Module II
Data Representation

Dr. Arijit Roy
Computer Science and Engineering Group
Indian Institute of Information Technology Sri City

Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - **Representation: unsigned and signed**
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
- Summary

Encoding Integers

w : Length
x: either 0 or 1
2: As we are considering binary value
= : LHS is the same as RHS

Unsigned

$$B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

Two's Complement

$$B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

Sign Bit

```
short int x =  15213;
short int y = -15213;
```

The most common computer representation of signed numbers is known as two's complement

- C short 2 bytes long

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

- Sign Bit
  - For 2's complement, most significant bit indicates sign
    - 0 for nonnegative
    - 1 for negative

Encoding Example (Cont.)

```
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**

Typical 32-bit Program

C data type	Minimum	Maximum
`char`	−128	127
`unsigned char`	0	255
`short [int]`	−32,768	32,767
`unsigned short [int]`	0	65,535
`int`	−2,147,483,648	2,147,483,647
`unsigned [int]`	0	4,294,967,295
`long [int]`	−2,147,483,648	2,147,483,647
`unsigned long [int]`	0	4,294,967,295
`long long [int]`	−9,223,372,036,854,775,808	9,223,372,036,854,775,807
`unsigned long long [int]`	0	18,446,744,073,709,551,615

Figure 2.8 **Typical ranges for C integral data types on a 32-bit machine.** Text in square brackets is optional.

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
- Other Values
  - Minus 1

    111…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	65,535	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
  - $|TMin| \;=\; TMax + 1$
    - Asymmetric range
  - $UMax \;=\; 2 * TMax + 1$

- **C Programming**
  - #include <limits.h>
  - Declares constants, e.g.,
    - ULONG_MAX
    - LONG_MAX
    - LONG_MIN
  - Values platform specific

- Asymmetric: There is no positive counter part to *TMin*
- 0 is non negative, therefore, considered in the set of positive numbers

Unsigned & Signed Numeric Values

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

- Equivalence
  - Same encodings for nonnegative values
- Uniqueness
  - Every bit pattern represents unique integer value
  - Each representable integer has unique bit encoding

Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
    - Representation: unsigned and signed
    - **Conversion, casting**
    - Expanding, truncating
    - Addition, negation, multiplication, shifting
- Summary

Signed numbers

- Computer programs calculate both positive and negative numbers, so we need a representation that distinguishes the positive from the negative.
- Add a separate sign, which conveniently can be represented in a single bit; the name for this representation is ***sign and magnitude***.
  - where to put the sign bit. To the right? To the left ?
  - adders for sign and magnitude may need an extra step to set the sign because we can’t know in advance what the proper sign will be.
  - separate sign bit means that sign and magnitude has both a positive and a negative zero

2s-Complement Signed Integers

- Given an n-bit number

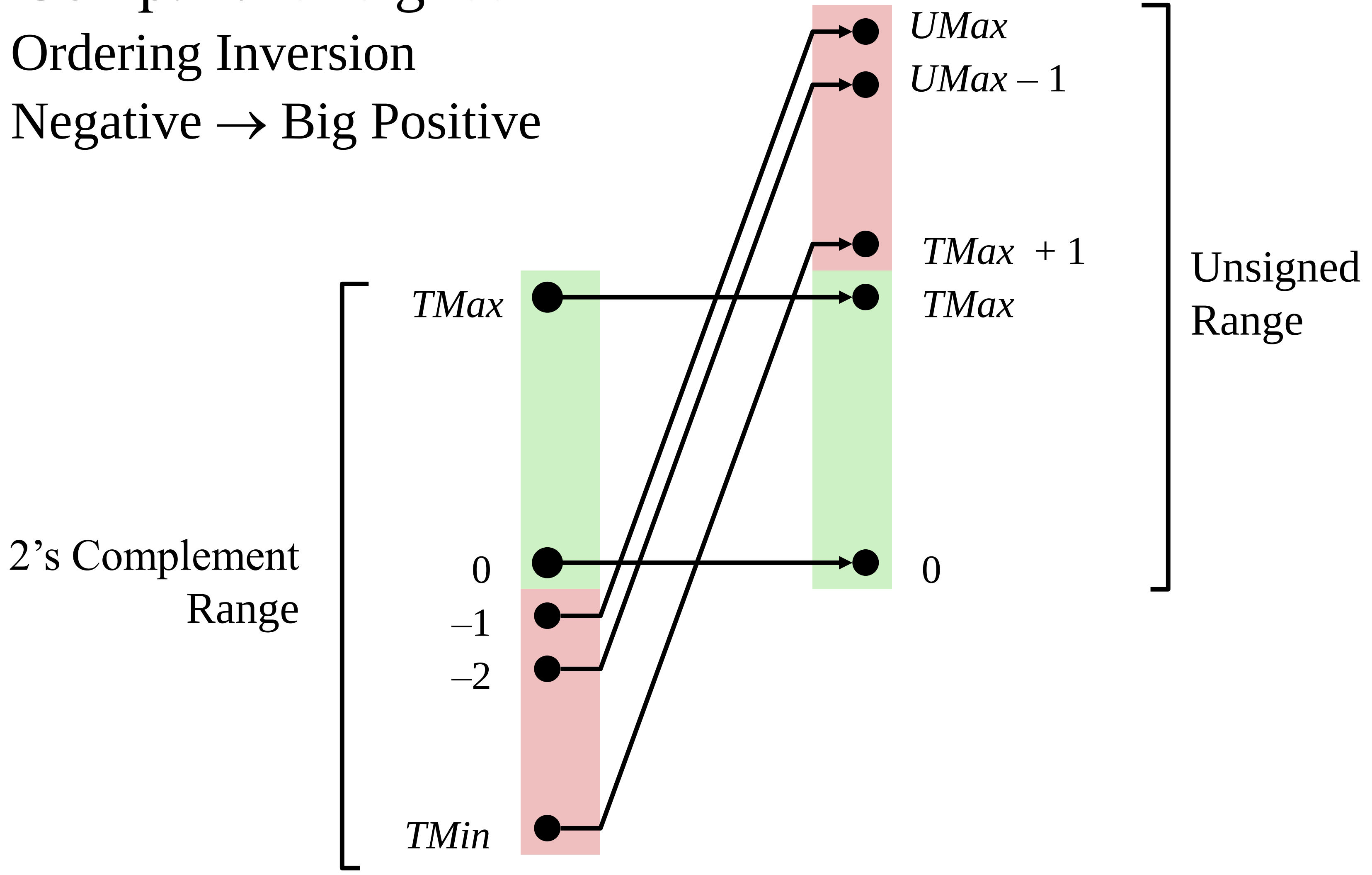
$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- Range: -2^{n-1} to $+2^{n-1} - 1$
- Example
  - $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
    $= -1\times2^{31} + 1\times2^{30} + \ldots + 1\times2^{2} + 0\times2^{1} + 0\times2^{0}$
    $= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$
- Using 32 bits
  - $-2{,}147{,}483{,}648$ to $+2{,}147{,}483{,}647$

leading 0s mean positive, and leading 1s mean negative

Conversion Visualized

- 2’s Comp. → Unsigned
  - Ordering Inversion
  - Negative → Big Positive



2s-Complement Signed Integers

- Bit 31 is sign bit
  - 1 for negative numbers
  - 0 for non-negative numbers

- Non-negative numbers have the same unsigned and 2s-complement representation
- Some specific numbers
  - 0: 0000 0000 … 0000
  - –1: 1111 1111 … 1111
  - Most-negative: 1000 0000 … 0000
  - Most-positive: 0111 1111 … 1111

EXAMPLE

Binary to Decimal Conversion

What is the decimal value of this 32-bit two's complement number?

$$1111\ \ 1111\ \ 1111\ \ 1111\ \ 1111\ \ 1111\ \ 1111\ \ 1100_{\text{two}}$$

ANSWER

Substituting the number's bit values into the formula above:

$$(1 \times -2^{31}) + (1 \times 2^{30}) + (1 \times 2^{29}) + \ldots + (1 \times 2^{1}) + (0 \times 2^{1}) + (0 \times 2^{0})$$
$$= -2^{31} + 2^{30} + 2^{29} + \ldots + 2^{2} + 0 + 0$$
$$= -2{,}147{,}483{,}648_{\text{ten}} + 2{,}147{,}483{,}644_{\text{ten}}$$
$$= -4_{\text{ten}}$$

[illegible]

Signed Negation - shortcut

- Complement and add 1
  - Complement means 1 → 0, 0 → 1

$$x + \overline{x} = 1111\ldots111_2 = -1$$

$$\overline{x} + 1 = -x$$

- Example: negate +2
  - $+2 = 0000\ 0000\ \ldots\ 0010_2$
  - $-2 = 1111\ 1111\ \ldots\ 1101_2 + 1$
    $= 1111\ 1111\ \ldots\ 1110_2$

Signed vs. Unsigned in C

- Constants
  - By default are considered to be **signed** integers
  - Unsigned if have "U" as suffix

```
0U, 4294967259U
```

C supports both signed and unsigned arithmetic

- Casting
  - Explicit casting between signed & unsigned same as U2T and T2U

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

```
float a = 1.2;
int b = a; //Compiler will throw an error for this
int b = (int)a + 1;
```

  - Implicit casting also occurs via assignments and procedure calls

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

Implicit type casting indicates the conversion of one data type to another, where the original meaning will be the same after conversion.

```
short    int     v  = -12345;
unsigned short uv = (unsigned short) v;
printf("v = %d, uv = %u\n", v, uv);
```

When run on a two's-complement machine, it generates the following output:

```
v = -12345, uv = 53191
```

```
int x = -1;
unsigned u = 2147483648; /* 2 to the 31st */

printf("x = %u = %d\n", x, x);
printf("u = %u = %d\n", u, u);
```

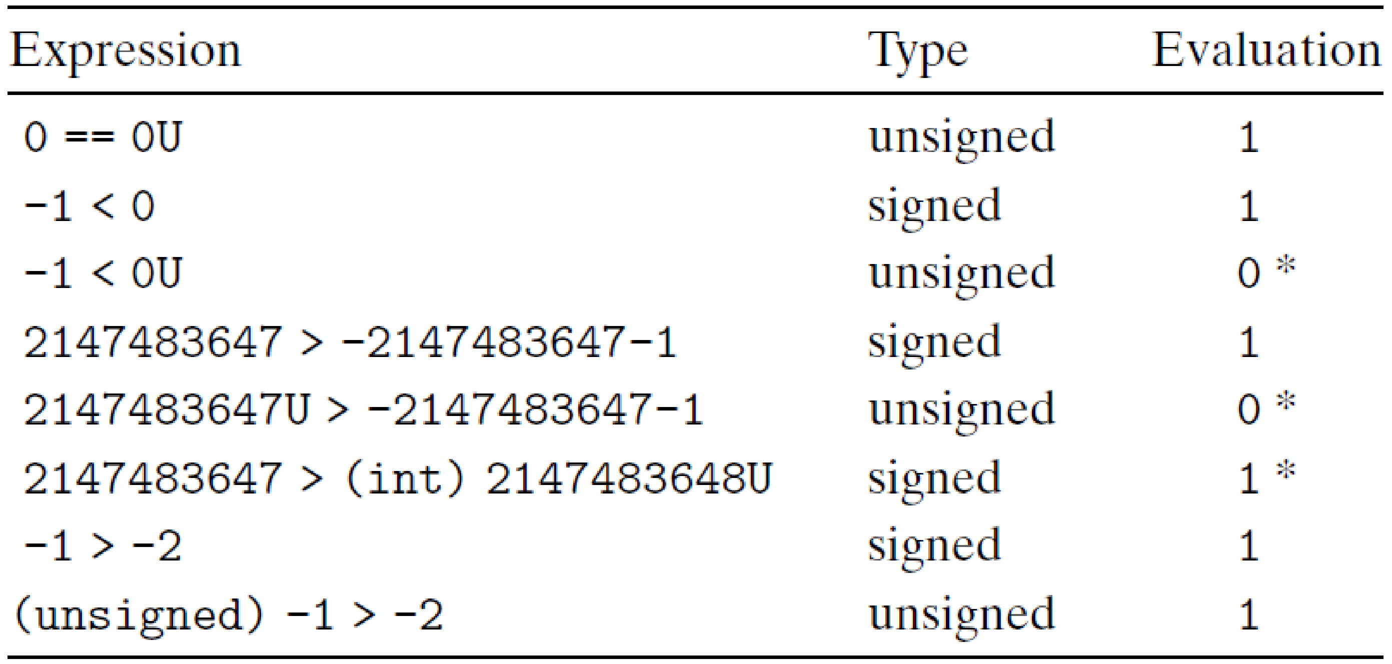
When run on a 32-bit machine, it prints the following:

```
x = 4294967295 = -1
u = 2147483648 = -2147483648
```

In both cases, `printf` prints the word first as if it represented an unsigned number, and second as if it represented a signed number. We can see the conversion routines in action: $T2U_{32}(-1) = UMax_{32} = 2^{32} - 1$ and $U2T_{32}(2^{31}) = 2^{31} - 2^{32} = -2^{31} = TMin_{32}$.

Casting Surprises

When an operation is performed where one operand is signed and the other is unsigned, C implicitly casts the signed argument to unsigned and performs the operations assuming the numbers are nonnegative.



Expression	Type	Evaluation
`0 == 0U`	unsigned	1
`-1 < 0`	signed	1
`-1 < 0U`	unsigned	0 *
`2147483647 > -2147483647-1`	signed	1
`2147483647U > -2147483647-1`	unsigned	0 *
`2147483647 > (int) 2147483648U`	signed	1 *
`-1 > -2`	signed	1
`(unsigned) -1 > -2`	unsigned	1

Figure 2.18 **Effects of C promotion rules.** Nonintuitive cases marked by '*'.

Code Security Example

User can read from

Length of the buffer allocate by the user

```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Declaration of library function memcpy */
void *memcpy(void *dest, void *src, size_t n);

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

Designed to copy a specific number of bytes n from one region of memory to another

Designed to copy some of the data maintained by the OS Kernel to a designated region of the memory accessible to the users

- Similar to code found in FreeBSD's Open-source OS implementation of *getpeername*

A library function

- There are legions of smart people trying to find vulnerabilities in programs

Typical Usage

```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

```
#define MSIZE 528

void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, MSIZE);
    printf("%s\n", mybuf);
}
```

No more file are copied than are available in the source or the destination buffer

Malicious Usage

3 Data type for n is **`size_t`**

```
/* Declaration of library function memcpy */
void *memcpy(void *dest, void *src, size_t n);
```

2 The value is computed for **`len`** which will then passed as parameter **n** to **memcpy**

```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

```
#define MSIZE 528

void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, -MSIZE);
    . . .
}
```

1 Malicious programmer use with negative value

4 **`size_t`** is declared using typedef in **`stdio.h`** library file. Defined to be **`unsigned`** for 32 bit and **`unsigned long`** for 64 bits program

5 Since argument **`n`** is **`unsigned`**, **`memcpy`** will treat it as a very large number and will copy more number of bytes from the kernel to the buffer

Summary
Casting Signed ↔ Unsigned: Basic Rules

- Bit pattern is maintained
- But reinterpreted
- Can have unexpected effects: adding or subtracting 2^w

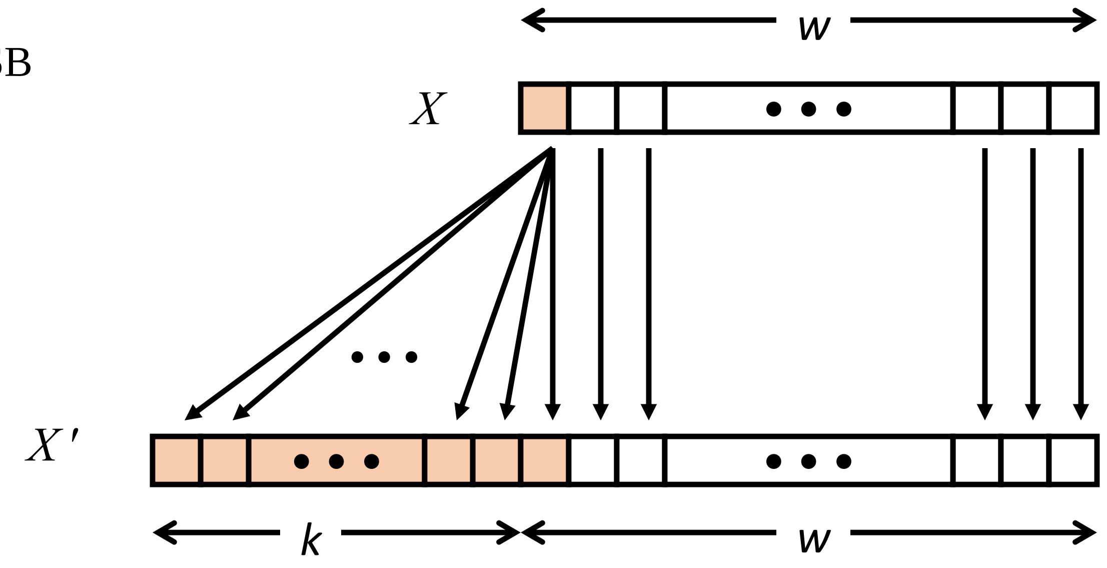
- Expression containing signed and unsigned int
  - int is cast to unsigned!!

Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - **Expanding, truncating**
  - Addition, negation, multiplication, shifting
- Summary

Sign Extension

- 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 Example

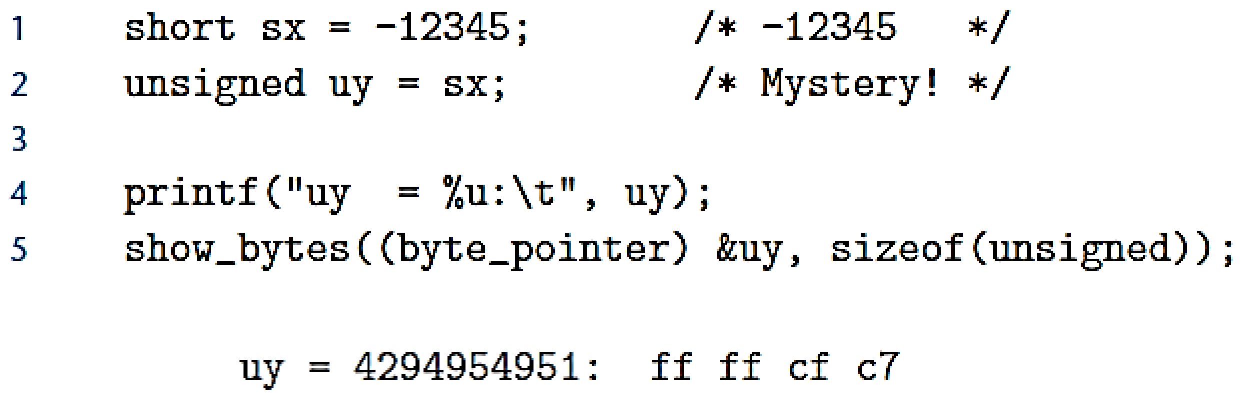
```c
short sx = -12345;         /* -12345 */
unsigned short usx = sx; /*  53191 */
int   x = sx;              /* -12345 */
unsigned  ux = usx;        /*  53191 */

printf("sx  = %d:\t", sx);
show_bytes((byte_pointer) &sx, sizeof(short));
printf("usx = %u:\t", usx);
show_bytes((byte_pointer) &usx, sizeof(unsigned short));
printf("x   = %d:\t", x);
show_bytes((byte_pointer) &x, sizeof(int));
printf("ux  = %u:\t", ux);
show_bytes((byte_pointer) &ux, sizeof(unsigned));
```

When run on a 32-bit big-endian machine using a two's-complement representation, this code prints the output

```
sx  = -12345:  cf c7
usx = 53191:   cf c7
x   = -12345:  ff ff cf c7
ux  = 53191:   00 00 cf c7
```

Sign Extension

One point worth making is that the relative order of conversion from one data size to another and between unsigned and signed can affect the behavior of a program.



```
short sx = -12345;        /* -12345   */
unsigned uy = sx;         /* Mystery! */

printf("uy  = %u:\t", uy);
show_bytes((byte_pointer) &uy, sizeof(unsigned));
```

```
uy = 4294954951:  ff ff cf c7
```

This shows that, when converting from short to unsigned, the program first changes the size and then the type. That is, (unsigned) sx is equivalent to (unsigned) (int) sx, evaluating to 4,294,954,951, not (unsigned) (unsigned short) sx, which evaluates to 53,191.

Truncating

```
int x = 53191;
short sx = (short) x; /* -12345 */
int y = sx; /* -12345 */
```

53191 = 00000000 00000000 11001111 11000111

-12345 = 11001111 11000111

- When truncating a w-bit number $x = [xw-1, xw-2, \ldots, x0]$ to a k-bit number, we drop the high-order $w - k$ bits
- Truncating a number can alter its value -- a form of overflow.
- For an unsigned number x, the result of truncating it to k bits is equivalent to computing x mod 2^k

$$\begin{aligned} B2U_w([x_{w-1}, x_{w-2}, \ldots, x_0]) \bmod 2^k &= \left[\sum_{i=0}^{w-1} x_i 2^i\right] \bmod 2^k \\ &= \left[\sum_{i=0}^{k-1} x_i 2^i\right] \bmod 2^k \\ &= \sum_{i=0}^{k-1} x_i 2^i \\ &= B2U_k([x_{k-1}, x_{k-2}, \ldots, x_0]) \end{aligned}$$

In this derivation, we make use of the property:

$$: 2^i \bmod 2^k = 0 \text{ for any } i \geq k$$

The same is applicable for signed numbers

Summary:
Expanding, Truncating: Basic Rules

- Expanding (e.g., short int to int)
  - Unsigned: zeros added
  - Signed: sign extension
  - Both yield expected result

- Truncating (e.g., unsigned to unsigned short)
  - Unsigned/signed: bits are truncated
  - Result reinterpreted
  - Unsigned: mod operation
  - Signed: similar to mod
  - For small numbers yields expected behaviour

Practice

Suppose we truncate a 4-bit value (represented by hex digits 0 through F) to a 3-bit value (represented as hex digits 0 through 7). Fill in the table below showing the effect of this truncation for some cases, in terms of the unsigned and two's-complement interpretations of those bit patterns.

Hex		Unsigned		Two's complement	
Original	Truncated	Original	Truncated	Original	Truncated
0	0	0		0	
2	2	2		2	
9	1	9		–7	
B	3	11		–5	
F	7	15		–1	

Today: Bits, Bytes, and Integers

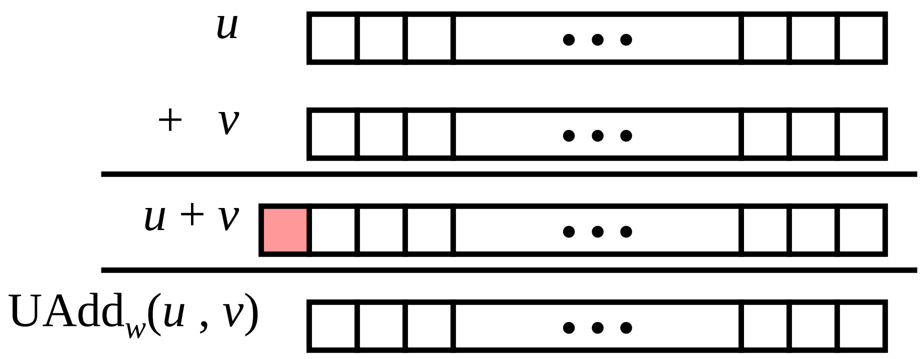
- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - **Addition, negation, multiplication, shifting**
- Summary

Unsigned Addition

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits



- Standard Addition Function
  - Ignores carry output
- Implements Modular Arithmetic

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

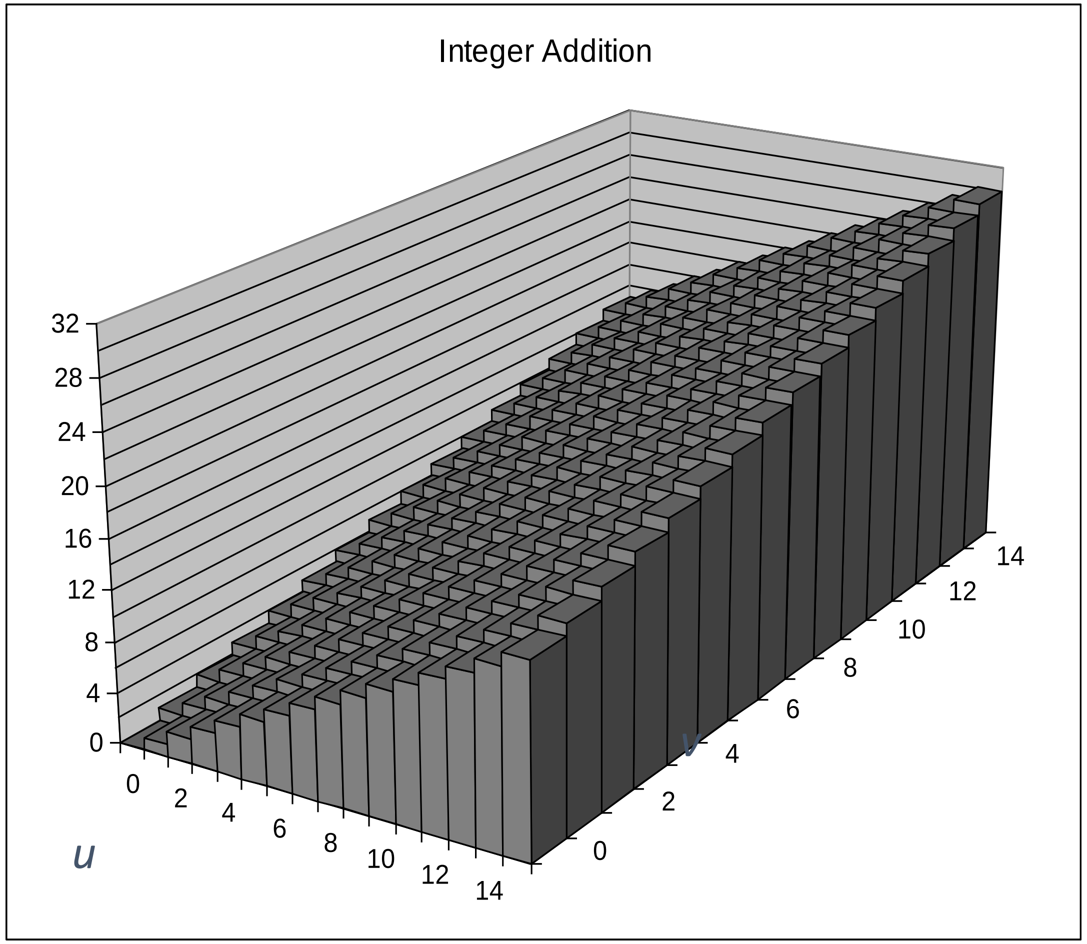
$$UAdd_w(u,v) \quad = \begin{cases} u+v & u+v < 2^w \\ u+v-2^w & u+v \geq 2^w \end{cases}$$

For example, consider a 4-bit number representation with $x = 9$ and $y = 12$, having bit representations [1001] and [1100], respectively. Their sum is 21, having a 5-bit representation [10101]. But if we discard the high-order bit, we get [0101], that is, decimal value 5. This matches the value 21 mod 16 = 5

Visualizing (Mathematical) Integer Addition

- Integer Addition
  - 4-bit integers *u*, *v*
  - Compute true sum $\text{Add}_4(u, v)$
  - Values increase linearly with *u* and *v*
  - Forms planar surface

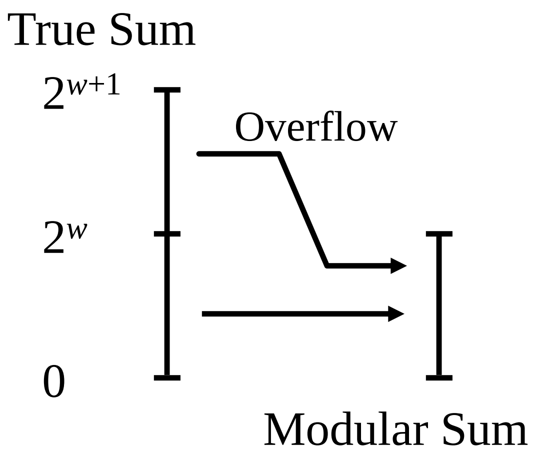
$\text{Add}_4(u, v)$

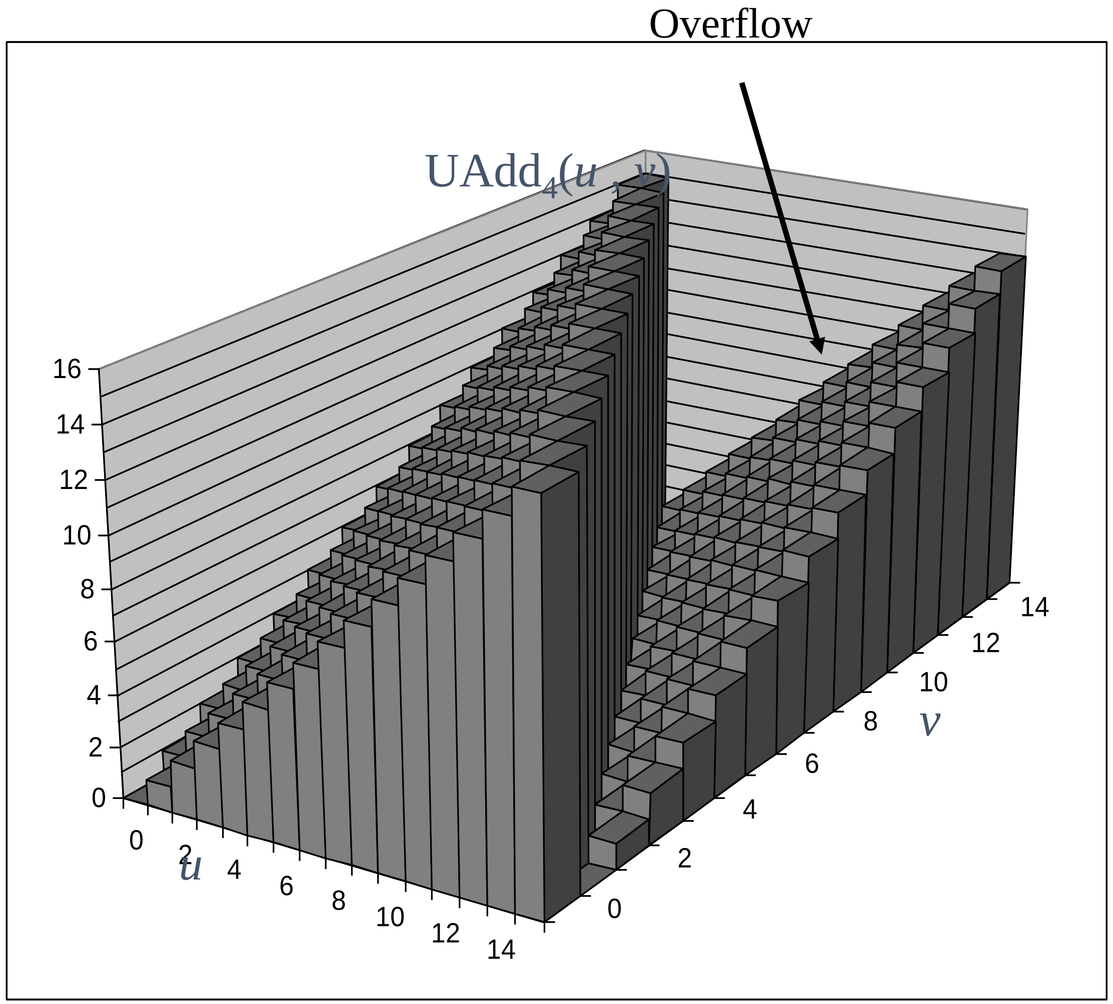


Visualizing Unsigned Addition

- Wraps Around
  - If true sum $\geq 2^w$
  - At most once

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





When executing C programs, overflows are not signaled as errors. At times, however, we might wish to determine whether overflow has occurred. For example, suppose we compute *s* .=*x* +*y*, and we wish to determine whether *s* equals *x* + *y*. We claim that overflow has occurred if and only if *s* < *x* (or equivalently, *s* < *y*)

Mathematical Properties

- Modular Addition Forms an ***Abelian Group***
  - **Closed** under addition

    $0 \le \mathrm{UAdd}_w(u, v) \le 2^w - 1$
  - **Commutative**

    $\mathrm{UAdd}_w(u, v) = \mathrm{UAdd}_w(v, u)$
  - **Associative**

    $\mathrm{UAdd}_w(t, \mathrm{UAdd}_w(u, v)) = \mathrm{UAdd}_w(\mathrm{UAdd}_w(t, u), v)$
  - **0** is additive identity

    $\mathrm{UAdd}_w(u, 0) = u$
  - Every element has additive **inverse**
    - Let $\mathrm{UComp}_w(u) = 2^w - u$

      $\mathrm{UAdd}_w(u, \mathrm{UComp}_w(u)) = 0$

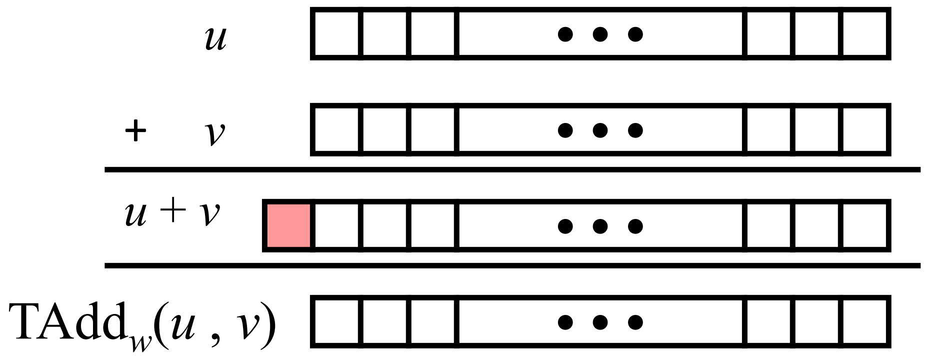
The result of applying the group operation to two group elements does not depend on the order in which they are written

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:

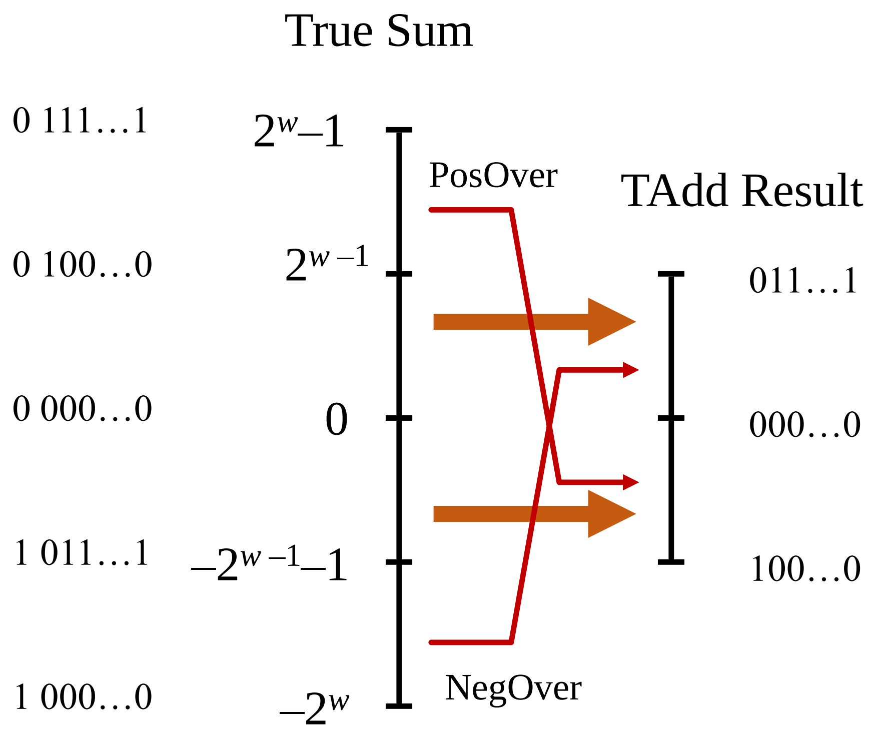
    int s, t, u, v;

    s = (int) ((unsigned) u + (unsigned) v);

    t = u + v
  - Will give **s == t**

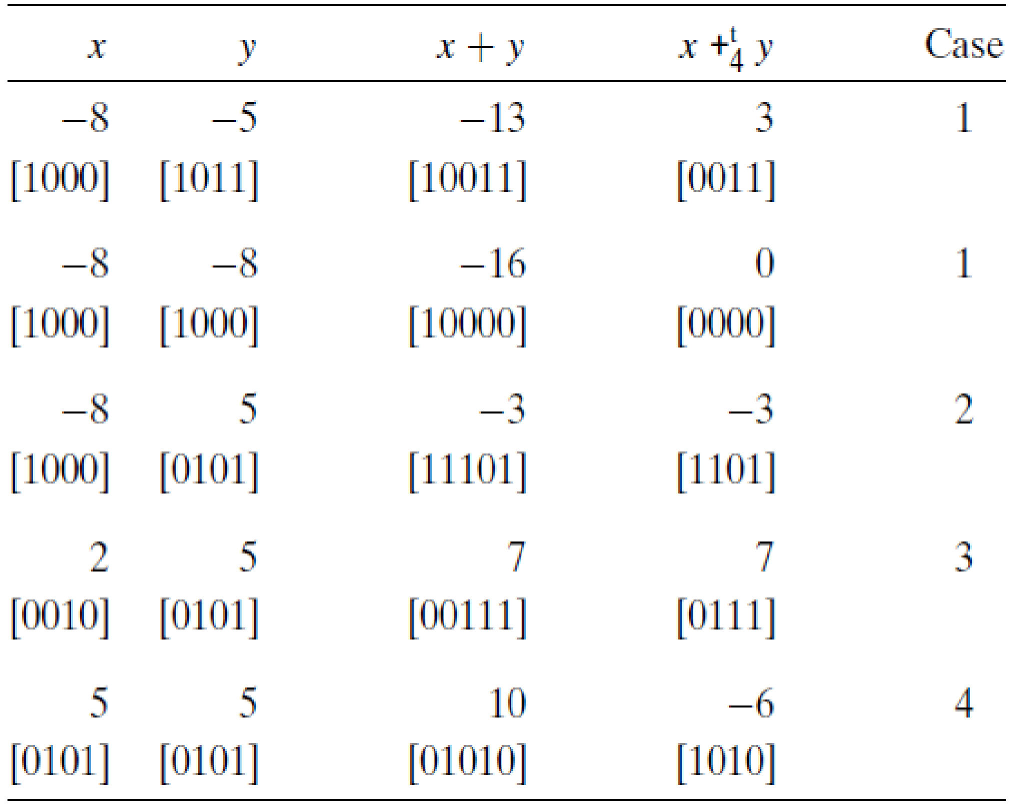
TAdd Overflow

- Functionality
  - True sum requires w+1 bits
  - Drop off MSB
  - Treat remaining bits as 2's comp. integer



Visualizing 2's Complement Addition

- Values
  - 4-bit two's comp.
  - Range from -8 to +7
- Wraps Around
  - If sum ≥ 2^{w-1}
    - Becomes negative
    - At most once
  - If sum < -2^{w-1}
    - Becomes positive
    - At most once

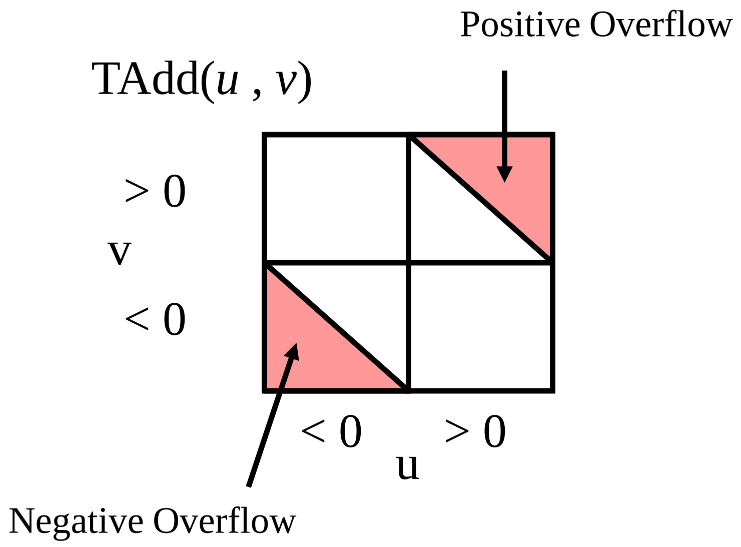


x	y	$x + y$	$x +^{t}_{4} y$	Case
−8 [1000]	−5 [1011]	−13 [10011]	3 [0011]	1
−8 [1000]	−8 [1000]	−16 [10000]	0 [0000]	1
−8 [1000]	5 [0101]	−3 [11101]	−3 [1101]	2
2 [0010]	5 [0101]	7 [00111]	7 [0111]	3
5 [0101]	5 [0101]	10 [01010]	−6 [1010]	4

Figure 2.24 **Two's-complement addition examples.** The bit-level representation of the 4-bit two's-complement sum can be obtained by performing binary addition of the operands and truncating the result to 4 bits.

Characterizing TAdd

- Functionality
  - True sum requires *w*+1 bits
  - Drop off MSB
  - Treat remaining bits as 2's comp. integer



$$TAdd_w(u,v) = \begin{cases} u+v+2^w & u+v < TMin_w \quad \text{(NegOver)} \\ u+v & TMin_w \le u+v \le TMax_w \\ u+v-2^w & TMax_w < u+v \quad \text{(PosOver)} \end{cases}$$

Mathematical Properties of TAdd

- Isomorphic Group to unsigneds with UAdd
  - $\text{TAdd}_w(u, v) = \text{U2T}(\text{UAdd}_w(\text{T2U}(u), \text{T2U}(v)))$
    - Since both have identical bit patterns

- Two's Complement Under TAdd Forms a Group
  - Closed, Commutative, Associative, 0 is additive identity
  - Every element has additive inverse

$$TComp_w(u) = \begin{cases} -u & u \neq TMin_w \\ TMin_w & u = TMin_w \end{cases}$$

Practice

x	y	$x + y$	$x +^{t}_{5} y$	Case
[10100]	[10001]			
[11000]	[11000]			
[10111]	[01000]			
[00010]	[00101]			
[01100]	[00100]			