[CSED211] Introduction to Computer Software Systems

Lecture 2: Bits, Bytes, Integers

Prof. Jisung Park

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

CAOS
COMPUTER ARCHITECTURE & OPERATING SYSTEMS LABORATORY

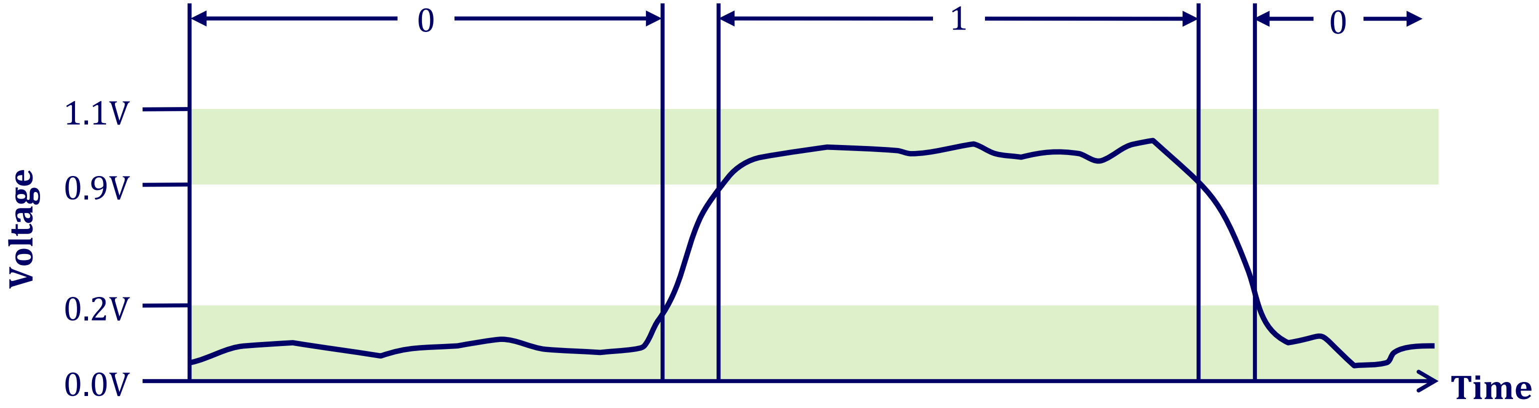
2023.09.06

Lecture Agenda: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
- Representation in memory, pointers, strings

Everything is bits

- Each bit is 0 or 1
- By encoding and interpreting sets of bits in various ways, a computer
  - Determines what to do (i.e., instructions)
  - Represents and manipulates numbers, sets, strings, etc.
- Why bits? Electronic implementation
  - Easy to represent data with bi-stable elements
  - Reliably transmitted on noisy and inaccurate wires



Example: Representation of Numerical Values

- Base 2 number representation
  - Represent $\mathtt{15213}_{(10)}$ as $\mathtt{11101101101101}_{(2)}$
  - Represent $\mathtt{1.20}_{(10)}$ as $\mathtt{1.0011001100110011[0011…]}_{(2)}$
  - Represent $\mathtt{1.5213}_{(10)}\times\mathtt{10}^{4}$ as $\mathtt{1.1101101101101}_{(2)}\times\mathtt{2}^{13}$

Encoding Byte Values

- Byte = 8 bits
  - Binary: `00000000`$_{(2)}$ to `11111111`$_{(2)}$
  - Decimal: `000`$_{(10)}$ to `255`$_{(10)}$
  - Hexadecimal: `00`$_{(16)}$ to **`FF`**$_{(16)}$
    - Base 16 number representation
    - Use symbols '`0`' to '`9`' and '**`A`**' to '**`F`**'
    - Write **`FA1D37B`**$_{(16)}$ in C as
      - `0xFA1D37B`
      - `0xfa1d37b`
      - `0xFa1D37b`

Decimal	Binary	HEX
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Example Data Representations

C Data Type	Typical 32-bit	Typical 64-bit	x86-64
`char`	1	1	1
`short`	2	2	2
`int`	4	4	4
`long`	4	8	8
`float`	4	4	4
`double`	8	8	8
`long double`	–	–	10/16
Pointer (word)	4	8	8

Lecture Agenda: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
- Representation in memory, pointers, strings

Boolean Algebra

- Developed by George Boole in 19th century
  - Algebraic representation of logic
    - Encode "True" as `1` and "False" as `0`

And

- **`A&B=1`** when both **`A=1`** and **`B=1`**

&	0	1
0	0	0
1	0	1

Or

- **`A|B=1`** when either **`A=1`** or **`B=1`**

|	0	1
0	0	1
1	1	1

Not

- **`~A=1`** when **`A=0`**

~	
0	1
1	0

Exclusive-Or (Xor)

- **`A^B=1`** when either **`A=1`** or **`B=1`**, but not both

^	0	1
0	0	1
1	1	0

General Boolean Algebras

- Operate on bit vectors
  - Operations applied bitwise

```
   01101001      01101001      01101001
&  01010101   |  01010101   ^  01010101   ~  01010101
   --------      --------      --------      --------
   01000001      01111101      00111100      10101010
```

- All of the properties of boolean algebra hold (e.g., De Morgan's Laws)

Example: Representing & Manipulating Sets

- Representation
  - Width w bit vector represents subsets of $\{0, \ldots, w-1\}$
  - a_j = **1** if $j \in$ A
    - a = **01101001** for A = {0, 3, 5, 6}
    - b = **01010101** for B = {0, 2, 4, 6}

- Operations
  - **&** Intersection a **&** b = **01000001** A ∩ B = {0, 6}
  - **|** Union a **|** b = **01111101** A ∪ B = {0, 2, 3, 4, 5, 6}
  - **^** Symmetric difference a **^** b = **00111100** (A ∪ B) − (A ∩ B) = {2, 3, 4, 5}
  - **~** Complement **~**a = **10010110** A^C = {1, 2, 4, 7}

Bit-Level Operations in C

- Operations `&`, `|`, `~`, `^` are available in C
  - Apply to any “integral” data type
    - `long`, `int`, `short`, `char`, `unsigned`
  - View arguments as bit vectors
  - Arguments applied bitwise
- Examples (`char` data type)
  - `~0x41 → 0xBE`
    - `~01000001 → 10111110`
  - `~0x00 → 0xFF`
    - `~00000000 → 11111111`
  - `0x69 & 0x55 → 0x41`
    - `01101001 & 01010101 → 01000001`
  - `0x69 | 0x55 → 0x7D`
    - `01101001 | 01010101 → 01111101`

Comparison: Logic Operations in C

- Logical operators
  - `&&`, `||`, `!`
    - View `0` as "False"
    - Anything nonzero as "True"
    - Always return `0` or `1`
    - Early termination
- Examples (`char` data type)
  - `!0x41` → `0x00`
  - `!0x00` → `0x01`
  - `!!0x41` → `0x01`
  - `0x69 && 0x55` → `0x01`
  - `0x69 || 0x55` → `0x01`
  - `p && *p` (avoids null pointer access)

Watch out for `&&` vs. `&` (and `||` vs. `|`, ...), one of the common oopsies in C programming

Shift Operations

- Left shift: `x << y`
  - Shift bit-vector x to left by y positions
    - Throw away extra bits on left
  - Fill with `0`'s on right
- Right shift: `x >> y`
  - Shift bit-vector x to right by y positions
    - Throw away extra bits on right
  - Logical shift
    - Fill with `0`'s on left
  - Arithmetic shift
    - Replicate most significant bit on left
- Undefined behavior (machine-specific)
  - When shift amount < 0 or ≥ word size

Argument `x`	`01100010`
`<< 3`	`00010000`
Log. `>> 2`	`00011000`
Arith. `>> 2`	`00011000`

Argument `x`	`10100010`
`<< 3`	`00010000`
Log. `>> 2`	`00101000`
Arith. `>> 2`	`11101000`

Lecture Agenda: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
- Representation in memory, pointers, strings

Encoding Integers

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` in C: 2-byte 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 ($\because 2^{w-1} > \sum_{i=0}^{w-2} 2^i$)
    - `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	Sum	-15213

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	65,535	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 are platform-specific

Unsigned & Signed Numeric Values

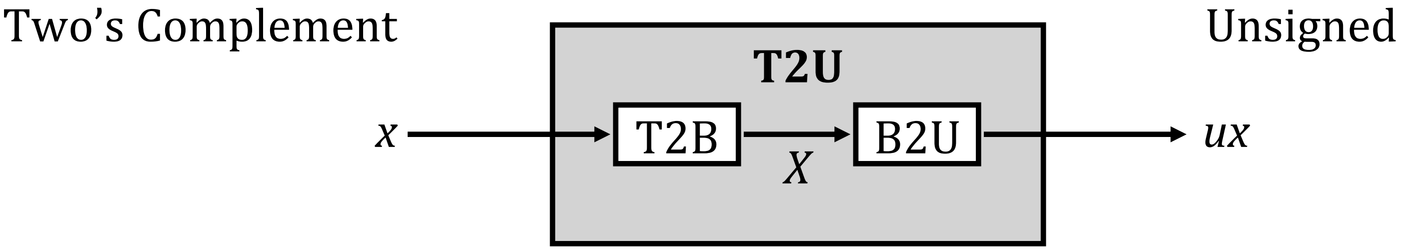
X	*B2U(X)*	*B2T(X)*
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	–8
1001	9	–7
1010	10	–6
1011	11	–5
1100	12	–4
1101	13	–3
1110	14	–2
1111	15	–1

- Equivalence
  - Same encodings for positive values
- Uniqueness
  - Every bit pattern represents a unique integer value
  - Each representable integer has a unique bit encoding
- $\Rightarrow$ Invert mappings
  - $U2B(x) = B2U^{-1}(x)$
    - Bit pattern for unsigned integer
  - $T2B(x) = B2T^{-1}(x)$
    - Bit pattern for 2's complement integer

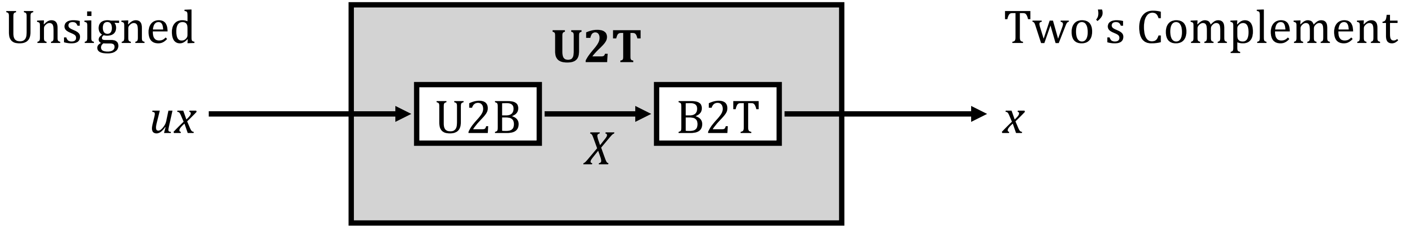
Lecture Agenda: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
- Representation in memory, pointers, strings

Mapping Between Signed & Unsigned



Keep the same bit pattern



Keep the same bit pattern

- Mappings b/w unsigned and two's complement numbers: keep the same bit representations but reinterpret

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

Signed $\xrightarrow{\text{T2U}}$ Unsigned

Signed $\xleftarrow{\text{U2T}}$ Unsigned

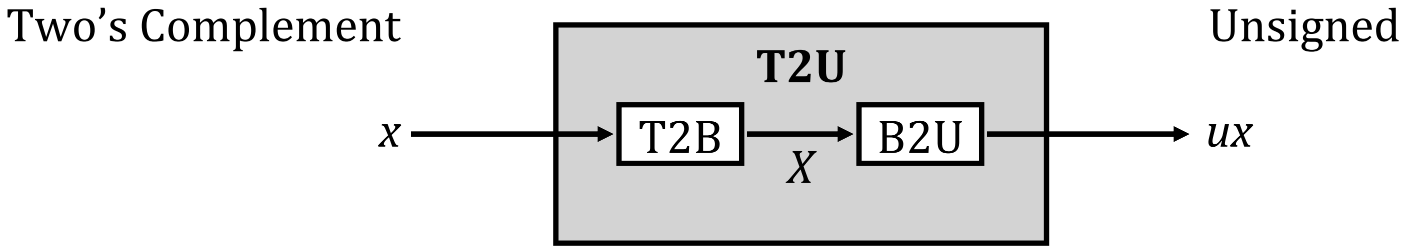
Mapping Signed ↔ Unsigned

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

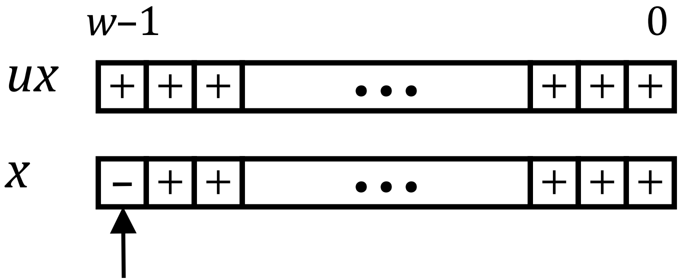
=

+/- 16

Relation between Signed & Unsigned



Keep the same bit pattern



Large negative weight

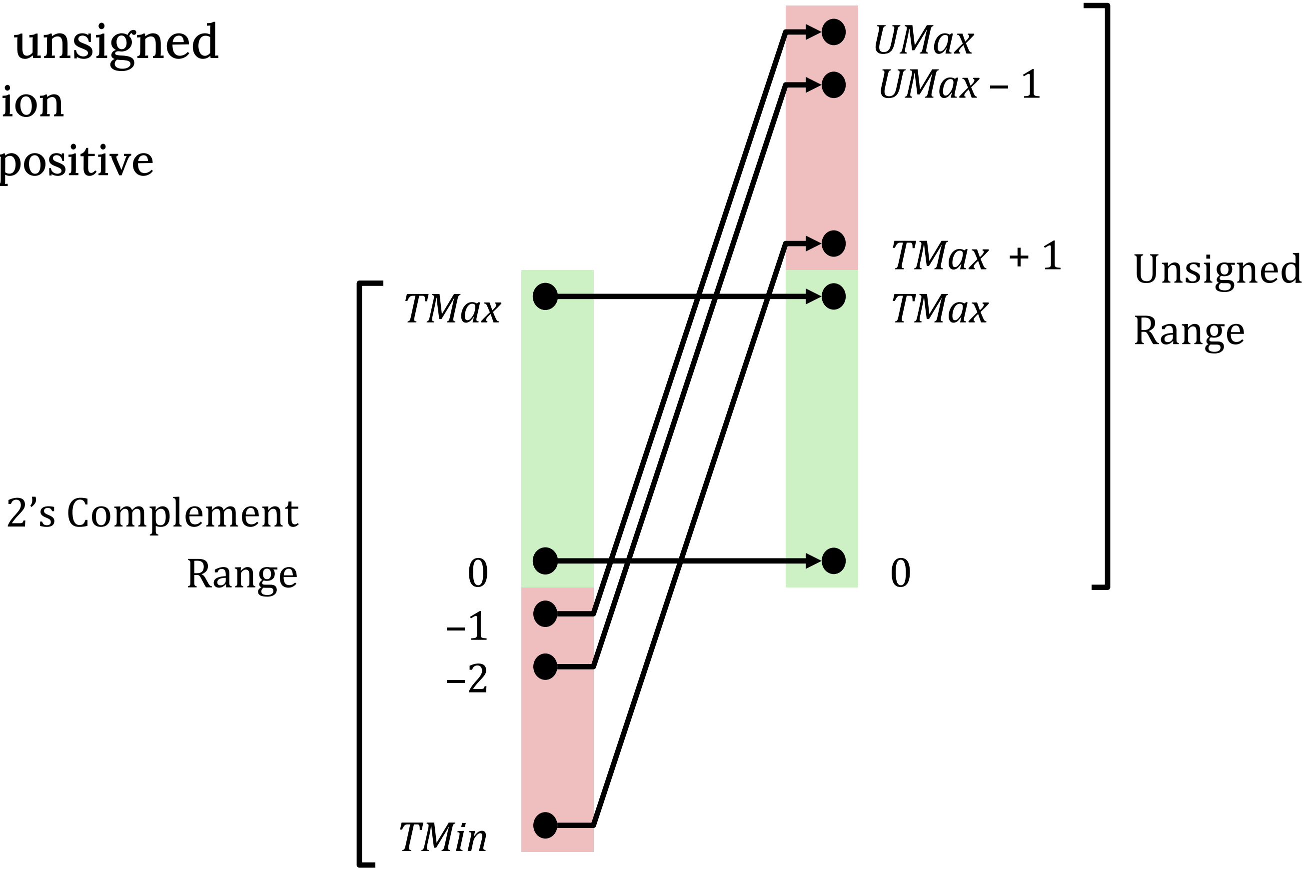
becomes

Large positive weight

$$ux = \begin{cases} x & x \geq 0 \\ x + 2^w & x < 0 \end{cases}$$

Conversion Visualized

- 2's complement → unsigned
  - Ordering inversion
  - Negative → big positive



Signed vs. Unsigned in C

- Constants
  - Considered to be signed integers by default
  - Unsigned if have "`U`" as a suffix

    `0U`, `4294967259U`

- Casting
  - Explicit casting between signed & unsigned: 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 there is a mix of unsigned and signed in single expression, signed values implicitly cast to unsigned
  - Including comparison operations <, >, ==, <=, >=
  - Examples for $w = 32$: $TMin$ = -2,147,483,648, $TMax$ = 2,147,483,647

Constant#1	Constant#2	Relation	Evaluation
`0`	`0U`	`==`	Unsigned
`-1`	`0`	`<`	Signed
`-1`	`0U`	`>`	Unsigned
`2147483647`	`-2147483648`	`>`	Signed
`2147483647U`	`-2147483648`	`<`	Unsigned
`-1`	`-2`	`>`	Signed
`-1`	`(unsigned) -2`	`>`	Unsigned
`2147483647`	`2147483648U`	`<`	Unsigned
`2147483647`	`(int) 2147483648U`	`>`	Signed

Summary: Basic Signed-Unsigned Casting Rules

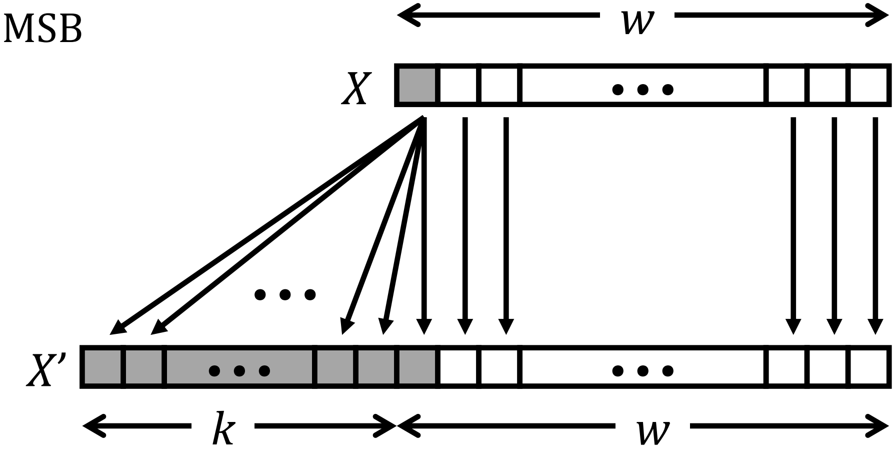
- Keep the same bit pattern
- But reinterpret
- Can have unexpected effects: adding or subtracting 2^w
- Expression containing signed and unsigned: cast to unsigned

Lecture Agenda: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
    - Representation: unsigned and signed
    - Conversion, casting
    - Expanding, truncating
    - Addition, negation, multiplication, shifting
- Representation in memory, pointers, strings

Sign Extension

- Task:
  - Given w-bit signed integer x
  - Convert it to (w+k)-bit integer with the 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

```
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

Summary: Basic Expansion & Truncation Rules

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

- Truncating (e.g., `unsigned int` to `unsigned short`)
  - Unsigned/signed: bits are truncated
  - Result reinterpreted
  - Unsigned: mod (`%`) operation
  - Signed: similar to mod
    - Expected behaviour only for small numbers yield

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
- Representation in memory, pointers, strings

Negation: Inversion & Increment

- Claim: the following holds for 2's complement

  `~x + 1 = -x`

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

	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

Inversion & Increment Examples

`x = 15213`

	Decimal	HEX	Binary
`x`	15213	3B 6D	00111011 01101101
`~x`	-15214	C4 92	11000100 10010010
`~x+1`	-15213	C4 93	11000100 10010011

`x = 0`

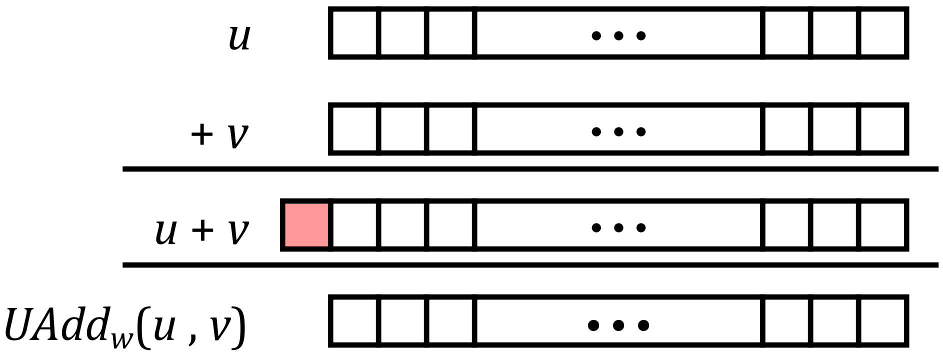
	Decimal	HEX	Binary
`x`	0	3B 6D	00000000 00000000
`~x`	-1	FF FF	11111111 11111111
`~x+1`	0	00 00	00000000 00000000

Unsigned Addition

Operands: w bits

True Sum: w+1 bits

Discard Carry: w bits



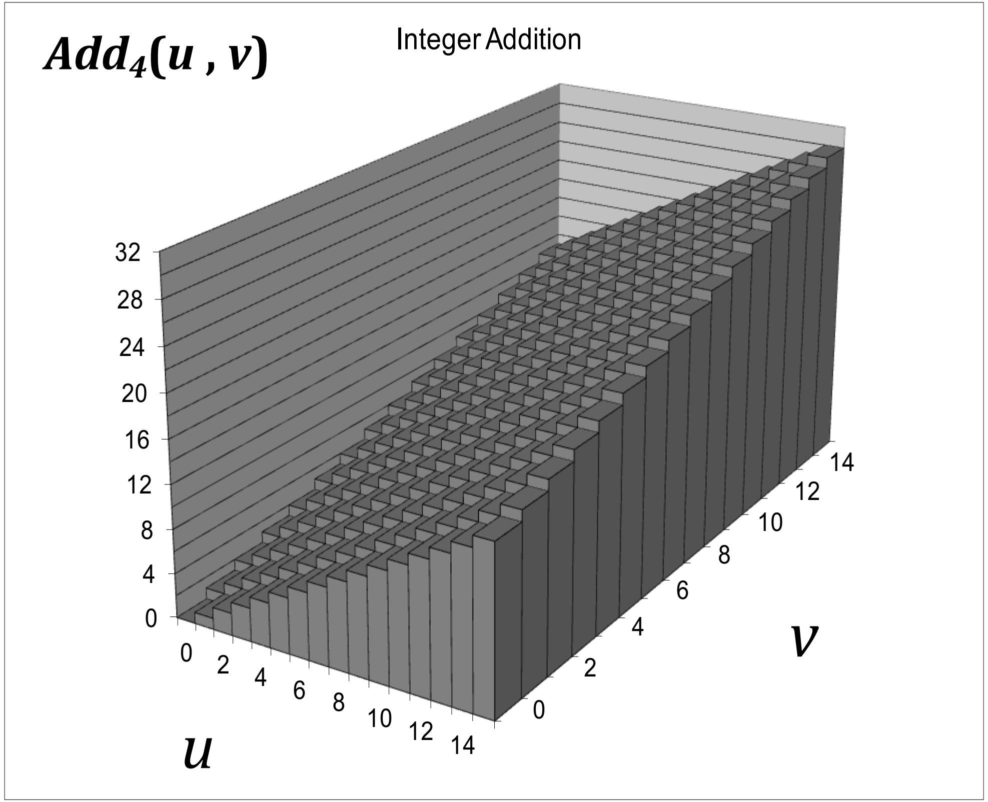
- Standard addition function
  - Ignores carry output
- Implements modular arithmetic

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

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

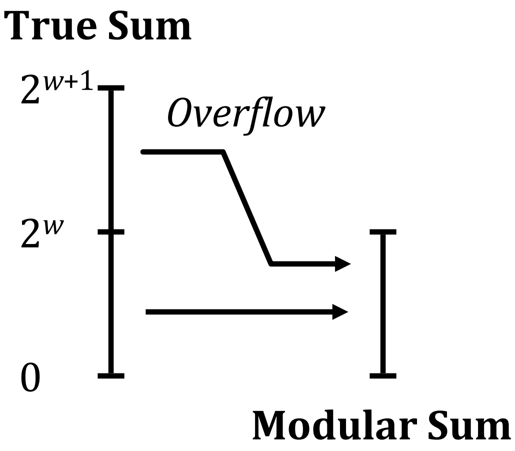
Visualizing (Mathematical) Integer Addition

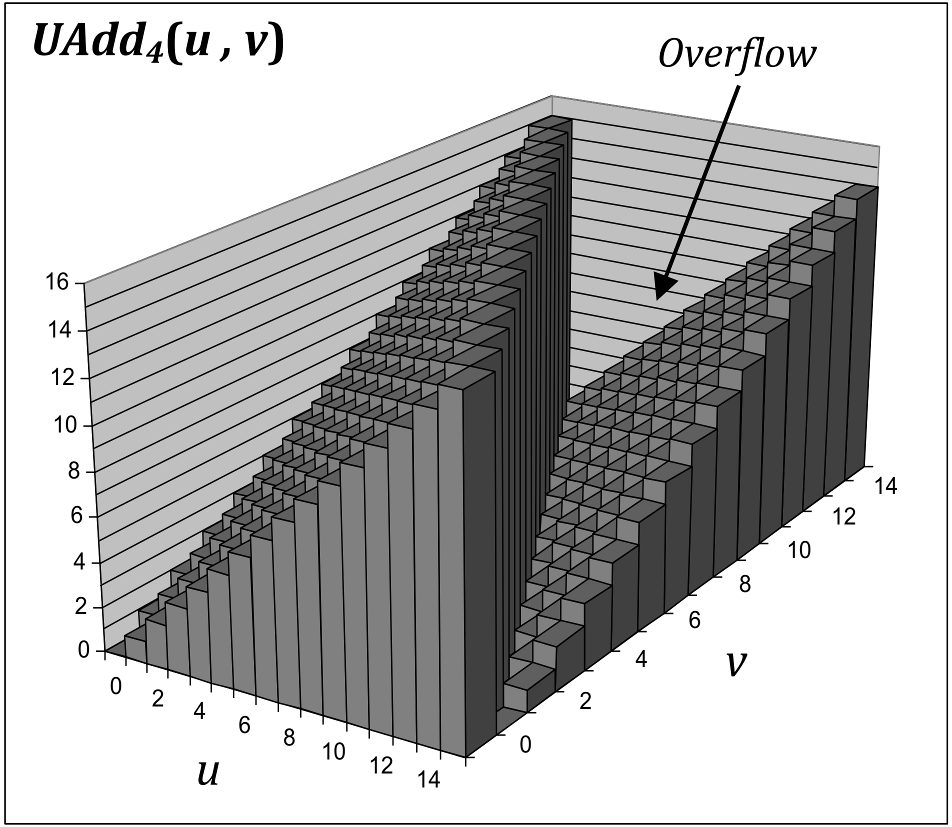
- Integer addition
  - 4-bit integers u, v
  - Compute true sum $Add_4(u\ ,\ v)$
  - Values increase linearly with u and v
  - Forms planar surface



Visualizing Unsigned Addition

- Wraps around
  - If true sum ≥ 2^w
  - At most once



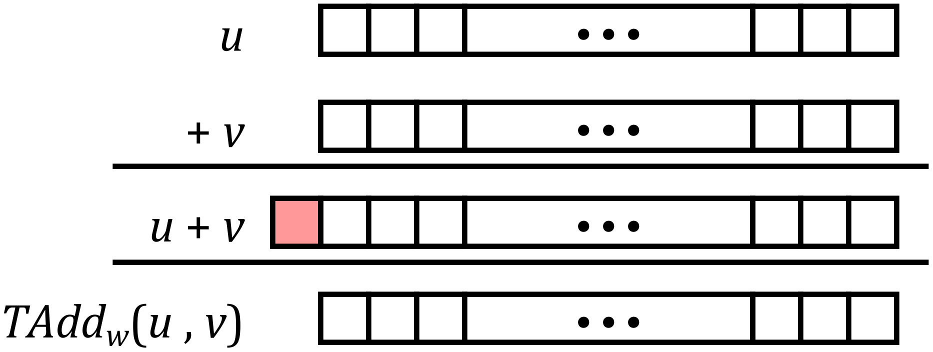


Two’s Complement Addition

Operands: w bits

True Sum: w+1 bits

Discard Carry: w bits



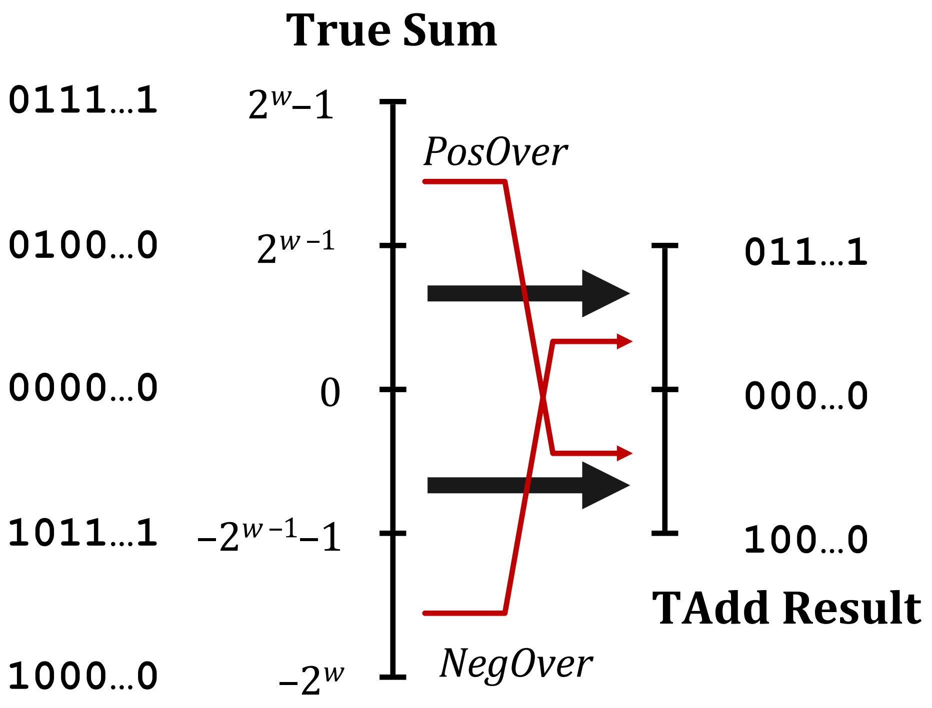
- *TAdd* and *UAdd* have identical bit-level behaviour
  - Signed vs. unsigned addition in C

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

  - Will give `s == t`

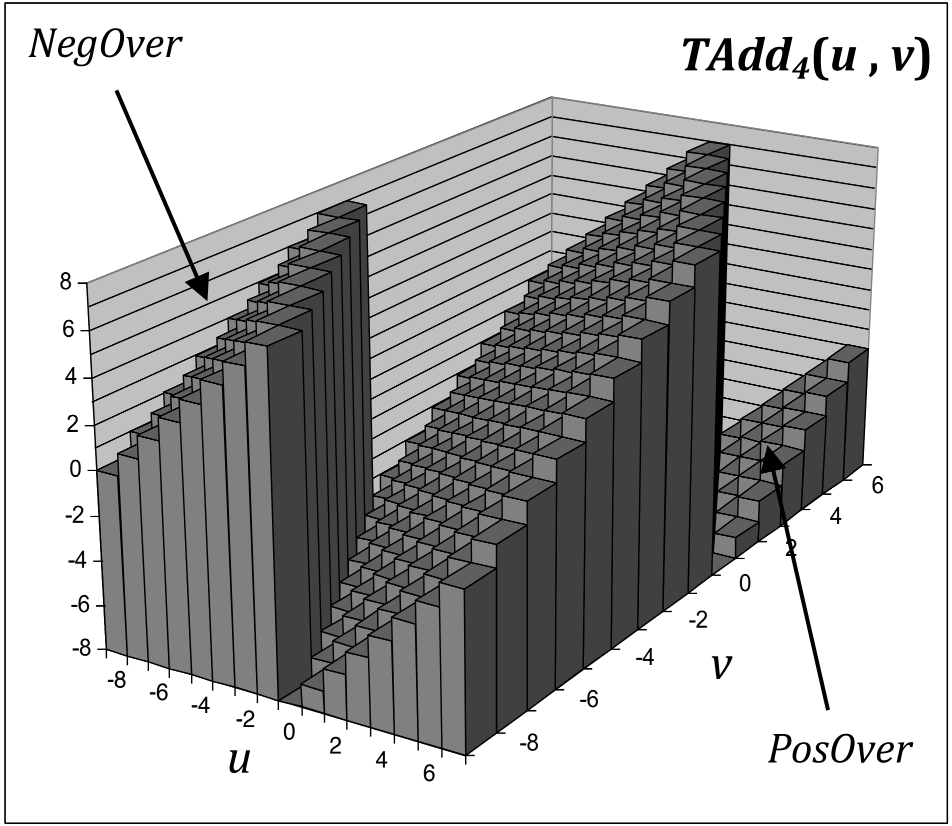
TAdd Overflow

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



Visualizing 2's Complement Addition

- Values
  - 4-bit 2's complement
  - 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



Multiplication

- Computing exact product of w-bit numbers x, y
  - Either signed or unsigned

- Problem: exact results may require more than w bits
  - Unsigned: up to $2w$ bits
    - Result range: $0 \le x \times y \le (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
  - Two's complement min: up to $(2w-1)$ bits
    - Result range: $x \times y \ge (-2^{w-1}) \times (2^{w-1} - 1) = -2^{2w-2} + 2^{w-1}$
  - Two's complement max: up to $2w$ bits, but only for $(TMin_w)^2$
    - Result range: $x \times y \le (-2^{w-1})^2 = 2^{2w-2}$

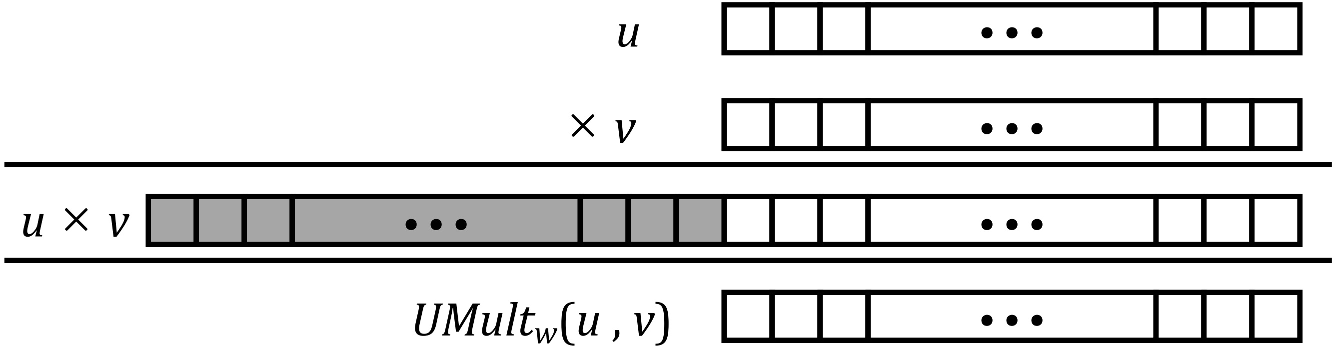
- Maintaining exact results
  - Would need to keep expanding word size with each product computed
  - Done in software by arbitrary-precision (or bignum) arithmetic packages

Unsigned Multiplication in C

Operands: w bits

True Product: $2w$ bits

Discard *w* MSBs: w bits



- Standard Multiplication Function
  - Ignores high order w bits
- Implements Modular Arithmetic

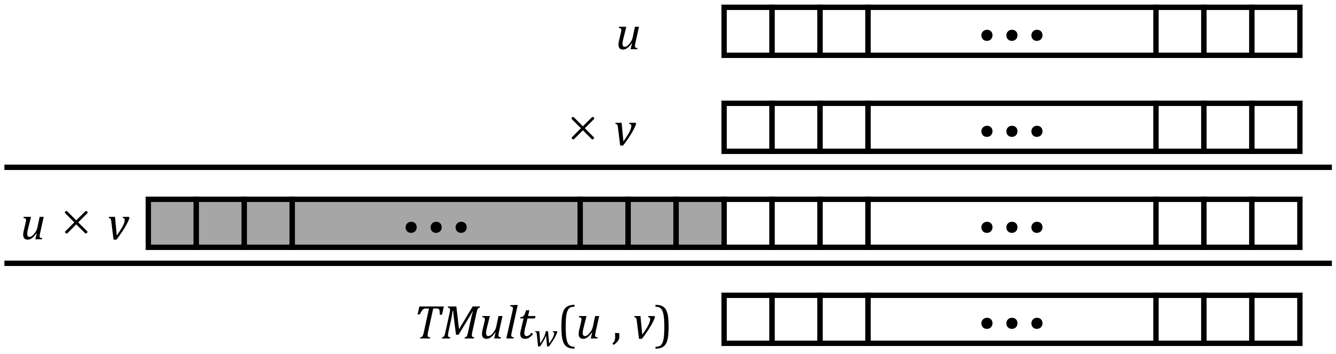
  $UMult_w(u , v) = (u \times v) \bmod 2^w$

Signed Multiplication in C

Operands: w bits

True Product: $2w$ bits

Discard w MSBs: w bits



- Standard Multiplication Function
    - Ignores high order w bits
    - Some of which are different for signed vs. unsigned multiplication
    - Lower bits are the same

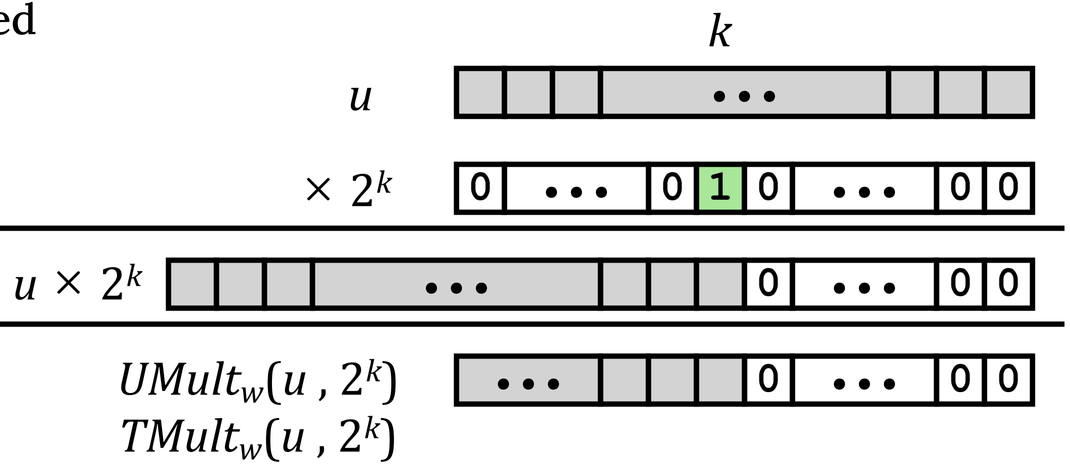
Power-of-2 Multiply with Shift

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

Operands: w bits

True Product: $w+k$ bits

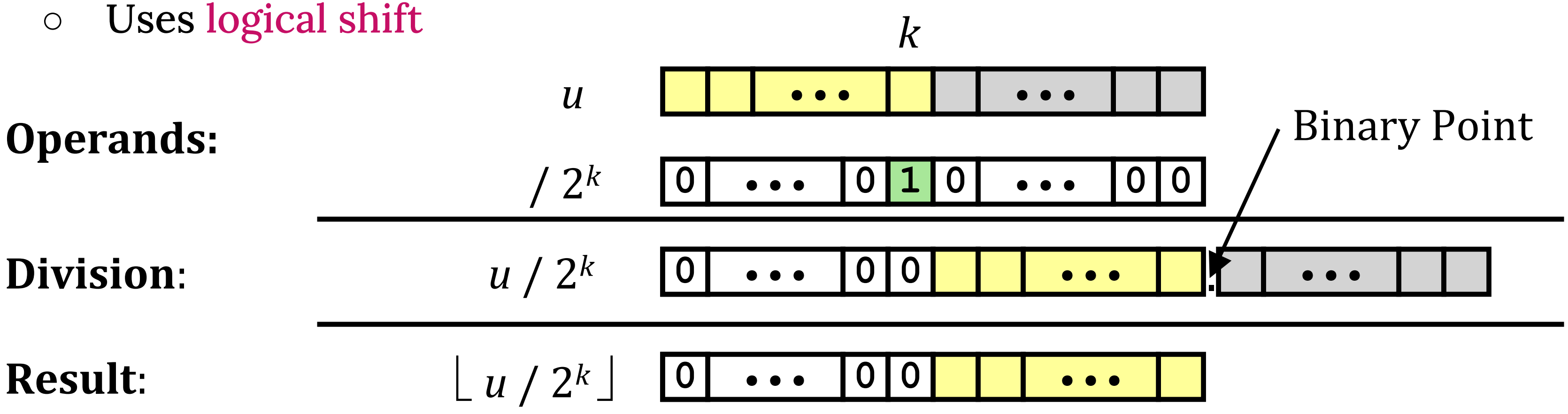
Discard k bits: w bits



- Examples
  - `u << 3 == u * 8`
  - `u << 5 - u << 3 == u * 24`
  - Most machines shift and add faster than multiply: compiler may optimize this

Unsigned Power-of-2 Divide with Shift

- Quotient of unsigned by power of 2
  - `u >> k` gives ⌊ u / $2k$ ⌋
  - Uses logical shift



	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

Arithmetic: Basic Rules

- Addition
  - Unsigned/signed: normal addition followed by truncate, the 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 a proper range)
    - Mathematical addition + possible addition or subtraction of 2^w

- Multiplication
  - Unsigned/signed: normal multiplication followed by truncate, the same operation on bit level
  - Unsigned: multiplication mod 2^w
  - Signed: modified multiplication mod 2^w (result in a proper range)

Why to Use Unsigned?

- Do NOT use signed w/o understanding implications
  - Easy to make a mistake

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

  - Can be very subtle

```
#define DELTA sizeof(int)
int i;
for (i = CNT; i-DELTA >= 0; i-= DELTA)
    ...
```

- Do use unsigned to perform modular arithmetic
  - Multi-precision arithmetic
- Do use unsigned to represent sets
  - Logical right shift, no sign extension

Counting Down with Unsigned

- Proper way to use unsigned as loop index

```
unsigned i;
for (i = cnt-2; i < cnt; i--)
    a[i] += a[i+1];
```

- See Robert Seacord, *Secure Coding in C and C++*
  - C Standard guarantees that unsigned addition will behave like modular arithmetic
    - 0 – 1 → *UMax*
- Even better

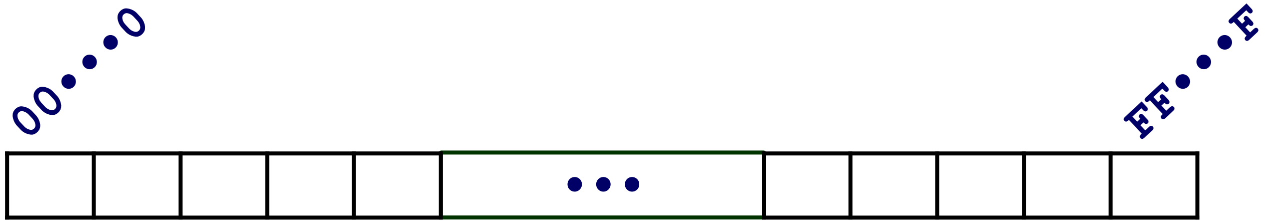
```
size_t i;
for (i = cnt-2; i < cnt; i--)
 a[i] += a[i+1];
```

  - Data type **`size_t`** defined as unsigned value with length = word size
  - Code will work even if **`cnt`** = *UMax*
  - What if **`cnt`** is signed and < 0?

Lecture Agenda: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
- Representation in memory, pointers, strings

Byte-Oriented Memory Organization



- Programs refer to virtual addresses
  - Conceptually, consider main memory as a very large array of bytes
    - In reality, it is not true, but a program can think of it that way
  - An address is like an index into that array
    - A pointer variable stores an address

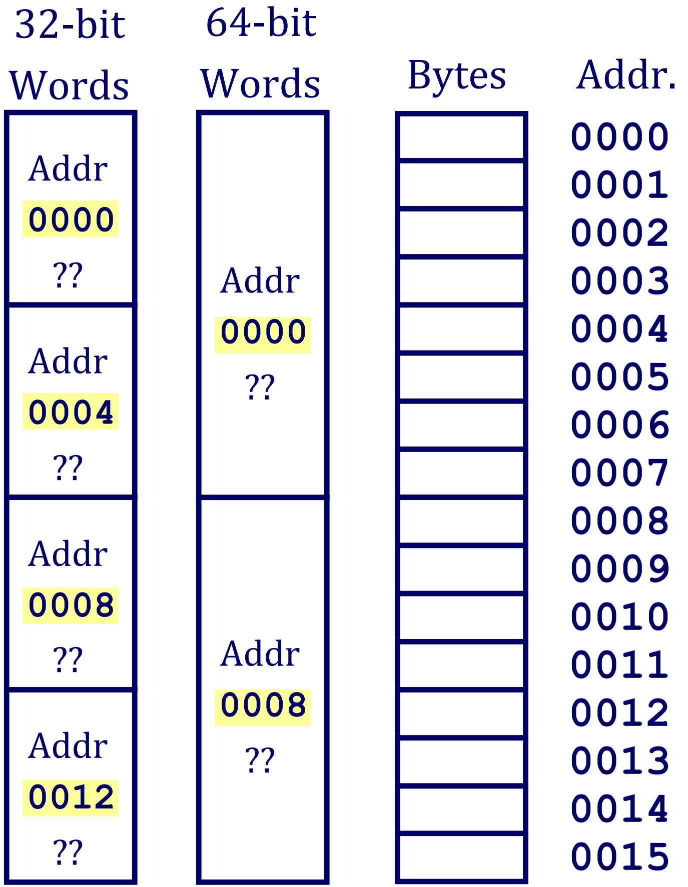
- Note: a system can provide private address spaces to each process
  - Think of a process as a program being executed
  - So, a program can clobber its own data, but not that of others

Machine Words

- Any given computer has a word size
  - The nominal size of integer-valued data
    - More precisely: the size of addresses
  - Until recently, most machines used 32 bits (4 bytes) as word size
    - Limits addresses to 4 GiB (2^{32} bytes)
  - Increasingly, machines have 64-bit word size
    - Potentially, could have 18 EB (exabytes) of addressable memory
    - Around 18.4×10^{18}
  - Machines still support multiple data formats
    - Fractions or multiples of word size
    - Always integral number of bytes

Word-Oriented Memory Organization

- Addresses specify byte locations
  - Address of the first byte in word
  - Addresses of successive words differ by 4 (32-bit) or 8 (64-bit)



Example Data Representations

C Data Type	Typical 32-bit	Typical 64-bit	x86-64
`char`	1	1	1
`short`	2	2	2
`int`	4	4	4
`long`	4	8	8
`float`	4	4	4
`double`	8	8	8
`long double`	–	–	10/16
Pointer (word)	4	8	8

Byte Ordering

- How should bytes within a multi-byte word be ordered in memory?
- Conventions
  - Big endian: Sun, PowerPC Mac, internet
    - The most significant byte comes first
  - Little endian: x86
    - The least significant byte comes first

Byte Ordering Example

- Big endian
  - The most significant byte has the highest (smallest) address
- Little endian
  - The least significant byte has the highest (smallest) address
- Example
  - Variable **`x`** has 4-byte representation **`0x01234567`**
  - Address given by **`&x`** is **`0x100`**

Big Endian

		0x100	0x101	0x102	0x103		
		01	23	45	67		

Little Endian

		0x100	0x101	0x102	0x103		
		67	45	23	01		

Reading Byte-Reversed Listings

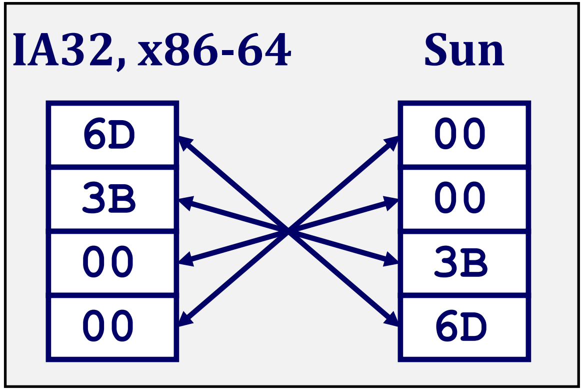
- Disassembly
  - Text representation of binary machine code
  - Generated by program that reads the machine code
- Example fragment

Address	Instruction Code	Assembly Rendition
`8048365:`	`5b`	`pop %ebx`
`8048366:`	`81 c3 ab 12 00 00`	`add $0x12ab,%ebx`
`804836c:`	`83 bb 28 00 00 00 00`	`cmpl $0x0,0x28(%ebx)`

- Deciphering numbers
  - Value: `0x12ab`
  - Pad to 32 bits: `0x000012ab`
  - Split into bytes: `00 00 12 ab`
  - Reverse: `ab 12 00 00`

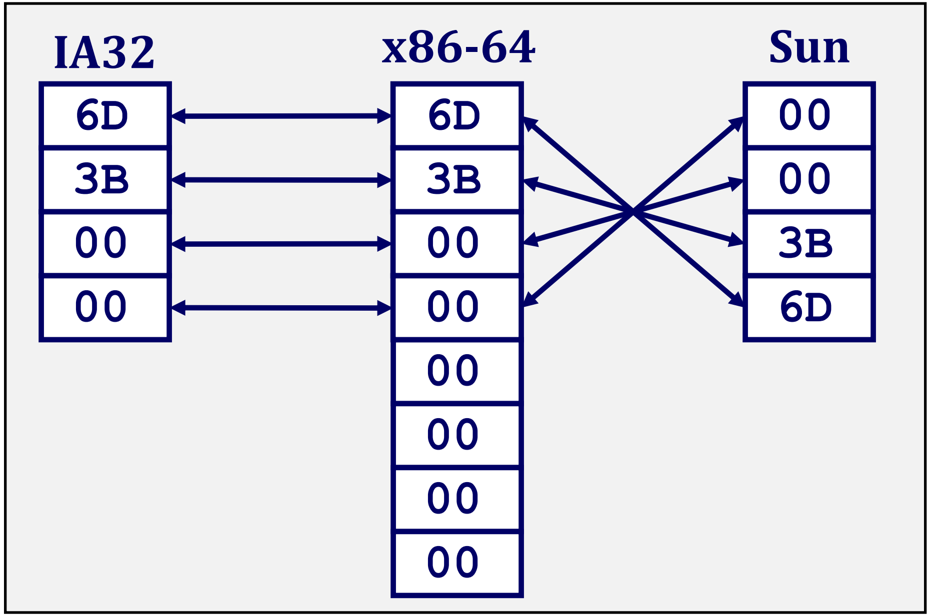
Representing Integers

`int a = 15213;`

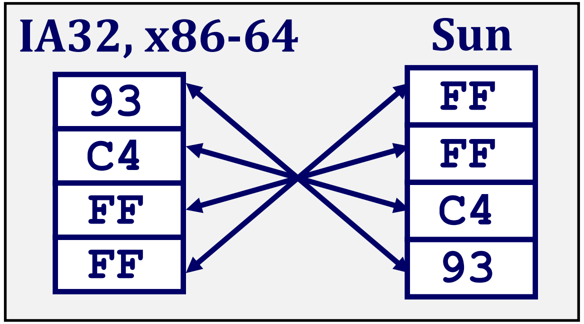


Decimal:	15213			
Binary:	0011	1011	0110	1101
Hex:	3	B	6	D

`long int c = 15213;`



`int b = -15213;`



Examining Data Representations

- Code to print byte representation of data
  - Casting pointer to **`unsigned char *`** creates a byte array

```
typedef unsigned char *pointer;

void show_bytes(pointer start, int len){
  int i;
  for (i = 0; i < len; i++)
    printf("%p\t0x%.2x\n",start+i, start[i]);
  printf("\n");
}
```

`printf` directives:

`%p`: Print pointer

`%x`: Print hexadecimal

show_bytes Execution Example

```
int a = 15213;
printf("int a = 15213;\n");
show_bytes((pointer) &a, sizeof(int));
```

Result (Linux):

```
int a = 15213;
0x11ffffcb8    0x6d
0x11ffffcb9    0x3b
0x11ffffcba    0x00
0x11ffffcbb    0x00
```

Representing Pointers

```
int b = -15213;
int *p = &b;
```

Sun	IA32	x86-64
EF	D4	0C
FF	F8	89
FB	FF	EC
2C	BF	FF
		FF
		7F
		00
		00

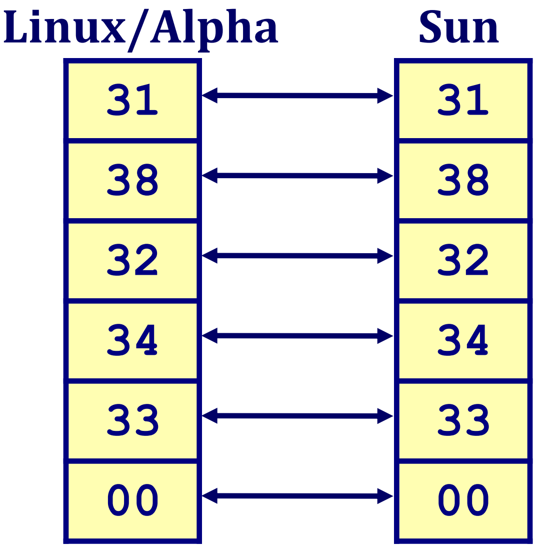
- Different compilers & machines assign different locations to objects
- Even get different results each time run program

Representing Strings

- Strings in C
  - Represented by an array of characters
  - Each character encoded in ASCII format
    - Standard 7-bit encoding of character set
    - Character “`0`” has code `0x30`
      - Digit “`i`” has code `0x30+i`
  - String should be null-terminated
    - Final character = “`0`”

- Compatibility
  - Byte ordering not an issue

```
char s[6] = "18243";
```



Integer C Puzzles

- Assume 32-bit word size, two's complement integers
- For each of the following C expressions: true or false? Why?

Initialization

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

- ✗ `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`
- ✗ `(x|-x)>>31 == -1`
- ✓ `ux >> 3 == ux/8`
- ✗ `x >> 3 == x/8`
- ✗ `x & (x-1) != 0`

[CSED211] Introduction to Computer Software Systems

Lecture 2: Bits, Bytes, Integers

Prof. Jisung Park

COMPUTER ARCHITECTURE & OPERATING SYSTEMS LABORATORY

2023.09.06