CS 370: Computer Organization

Integer Arithmetic Operations

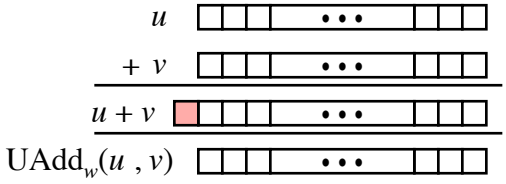
Instructor:
Professor Stephen P. Carl

Unsigned Addition

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry



Standard addition ignores carry output; this is Modular Arithmetic

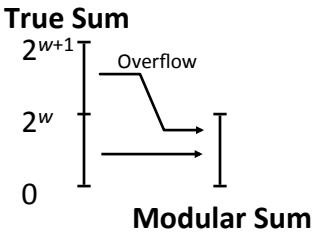
s = $\text{UAdd}_w(u, v)$ = $u + v \mod 2^w$

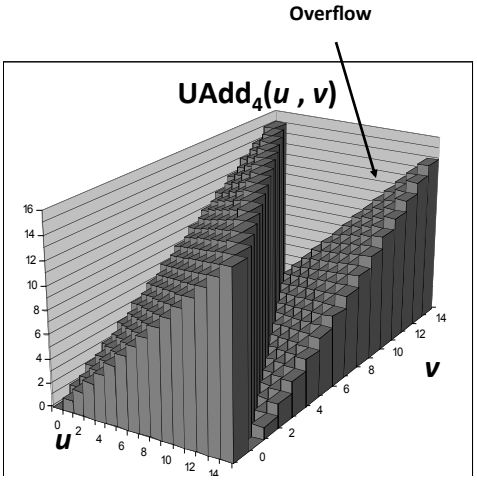
$$UAdd_w(u,v) = \begin{cases} u+v & u+v < 2^w \\ u+v-2^w & u+v \geq 2^w \end{cases}$$

Visualizing Unsigned Addition

Modulus wraps around at most once

- IF true sum ≥ 2^w



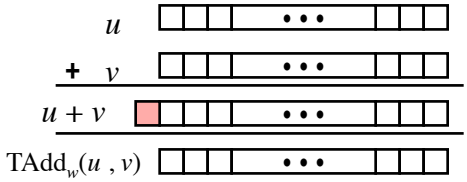


Two's Complement Addition

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:

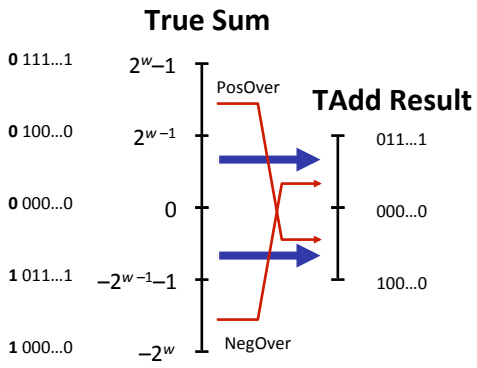
```
int s, t, u, v;
s = (int) ((unsigned) u + (unsigned) v);
t = u + v
```

Will give `s == t`

TAdd Overflow

Functionality

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



Multiplication

Computing Exact Product of w-bit numbers x, y

- Either signed or unsigned

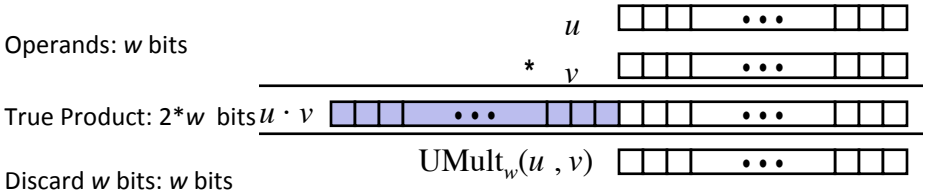
Ranges

- Unsigned: $0 \le x * y \le (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
  - Up to $2w$ bits
- Two's complement min: $x * y \ge (-2^{w-1})*(2^{w-1}-1) = -2^{2w-2} + 2^{w-1}$
  - Up to $2w-1$ bits
- Two's complement max: $x * y \le (-2^{w-1})^2 = 2^{2w-2}$
  - 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

Unsigned Multiplication in C



Standard Multiplication function ignores high order *w* bits

Again, implements Modular Arithmetic

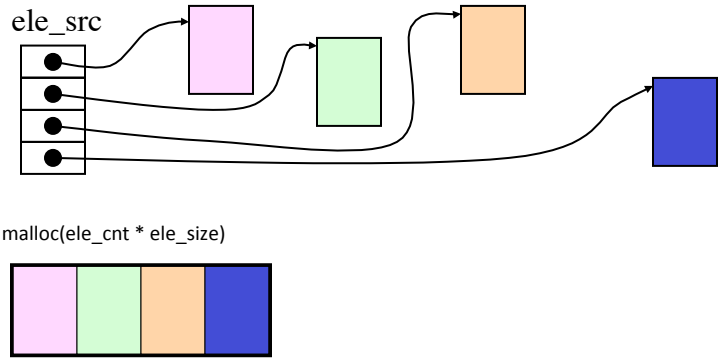
$\text{UMult}_w(u , v) = u \cdot v \mod 2^w$

Another Code Security Example

SUN XDR library

Widely used library for transferring data between machines

```
void* copy_elements(void *ele_src[], int ele_cnt, size_t ele_size);
```



XDR Code

```
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;
    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;
}
```


XDR Vulnerability

malloc(ele_cnt * ele_size)

What if:

- **ele_cnt** = $2^{20} + 1$
- **ele_size** = 4096 = 2^{12}
- Allocation = ??

Expected allocation size is $2^{32} + 2^{12}$ =

XDR Vulnerability

malloc(ele_cnt * ele_size)

Actual allocation:

- Allocation = **ele_cnt * ele_size mod 2^{32}**
- Allocation = **$(2^{20} + 1) * 2^{12}$ mod 2^{32}** **(due to 32-bit unsigned integer multiplication)**
- Allocation =_____

What is the effect of this error?

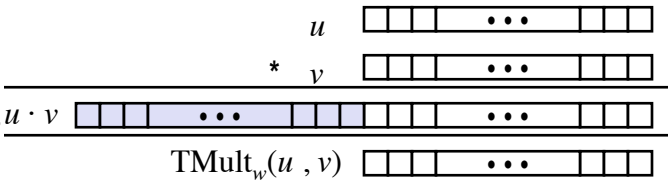
How can I make this function secure?

Signed Multiplication in C

Operands: *w* bits

True Product: 2**w* bits

Discard *w* bits: *w* bits



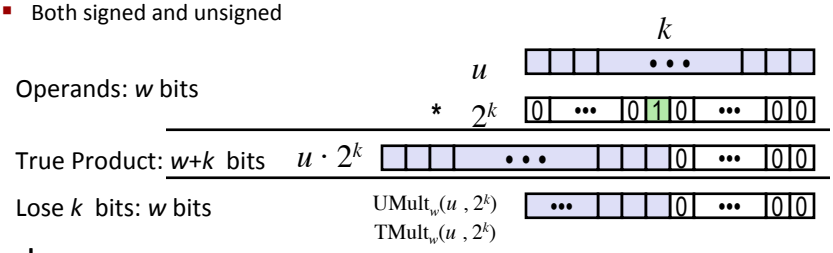
Standard Multiplication Function

- 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



Examples

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

Compiled Multiplication Code

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

In C Function

```
int mul12;
mul12 = x*12;
```

Compiled Arithmetic Operations

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

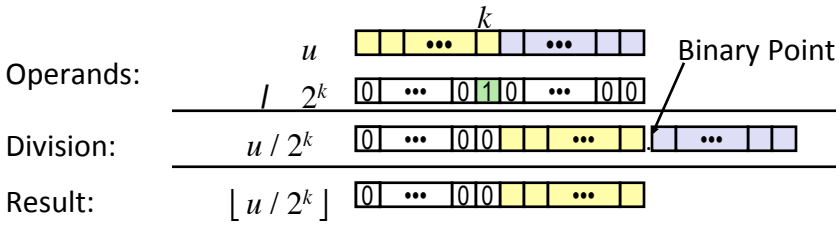
Explanation

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

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	00011101 10110110
x >> 4	950.8125	950	03 B6	00000011 10110110
x >> 8	59.4257813	59	00 3B	00000000 00111011

Compiled Unsigned Division Code

C Function

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

Compiled Arithmetic Operations

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

Explanation

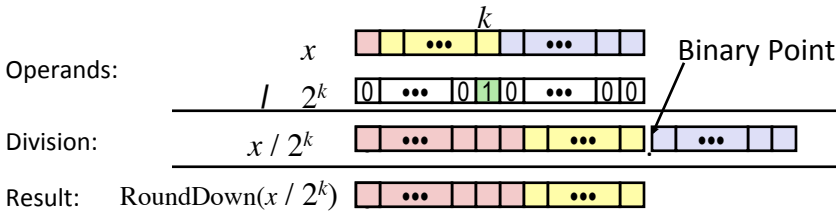
```
  # Logical shift
  return x >> 3;
```

- **Uses logical shift for unsigned**
- **In Java, logical shift written as >>>**

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 `x < 0`



	Division	Computed	Hex	Binary
y	-15213	-15213	C4 93	11000100 10010011
y >> 1	-7606.5	-7607	E2 49	11100010 01001001
y >> 4	-950.8125	-951	FC 49	11111100 01001001
y >> 8	-59.4257813	-60	FF C4	11111111 11000100

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 >>

Arithmetic: Basic Rules

Addition:

- Unsigned/signed: Normal addition followed by truncate, same operation on bit level
- Unsigned: addition mod 2^w
  - Mathematical addition + possible subtraction of 2w
- Signed: modified addition mod 2^w (result in proper range)
  - Mathematical addition + possible addition or subtraction of 2w

Multiplication:

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

Arithmetic: Basic Rules

Left shift

- Unsigned/signed: same as multiplication by 2^k
- Always logical shift

Right shift

- Unsigned: logical shift, div (division + round to zero) by 2^k
- Signed: arithmetic shift
  - Positive numbers: div (division + round to zero) by 2^k
  - Negative numbers: div (division + round away from zero) by 2^k, use biasing to fix

Why Use Unsigned?

***Not* just because you have non-negative numbers!**

- Easy to make mistakes

```
unsigned i;
for (i = cnt-2; i >= 0; i--) // does the loop terminate?
  a[i] += a[i+1];
```

- Mistakes can be very subtle

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


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?

Why Should I Use Unsigned? (cont.)

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