15-213

"The Class That Gives CMU Its Zip!"

Bits, Bytes, and Integers
August 27, 2009

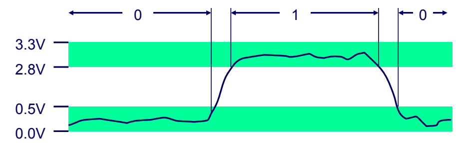
Topics
- Representing information as bits
- Bit-level manipulations
  - Boolean algebra
  - Expressing in C
- Representations of Integers
  - Basic properties and operations
  - Implications for C

`lecture-02.ppt`

Binary Representations

Base 2 Number Representation
- Represent 15213_{10} as 11101101101101_2
- Represent 1.20_{10} as $1.0011001100110011[0011]..._2$
- Represent 1.5213×10^4 as $1.1101101101101_2 \times 2^{13}$

Electronic Implementation
- Easy to store with bistable elements
- Reliably transmitted on noisy and inaccurate wires



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`
    - » Or `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

Back to bits: 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

Not
- ~A = 1 when A=0

~	
0	1
1	0

Or
- A|B = 1 when either A=1 or B=1

|	0	1
0	0	1
1	1	1

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 Apply

Bit-Level Operations in C

Operations &, |, ~, ^ Available in C
- Apply to any "integral" data type
  - `long, int, short, char, unsigned`
- View arguments as bit vectors
- Arguments applied bit-wise

Examples (Char data type)
- `~0x41 --> 0xBE`
  - $\sim 01000001_2$ --> 10111110_2
- `~0x00 --> 0xFF`
  - $\sim 00000000_2$ --> 11111111_2
- `0x69 & 0x55 --> 0x41`
  - 01101001_2 & 01010101_2 --> 01000001_2
- `0x69 | 0x55 --> 0x7D`
  - 01101001_2 | 01010101_2 --> 01111101_2

Contrast: Logic Operations in C

Contrast to Logical Operators

- &&, ||, !
  - View 0 as "Fa
  - Anything n   rue"
  - Al

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

Exam

- !
- !
- !
- `0x69 && 0x55  -->  0x01`
- `0x69 || 0x55  -->  0x01`
- `p && *p` (avoids null pointer access)

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 on right

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

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

Memory organization

Programs refer to data by address

- address space viewed as a large array of bytes
- an address is like an index into that array

Any given computer has a "Word Size"

- nominal size of integer-valued data
  - and, usually, of addresses
- 32 bits is still most common
  - though 64 bits is emerging

Data Representations

Sizes of C Objects (in Bytes)

C Data Type	Typical 32-bit	Intel IA32	x86-64
unsigned [int]	4	4	4
int	4	4	4
long int	4	4	4
char	1	1	1
short	2	2	2
float	4	4	4
double	8	8	8
long double	–	10/12	10/12
char *	4	4	8

» Or any other pointer

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

- 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**

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

Signed vs. unsigned ints in C

Constants

- By default, considered to be signed integers
- Unsigned if have "U" as suffix

```
0U, 4294967259U
```

Casting

- Can explicitly cast between signed & unsigned

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

- Implicit casting also occurs via assignments (and function calls)

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


Casting Surprises

Expression Evaluation

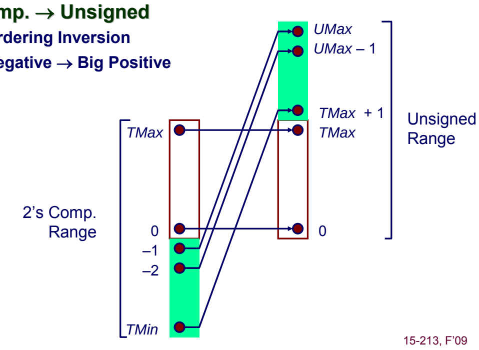
- If mix unsigned and signed in single expression, signed values implicitly cast to unsigned
- Including comparison operations <, >, ==, <=, >=
- Examples for *W* = 32

Constant$_1$	Constant$_2$	Relation	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

Visual of casting

2's Comp. → Unsigned

- Ordering Inversion
- Negative → Big Positive



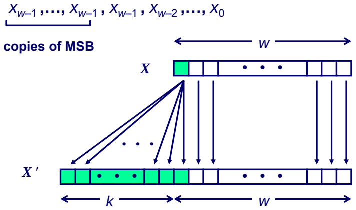
Sign Extension

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

```
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

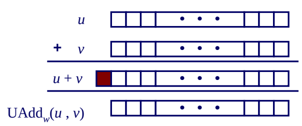
- Converting from smaller to larger integer data type
- C automatically performs sign extension

Unsigned Addition

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits



Standard Addition Function

- Ignores carry output

Implements Modular Arithmetic

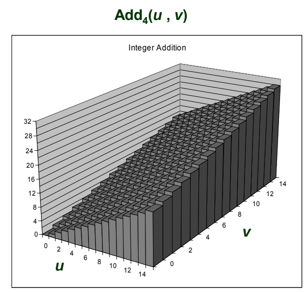
$s = \mathrm{UAdd}_w(u, v) = u + v \mod 2^w$

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

Visualizing what should happen

Integer Addition

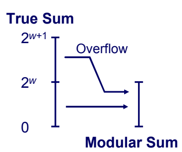
- As in, actual math, not computer math
- 4-bit integers *u*, *v*
- Compute true sum $\mathrm{Add}_4(u, v)$
- Values increase linearly with *u* and *v*
- Forms planar surface

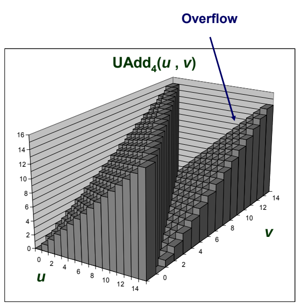


Visualizing actually happens

Wraps Around

- If true sum ≥ 2^w
- At most once



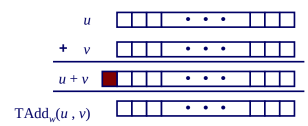


Two's Complement Addition

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits



TAdd and UAdd have Identical Bit-Level Behavior

- Signed vs. unsigned addition in C:

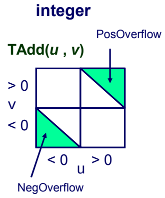
```
int s, t, u, v;
s = (int) ((unsigned) u + (unsigned) v);
 t = u + v
```

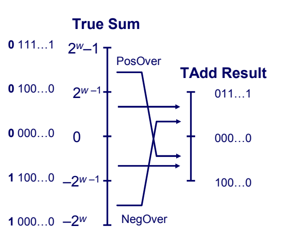
- Will give `s == t`

Characterizing TAdd

Functionality

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





$$TAdd_w(u,v) = \begin{cases} u+v+2^{w-1} & u+v < TMin_w \text{ (NegOver)} \\ u+v & TMin_w \leq u+v \leq TMax_w \\ u+v-2^{w-1} & TMax_w < u+v \text{ (PosOver)} \end{cases}$$

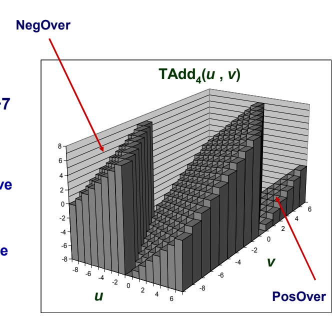
Visualizing what actually happens

Values

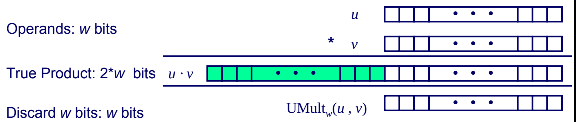
- 4-bit two's comp.
- Range from -8 to +7

Wraps Around

- If sum ≥ 2^{w-1}
  - Becomes negative
  - At most once
- If sum < -2^{w-1}
  - Becomes positive
  - At most once



Unsigned Multiplication in C



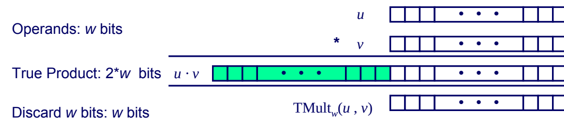
Standard Multiplication Function

- Ignores high order w bits

Implements Modular Arithmetic

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

Signed Multiplication in C



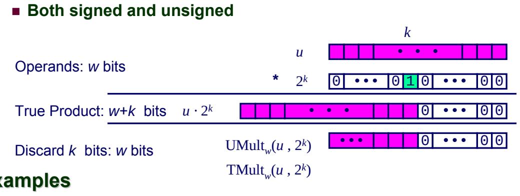
Standard Multiplication Function

- 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



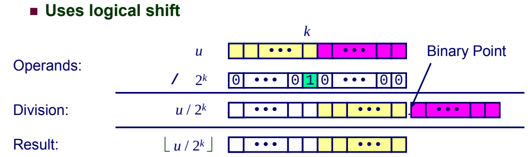
Examples

- `u << 3 == u * 8`
- `u << 5 - u << 3 == u * 24`
- Most machines shift and add faster than multiply
  - Compiler generates this code automatically

Unsigned Power-of-2 Divide with Shift

Quotient of Unsigned by Power of 2

- `u >> k` gives $\lfloor u / 2^k \rfloor$
- Uses logical shift

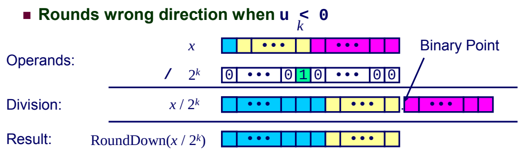


	Division	Computed	Hex	Binary
x	15213	15213	3B 6D	00111011 01101101
x >> 1	7606.5	7606	1D B6	**0**0011101 10110110
x >> 4	950.8125	950	03 B6	**0000**0011 10110110
x >> 8	59.4257813	59	00 3B	**00000000** 00111011

Signed Power-of-2 Divide with Shift

Quotient of Signed by Power of 2

- `x >> k` gives $\lfloor x / 2^k \rfloor$
- Uses arithmetic shift
- Rounds wrong direction when `u < 0`



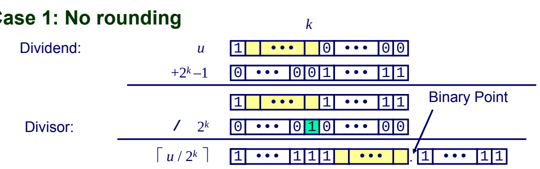
	Division	Computed	Hex	Binary
y	-15213	-15213	C4 93	11000100 10010011
y >> 1	-7606.5	-7607	E2 49	**1**1100010 01001001
y >> 4	-950.8125	-951	FC 49	**1111**1100 01001001
y >> 8	-59.4257813	-60	FF C4	**11111111** 11000100

Correct Power-of-2 Divide

Quotient of Negative Number by Power of 2

- Want $\lceil x / 2^k \rceil$ (Round Toward 0)
- Compute as $\lfloor (x+2^k-1) / 2^k \rfloor$
  - In C: `(x + (1<<k)-1) >> k`
  - Biases dividend toward 0

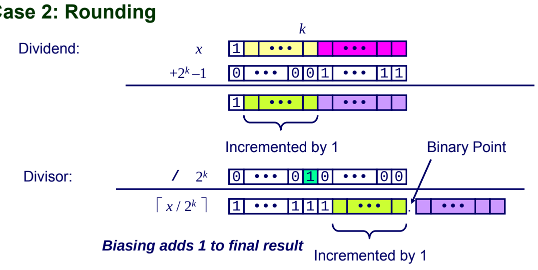
Case 1: No rounding



Biasing has no effect

Correct Power-of-2 Divide (Cont.)

Case 2: Rounding



Representing Strings

`char S[6] = "15213";`

Strings in C

- Represented by 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

31
35
32
31
33
00

What's next

Recitation on Monday

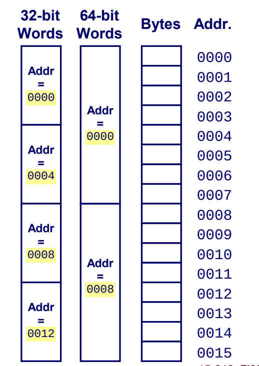
- autolab, fish machines, datalab
- get started on lab #1 before then…

Floating point (Tues): representations and arithmetic

Word-Oriented Memory Organization

Addresses Specify Byte Locations

- Address of first byte in word
- Addresses of successive words differ by word size
  - e.g., 4 (32-bit) or 8 (64-bit)



Byte ordering in multi-byte "words"

Big Endian (e.g., SPARC, Power PC)

- Least significant byte has highest address

Little Endian (e.g., x86)

- Least significant byte has lowest 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		

Examining Data Representations

Code to Print Byte Representation of Data

- Casting pointer to `unsigned char *` creates byte array

```
typedef unsigned char *pointer;

void show_bytes(pointer start, int len)
{
  int i;
  for (i = 0; i < len; i++)
    printf("0x%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 Integers

```
int A = 15213;
int B = -15213;
long int C = 15213;
```

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

IA32, x86-64 A	Sun A
6D	00
3B	00
00	3B
00	6D

IA32 C	x86-64 C	Sun C
6D	6D	00
3B	3B	00
00	00	3B
00	00	6D
	00	
	00	
	00	
	00	

IA32, x86-64 B	Sun B
93	FF
C4	FF
FF	C4
FF	93

Two's complement representation
(Covered later)

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 4 bytes: `0x000012ab`
- Split into bytes: `00 00 12 ab`
- Reverse: `ab 12 00 00`

Representing Pointers

```
int B = -15213;
int *P = &B;
```

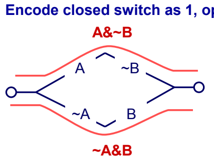
Sun P	IA32 P	x86-64 P
EF	D4	0C
FF	F8	89
FB	FF	EC
2C	BF	FF
		FF
		7F
		00
		00

Different compilers & machines assign different locations to objects

Application of Boolean Algebra

Applied to Digital Systems by Claude Shannon

- 1937 MIT Master's Thesis
- Reason about networks of relay switches
  - Encode closed switch as 1, open switch as 0



Connection when

A&~B | ~A&B

= A^B

Integer C Puzzles

- Assume 32-bit word size, two's complement integers
- For each of the following C expressions, either:
  - Argue that is true for all argument values
  - Give example where not true

Initialization

```
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`

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>`
  - K&R App. B11
- Declares constants, e.g.,
  - `ULONG_MAX`
  - `LONG_MAX`
  - `LONG_MIN`
- Values platform-specific

Unsigned & Signed Numeric Values

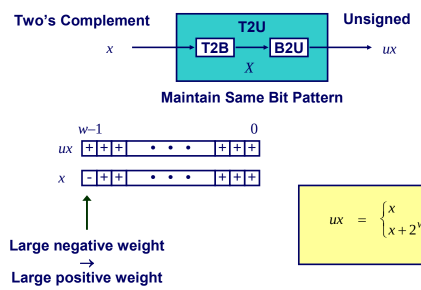
X	B2U(*X*)	B2T(*X*)
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	–8
1001	9	–7
1010	10	–6
1011	11	–5
1100	12	–4
1101	13	–3
1110	14	–2
1111	15	–1

Equivalence
- Same encodings for nonnegative values

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

⇒ Can Invert Mappings
- U2B(*x*) = $B2U^{-1}(x)$
  - Bit pattern for unsigned integer
- T2B(*x*) = $B2T^{-1}(x)$
  - Bit pattern for two's comp integer

Relation between Signed & Unsigned



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

When should I use unsigned?

Don't Use Just Because Number Nonzero
- 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)
  . . .
```

Do Use When Performing Modular Arithmetic
- Multiprecision arithmetic

Do Use When Need Extra Bit's Worth of Range
- Working right up to limit of word size

Negating with Complement & Increment

Claim: Following Holds for 2's Complement
`~x + 1 == -x`

Complement
- Observation: `~x + x == 1111…11`$_2$ `== -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

Increment
- `~x + ~~x~~ + (~~-x~~ + 1)   == ~~-1~~ + (-x + ~~1~~)`
- `~x + 1   ==   -x`

Warning: Be cautious treating `int`'s as integers
- OK here

Comp. & Incr. 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
y	-15213	C4 93	11000100 10010011

0

	Decimal	Hex	Binary
0	0	00 00	00000000 00000000
~0	-1	FF FF	11111111 11111111
~0+1	0	00 00	00000000 00000000

Mathematical Properties

Modular Addition Forms an *Abelian Group*

- Closed under addition
  - $0 \leq UAdd_w(u, v) \leq 2^w - 1$
- Commutative
  - $UAdd_w(u, v) = UAdd_w(v, u)$
- Associative
  - $UAdd_w(t, UAdd_w(u, v)) = UAdd_w(UAdd_w(t, u), v)$
- 0 is additive identity
  - $UAdd_w(u, 0) = u$
- Every element has additive inverse
  - Let $UComp_w(u) = 2^w - u$
  - $UAdd_w(u, UComp_w(u)) = 0$

Mathematical Properties of TAdd

Isomorphic Algebra to UAdd

- $TAdd_w(u, v) = U2T(UAdd_w(T2U(u), T2U(v)))$
  - 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

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

Multiplication

Computing Exact Product of *w*-bit numbers *x*, *y*

- Either signed or unsigned

Ranges

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

Maintaining Exact Results

- Would need to keep expanding word size with each product computed
- Done in software by "arbitrary precision" arithmetic packages

Compiled Multiplication Code

C Function

```
int mul12(int x)
{
  return x*12;
}
```

Compiled Arithmetic Operations

```
 leal (%eax,%eax,2), %eax
 sall $2, %eax
```

Explanation

```
 t <- x+x*2
 return t << 2;
```

- C compiler automatically generates shift/add code when multiplying by constant

Compiled Unsigned Division Code

C Function

```
unsigned udiv8(unsigned x)
{
  return x/8;
}
```

Compiled Arithmetic Operations

```
 shrl $3, %eax
```

Explanation

```
 # Logical shift
 return x >> 3;
```

- Uses logical shift for unsigned

For Java Users

- Logical shift written as >>>

Compiled Signed Division Code

C Function

```
int idiv8(int x)
{
  return x/8;
}
```

Compiled Arithmetic Operations

```
 testl %eax, %eax
 js   L4
L3:
 sarl $3, %eax
 ret
L4:
 addl $7, %eax
 jmp  L3
```

Explanation

```
 if x < 0
   x += 7;
 # Arithmetic shift
 return x >> 3;
```

- Uses arithmetic shift for int

For Java Users

- Arith. shift written as >>

Properties of Unsigned Arithmetic

Unsigned Multiplication with Addition Forms Commutative Ring

- Addition is commutative group
- Closed under multiplication
  - $0 \leq UMult_w(u, v) \leq 2^w - 1$
- Multiplication Commutative
  - $UMult_w(u, v) = UMult_w(v, u)$
- Multiplication is Associative
  - $UMult_w(t, UMult_w(u, v)) = UMult_w(UMult_w(t, u), v)$
- 1 is multiplicative identity
  - $UMult_w(u, 1) = u$
- Multiplication distributes over addtion
  - $UMult_w(t, UAdd_w(u, v)) = UAdd_w(UMult_w(t, u), UMult_w(t, v))$

Properties of Two's Comp. Arithmetic

Isomorphic Algebras

- Unsigned multiplication and addition
  - Truncating to *w* bits
- Two's complement multiplication and addition
  - Truncating 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.,
  - $u > 0 \quad \Rightarrow \quad u + v > v$
  - $u > 0, v > 0 \quad \Rightarrow \quad u \cdot v > 0$
- These properties are not obeyed by two's comp. arithmetic
  - *TMax* `+ 1` `==` *TMin*
  - `15213 * 30426 == -10030` (16-bit words)

Integer C Puzzles Revisited

Initialization

```
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`