### Numeric Ranges

- **Unsigned Values**
  - $U_{\text{Min}} = 0$
  - $000...0$
  - $U_{\text{Max}} = 2^w - 1$
  - $111...1$

- **Two’s Complement Values**
  - $T_{\text{Min}} = -2^{w-1}$
  - $100...0$
  - $T_{\text{Max}} = 2^{w-1} - 1$
  - $011...1$

- **Other Values**
  - Minus 1
  - $111...1$

#### Values for $W = 16$

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>UMax</td>
<td>65535</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>Tmax</td>
<td>32767</td>
<td>7F FF</td>
<td>01111111 11111111</td>
</tr>
<tr>
<td>Tmin</td>
<td>-32768</td>
<td>80 00</td>
<td>10000000 00000000</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 00</td>
<td>00000000 00000000</td>
</tr>
</tbody>
</table>
Values for Different Word Sizes

<table>
<thead>
<tr>
<th></th>
<th>W</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8</td>
<td>16</td>
<td>32</td>
<td>64</td>
<td></td>
</tr>
<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>
<td></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>
<td></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>
<td></td>
</tr>
</tbody>
</table>

- Observations
  
  \[|TMin| = Tmax + 1\]
  
  » Asymmetric range
  
  \[UMax = 2 \times Tmax + 1\]

- C Programming
  
  - `#include <limits.h>`
  
  - declares constants, e.g.,
    
    - ULONG_MAX
    - LONG_MAX
    - LONG_MIN
  
  - values platform-specific
Quiz 1

• What is $-\text{TMin}$ (assuming two’s complement signed integers)?
  a) TMin
  b) TMax
  c) 0
  d) 1
4-Bit Computer Arithmetic
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; // “unsigned” means “unsigned int”
    tx = (int) ux;
    uy = (unsigned int) 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*
  - including comparison operations `<`, `>`, `==`, `<=`, `>=`
  - examples for \( W = 32 \): \( T_{\text{MIN}} = -2,147,483,648 \), \( T_{\text{MAX}} = 2,147,483,647 \)

<table>
<thead>
<tr>
<th>Constant(_1)</th>
<th>Constant(_2)</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>-2147483647-1</td>
<td>&gt;</td>
<td>signed</td>
</tr>
<tr>
<td>2147483647U</td>
<td>-2147483647-1</td>
<td>&lt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>-1</td>
<td>-2</td>
<td>&gt;</td>
<td>unsigned</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>
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:
  – \( X' = x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_0 \)
### Sign Extension Example

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

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>15213</td>
<td>3B 6D</td>
<td>00111011 01101101</td>
</tr>
<tr>
<td>ix</td>
<td>15213</td>
<td>00 00 3B 6D</td>
<td>00000000 00000000 00111011 01101101</td>
</tr>
<tr>
<td>y</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
<tr>
<td>iy</td>
<td>-15213</td>
<td>FF FF C4 93</td>
<td>11111111 11111111 11000100 10010011</td>
</tr>
</tbody>
</table>

- Converting from smaller to larger integer data type
  - C automatically performs sign extension
Does it Work?

\[ \text{val}_w = -2^{w-1} + \sum_{i=0}^{w-2} b_i \cdot 2^i \]

\[ \text{val}_{w+1} = -2^w + 2^{w-1} + \sum_{i=0}^{w-2} b_i \cdot 2^i \]

\[ = -2^{w-1} + \sum_{i=0}^{w-2} b_i \cdot 2^i \]

\[ \text{val}_{w+2} = -2^{w+1} + 2^w + 2^{w-1} + \sum_{i=0}^{w-2} b_i \cdot 2^i \]

\[ = -2^w + 2^{w-1} + \sum_{i=0}^{w-2} b_i \cdot 2^i \]

\[ = -2^{w-1} + \sum_{i=0}^{w-2} b_i \cdot 2^i \]
Power-of-2 Multiply with Shift

• Operation
  – \( u \ll k \) gives \( u \times 2^k \)
  – both signed and unsigned

\[
\begin{array}{c}
\text{operands: } w \text{ bits} \\
\hline
u \\
\times 2^k \\
\hline
\text{true product: } w+k \text{ bits} \\
appears as \ u \times 2^k \\
\text{discard } k \text{ bits: } w \text{ bits}
\end{array}
\]

\[
\begin{array}{c}
\text{UMult}_w(u, 2^k) \\
\text{TMult}_w(u, 2^k)
\end{array}
\]

• Examples
  
  \[
  u \ll 3 == u \times 8
  \]
  
  \[
  u \ll 5 - u \ll 3 == u \times 24
  \]

– most machines shift and add faster than multiply
  » compiler generates this code automatically
Unsigned Power-of-2 Divide with Shift

- Quotient of unsigned by power of 2
  - $u \gg k$ gives $\lceil u / 2^k \rceil$
  - uses logical shift

<table>
<thead>
<tr>
<th>Division</th>
<th>Computed</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>15213</td>
<td>15213</td>
<td>3B 6D 00111011 01101101</td>
</tr>
<tr>
<td>$x \gg 1$</td>
<td>7606.5</td>
<td>7606</td>
<td>1D B6 00011101 10110110</td>
</tr>
<tr>
<td>$x \gg 4$</td>
<td>950.8125</td>
<td>950</td>
<td>03 B6 00000011 10110110</td>
</tr>
<tr>
<td>$x \gg 8$</td>
<td>59.4257813</td>
<td>59</td>
<td>00 3B 00000000 00111011</td>
</tr>
</tbody>
</table>
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 \( x < 0 \)

\[
\begin{array}{c|cccc|c}
\hline
\text{Division} & \text{Computed} & \text{Hex} & \text{Binary} \\
\hline
y & -15213 & -15213 & C4 93 & 11000100 10010011 \\
y >> 1 & -7606.5 & -7607 & E2 49 & 11100010 01001001 \\
y >> 4 & -950.8125 & -951 & FC 49 & 11111100 01001001 \\
y >> 8 & -59.4257813 & -60 & FF C4 & 11111111 11000100 \\
\hline
\end{array}
\]
Correct Power-of-2 Divide

- Quotient of negative number by power of 2
  - want $\left\lfloor \frac{x}{2^k} \right\rfloor$ (round toward 0)
  - compute as $\left\lfloor \frac{(x+2^k-1)}{2^k} \right\rfloor$
    - in C: `(x + (1<<k)-1) >> k`
    - biases dividend toward 0

**Case 1: no rounding**

<table>
<thead>
<tr>
<th>Dividend:</th>
<th>$u$</th>
<th>$k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$+2^k-1$</td>
<td>0 1 1 1 1 0 0</td>
<td></td>
</tr>
<tr>
<td>divisor:</td>
<td>0 0 1 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>$u/2^k$</td>
<td>1 1 1 1 1 1 1</td>
<td></td>
</tr>
<tr>
<td>$\left\lfloor u/2^k \right\rfloor$</td>
<td>1 1 1 1 1 1 1</td>
<td></td>
</tr>
</tbody>
</table>

**Biasing has no effect**

*Binary point*
Correct Power-of-2 Divide (Cont.)

Case 2: rounding

dividend: 
\[ x = \underbrace{1\ldots k}_{k} \]
\[ +2^k - 1 = \overbrace{0\ldots 011}^{k} \]

\[ \overbrace{1\ldots k}^{k} \]

incremented by 1

divisor: 
\[ \left\lfloor \frac{x}{2^k} \right\rfloor = \underbrace{1\ldots 111}_{k} \]
\[ / 2^k = \overbrace{0\ldots 010}^{k} \]

\[ \underbrace{1\ldots}_{k} \]

incremented by 1

Biasing adds 1 to final result
Why Should I Use Unsigned?

• *Don’t* use just because number nonnegative
  
  – easy to make mistakes
    
    ```c
    unsigned i;
    for (i = cnt-2; i >= 0; i--)
      a[i] += a[i+1];
    ```
  
  – can be very subtle
    
    ```c
    #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
Byte-Oriented Memory Organization

• Programs refer to data by address
  – conceptually, envision it as a very large array of bytes
    » in reality, it’s not, but can think of it that way
  – an address is like an index into that array
    » and, a pointer variable stores an address

• Note: system provides private address spaces to each “process”
  – think of a process as a program being executed
  – so, a program can clobber its own data, but not that of others
Machine Words

- Any given computer has a “word size”
  - nominal size of integer-valued data
    » and of addresses
  - until recently, most machines used 32 bits (4 bytes) as word size
    » limits addresses to 4GB ($2^{32}$ bytes)
    » becomes too small for memory-intensive applications
      • leading to emergence of computers with 64-bit word size
  - machines still support multiple data formats
    » fractions or multiples of word size
    » always integral number of bytes
**Word-Oriented Memory Organization**

- Addresses specify byte locations
  - address of first byte in word
  - addresses of successive words differ by 4 (32-bit) or 8 (64-bit)
Byte Ordering

- **Four-byte integer**
  - 0x76543210

- **Stored at location 0x100**
  - which byte is at 0x100?
  - which byte is at 0x103?

<table>
<thead>
<tr>
<th>10</th>
<th>32</th>
<th>54</th>
<th>76</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>0x101</td>
<td>0x102</td>
<td>0x103</td>
</tr>
</tbody>
</table>

- **Little-endian**

<table>
<thead>
<tr>
<th>76</th>
<th>54</th>
<th>32</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>0x101</td>
<td>0x102</td>
<td>0x103</td>
</tr>
</tbody>
</table>

- **Big-endian**
Byte Ordering (2)

Big Endian

00 00 00 01

Little Endian
Quiz 2

```c
int main() {
    long x=1;
    proc(x);
    return 0;
}

void proc(int arg) {
    printf("%d\n", arg);
}
```

What value is printed on a big-endian 64-bit computer?

a) 0  
b) 1  
c) $2^{32}$  
d) $2^{32}-1$