Today: Integers

• Representation: unsigned and signed
• Conversion, casting
• Expanding, truncating
• Addition, negation, multiplication, shifting
• Summary

Encoding integers

Unsigned

\[ B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i \]

Two’s complement

\[ B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i \]

• A C `short` is 2 bytes long

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>15213</td>
<td>00111011 01101101</td>
</tr>
<tr>
<td>y</td>
<td>-15213</td>
<td>C4 93 11000100 10010011</td>
</tr>
</tbody>
</table>

• Sign bit
  - For 2’s complement, most significant bit indicates sign
  - 0 for nonnegative
  - 1 for negative

Encoding example (cont.)

\[ x = 15213: \quad 00111011 \quad 01101101 \]
\[ y = -15213: \quad 11000100 \quad 10010011 \]

<table>
<thead>
<tr>
<th>Weight</th>
<th>15213</th>
<th>-15213</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>32</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>64</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>128</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>256</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>512</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1024</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2048</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4096</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>8192</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>16384</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>-32768</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Sum</td>
<td>15213</td>
<td>-15213</td>
</tr>
</tbody>
</table>

Numeric ranges

• Unsigned values
  - \( UMin = 0 \)
    - 000...0
  - \( UMax = 2^w - 1 \)
    - 111...1

• Two’s complement values
  - \( TMin = -2^{w-1} \)
    - 100...0
  - \( TMax = 2^{w-1} - 1 \)
    - 011...1

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>UMax</td>
<td>65535</td>
<td>FF FF</td>
</tr>
<tr>
<td>TMax</td>
<td>32767</td>
<td>7F FF</td>
</tr>
<tr>
<td>TMin</td>
<td>-32768</td>
<td>80 00</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>FF FF</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 00</td>
</tr>
<tr>
<td>Values for ( w=16 )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Values for different word sizes

<table>
<thead>
<tr>
<th></th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>UMax</td>
<td>255</td>
<td>65,535</td>
<td>4,294,967,295</td>
<td>18,446,744,073,709,551,615</td>
</tr>
<tr>
<td>Tmax</td>
<td>127</td>
<td>32,767</td>
<td>2,147,483,647</td>
<td>9,223,372,036,854,775,807</td>
</tr>
<tr>
<td>Tmin</td>
<td>-128</td>
<td>-32,768</td>
<td>-2,147,483,648</td>
<td>-9,223,372,036,854,775,808</td>
</tr>
</tbody>
</table>

- | TMin | = | Tmax + 1 |
  - | Asymmetric range |
  - | UMax = 2 * Tmax + 1 |

- Observations
  - | TMin | = | Tmax + 1 |
    - | Asymmetric range |
    - | UMax = 2 * Tmax + 1 |

- C Programming
  - | #include <limits.h> |
    - | Declares constants, e.g., |
      - | ULONG_MAX |
      - | LONG_MAX |
      - | LONG_MIN |
  - Values platform specific

---

Unsigned & signed numeric values

<table>
<thead>
<tr>
<th>X</th>
<th>B2(U)(X)</th>
<th>B2(T)(X)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
<td>-8</td>
</tr>
<tr>
<td>1001</td>
<td>9</td>
<td>-7</td>
</tr>
<tr>
<td>1010</td>
<td>10</td>
<td>-6</td>
</tr>
<tr>
<td>1011</td>
<td>11</td>
<td>-5</td>
</tr>
<tr>
<td>1100</td>
<td>12</td>
<td>-4</td>
</tr>
<tr>
<td>1101</td>
<td>13</td>
<td>-3</td>
</tr>
<tr>
<td>1110</td>
<td>14</td>
<td>-2</td>
</tr>
<tr>
<td>1111</td>
<td>15</td>
<td>-1</td>
</tr>
</tbody>
</table>

- Equivalence
  - | Same encodings for nonnegative values |
- Uniqueness
  - | Every bit pattern represents unique integer value |
  - | Each representable integer has unique bit encoding |
  - | Can invert mappings |
    - | U2B(x) = B2U⁻¹(x) |
    - | Bit pattern for unsigned integer |
    - | T2B(x) = B2T⁻¹(x) |
    - | Bit pattern for two’s comp integer |

---

Today: Integers

- Representation: unsigned and signed
- Conversion, casting
- Expanding, truncating
- Addition, negation, multiplication, shifting
- Summary

---

Mapping between signed & unsigned

Two's complement

<table>
<thead>
<tr>
<th>x</th>
<th>T2U</th>
<th>B2U</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>T2B</td>
<td>B2T</td>
</tr>
</tbody>
</table>

- Maintain same bit pattern

Unsigned

<table>
<thead>
<tr>
<th>x</th>
<th>U2B</th>
<th>B2T</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>U2T</td>
<td>B2T</td>
</tr>
</tbody>
</table>

- Maintain same bit pattern

- Mappings between unsigned and two's complement numbers: keep bit representations and reinterpret

---

Mapping signed ↔ unsigned

<table>
<thead>
<tr>
<th>Bits</th>
<th>Signed</th>
<th>Unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>-8</td>
<td>8</td>
</tr>
<tr>
<td>1001</td>
<td>-7</td>
<td>9</td>
</tr>
<tr>
<td>1010</td>
<td>-6</td>
<td>10</td>
</tr>
<tr>
<td>1011</td>
<td>-5</td>
<td>11</td>
</tr>
<tr>
<td>1100</td>
<td>-4</td>
<td>12</td>
</tr>
<tr>
<td>1101</td>
<td>-3</td>
<td>13</td>
</tr>
<tr>
<td>1110</td>
<td>-2</td>
<td>14</td>
</tr>
<tr>
<td>1111</td>
<td>-1</td>
<td>15</td>
</tr>
</tbody>
</table>

Mapping signed ↔ unsigned

<table>
<thead>
<tr>
<th>Bits</th>
<th>Signed</th>
<th>Unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>-8</td>
<td>8</td>
</tr>
<tr>
<td>1001</td>
<td>-7</td>
<td>9</td>
</tr>
<tr>
<td>1010</td>
<td>-6</td>
<td>10</td>
</tr>
<tr>
<td>1011</td>
<td>-5</td>
<td>11</td>
</tr>
<tr>
<td>1100</td>
<td>-4</td>
<td>12</td>
</tr>
<tr>
<td>1101</td>
<td>-3</td>
<td>13</td>
</tr>
<tr>
<td>1110</td>
<td>-2</td>
<td>14</td>
</tr>
<tr>
<td>1111</td>
<td>-1</td>
<td>15</td>
</tr>
</tbody>
</table>

+16
Relation between signed & unsigned

Two's complement

T2U

T2B

B2U

Maintain same bit pattern

Signed vs. unsigned in C

- Constants
  - By default are considered to be signed integers
  - Unsigned if have "U" as suffix:

```
0U
4294967299U
```

- Casting
  - Explicit casting between signed & unsigned same as U2T and T2U:

```
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;
```

Conversion visualized

- 2's Comp. → Unsigned
  - Ordering inversion
  - Negative → big positive

Signed vs. unsigned in C

- Casting surprises

  - Expression evaluation
    - If mix unsigned and signed in single expression, signed values implicitly cast to unsigned
    - Including comparison operations <, >, ==, <=, >=

```
Constant 1         Constant 2         Relation         Evaluation

0                0U                    ==                Unsigned
-1               0U                    <                  Signed
-1                2147483647U        >                  Signed
2147483647U      -2147483647-1      >                  Signed
-1                -2                   >                  Signed
(unsigned)-1      -2                   >                  Signed
2147483647U      2147483648U        <                  Signed
2147483647U      (int) 2147483648U   >                  signed
```

Code security example

- Similar to code found in FreeBSD's implementation of `getpeername`
- There are legions of smart people trying to find vulnerabilities in programs

```
/* 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;
}
```

Typical 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);
    printf("%s\n", mybuf);
}
```
Malicious usage

```c
/* 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;
}
```

```c
#define MSIZE 528
void getstuff() { 
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, -MSIZE);
    ... 
}
```

Summary: casting signed ↔ unsigned
- 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!!

Today: Integers
- Representation: unsigned and signed
- Conversion, casting
- Expanding, truncating
- Addition, negation, multiplication, shifting
- Summary

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

Sign extension example

```c
short int x = 15213;
int ix = (int) x;
short int y = -15213;
int iy = (int) y;
```

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>15213</td>
<td>00111101 01101101</td>
</tr>
<tr>
<td>ix</td>
<td>15213</td>
<td>00000000 00000000 00000000 00000000 00111010 01101010</td>
</tr>
<tr>
<td>y</td>
<td>-15213</td>
<td>11000100 10010011</td>
</tr>
<tr>
<td>iy</td>
<td>-15213</td>
<td>FF FF 00111010 01101010</td>
</tr>
</tbody>
</table>

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

Summary:
Expanding, truncating: basic rules
- Expanding (e.g., short int to int)
  - Unsigned: zeros added
  - Signed: sign extension
  - Both yield expected result
- Truncating (e.g., unsigned to unsigned short)
  - Unsigned/signed: bits are truncated
  - Result reinterpreted
  - Signed: mod operation
- Signed: similar to mod
  - For small numbers yields expected behaviour
Today: Integers

- Representation: unsigned and signed
- Conversion, casting
- Expanding, truncating
- Addition, negation, multiplication, shifting
- Summary

Negation: complement & increment

- Claim: the following holds for 2’s complement:
  \[ \neg x + 1 = -x \]
- Complement
  - Observation:
    \[ \neg x + x = 111\ldots11 == -1 \]

Complement & increment examples

\[ x = 15213 \]

<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</td>
</tr>
<tr>
<td>\neg x</td>
<td>-15214</td>
<td>C4 92</td>
</tr>
<tr>
<td>\neg x + 1</td>
<td>-15213</td>
<td>C4 93</td>
</tr>
<tr>
<td>y</td>
<td>-15213</td>
<td>C4 93</td>
</tr>
</tbody>
</table>

\[ x = 0 \]

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>00 00</td>
</tr>
<tr>
<td>\neg 0</td>
<td>-1</td>
<td>FF FF</td>
</tr>
<tr>
<td>\neg 0 + 1</td>
<td>0</td>
<td>00 00</td>
</tr>
</tbody>
</table>

Unsigned addition

- Standard addition function
  - Ignores carry output
- Implements modular arithmetic
  \[ s = \text{UAdd}_w(u, v) = u + v \mod 2^w \]
  \[ \text{UAdd}_w(u, v) = \begin{cases} u + v, & u + v < 2^w \ 
  u + v - 2^w, & u + v \geq 2^w \end{cases} \]

Visualizing (mathematical) integer addition

- Integer addition
  - 4-bit integers \( u, v \)
  - Compute true sum \( \text{Add}_4(u, v) \)
  - Values increase linearly with \( u \) and \( v \)
  - Forms planar surface

Visualizing unsigned addition

- Wraps around
  - If true sum \( \geq 2^w \)
  - At most once
Mathematical properties

- Modular addition forms an Abelian group
  - **Closed** under addition
    \[ 0 \leq UAdd_w(u, v) \leq 2^w - 1 \]
  - **Commutative**
    \[ UAdd_w(u, v) = UAdd_w(v, u) \]
  - **Associative**
    \[ UAdd_w(t, UAdd_w(u, v)) = UAdd_w(UAdd_w(t, u), v) \]
  - 0 is additive **identity**
    \[ UAdd_w(u, 0) = u \]
  - Every element has additive **inverse**
    Let: \[ UComp_w(u) = 2^w - u \]
    Then: \[ UAdd_w(u, UComp_w(u)) = 0 \]

Two’s complement addition

- **Operands**: \( w \) bits
- **True sum**: \( w+1 \) bits
- **Discard carry**: \( w \) bits

\[ u + v \]
\[ TAdd_w(u, v) \]

- **TAdd and UAdd have identical bit-level behavior**
  - Signed vs. unsigned addition in C:
    ```c
    int s, t, u, v;
    s = (int) ((unsigned) u + (unsigned) v);
    t = u + v;
    s == t
    ```

TAdd overflow

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

Visualizing 2’s compl. 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

Characterizing TAdd

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

Mathematical properties of TAdd

- **Group isomorphic to unsigneds with UAdd**
  - Since both have identical bit patterns
    \[ TAdd_w(u, v) = U2T(UAdd_w(T2U(u), T2U(v))) \]
- **2’s complement under TAdd forms a group**
  - Closed, commutative, associative,
  - 0 is additive identity
  - Every element has additive inverse
    \[ TComp_w(u) = \begin{cases} -u & u < TMin_w \\ u & u = TMin_w \end{cases} \]
Multiplication

• Computing exact product of w-bit numbers x, y
  – Either signed or unsigned

• Ranges
  – Unsigned (up to $2^w$ bits):
    $0 \leq x \cdot y \leq (2^w - 1)2 = 2^{2w} - 2^{w+1} + 1$
  – Two’s complement min (up to $2^{w-1}$ bits):
    $x \cdot y \geq (-2^{w-1}) \cdot (2^{w-1} - 1) = -2^{2w-2} + 2^{w-1}$
  – Two’s complement max (up to $2^w$ bits, but only for $(\text{Min}_w)^2$):
    $x \cdot y \leq (-2^{w-1})^2 = 2^{2w-2}$

• 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

Operands: w bits

True product: $2^w \cdot u \cdot v$

Discard w bits: w bits

• Standard multiplication function
  – Ignores high order w bits
  – Implements modular arithmetic

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

Code security example #2

• SUN XDR library
  – Widely used library for transferring data between machines

```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 */
    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 */
        /* Move pointer to next memory region */
        next += ele_size;
        return result;
    }
}
```

XDR code

```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 */
    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 */
        /* Move pointer to next memory region */
        next += ele_size;
    }
    return result;
}
```

XDR vulnerability

```c
malloc(ele_cnt * ele_size)
```

• What if:
  – ele_cnt = $2^{20} + 1$
  – ele_size = 4096, $2^{12}$
  – Allocation = ??

• How can I make this function secure?

Signed multiplication in C

Operands: w bits

True product: $2^w \cdot u \cdot v$

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
  \[ u \ll k \] gives \[ u \times 2^k \]
  - Both signed and unsigned

<table>
<thead>
<tr>
<th>Operands: ( w ) bits</th>
<th>True product: ( w \times 2^k ) with ( k ) bits</th>
<th>Discard ( k ) bits: ( w ) bits</th>
</tr>
</thead>
</table>
| \[ u \] | \[ \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ ld
Correct power-of-2 divide (Cont.)

- Case 2: Rounding:

<table>
<thead>
<tr>
<th>Dividend:</th>
<th>Divisor:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\frac{u}{2^k}$</td>
<td>$/2^k$</td>
</tr>
</tbody>
</table>

Biasing adds 1 to final result

Compiled signed division code

C function

```c
int idiv8(int x)
{
    return x/8;
}
```

Compiled arithmetic operations

```c
testl %eax, %eax
js L4
L3:
sarl $3, %eax
ret
L4:
addl $7, %eax
jmp L3
```

- 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 $2^w$
  - Signed: modified addition mod $2^w$ (result in proper range)
    - Mathematical addition + possible addition or subtraction of $2^w$

- 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

- Unsigned ints, 2’s complement ints are isomorphic rings:
  $\Rightarrow$ isomorphism = casting

- Left shift
  - Unsigned/signed: 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

Today: Integers

- Representation: unsigned and signed
- Conversion, casting
- Expanding, truncating
- Addition, negation, multiplication, shifting
- Summary

Unsigned arithmetic

Unsigned multiplication with addition forms a commutative ring:

- Addition is a commutative group
- Closed under multiplication
  $0 \leq U\text{Mult}_w(u,v) \leq 2^w - 1$
- Multiplication is commutative
  $U\text{Mult}_w(u,v) = U\text{Mult}_w(v,u)$
- Multiplication is associative
  $U\text{Mult}_w(t, U\text{Mult}_w(u,v)) = U\text{Mult}_w(U\text{Mult}_w(t,u),v)$
- 1 is multiplicative identity
  $U\text{Mult}_w(u,1) = u$
- Multiplication distributes over addition
  $U\text{Mult}_w(t, U\text{Add}_w(u,v)) = U\text{Add}_w(U\text{Mult}_w(t,u),U\text{Mult}_w(t,v))$
Two’s complement arithmetic

- Isomorphic algebras
  - Unsigned multiplication and addition
    - Truncating to w bits
  - Two’s complement multiplication and addition
    - Truncating to w bits
- Both form rings
  - Isomorphic to ring of integers mod \(2^w\)
- Comparison to (mathematical) integer arithmetic
  - Both are rings
  - True integers obey ordering properties, e.g.,
    \[ u > 0 \Rightarrow u + v > v \]
    \[ u > 0, v > 0 \Rightarrow u \cdot v > 0 \]
  - These properties are not obeyed by two’s complement arithmetic
    \[ T_{\text{Max}} + 1 = T_{\text{Min}} \]
    \[ 15213 \times 30426 = -10030 \] (e.g. 16-bit words)

Why should I use unsigned?

- Don’t use just because the number is non-negative
  - Easy to make mistakes
    \[ \text{unsigned } i; \]
    \[ \text{for } (i = \text{cnt-2}; i >= 0; i--) \]
    \[ a[i] += a[i+1]; \]
  - Can be very subtle
    - \#define DELTA sizeof(int)
    - int i;
    - for (i = CNT; i-DELTA >= 0; i-= DELTA)
  - Do use when performing modular arithmetic
    - Multiprecision arithmetic
  - Do use when using bits to represent sets
    - Logical right shift, no sign extension

Next time: Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- Rounding, addition, multiplication
- Floating point in C