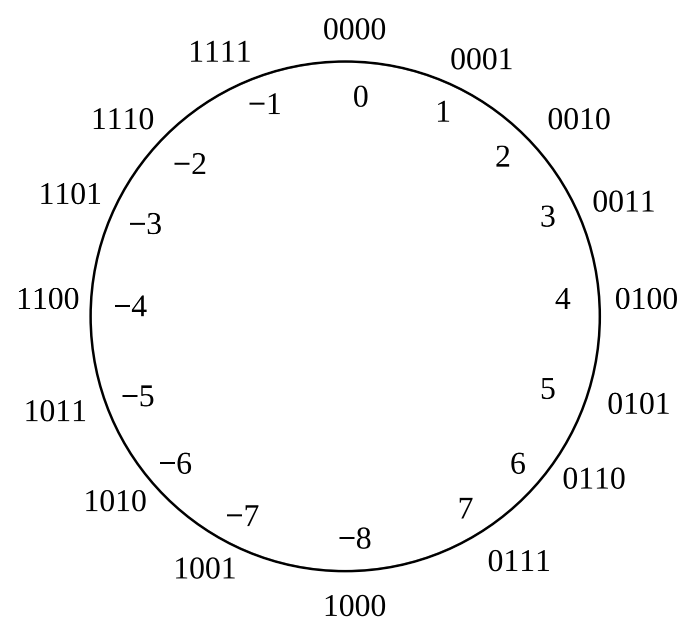
- 0 for nonnegative
- 1 for negative

Encoding Example

```
x =    15213:  00111011 01101101
y =   −15213:  11000100 10010011
```

Weight	15213		-15213	
1	1	1	1	1
2	0	0	1	2
4	1	4	0	0
8	1	8	0	0
16	0	0	1	16
32	1	32	0	0
64	1	64	0	0
128	0	0	1	128
256	1	256	0	0
512	1	512	0	0
1024	0	0	1	1024
2048	1	2048	0	0
4096	1	4096	0	0
8192	1	8192	0	0
16384	0	0	1	16384
-32768	0	0	1	-32768
Sum	**15213**		**-15213**	

Signed Integers: 4-bit System



Numeric Ranges

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

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	65,525	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

- $|TMin| = TMax + 1$
- $UMax = 2 \times TMax + 1$

C Programming

```
#include <limits.h>
```

Declares various constants: `ULONG_MAX`, `LONG_MAX`, `LONG_MIN`, etc. *The values are platform-specific.*

Unsigned and Signed Numeric Values

Equivalence: Same encoding for nonnegative values

Uniqueness:

- Every bit pattern represents a unique integer value
- Each representable integer has unique encoding

Can Invert Mappings:

- inverse of B2U(X) is U2B(X)
- inverse of B2T(X) is T2B(X)

X	B2U(X)	B2T(X)
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

Casting Signed to Unsigned

C allows conversions from signed to unsigned.

```
short int          x = 15213;
unsigned short ux = (unsigned short) x;
short int          y = -15213;
unsigned short uy = (unsigned short) y;
```

Resulting Values:

- *The bit representation stays the same.*
- Nonnegative values are unchanged.
- Negative values change into (large) positive values.

Signed vs Unsigned in C

Constants

- By default, constants are considered to be signed integers.
- They are unsigned if they have "U" as a suffix: `0U`, `4294967259U`.

Casting

- Explicit casting between signed and unsigned is the same as U2T and T2U:

```
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;
```


Casting Surprises

Expression Evaluation

- If you mix unsigned and signed in a single expression, signed values implicitly cast to unsigned.
- This includes when you compare using `<`, `>`, `==`, `<=`, `>=`.

Const 1	Const 2	Rel.	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

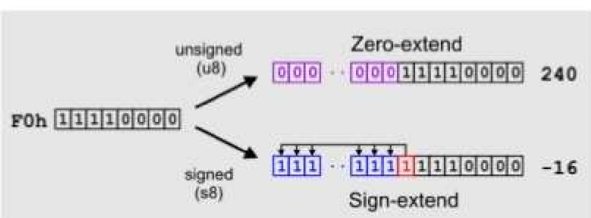
Sign Extension

Task: Given a w-bit signed integer x, convert it to a w+k-bit integer with the *same value*.

Rule: Make k copies of the sign bit :

$$x' = x_{w-1}, \ldots x_{w-1}, x_{w-2}, \ldots, w_0$$

Why does this work?



- zero-extend extends unsigned values to wider mode
- sign-extend extends signed values to wider mode

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

In converting from smaller to larger signed integer data types, C automatically performs sign extension.

Why Use Unsigned?

Don't use just to ensure numbers are nonzero.

- Some C compilers generate less efficient code for unsigned.

```
unsigned i;
for (i=1; i < cnt; i++)
    a[i] += a[i-1]
```

- It's easy to make mistakes.

```
for (i = cnt-2; i >= 0; i--)
    a[i] += a[i+1]
```

Do use when performing modular arithmetic.

- multiprecision arithmetic
- other esoteric stuff

Do use when you need extra bits of range.

Negating Two's Complement

To find the negative of a number in two's complement form: complement the bit pattern and add 1:

$$\sim x + 1 = -x$$

Example:

10011101 = 0x9C = -99_{10}

complement:

01100010 = 0x62 = 98_{10}

add 1:

01100011 = 0x63 = 99_{10}

Try it with: 11111111 and 00000000.

Complement and Increment Examples

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
~x	-15214	C4 92	11000100 10010010
~x+1	-15213	C4 93	11000100 10010011
0	0	00 00	00000000 00000000
~0	-1	FF FF	11111111 11111111
~0+1	0	00 00	00000000 00000000

Unsigned Addition

Given two w-bit unsigned quantities u, v, the true sum may be a w+1-bit quantity.

Discard the carry bit and treat the result as an unsigned integer.

Thus, unsigned addition implements **modular addition**.

$$\mathrm{UAdd}_w(u, v) = (u + v) \bmod 2^w$$

$$\mathrm{UAdd}_w(u, v) = \begin{cases} u + v & u + v < 2^w \\ u + v - 2^w & u + v \geq 2^w \end{cases}$$

Detecting Unsigned Overflow

Task:
Determine if $s = \text{UAdd}_w(u, v) = u + v$.

Claim: We have overflow iff:

$$s < u \textit{ or } s < v.$$

BTW: $s < u$ *iff* $s < v$. So it's OK to check only one of these conditions because both will be true when there's an overflow.

On the machine, this causes the **carry flag** to be set.

Properties of Unsigned Addition

W-bit unsigned addition is:

- Closed under addition:
  $$0 \leq \text{UAdd}_w(u, v) \leq 2^w - 1$$
- Commutative
  $$\text{UAdd}_w(u, v) = \text{UAdd}_w(v, u)$$
- Associative
  $$\text{UAdd}_w(t, \text{UAdd}_w(u, v)) = \text{UAdd}_w(\text{UAdd}_w(t, u), v)$$
- 0 is the additive identity
  $$\text{UAdd}_w(u, 0) = u$$
- Every element has an additive inverse
  Let $\text{UComp}_w(u) = 2^w - u$, then
  $$\text{UAdd}_w(u, \text{UComp}_w(u)) = 0$$

Two's Complement Addition

Given two w-bit signed quantities u, v, the true sum may be a w+1-bit quantity.

Discard the carry bit and treat the result as a two's complement number.

$$\text{TAdd}_w(u, v) = \begin{cases} u + v + 2^w & u + v < \text{TMin}_w \textbf{ (NegOver)} \\ u + v & \text{TMin}_w \leq u + v \leq \text{TMax}_w \\ u + v - 2^w & \text{TMax}_w < u + v \textbf{ (PosOver)} \end{cases}$$

Two's Complement Addition

TAdd and UAdd have identical bit-level behavior.

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

This will give `s == t`.

Detecting 2's Complement Overflow

Task:
Determine if $s = \text{TAdd}_w(u, v) = u + v$.

Claim: We have overflow iff either:

- $u, v < 0$ but $s \geq 0$ (NegOver)
- $u, v \geq 0$ but $s < 0$ (PosOver)

Can compute this as:

```
ovf = (u<0 == v<0) && (u<0 != s<0);
```

On the machine, this causes the **overflow flag** to be set.

Why don't we have to worry about the case where one input is positive and one negative?

Properties of TAdd

TAdd is Isomorphic to UAdd.
This is clear since they have identical bit patterns.

$$\text{Tadd}_w(u, v) = \text{U2T}(\text{UAdd}_w(\text{T2U}(u), \text{T2U}(v)))$$

Two's Complement under TAdd forms a group.

- Closed, commutative, associative, 0 is additive identity.
- Every element has an additive inverse:

  Let $\text{TComp}_w(u) = \text{U2T}(\text{UComp}_w(\text{T2U}(u))$, then $\text{TAdd}_w(u, \text{UComp}_w(u)) = 0$

$$\text{TComp}_w(u) = \begin{cases} -u & u \neq \text{TMin}_w \\ \text{TMin}_w & u = \text{TMin}_w \end{cases}$$

Multiplication

Computing the exact product of two w-bit numbers x, y. This is the same for both signed and unsigned.

Ranges:

- Unsigned: $0 \leq x * y \leq (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$, requires up to $2w$ bits.
- Two's comp. min: $x * y \geq (-2^{w-1}) * (2^{w-1} - 1) = -2^{2w-2} + 2^{w-1}$, requires up to $2w - 1$ bits.
- Two's comp. max: $x * y \leq (-2^{w-1})^2 = 2^{2w-2}$, requires up to $2w$ (but only for TMin_w^2).

Maintaining the exact result

- Would need to keep expanding the word size with each product computed.
- Can be done in software with "arbitrary precision" arithmetic packages.

Unsigned Multiplication in C

Given two w-bit unsigned quantities u, v, the true sum may be a 2w-bit quantity.

We just discard the most significant w bits, treat the result as an unsigned number.

Thus, unsigned multiplication implements **modular multiplication**.

$$\text{UMult}_w(u, v) = (u \times v) \bmod 2^w$$

Unsigned vs. Signed Multiplication

Unsigned Multiplication

```
unsigned ux = (unsigned) x;
unsigned uy = (unsigned) y;
unsigned up = ux * uy;
```

- Truncates product to w-bit number: $up = \text{UMult}_w(ux, uy)$
- Modular arithmetic: $up = (ux \cdot uy) \bmod 2^w$

Two's Complement Multiplication

```
int x, y;
int p = x * y;
```

- Compute exact product of two w-bit numbers x, y.
- Truncate result to w-bit number: $p = \text{TMult}_w(x, y)$

Unsigned vs. Signed Multiplication

Unsigned Multiplication

```
unsigned ux = (unsigned) x;
unsigned uy = (unsigned) y;
unsigned up = ux * uy;
```

Two's Complement Multiplication

```
int x, y;
int p = x * y;
```

Relation

- Signed multiplication gives same bit-level result as unsigned.
- `up == (unsigned) p`

Multiply with Shift

A left shift by k, is equivalent to multiplying by 2^k. This is true for both signed and unsigned values.

$$
\begin{aligned}
\texttt{u << 1} &\rightarrow u \times 2\\
\texttt{u << 2} &\rightarrow u \times 4\\
\texttt{u << 3} &\rightarrow u \times 8\\
\texttt{u << 4} &\rightarrow u \times 16\\
\texttt{u << 5} &\rightarrow u \times 32\\
\texttt{u << 6} &\rightarrow u \times 64
\end{aligned}
$$

Compilers often use shifting for multiplication, since shift and add is much faster than multiply (on most machines).

`u << 5 - u << 3 == u * 24`

Aside: Floor and Ceiling Functions

Two useful functions on real numbers are the *floor* and *ceiling* functions.

Definition: The floor function $\lfloor r \rfloor$, is the greatest integer less than or equal to r.

$$\lfloor 3.14 \rfloor = 3$$
$$\lfloor -3.14 \rfloor = -4$$
$$\lfloor 7 \rfloor = 7$$

Definition: The ceiling function $\lceil r \rceil$, is the smallest integer greater than or equal to r.

$$\lceil 3.14 \rceil = 4$$
$$\lceil -3.14 \rceil = -3$$
$$\lceil 7 \rceil = 7$$

Unsigned Divide by Shift

A right shift by k, is (approximately) equivalent to dividing by 2^k, but the effects are different for the unsigned and signed cases.

Quotient of unsigned value by power of 2.

$$\texttt{u >> k} == \lfloor u/2^k \rfloor$$

Uses logical shift.

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

Signed Divide by Shift

Quotient of signed value by power of 2.

$$\texttt{u >> k} == \lfloor u/2^k \rfloor$$

- Uses arithmetic shift. What does that mean?
- Rounds in wrong direction when $u < 0$.

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

Correct Power-of-2 Division

We've seen that right shifting a negative number gives the wrong answer because it rounds away from 0.

$$\texttt{x >> k} == \lfloor x/2^k \rfloor$$

We'd really like $\lceil x/2^k \rceil$ instead.

You can compute this as: $\lfloor (x + 2^k - 1)/2^k \rfloor$. In C, that's:

```
(x + (1<<k) -1) >> k
```

This biases the dividend toward 0.

Properties of Unsigned Arithmetic

Unsigned multiplication with addition forms a **Commutative Ring.**

- Addition is commutative
- Closed under multiplication

$$0 \leq \text{UMult}_w(u, v) \leq 2^w - 1$$

- Multiplication is commutative

$$\text{UMult}_w(u, v) = \text{UMult}_w(v, u)$$

- Multiplication is associative

$$\text{UMult}_w(t, \text{UMult}_w(u, v)) = \text{UMult}_w(\text{UMult}_w(t, u), v)$$

- 1 is the multiplicative identity

$$\text{UMult}_w(u, 1) = u$$

- Multiplication distributes over addition

$$\text{UMult}_w(t, \text{UAdd}_w(u, v)) = \text{UAdd}_w(\text{UMult}_w(t, u), \text{UMult}_w(t, v))$$

Isomorphic Algebras

- Unsigned multiplication and addition: truncate to w bits
- Two's complement multiplication and addition: truncate to w bits

Both form rings isomorphic to ring of integers mod 2^w
Comparison to Integer Arithmetic

- Both are rings
- Integers obey ordering properties, e.g.
  $$u > 0 \rightarrow u + v > v$$
  $$u > 0, v > 0 \rightarrow u \cdot v > 0$$
- These properties are not obeyed by two's complement arithmetic.
  `TMax + 1 == TMin`
  `15213 * 30426 == -10030` (for 16-bit words)

Assume a machine with 32-bit word size, two's complement integers.

```
int x = foo();
int y = bar();
unsigned ux = x;
unsigned uy = y;
```

`x < 0`	$\rightarrow$ `((x*2) < 0`	False: TMin
`ux >= 0`		True: $0 =$ UMin
`(x & 7) == 7`	$\rightarrow$ `(x<<30) < 0`	True: $x_1 = 1$
`ux > -1`		False: 0
`x > y`	$\rightarrow$ `-x < -y`	False: -1, TMin
`x * x >= 0`		False: 30426
`x > 0 && y > 0`	$\rightarrow$ `x + y > 0`	False: TMax, TMax
`x >= 0`	$\rightarrow$ `-x <= 0`	True: -TMax < 0
`x <= 0`	$\rightarrow$ `-x >= 0`	False: TMin