INTEGER ARITHMETIC

CAS CS 210 Computer System
CSAPP 2.3

BASICS OF ADDITION

- 1+ 1 = 2 : 1b + 1b = 10b
- addition of 3 bits

  1b + 1b + 1b = (1b + 1b) + 1b = 10b + 1b = 11b
- addition of vectors of bits:

x	y	x+y
0	0	0
1	0	1
0	1	1
1	1	10

Result can OVERFLOW word size of inputs

```
                         1 carry 11
  1110 | 1110 | 1110 |  1110 | 1110
  0101 | 0101 | 0101 |  0101 | 0101
+----- |----- |----- |----- |-----
       |    1 |   11 |   011 |10011 overflows
```

At most 1 bit is need to track overflow

UNSIGNED ADDITION

```
unsigned long long;
unsigned int;
unsigned short;
unsigned char;
```

w=3
2^3=8
Unsigned:
min=000
max=2^3-1
=1000-1
=111

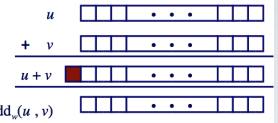
x	y	x+y	result	3bit
0	0	000 000 000	0	000=0
1	1	1 001 001 010	2	010=2
6	1	110 001 111	7	111=7
7	1	111 111 001 1000	8	000=0
6	4	1 110 100 1010	10	010=2
7	7	111 111 111 1110	14	110=6

Unsigned Addition

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits



Standard Addition Function

- Ignores carry output

Implements Modular Arithmetic

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

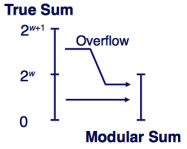
OVERFLOW

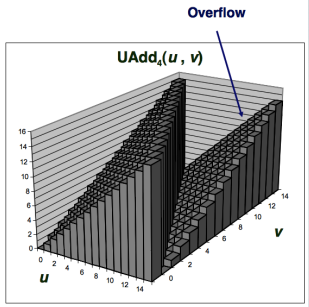
- OVERFLOW: when the full integer result cannot fit within the word size limits of the data type
- occurs when x+y >= 2^w
- C does **not** signal overflow errors they occur silently. Unsigned overflow can be detected if (x+y) < x
- We will see that most processors do indicate if an overflow occurs
- Both addition and multiplication can suffer from it

Visualizing Unsigned Addition

Wraps Around

- If true sum ≥ 2^w
- At most once





SIGNED ADDITION

```
long long;
    int;
  short;
   char;
```

w=3
2^3=8
Signed:
min=100=-4
max=011=3

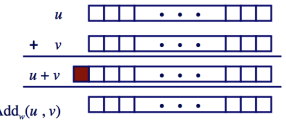
x	y	x+y	result	3bit
0	0	000 000 000	0	000=0
2	1	010 001 011	3	011=3
3	1	11 011 001 100	4	100=-4
-3	-1	111 101 111 1100	-4	100=-4
-4	-1	1 100 111 1011	-5	011=3
-4	-4	1 100 100 1000	-8	000=0

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`**

OVERFLOW & "UNDER-FLOW"

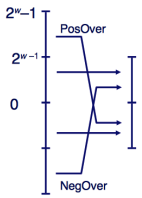
- OVERFLOW: when the result exceeds 2's Complement Max
- UNDERFLOW: when result exceeds 2's Complement Min
- C does **not** signal either errors they occur silently.
- Again most processors do indicate if either occurs
- Both addition and multiplication can suffer from it

Detecting 2's Comp. Overflow

Task

- **Given** $s = \text{TAdd}_w(u , v)$
- **Determine if** $s = \text{Add}_w(u , v)$
- **Example**

```
int s, u, v;
s = u + v;
```



Claim

- **Overflow iff either:**
  - $u, v < 0, s \geq 0$ **(NegOver)**
  - $u, v \geq 0, s < 0$ **(PosOver)**

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

state expression in english

(both +or both -) && (sum does not match sign of u)

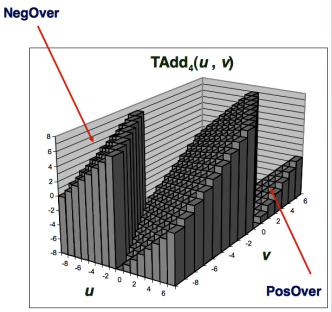
Visualizing 2's Comp. Addition

Values

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

Wraps Around

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



MULTIPLYING BY A CONSTANT (2.3.6)

- HW support for multiplication but slow compared to ~, &, |, +, >>, <<
- So when it can the compiler tries to replace multiplication with equivalent computations using the more primitive ops
- key rule used is that both for unsigned and signed

  $x * 2^k = x << k$ for $0 <= k < w$ (see 2.3.6 for proof):

  eg. $3 * 2 = 3 << 1$ and $3 * 4 = (3 << 1) << 1 = 3<<2$

`x*14;` compiler → `x * (2^3 + 2^2 + 2^1 )=(x<<3) + (x<<2) + (x<<1)`

or even better

`x * (2^4 - 2^1 )= (x<<4) - (x<<1)`

DIVIDING BY PWRS OF 2 (2.3.7)

Integer division defined to always round towards zero
7/2 = 3 and -7/2 = -3

- like multiplication we would like to exploit right shifts to replace divisions by powers of two: $x / 2^k \rightarrow x >> k : 0>k<w$: eg.
  7/2 →7 >> 1→[0111] >> 1 →[0011]=3
- this works for positive x as truncation will round towards 0 but it does not work for negative x that requires rounding eg.:

  -7/2 = [1001]/2 supposed be -3 but

  [1001] >> 1 = [1100] = -4
- truncation operates like floor but for negative division we need ceiling.

DIVIDING BY PWRS OF 2 (2.3.7)

Integer division defined to always round towards zero
7/2 = 3 and -7/2 = -3

- like multiplication we would like to exploit right shifts to replace divisions by powers of two: x / 2^k → x >>k : 0>k<w: eg.
  7/2 →7 >> 1→[0111] >> 1 →[0011]=3
- this works for positive x as truncation will round towards 0 but it does not work for negative x that requires rounding eg.:
  -7/2 = [1001]/2 supposed be -3 but
  [1001] **>>** 1 = [**1**100] = -4
- truncation operates like floor but for negative division we need ceiling. We can exploit the fact that: $\lceil x/y \rceil = \lfloor (x+y-1)/y \rfloor$ so that x / 2^k →`(x<0 ? x+(1<<k)-1 : x) >> k`

for 2's comp machine using arithmetic right shifts

BRIEF INTRO TO FLOATING POINT

REAL NUMBERS : FLOATING POINT

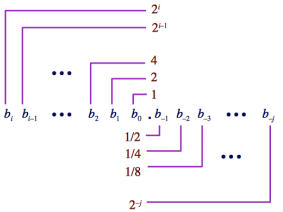
IEEE Standard 754

- Established in 1985 as uniform standard for floating point arithmetic
  - Before that, many idiosyncratic formats
- Supported by all major CPUs

Driven by Numerical Concerns

- Nice standards for rounding, overflow, underflow
- Hard to make go fast
  - Numerical analysts predominated over hardware types in defining standard

Fractional Binary Numbers



Representation

- Bits to right of "binary point" represent fractional powers of 2

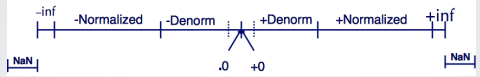
Representable Numbers

Limitation

- Can only exactly represent numbers of the form $x/2^k$
- Other numbers have repeating bit representations

Value	Representation
1/3	$0.0101010101[01]\ldots_2$
1/5	$0.001100110011[0011]\ldots_2$
1/10	$0.0001100110011[0011]\ldots_2$

Summary of Floating Point Real Number Encodings

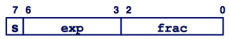


Tiny Floating Point Example

8-bit Floating Point Representation

- the sign bit is in the most significant bit.
- the next four bits are the exponent, with a bias of 7.
- the last three bits are the `frac`

- **Same General Form as IEEE Format**
  - normalized, denormalized
  - representation of 0, NaN, infinity



Values Related to the Exponent

Exp	exp	E	2^E	
0	0000	-6	1/64	(denorms)
1	0001	-6	1/64	
2	0010	-5	1/32	
3	0011	-4	1/16	
4	0100	-3	1/8	
5	0101	-2	1/4	
6	0110	-1	1/2	
7	0111	0	1	
8	1000	+1	2	
9	1001	+2	4	
10	1010	+3	8	
11	1011	+4	16	
12	1100	+5	32	
13	1101	+6	64	
14	1110	+7	128	
15	1111	n/a		(inf, NaN)

Dynamic Range

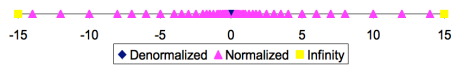
	s	exp	frac	*E*	Value	
Denormalized numbers	0	0000	000	-6	0	
	0	0000	001	-6	1/8*1/64 = 1/512	← closest to zero
	0	0000	010	-6	2/8*1/64 = 2/512	
	…					
	0	0000	110	-6	6/8*1/64 = 6/512	
	0	0000	111	-6	7/8*1/64 = 7/512	← largest denorm
	0	0001	000	-6	8/8*1/64 = 8/512	← smallest norm
	0	0001	001	-6	9/8*1/64 = 9/512	
	…					
	0	0110	110	-1	14/8*1/2 = 14/16	
	0	0110	111	-1	15/8*1/2 = 15/16	← closest to 1 below
Normalized numbers	0	0111	000	0	8/8*1 = 1	
	0	0111	001	0	9/8*1 = 9/8	← closest to 1 above
	0	0111	010	0	10/8*1 = 10/8	
	…					
	0	1110	110	7	14/8*128 = 224	
	0	1110	111	7	15/8*128 = 240	← largest norm
	0	1111	000	n/a	inf	

Distribution of Values

6-bit IEEE-like format

- e = 3 exponent bits
- f = 2 fraction bits
- Bias is 3

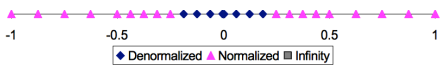
Notice how the distribution gets denser toward zero.



Distribution of Values (close-up view)

6-bit IEEE-like format

- e = 3 exponent bits
- f = 2 fraction bits
- Bias is 3



EXTRA

WHY DOES 2'S COMPLEMENT WORK

WHY IS TAdd SAME AS UAdd?

Single-bit subtraction

a	b	a-b
0	0	0
1	0	1
0	1	1
1	1	0

Multi-bit subtraction

```
10b - 1b = 1
10 - 1b - 1b = 0
```

What Does -x mean? 0 subtract x

Consider Binary case of -5 with w=4

Remember h 10b - 1 = 1 raction works

Borrow as needed and marking extra subtraction of 1

	0000	0000	0000	0000
subtract	0101	0101	0101	0101
	-----	-----	-----	-----
		1	11	011	1011

GIVEN FIXED WIDTH "w"

	0000	=	10000
subtract	0101		0101
	-----		-----
	1011		01011

subtract from 2^w is the same

So a negative of number in a computer is the same as subtracting it from 2^w

AND

$2^w = 2^w - 1 + 1$

subtract { add { 1 ← + 1 ; 1111 ← $2^w - 1$ } ; 0101 }

BUT

subtract { add { 1 ← + 1 ; 1111 ← 2^w - 1 ; 0101 } }

by Associativity

(1 + 1111) - 0101 =

1 + (1111 - 0101)

for any binary x of length w bits $(2^w - 1) - x$ is the same as inverting all the bits of x (eg ~x). Consider every bitwise subtraction will result in 0 where there is a 1 in x and 1 every where there is a 0 in x.

a	b	a-b
0	0	0
1	0	1
0	1	-1
1	1	0

Therefore :

(1 + 1111) - 0101 = 1 + ~0101

= 1 + 1010

= 1011

SO

So a negative of number in a computer is the same as subtracting it from $2^w = (2^w - 1) + 1$

$$\begin{array}{r} 0000 \\ -\ 0101 \\ \hline \end{array} = \begin{array}{r} 10000 \\ -\ 0101 \\ \hline \end{array} = \left\{ \begin{array}{r} 1 \\ +1111 \\ 0101 \\ \hline \end{array} \right. = \begin{array}{r} 1 \\ +\ \sim 0101 \\ \hline \end{array}$$

taking the negative of a number, that is, subtracting a number from 0, is the same as inverting the bits and adding one which is the definition of 2's complement.
SO a 2's complement is mathematically the negative of the number!

CONSTRUCTED

$$B2T_w(\vec{x}) \doteq -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i$$

1. 0
2. $1 <= x < 2^{w-1}$
3. -2^{w-1}
4. $(\sim x + 1)$ for $1 <= x < 2^{w-1}$

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

~x + 1 mappings

Since ~x+1 is the negative of x normal binary addition (UAdd) just works!

MULTIPLICATION

UNSIGNED MULTIPLICATION (2.3.4)

$0 \leq x, y \leq 2^{w-1} \Rightarrow 0 \leq x \cdot y \leq (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$

This could require 2*w bits to represent

- straight forward truncation of x times y to w bits:

$$x *_w^{\mathtt{u}} y = (x \cdot y) \mathtt{mod} 2^w$$

TWO'S-COMPLEMENT MULTIPLICATION (2.3.5)

bit-level representation of unsigned and 2's complement multiplication the same (see 2.3.5 for simple proof):

$$x *_w^{\mathtt{t}} y = U2T_w((x \cdot y) \mathtt{mod} 2^w)$$

some simple 3 bit examples

mode	x		y		x * y		trunc x * y	
unsigned	5	[101]	3	[011]	15	[001111]	7	[111]
2's comp.	-3	[101]	3	[011]	-9	[110111]	-1	[111]
unsigned	4	[100]	7	[111]	28	[011100]	4	[100]
2's comp.	-4	[100]	-1	[111]	4	[000100]	-4	[100]
unsigned	3	[011]	3	[011]	9	[001001]	1	[001]
2's comp.	3	[011]	3	[011]	9	[001001]	1	[001]

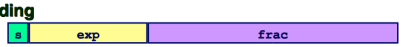
FLOATING POINT

Floating Point Representation

Numerical Form

- $-1^s M 2^E$
  - Sign bit *s* determines whether number is negative or positive
  - Significand *M* normally a fractional value in range [1.0,2.0).
  - Exponent *E* weights value by power of two

Encoding



- MSB is sign bit
- `exp` field encodes *E*
- `frac` field encodes *M*

Floating Point Precisions

Encoding

- MSB is sign bit
- `exp` field encodes *E*
- `frac` field encodes *M*

Sizes

- Single precision: 8 `exp` bits, 23 `frac` bits
  - 32 bits total
- Double precision: 11 `exp` bits, 52 `frac` bits
  - 64 bits total
- Extended precision: 15 `exp` bits, 63 `frac` bits
  - Only found in Intel-compatible machines
  - Stored in 80 bits
    - 1 bit wasted

"Normalized" Numeric Values

Condition

- exp = 000…0 and exp = 111…1

Exponent coded as *biased* value

E = Exp – Bias

- *Exp* : unsigned value denoted by exp
- *Bias* : Bias value
  - » Single precision: 127 (*Exp*: 1…254, *E*: -126…127)
  - » Double precision: 1023 (*Exp*: 1…2046, *E*: -1022…1023)
  - » in general: *Bias* = 2^{e-1} - 1, where e is number of exponent bits

Significand coded with implied leading 1

M = $1.xxx…x_2$

- xxx…x: bits of frac
- Minimum when 000…0 (*M* = 1.0)
- Maximum when 111…1 (*M* = 2.0 – ⊙)
- Get extra leading bit for "free"

Normalized Encoding Example

Value

```
Float F = 15213.0;
```

- $15213_{10} = 11101101101101_2 = 1.1101101101101_2 \times 2^{13}$

Significand

M = 1.1101101101101_2

frac = $11011011011010000000000_2$

Exponent

E = 13

Bias = 127

Exp = 140 = 10001100_2

Floating Point Representation (Class 02):

```
Hex:      4    6    6    D    B    4    0    0
 Binary:      0100 0110 0110 1101 1011 0100 0000
0000

140:      100 0110 0

15213:              1110 1101 1011 01
```

Denormalized Values

Condition

- exp = 000…0

Value

- Exponent value *E* = –*Bias* + 1
- Significand value *M* = $0.xxx…x_2$
  - xxx…x: bits of frac

Cases

- exp = 000…0, frac = 000…0
  - Represents value 0
  - Note that have distinct values +0 and –0
- exp = 000…0, frac = 000…0
  - Numbers very close to 0.0
  - Lose precision as get smaller
  - "Gradual underflow"

Special Values

Condition

- `exp` = `111...1`

Cases

- `exp` = `111...1`, `frac` = `000...0`
  - Represents value +/– (infinity)
  - Operation that overflows
  - Both positive and negative
- `exp` = `111...1`, `frac` = `000...0`
  - Not-a-Number (NaN)
  - Represents case when no numeric value can be determined

Special Properties of Encoding

FP Zero Same as Integer Zero

- All bits = 0

Can (Almost) Use Unsigned Integer Comparison

- Must first compare sign bits
- Must consider -0 = 0
- NaNs problematic
  - Will be greater than any other values
  - What should comparison yield?
- Otherwise OK
  - Denorm vs. normalized
  - Normalized vs. infinity