w07-Lec

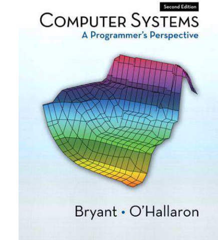
Data Representation

Part I – Bits, Bytes and Integers

Assembled for 204111
by Kittipitch Kuptavanich

Bits, Bytes and Integers

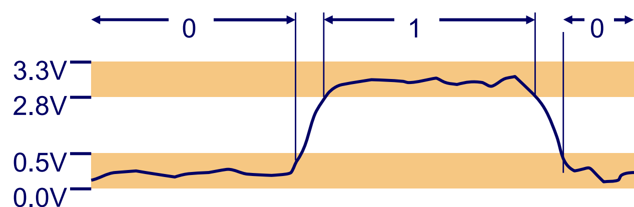
- **Binary Representation**
- Integral Data Type Ranges
- Encodings
- Signed and Unsigned Conversions, Casting
- Expanding
- Truncating
- Integer Arithmetic



from Carnegie Mellon University's 15213 course slide by Greg Kesden

Binary Representation

- **Base 2 Number Representation**
  - **Represent 15213_{10} as 11101101101101_2**
  - **Represent 1.20_{10} as $1.0011001100110011[0011]..._2$**
  - **Represent 1.5213_{10} X 10^4 as 1.1101101101101_2 X 2^{13}**
- **Electronic Implementation**
  - **Easy to store with bistable elements (media ที่มีสถานะเสถียร 2 สถานะ)**
  - **Reliably transmitted on noisy and inaccurate wires**



Bits, Bytes and Words

- **Modern computers store and process 2-valued signals called *binary digits*, or *bits*,**
- **Most computers access bits in memory in *blocks of eight bits*, or *bytes* (rather than individual bits)**
- **Information are transferred in fixed-sized chunks of bytes known as *words* with the size of either 4 bytes (32 bits) or 8 bytes (64 bits)**
- **In this class we assume a word size of *4* bytes**

from Carnegie Mellon University's 15213 course slide by Greg Kesden

Encoding Byte Values

- **Byte = 8 bits**
  - **Binary 00000000_2 to 11111111_2**
  - **Decimal: 0_{10} to 255_{10}**
  - **Hexadecimal 00_{16} to FF_{16}**
    - **Base 16 number representation**
    - **Use characters '0' to '9' and 'A' to 'F'**
    - **Write $FA1D37B_{16}$ in C as**
      - `0xFA1D37B`
      - `0xfa1d37b`

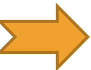
Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Conversion Exercise

2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
256	128	64	32	16	8	4	2	1

Practice Problem 2.3: Fill in the missing entries

Decimal	Binary	Hexadecimal
0	0000 0000	0x00
167	______	______
62	______	______
188	______	______
______	0011 0111	______
______	1000 1000	______
______	1111 0011	______
______	______	0x52
______	______	0xAC
______	______	0xE7



from Carnegie Mellon University's 15213 course slide by Greg Kesden

Boolean Algebra Revisited

- **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

from Carnegie Mellon University's 15213 course slide by Greg Kesden

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 Apply**

from Carnegie Mellon University's 15213 course slide by Greg Kesden

Contrast: Logic Operations in C

- **Contrast to 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)**

from Carnegie Mellon University's 15213 course slide by Greg Kesden

Shift Operations

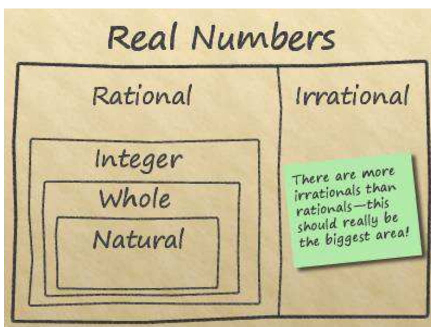
- **Left Shift: x << y**
  - **Shift bit-vector x left y positions**
    - **Throw away extra bits on left**
    - **Fill with 0's on right**
- **Right Shift: x >> y**
  - **Shift bit-vector x right y positions**
    - **Throw away extra bits on right**
  - **Logical shift**
    - **Fill with 0's on left**
  - **Arithmetic shift**
    - **Replicate most significant bit (MSB) on left**
- **Undefined Behavior**
  - **Shift amount < 0 or $\geq$ word size**

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

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

Numbers Revisited

- **Number Types**



- **C Data Types (bytes)**

C Data Type	Typical 32-bit	Intel IA32 (i386)	x86-64
char	1	1	1
short	2	2	2
int	4	4	4
long	4	4	8
long long	8	8	8
float	4	4	4
double	8	8	8
long double	8	10/12	10/16
pointer	4	4	8

Reality vs Abstraction

- **Ints are not integers**
- **For most computer (32-bit `int`)**
  - **The expression `200 * 300 * 400 * 500`**
  - **Yields: `-884,901,888`**
- **Why??? – Overflowing**
- **For example, multiplication is associative and commutative so any of the following yields `-884,901,888`**

```
(500 * 400) * (300 * 200)
((500 * 400) * 300) * 200
((200 * 500) * 300) * 400
400 * (200 * (300 * 500))
```

Reality vs Abstraction [2]

- Floats are not reals
- Overflowing will yield a special value: +∞
- Arithmetic is not associative (due to the finite precision)
- For example:

```
printf("%f\n",(3.14+1e20)-1e20);
                vs
printf("%f\n",3.14+(1e20-1e20));
```

- `(3.14+1e20)-1e20` will evaluate to 0.0
- `3.14+(1e20-1e20)` will evaluate to 3.14

Representation Limitations

- Integer representations (`char, short, int, long,…`)
  - Small ranges of values
  - Precise values
- Floating-point representations (float, double,…)
  - Wide ranges of values
  - Approximate values

Later in the course

Bits, Bytes and Integers

- Binary Representation
- **Integral Data Type Ranges**
- Encodings
- Signed and Unsigned Conversions, Casting
- Expanding
- Truncating
- Integer Arithmetic

Range Calculation

- Consider an unsigned environment (0 and positive integer) For 4 bits
  - We can represent 2^4 different numbers : 0000, 0001, 0010, …… 1111
  - The maximum is
    - $1111_2 = 15_{10} = 2^4 - 1$
  - In signed representations, about half of the range will be assigned to represent the negative numbers for w bits the range will be -2^{w-1} to $2^{w-1} - 1$

Bits	unsigned data range	value
2	0 - (2^2 - 1)	0 - 3
4	0 – (2^4 – 1)	0 – 15
…	…	…
w	0 – (2^w – 1)	

Range Calculation Exercise

- Fill in the rest of the entries

Bits	Unsigned range	Unsigned value	Signed range	Signed value
2	0 to $2^2 - 1$	0 to 3		
4	0 to $2^4 - 1$	0 to 15		
8	0 to $2^8 - 1$	0 to 255		
16				
32				
w	0 to $2^w - 1$			

Integral Data Type Ranges

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

Typical ranges for C integral data types o a 32-bit machine

Integral Data Type Ranges [2]

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

Guaranteed ranges for C integral data types.

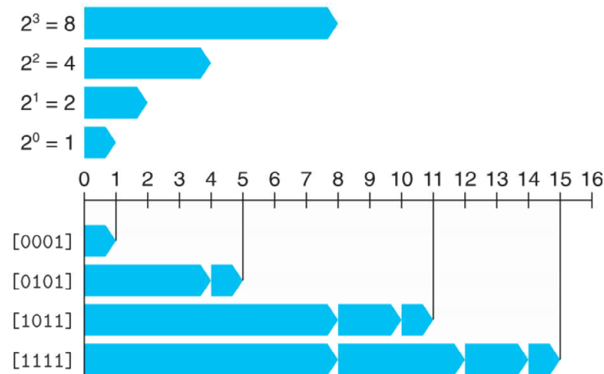
Bits, Bytes and Integers

- Binary Representation
- Integral Data Type Ranges
- **Encodings**
- Signed and Unsigned Conversions, Casting
- Expanding
- Truncating
- Integer Arithmetic

Unsigned Encodings

- พิจารณาการแสดงค่าแบบ unsigned บนเส้นจำนวน ในกรณีมี 4 Bit

Figure 2.11
Unsigned number examples for $w = 4$. When bit i in the binary representation has value 1, it contributes 2^i to the value.



Representing Negatives

- เราจำเป็นต้องหาวิธีในการแทนข้อมูล negative ด้วยเลขฐานสอง
- วิธีที่ง่ายที่สุด ใช้ Bit ที่สำคัญที่สุด (most significant bit: MSB) เป็นตัวบอกเครื่องหมาย ถ้า MSB (ซ้ายสุด) เป็น 1 แสดงว่าเป็นจำนวนลบ ถ้าเป็น 0 แสดงว่าเป็นจำนวนบวก วิธีนี้เรียกว่า sign and magnitude
- ตัวอย่าง (4 Bit)
  - ` 5 = 0101`
  - `-5 = 1101`

Signed Bit

- Let's try some operation
- 5 + 2

```
    0 1 0 1       5
+   0 0 1 0       2
=   0 1 1 1       7 (correct)
```

- 5 - 2 = 5 + -2

```
    0 1 0 1       5
+   1 0 1 0      -2
=   1 1 1 1      -7 (wrong)
```


Two's-Complement Encodings

- สิ่งที่ต้องคำนึงถึงในการแทนข้อมูลคือต้องสามารถทำ operation ทางคณิตศาสตร์ได้และให้ผลลัพธ์ถูกต้อง
- The most common computer representation of signed numbers is known as

two's-complement

Two's-Complement Encodings [2]

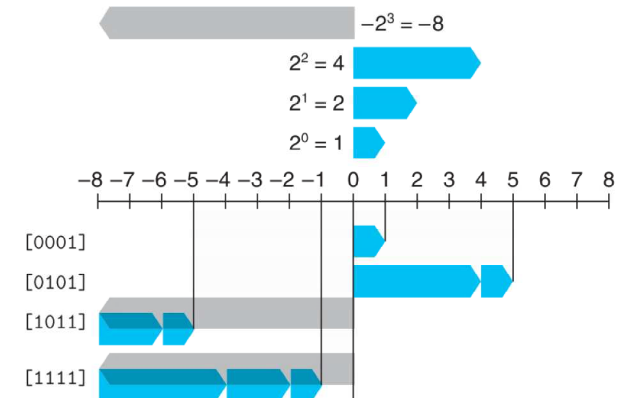
- แนวคิดคือ ในกรณี MSB เป็น 1 นอกจาก จะหมายความว่า จำนวนดังกล่าวเป็นจำนวนลบแล้ว bit นี้ยังมีน้ำหนักค่าเท่ากับ -2^{n-1}
- Bit อื่น ๆ มีค่าปรกติ เช่นกรณี 4 bit (-2^3 to 2^3-1)

$$
\begin{aligned}
0001 &= -0\cdot 2^3 + 0\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0 &= 0+0+0+1 &= 1\\
0101 &= -0\cdot 2^3 + 1\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0 &= 0+4+0+1 &= 5\\
1011 &= -1\cdot 2^3 + 0\cdot 2^2 + 1\cdot 2^1 + 1\cdot 2^0 &= -8+0+2+1 &= -5\\
1111 &= -1\cdot 2^3 + 1\cdot 2^2 + 1\cdot 2^1 + 1\cdot 2^0 &= -8+4+2+1 &= -1
\end{aligned}
$$

Two's-Complement Encodings [3]

- พิจารณาการแสดงค่าแบบ two's-complement บนเส้นจำนวน ในกรณีมี 4 Bit

Figure 2.12 **Two's-complement number examples for** $w = 4$. Bit 3 serves as a sign bit, and so, when set to 1, it contributes $-2^3 = -8$ to the value. This weighting is shown as a leftward-pointing gray bar.



Bit Pattern Comparison (2 bytes)

	12,345		−12,345		53,191	
Weight	Bit	Value	Bit	Value	Bit	Value
1	1	1	1	1	1	1
2	0	0	1	2	1	2
4	0	0	1	4	1	4
8	1	8	0	0	0	0
16	1	16	0	0	0	0
32	1	32	0	0	0	0
64	0	0	1	64	1	64
128	0	0	1	128	1	128
256	0	0	1	256	1	256
512	0	0	1	512	1	512
1,024	0	0	1	1,024	1	1,024
2,048	0	0	1	2,048	1	2,048
4,096	1	4,096	0	0	0	0
8,192	1	8,192	0	0	0	0
16,384	0	0	1	16,384	1	16,384
±32,768	0	0	1	−32,768	1	32,768
Total		12,345		−12,345		53,191

- 12345 (2's complement)
- -12345 (2's complement)
- 53,191 (unsigned)

Note that the latter two have identical bit representations.

2^{15} vs -2^{15}

Bits, Bytes and Integers

- Binary Representation
- Integral Data Type Ranges
- Encodings
- **Signed and Unsigned Conversions, Casting**
- Expanding
- Truncating
- Integer Arithmetic

Signed to Unsigned Conversions

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

Output:

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

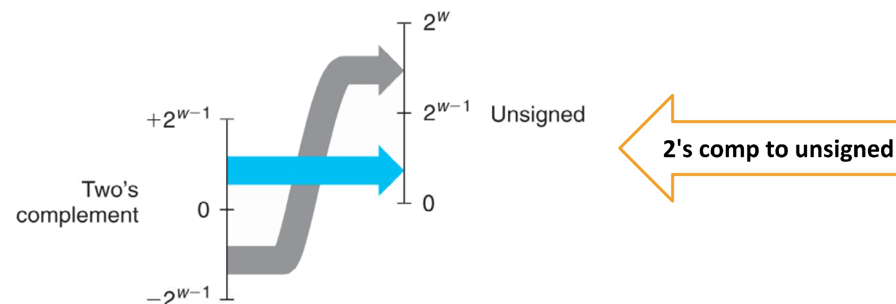
<u>ข้อสังเกต</u> $-12345 + 2^{15} + 2^{15} = 53191$

$-12345 + 2*2^{15} = 53191$

$-12345 + 2^{16} = 53191$

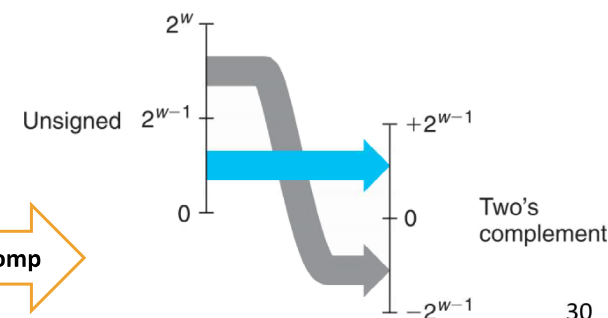
1. กรณีแปลงจาก 2's complement negative to unsigned ให้บวก 2^w (w = จำนวน bit ทั้งหมด)
2. กรณีแปลงจาก 2's complement positive ค่าใน unsigned จะมีค่าเท่ากัน (most significant bit = 0)

Unsigned to Signed Conversions



ในกรณี convert จาก unsigned เป็น 2's complement

- หากตัวเลขมีค่ามากกว่า 2^{w-1} เมื่อ convert จะกลายเป็นจำนวนลบ
- ค่าที่ได้คือ ค่า unsigned – 2^w

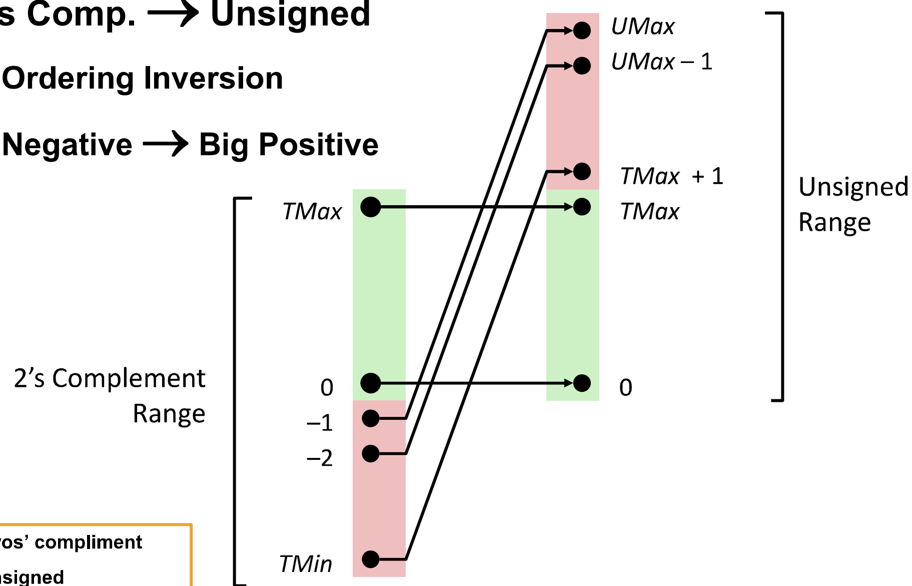


unsigned to 2's comp

from Carnegie Mellon University's 15213 course slide by Greg Kesden

Conversion Visualized

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



T= Twos' compliment
U= Unsigned

from Carnegie Mellon University's 15213 course slide by Greg Kesden

Signed vs. Unsigned in C

- **Constants**
  - **By default are considered to be signed integers**
  - **Unsigned if have "U" as suffix**
    - **0U, 4294967259U**
- **Casting**
  - **Explicit casting between signed & unsigned**

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

from Carnegie Mellon University's 15213 course slide by Greg Kesden

Signed vs. Unsigned in C [2]

- **Implicit casting also occurs via assignments and procedure calls**
  - `tx = ux;`
  - `uy = ty;`
- **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

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 '*'. When either operand of a comparison is unsigned, the other operand is implicitly cast to unsigned. See Web Aside DATA:TMIN for why we write $TMin_{32}$ as −2147483647−1.

from Carnegie Mellon University's 15213 course slide by Greg Kesden

Casting Summary

- **Casting Signed ↔ Unsigned: Basic Rules**
  - **Bit pattern is maintained**
  - **But reinterpreted**
  - **Can have unexpected effects: adding or subtracting** $2w$
- **Expression containing signed and unsigned int**
  - `int` **is cast to** `unsigned`**!!**

Implicit casting should be avoided

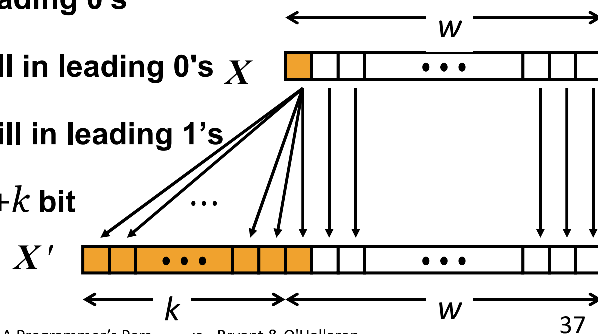
Bits, Bytes and Integers

- Binary Representation
- Integral Data Type Ranges
- Encodings
- Signed and Unsigned Conversions, Casting
- **Expanding**
- Truncating
- Integer Arithmetic

from Carnegie Mellon University's 15213 course slide by Greg Kesden

Expanding

- **Expanding คือกรณีที่มีการ cast จาก integer ที่มีจำนวน bit น้อยกว่าไปมากกว่า**
  เช่น short → int → long
  - **Unsigned: fill in leading 0's**
  - **Signed positive: fill in leading 0's**
  - **Signed negative: fill in leading 1's**
  - **จากรูป w bit to $w+k$ bit**



สรุป: Clone ค่าของ MSB ใส่ใน ทุก bit ที่เพิ่มมาทุกกรณี

from Carnegie Mellon University's 15213 course slide by Greg Kesden

Sign Extension Example

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

short 2 bytes
int 4 bytes

	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**
- **Got expected result (both sign and unsigned)**

Bits, Bytes and Integers

- Binary Representation
- Integral Data Type Ranges
- Encodings
- Signed and Unsigned Conversions, Casting
- Expanding
- **Truncating**
- Integer Arithmetic

from Carnegie Mellon University's 15213 course slide by Greg Kesden

Truncating

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

- **ในทางกลับกัน หากเป็นการ cast ที่ลดจำนวน bit ลง**
  - **Unsigned/signed: bits are truncated (ตัดทิ้ง)**
  - **Result reinterpreted**
  - **Unsigned: mod operation**
  - **Signed: similar to mod**
  - **For small numbers yields expected behavior**

Also should be avoided

Bits, Bytes and Integers

- Binary Representation
- Integral Data Type Ranges
- Encodings
- Signed and Unsigned Conversions, Casting
- Expanding
- Truncating
- **Integer Arithmetic**

Integer Arithmetic

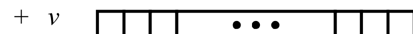
- **Addition**
  - **Unsigned Addition**
  - **Two's complement Addition**
- **Negation**

from Carnegie Mellon University's 15213 course slide by Greg Kesden

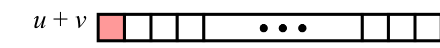
Unsigned Addition

Operands: *w* bits

u

$+ \; v$


True Sum: *w*+1 bits

$u + v$


Discard Carry: *w* bits

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

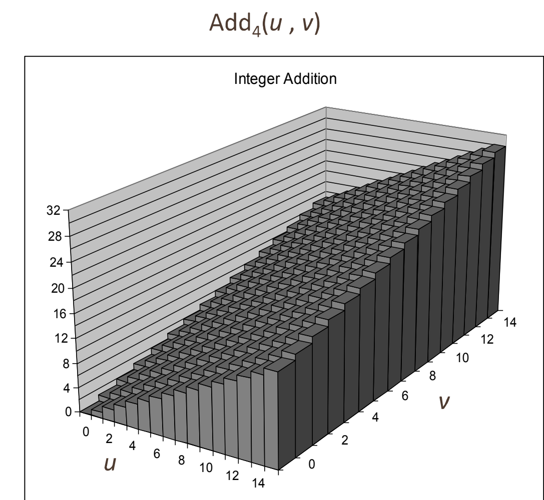
- **Standard Addition Function**
  - **Ignores carry output**
- **Implements Modular Arithmetic**

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

from Carnegie Mellon University's 15213 course slide by Greg Kesden

Visualizing (Mathematical) Integer Addition

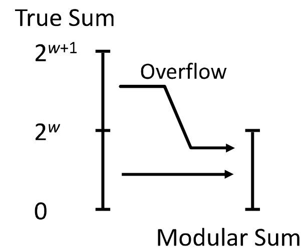
- **Integer Addition**
  - **4-bit integers *u*, *v***
  - **Compute true sum $\text{Add}_4(u , v)$**
  - **Values increase linearly with *u* and *v***
  - **Forms planar surface**

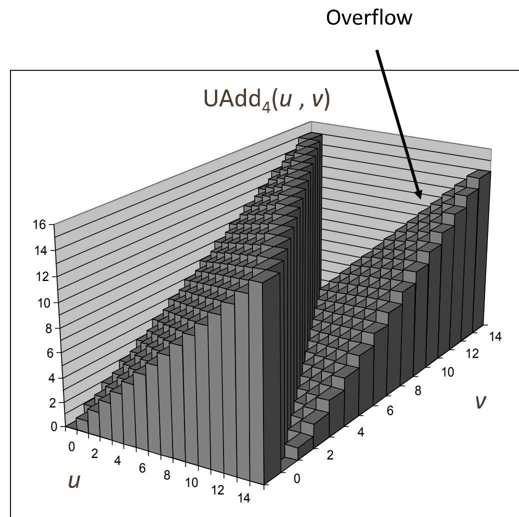


from Carnegie Mellon University's 15213 course slide by Greg Kesden

Visualizing Unsigned Addition

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

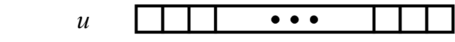


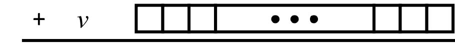


from Carnegie Mellon University's 15213 course slide by Greg Kesden

Two's Complement Addition

Operands: *w* bits

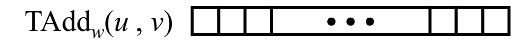




True Sum: *w*+1 bits

$u + v$

Discard Carry: *w* bits



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

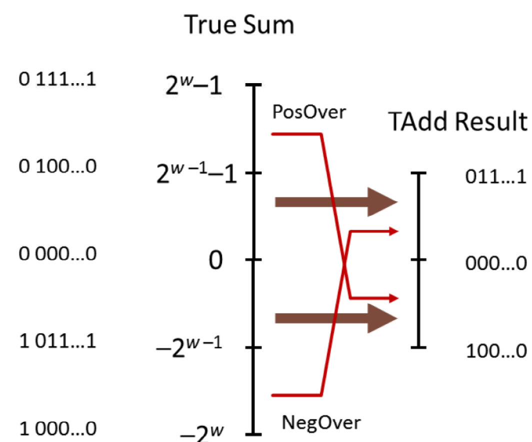
```
int s, t, u, v;
s = (int) ((unsigned) u + (unsigned) v);
t = u + v
```

  - **Will give** `s == t`

from Carnegie Mellon University's 15213 course slide by Greg Kesden

TAdd Overflow

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

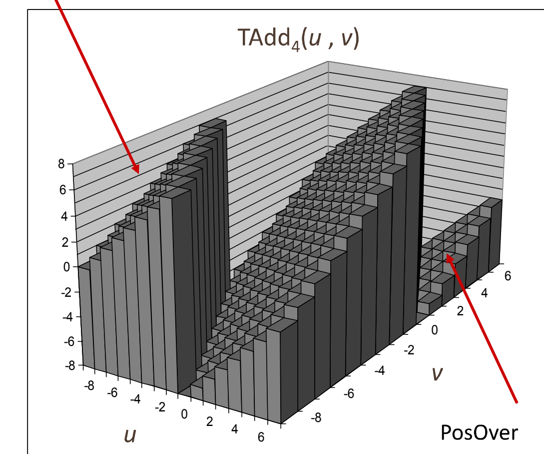


from Carnegie Mellon University's 15213 course slide by Greg Kesden

Visualizing 2's Complement Addition

NegOver

- **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**



2's Complement Negation

- พิจารณา $w = 4$ $(-8 \le x \le 7)$
- จำนวนทุกจำนวน ยกเว้น -8 (2^{w-1}) จะมี negation ตามค่าปกติในคณิตศาสตร์ $-x = 0 - x$

$$-^{t}_{w}\, x = \begin{cases} -2^{w-1}, & x = -2^{w-1} \\ -x, & x > -2^{w-1} \end{cases}$$

- Negation ของ 2^{w-1} จะมีค่า 2^{w-1} (ตัวมันเอง) ในการแทนค่าแบบ two's complement

Note: หากมีแค่การกลับ bit แต่ไม่มี การบวก 1 เรียก Ones' complement

Bit-level Representation of 2's-Complement Negation

Method1: complement the bits and then increment (กลับ bit แล้วบวก 1)

- **In C `-x` and `~x + 1` will give identical results.**

$\vec{x}$		$\sim\vec{x}$		$incr(\sim\vec{x})$	
[0101]	5	[1010]	−6	[1011]	−5
[0111]	7	[1000]	−8	[1001]	−7
[1100]	−4	[0011]	3	[0100]	4
[0000]	0	[1111]	−1	[0000]	0
[1000]	−8	[0111]	7	[1000]	−8

Bit-level Representation of 2's-Complement Negation [2]

Method2: Let k be the position of the rightmost 1, we complement each bit to the left of bit position k (หา 1 ตัวขวาสุด แล้วกลับ bit เฉพาะทางซ้ายของ 1 ตัวนั้น)

x		$-x$	
[1*100*]	−4	[0*100*]	4
[*1000*]	−8	[*1000*]	−8
[010*1*]	5	[101*1*]	−5
[011*1*]	7	[100*1*]	−7

from Carnegie Mellon University's 15213 course slide by Greg Kesden

Why Should I Use Unsigned?

- **Don't Use Just Because Number is Nonnegative**
  - **Easy to make mistakes**

```
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)
  . . .
```

from Carnegie Mellon University's 15213 course slide by Greg Kesden

Why Should I Use Unsigned? [2]

- Do Use When Performing Modular Arithmetic
  - Multiprecision arithmetic
- Do Use When Using Bits to Represent Sets
  - Logical right shift, no sign extension

```
Right shifting in C
unsigned int will result in a logical shift
signed int will result in an arithmetic shift
```


Conclusion

- Computers encode information as bits, generally organized as sequences of bytes.
- Different encodings are used for representing integers, real numbers, and character strings.
- Different models of computers use different conventions for encoding numbers and for ordering the bytes within multi-byte data.
- Most machines use two's-complement encoding of integers.

Conclusion [2]

- Understanding these encodings at the bit level, as well as understanding the mathematical characteristics of the arithmetic operations, is important for writing programs that operate correctly over the full range of numeric values.

Conversion Exercise (Ans.)

Practice Problem 2.3: Fill in the missing entries

Decimal	Binary	Hexadecimal
0	0000 0000	0x00
167	1010 0111	0xA7
62	0011 1110	0x3E
188	1011 1100	0xBC
55	0011 0111	0x37
136	1000 1000	0x88
243	1111 0011	0xF3
82	0101 0010	0x52
172	1010 1100	0xAC
231	1110 0111	0xE7

Range Calculation Exercise (Ans.)

- Fill in the rest of the entries

Bits	Unsigned range	unsigned value	Signed range	Signed value
2	0 to 2^2 – 1	0 to 3	-2^1 to 2^1–1	-2 to 1
4	0 to 2^4 – 1	0 to 15	-2^3 to 2^3–1	-8 to 7
8	0 to 2^8 – 1	0 to 255	-2^7 to 2^7–1	-128 to 127
16	0 to 2^{16} – 1	0 to 65535	-2^{15} to 2^{15}–1	-32768 to 32767
32	0 to 2^{32} – 1	0 to 4294967295	-2^{31} to 2^{31}–1	-2147483648 to - 2147483647
w	0 to 2^w – 1		-2^{w-1} to 2^{w-1}–1	