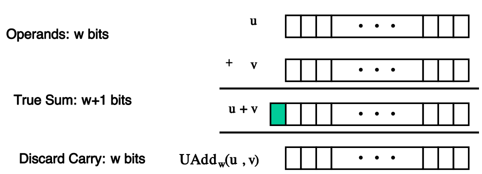
Arithmetic Operations

Outline

- Arithmetic Operations
  - Unsigned addition, multiplication
  - Signed addition, negation, multiplication
  - Using Shift to perform power-of-2 multiply
- Suggested reading
  - Chap 2.3

Unsigned Addition



Unsigned Addition

- Standard Addition Function
  - Ignores carry output
- Implements Modular Arithmetic
  - $s = \text{UAdd}_w(u, v) = (u + v) \bmod 2^w$

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

Unsigned Addition

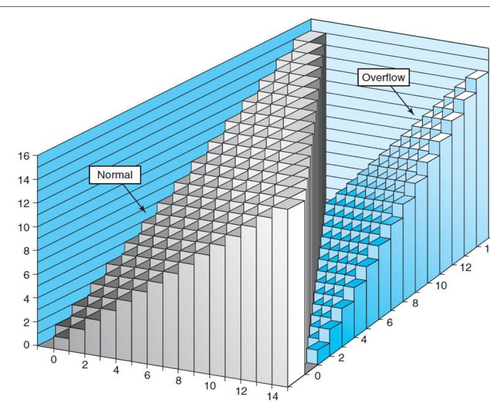
Practice Problem 2.27
Write a function with the following prototype:

/* Determine whether arguments can be added without overflow */

int uadd_ok(unsigned x, unsigned y);

This function should return 1 if arguments x and y can be added without causing overflow

Overflow iff (X+Y) < X



Unsigned Addition Forms an Abelian Group

- Closed under addition
  - $0 \leq UAdd_w(u, v) \leq 2^w - 1$
- Associative
  - $UAdd_w(t, UAdd_w(u, v)) = UAdd_w(UAdd_w(t, u), v)$
- 0 is additive identity
  - $UAdd_w(u, 0) = u$

7

Unsigned Addition Forms an Abelian Group

- Every element has additive inverse
  - Let $UComp_w(u) = 2^w - u$
  - $UAdd_w(u, UComp_w(u)) = 0$
- Commutative
  - $UAdd_w(u, v) = UAdd_w(v, u)$

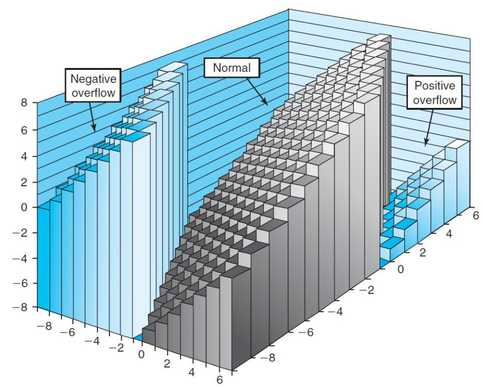
8

Signed Addition

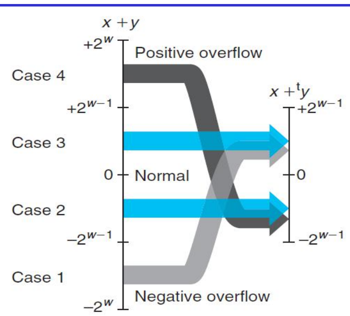
- Functionality
  - True sum requires $w+1$ bits
  - Drop off MSB
  - Treat remaining bits as 2's comp. integer

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

9



Signed Addition



11

Detecting Tadd Overflow

- Task
  - Given $s = TAdd_w(u, v)$
  - Determine if $s = Add_w(u, v)$
- 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);

12

Mathematical Properties of TAdd

- Two's Complement Under TAdd Forms a Group
  - Closed, Commutative, Associative, 0 is additive identity
  - Every element has additive inverse
    - Let $TComp_w(u) = \begin{cases} -u & u \neq TMin_w \\ TMin_w & u = TMin_w \end{cases}$
    - $TAdd_w(u, TComp_w(u)) = 0$

13

Detecting Tadd Overflow

```
int tadd_ok_bugy(int x, int y)
{
        int sum = x + y ;
        return (sum-x == y) && (sum-y == x)
}
```

Abelian group(x+y-x=y+x-x, always true)

```
int tsub_ok_bugy(int x, int y)
{
        return tadd_ok(x, -y) ;
}
```

Set y to TMIN, -y is also TMIN. If x is negative, add will always overflow, sub will not.

14

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

15

Negating with Complement & Increment

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

- In C
  - ~x + 1 == -x
- Complement
  - Observation: ~x + x == 1111…111 == -1
- Increment
  - ~x + x + (-x + 1) == -1 + (-x + 1)
  - ~x + 1 == -x

16

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 $TMin_w^2$

17

Multiplication

- Unsigned

$$x *^{u}_{w} y = (x \cdot y) \bmod 2^w$$

- Signed

$$x *^{t}_{w} y = U2T_w((x \cdot y) \bmod 2^w)$$

- Given two bit vectors $\vec{x}$ and $\vec{y}$

$B2U_w(\vec{x}) *^{u}_{w} B2U_w(\vec{y})$ is identical to $B2T_w(\vec{x}) *^{t}_{w} B2T_w(\vec{y})$ in binary

18

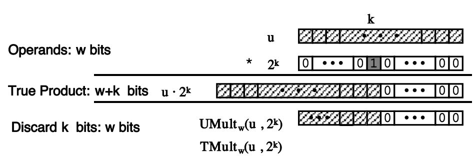
Multiplication

Mode	x		y		$x \cdot y$		Truncated $x \cdot y$	
Unsigned	5	[101]	3	[011]	15	[001111]	7	[111]
Two's complement	−3	[101]	3	[011]	−9	[110111]	−1	[111]
Unsigned	4	[100]	7	[111]	28	[011100]	4	[100]
Two's complement	−4	[100]	−1	[111]	4	[000100]	−4	[100]
Unsigned	3	[011]	3	[011]	9	[001001]	1	[001]
Two's complement	3	[011]	3	[011]	9	[001001]	1	[001]

Multiplication

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

Power-of-2 Multiply with Shift



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 much faster than multiply
    - Compiler will generate this code automatically

Security Vulnerability in the XDR Library

```
/*
 * Illustration of code vulnerability similar to that found in
 * Sun's XDR library.
 */
```


Security Vulnerability in the XDR Library

```
void* copy_elements(void *ele_src[], int ele_cnt, size_t ele_size) {
    /*
     * Allocate buffer for ele_cnt objects, each of ele_size bytes
     * and copy from locations designated by ele_src
     */
    void *result = malloc(ele_cnt * ele_size);
    if (result == NULL)
        /* malloc failed */
        return NULL;
```

Security Vulnerability in the XDR Library

```
    void *next = result;
    int i;
    for (i = 0; i < ele_cnt; i++) {
        /* Copy object i to destination */
        memcpy(next, ele_src[i], ele_size);
        /* Move pointer to next memory region */
        next += ele_size;
    }
    return result;
}
```

Consider ele_cnt = $2^{20} + 1$ and ele_size = 2^{12}

25