Integer Arithmetic

Kai Shen

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;

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

<table>
<thead>
<tr>
<th>Constant₁</th>
<th>Constant₂</th>
<th>Relation</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0U</td>
<td>==</td>
<td>unsigned</td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
<td>&lt;</td>
<td>signed</td>
</tr>
<tr>
<td>-1</td>
<td>0U</td>
<td>&gt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>-2147483647</td>
<td>&gt;</td>
<td>signed</td>
</tr>
<tr>
<td>2147483647U</td>
<td>-2147483647</td>
<td>&lt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>-1</td>
<td>-2</td>
<td>&gt;</td>
<td>signed</td>
</tr>
<tr>
<td>(unsigned)-1</td>
<td>-2</td>
<td>&gt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>2147483648U</td>
<td>&lt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>(int) 2147483648U</td>
<td>&gt;</td>
<td>signed</td>
</tr>
</tbody>
</table>
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;
}

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

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;
}

/* KSIZE, when interpreted as unsigned int by memcpy, becomes a very large integer */

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!!

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:
    $$X' = x_{w-1}, ..., x_{w-1}, x_{w-1}, x_{w-2}, ..., x_0$$
    - $k$ copies of MSB

- Example: $w=5$, $k=2$
  - $x = \ldots 10110$
  - $X' = \ldots 1110110$
  - $X'' = \ldots 0000011110110$

- Formula:
  - $X' = (x_{w-k}, ..., x_{w-k}, x_{w-k}, x_{w-k}, ..., x_0)$
  - $X'' = (0, 0, ..., 0, x_{w-k}, ..., x_{w-k}, x_{w-k}, x_{w-k}, ..., x_0)$
**Sign Extension Example**

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

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>-15213</td>
<td>3B 6D 00111011 01101101</td>
</tr>
<tr>
<td>ix</td>
<td>15213</td>
<td>00 00 00 00 00 00111011 01101101</td>
</tr>
<tr>
<td>y</td>
<td>-15213</td>
<td>CB 91 11000100 10010011</td>
</tr>
<tr>
<td>iy</td>
<td>15213</td>
<td>FF FF CB 91 11111111 11111111 11000100 10010011</td>
</tr>
</tbody>
</table>

- Short int x = 15213;
- int ix = (int) x;
- Short int y = -15213;
- Int iy = (int) y;

---

**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

---

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

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

---

**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:
    \[ \text{int} s, t, u, v; \]
    \[ s = (\text{int}) ((\text{unsigned}) u + (\text{unsigned}) v); \]
    \[ t = u + v \]
  - Will give \( s == t \)
### TAdd Overflow

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

<table>
<thead>
<tr>
<th>True Sum</th>
<th>TAdd Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0111...1</td>
<td>0111...1</td>
</tr>
<tr>
<td>0100...0</td>
<td>0100...0</td>
</tr>
<tr>
<td>0000...0</td>
<td>0000...0</td>
</tr>
<tr>
<td>1011...1</td>
<td>-2$^w$-1</td>
</tr>
<tr>
<td>1000...0</td>
<td>-2$^w$</td>
</tr>
</tbody>
</table>

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

### Negation

- **Semantics of negation ($\neg x$)**
  - Only meaningful for signed integer
  - TMIN is smallest negative integer, what is $-\text{TMIN}$?

<table>
<thead>
<tr>
<th>Observation: $-x + x = 1111...111 = -1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x \quad 10011101$</td>
</tr>
<tr>
<td>$+ \quad \neg x \quad 01100010$</td>
</tr>
<tr>
<td>$-1 \quad 11111111$</td>
</tr>
</tbody>
</table>

So we have: $\neg x + 1 = -x$

### Unsigned Multiplication

- **Operands:** $w$ bits
  - $u \quad \begin{array}{c} \cdot \end{array} \begin{array}{c} \cdot \end{array} \begin{array}{c} \cdot \end{array} \begin{array}{c} \cdot \end{array}$
  - $v \quad \begin{array}{c} \cdot \end{array} \begin{array}{c} \cdot \end{array} \begin{array}{c} \cdot \end{array} \begin{array}{c} \cdot \end{array}$

- **True product:** $2^w$ bits
  - $u \cdot v \quad \begin{array}{c} \cdot \end{array} \begin{array}{c} \cdot \end{array} \begin{array}{c} \cdot \end{array} \begin{array}{c} \cdot \end{array} \begin{array}{c} \cdot \end{array} \begin{array}{c} \cdot \end{array}$

- **Discard $w$ bits:** $w$ bits

- **Semantics:** standard multiplication, but ignore high order $w$ bits
- Implements modular arithmetic
  - $\text{UMult}_w(u, v) = u \cdot v \mod 2^w$

### Signed Multiplication

- **Under 2's complement, same bit-level computation as in unsigned case**
### A Practical Case Example

```c
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;
}
```

### Power-of-2 Multiply with Shift

- **Multiply is slow on most machines**
- **Operation**
  - \( u << k \) gives \( u \cdot 2^k \)
  - Both signed and unsigned
- **Operands**: \( w \) bits
- **True product**: \( w \cdot 2^k \) bits
- **Discard \( k \) bits**: \( w \) bits

- **Examples**
  - \( u << 3 \) == \( u \cdot 8 \)
  - \( u << 5 - u << 3 \) == \( u \cdot 24 \)

### Compiled Multiplication Code

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

### 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**
Unsigned Power-of-2 Divide with Shift

- Quotient of unsigned by power of 2
  - \( u \gg k \) gives \( \lfloor u / 2^k \rfloor \)
  - Uses logical shift

<table>
<thead>
<tr>
<th>Operands:</th>
<th>( u \gg k )</th>
<th>Binary Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>Division:</td>
<td>( u / 2^k )</td>
<td></td>
</tr>
<tr>
<td>Result:</td>
<td>( \lfloor u / 2^k \rfloor )</td>
<td></td>
</tr>
</tbody>
</table>

Signed Power-of-2 Divide with Shift

- Quotient of signed by power of 2
  - \( x \gg k \) gives \( \lfloor x / 2^k \rfloor \)
  - Uses arithmetic shift
  - Rounds wrong direction when \( u < 0 \)

<table>
<thead>
<tr>
<th>Operands:</th>
<th>( x \gg k )</th>
<th>Binary Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>Division:</td>
<td>( x / 2^k )</td>
<td></td>
</tr>
<tr>
<td>Result:</td>
<td>( \text{RoundDown}(x / 2^k) )</td>
<td></td>
</tr>
</tbody>
</table>

Correct Power-of-2 Divide

- Quotient of negative number by power of 2
  - Want \( \lceil x / 2^k \rceil \) (Round Toward 0)
  - Compute as \( \lceil (x+2^k-1)/2^k \rceil \)
  - In C: \( (x + (1<<k)-1) \gg k \)

Integer C Puzzles

Initialization

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

Is each of the following always true?

- \( ux >= 0 \)
- \( ux > -1 \)
- \( x > 0 \) && \( y > 0 \) \( \Rightarrow \) \( x + y > 0 \)
- \( ux >> 3 == ux/8 \)
- \( x >> 3 == x/8 \)
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.