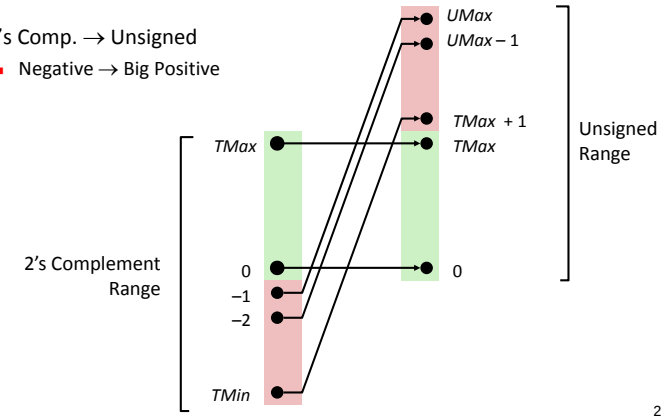
Integer Arithmetic

Kai Shen

1

Integer Conversion

- 2's Comp. → Unsigned
  - Negative → Big Positive



2

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;
```

  - Implicit casting also occurs via assignments and procedure calls

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

3

Casting Surprises

- Expression Evaluation
  - If there is a mix of unsigned and signed in single expression, ***signed values implicitly cast to unsigned***
  - Examples for *W* = 32: TMIN = -2,147,483,648 , TMAX = 2,147,483,647
- Constant$_1$ / Constant$_2$ / Relation / Evaluation

Constant$_1$	Constant$_2$	Relation	Evaluation
0	0U	==	unsigned
-1	0	<	signed
-1	0U	>	unsigned
2147483647	-2147483647	>	signed
2147483647U	-2147483647	<	unsigned
-1	-2	>	signed
(unsigned)-1	-2	>	unsigned
2147483647	2147483648U	<	unsigned
2147483647	(int) 2147483648U	>	signed

4

Security Vulnerability

```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

```
#define MSIZE 528

void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, MSIZE);
    printf("%s\n", mybuf);
}
```

- Real example: similar to code found in FreeBSD's implementation of getpeername

5

Malicious Usage

```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

```
#define MSIZE 528

void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, -MSIZE);
    . . .
}
```

- **`-MSIZE`**, when interpreted as **`unsigned int`** by **`memcpy`**, becomes a very large integer

6

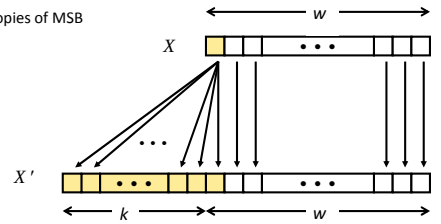
Casting Signed ↔ Unsigned: Basic Rules

- Bit pattern is maintained
- But reinterpreted
- Can have unexpected effects: adding or subtracting 2^w
- Expression containing signed and unsigned int
  - **`int`** is cast to **`unsigned`**!!

7

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:
  - $X' = \underbrace{x_{w-1}, \ldots, x_{w-1}}_{k \text{ copies of MSB}}, x_{w-1}, x_{w-2}, \ldots, x_0$



8

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

9

Integer Arithmetic

- Helpful for programmers to understand the semantics supported by current system (primarily hardware, may also involve compiler)
  - Main issue: limitation of the data type size
- We do not discuss hardware implementation (leave that the hardware engineers), but it doesn't hurt to know some implementation issues
  - Unsigned and signed (2's complement) integers are often computed in the same way, so their hardware implementation can share same components

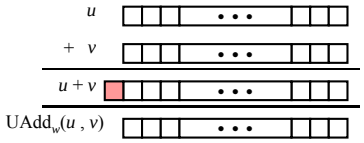
10

Unsigned Addition

Operands: *w* bits

True sum: *w*+1 bits

Discard carry: *w* bits



- Semantics: standard addition, but ignore overflowed carry
  - Still commutative and associative
- Implements modular arithmetic

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

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

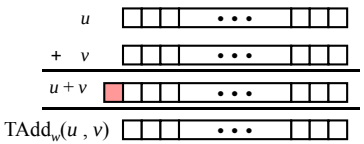
11

Two's Complement Addition

Operands: *w* bits

True sum: *w*+1 bits

Discard carry: *w* bits



- TAdd and UAdd have identical bit-level computation
  - Signed vs. unsigned addition in C:

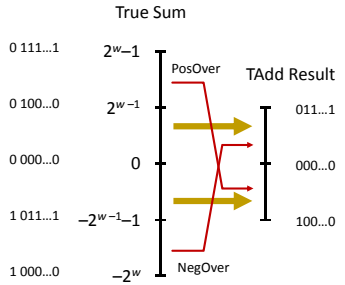
```
int s, t, u, v;
s = (int) ((unsigned) u + (unsigned) v);
t = u + v
```

  - Will give `s == t`

12

TAdd Overflow

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



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

13

Negation

- Semantics of negation (**`-x`**)
  - Only meaningful for signed integer
  - **`TMIN`** is smallest negative integer, what is **`-TMIN`**?
- Observation: **`~x + x == 1111…111 == -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

- So we have: **`~x + 1 == -x`**

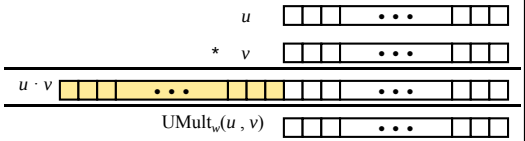
14

Unsigned Multiplication

Operands: *w* bits

True product: 2**w* bits

Discard *w* bits: *w* bits



- Semantics: standard multiplication, but ignore high order *w* bits
- Implements modular arithmetic

  $UMult_w(u, v) = u \cdot v \bmod 2^w$

15

Signed Multiplication

- Under 2's complement, same bit-level computation as in unsigned case

16

A Practical Case Example

```
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;
}
```

17

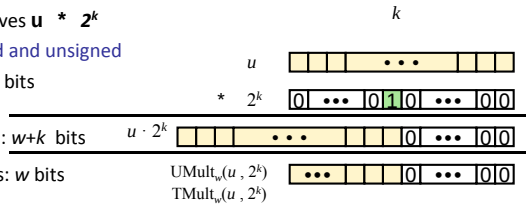
Power-of-2 Multiply with Shift

- Multiply is slow on most machines
- Operation
  - **u << k** gives **u * 2^k**
  - Both signed and unsigned

  Operands: *w* bits

  True product: *w+k* bits

  Discard *k* bits: *w* bits



- Examples
  - **u << 3 == u * 8**
  - **u << 5 - u << 3 == u * 24**

18

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

19

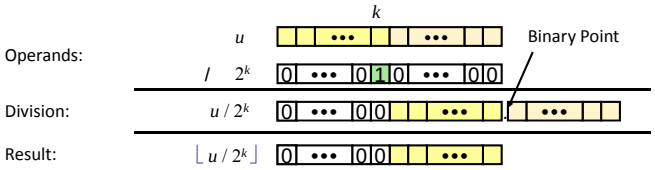
Division

- Integer division: divide one integer over another, output an integer
- Semantics:
  - Like standard division
  - No overflow problem (except divide by zero)
  - Round toward zero (round down on positive side, round up on negative side)
- Implementation for signed/unsigned division is very different
- Division is slower than multiply, so converting to shift etc. will help even more

20

Unsigned Power-of-2 Divide with Shift

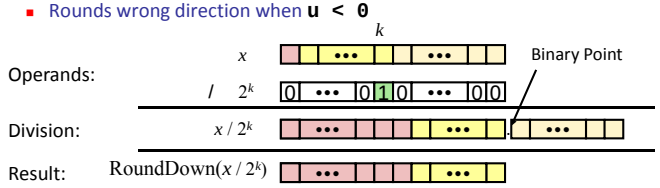
- Quotient of unsigned by power of 2
  - **u >> k** gives $\lfloor u / 2^k \rfloor$
  - Uses logical shift



21

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	11100010 01001001
y >> 4	-950.8125	-951	FC 49	11111100 01001001
y >> 8	-59.4257813	-60	FF C4	11111111 11000100

22

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**

23

Integer C Puzzles

Initialization

```
int x = foo();
int y = bar();
unsigned ux = x;
unsigned uy = y;
```

Is each of the following always true?

- ux >= 0
- ux > -1
- x > 0 && y > 0 ⇒ x + y > 0
- ux >> 3 == ux/8
- x >> 3 == x/8

24

Disclaimer

These slides were adapted from the CMU course slides provided along with the textbook of “Computer Systems: A programmer’s Perspective” by Bryant and O’Hallaron. The slides are intended for the sole purpose of teaching the computer organization course at the University of Rochester.

25