**Binary Representations**

**Base 2 Number Representation**
- Represent $15213_{10}$ as $11101101101101_2$
- Represent $1.20_{10}$ as $1.00110011001100110011_2$
- Represent $1.5213 \times 10^4$ as $1.1101101101101 \times 2^{13}$

**Electronic Implementation**
- Easy to store with bistable elements
- Reliably transmitted on noisy and inaccurate wires

---

**Encoding Byte Values**

**Byte = 8 bits**
- Binary: $00000000_2$ to $11111111_2$
- Decimal: $0_{10}$ to $255_{10}$
  - First digit must not be $0$ in C
- Hexadecimal: $00_{16}$ to $FF_{16}$
  - Base 16 number representation
  - Use characters '0' to '9' and 'A' to 'F'
  - Write $FA1D37B_{16}$ in C as $0xFA1D37B$
    » Or $0xfa1d37b$

---

**Byte-Oriented Memory Organization**

**Programs Refer to Virtual Addresses**
- Conceptually very large array of bytes
- Actually implemented with hierarchy of different memory types
- System provides address space private to particular "process"
  - Program being executed
  - Program can clobber its own data, but not that of others

**Compiler + Run-Time System Control Allocation**
- Where different program objects should be stored
- All allocation within single virtual address space
Machine Words

Machine Has "Word Size"
- Nominal size of integer-valued data
  - Including addresses
- Most current machines use 32 bits (4 bytes) words
  - Limits addresses to 4GB
    » Users can access 3GB
- Becoming too small for memory-intensive applications
- High-end systems use 64 bits (8 bytes) words
  - Potential address space ≈ 1.8 X 10^19 bytes
  - x86-64 machines support 48-bit addresses: 256 Terabytes
- Machines 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)

Data Representations

Sizes of C Objects (in Bytes)
- C Data Type
  - Typical 32-bit
  - Intel IA32
  - x86-64
- unsigned 4 4 4
- int 4 4 4
- long int 4 4 4
- char 1 1 1
- short 2 2 2
- float 4 4 4
- double 8 8 8
- long double 10/12 10/12
- char * 4 4 8
  » Or any other pointer

Byte Ordering

How should bytes within multi-byte word be ordered in memory?

Conventions
- Big Endian: Sun, PPC Mac
  - Least significant byte has highest address
- Little Endian: x86
  - Least significant byte has lowest address
Byte Ordering Example

Big Endian
- Least significant byte has highest address

Little Endian
- Least significant byte has lowest address

Example
- Variable \( x \) has 4-byte representation \( 0x01234567 \)
- Address given by \&\( x \) is \( 0x100 \)

<table>
<thead>
<tr>
<th></th>
<th>0x100</th>
<th>0x101</th>
<th>0x102</th>
<th>0x103</th>
</tr>
</thead>
<tbody>
<tr>
<td>Big Endian</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Little Endian</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Reading Byte-Reversed Listings

Disassembly
- Text representation of binary machine code
- Generated by program that reads the machine code

Example Fragment

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction Code</th>
<th>Assembly Rendition</th>
</tr>
</thead>
<tbody>
<tr>
<td>8048365:</td>
<td>5b</td>
<td>pop %ebx</td>
</tr>
<tr>
<td>8048366:</td>
<td>81 c3 ab 12 00 00</td>
<td>add $0x12ab,%ebx</td>
</tr>
<tr>
<td>804836c:</td>
<td>83 bb 28 00 00 00 00</td>
<td>cmpl $0x0,0x28(%ebx)</td>
</tr>
</tbody>
</table>

Deciphering Numbers
- Value: 0x12ab
- Pad to 4 bytes: 0x000012ab
- Split into bytes: 00 00 12 ab
- Reverse: ab 12 00 00

Examining Data Representations

Code to Print Byte Representation of Data
- Casting pointer to unsigned char * creates byte array

```c
typedef unsigned char *pointer;

void show_bytes(pointer start, int len)
{
    int i;
    for (i = 0; i < len; i++)
        printf("0x%p	0x%.2x\n", start+i, start[i]);
    printf("\n");
}
```

Printf directives:
- %p: Print pointer
- %x: Print Hexadecimal

show_bytes Execution Example

```c
int a = 15213;
printf("int a = 15213;\n");
show_bytes((pointer) &a, sizeof(int));
```

Result (Linux):
```
int a = 15213;
0x11ffffcb8 0x6d
0x11ffffcb9 0x3b
0x11ffffffca 0x00
0x11fffffffcb 0x00
```
### Representing Integers

- **Decimal:** 15213
- **Binary:** 0011 1011 0110 1101
- **Hex:** 3B 6D

- IA32, x86-64 A
  - Sun A
    - IA32: 6D 3B 00
    - x86-64: 00
  - Two’s complement representation (Covered later)

- IA32, x86-64 B
  - Sun B
    - IA32: C4 FF 00 00
    - x86-64: 93 00

### Representing Pointers

- **Decimal:** -15213
- **Binary:** 1110 1001 0101 1111
- **Hex:** EF D4 0C

- IA32, x86-64 P
  - Sun P
    - IA32: FF FB F8 00
    - x86-64: 2C BF 7F 00

**Different compilers & machines assign different locations to objects**

### Representing Strings

- **String in C:**
  - `char S[6] = "15213";`
  - Represented by array of characters
  - Each character encoded in ASCII format
    - Standard 7-bit encoding of character set
    - Character "0" has code 0x30
      - Digit i has code 0x30+i
  - String should be null-terminated
    - Final character = 0

**Compatibility**
- Byte ordering not an issue

### Boolean Algebra

- Developed by George Boole in 19th Century
- Algebraic representation of logic
  - Encode “True” as 1 and “False” as 0

**And**
- \( A \& B = 1 \) when both \( A = 1 \) and \( B = 1 \)

<table>
<thead>
<tr>
<th>&amp;</th>
<th>0 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 0</td>
</tr>
<tr>
<td>1</td>
<td>0 1</td>
</tr>
</tbody>
</table>

**Or**
- \( A \mid B = 1 \) when either \( A = 1 \) or \( B = 1 \)

<table>
<thead>
<tr>
<th>\mid</th>
<th>0 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 1</td>
</tr>
<tr>
<td>1</td>
<td>1 1</td>
</tr>
</tbody>
</table>

**Not**
- \( \sim A = 1 \) when \( A = 0 \)

<table>
<thead>
<tr>
<th>\sim</th>
<th>0 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1 0</td>
</tr>
<tr>
<td>1</td>
<td>0 1</td>
</tr>
</tbody>
</table>

**Exclusive-Or (Xor)**
- \( A \oplus B = 1 \) when either \( A = 1 \) or \( B = 1 \), but not both

<table>
<thead>
<tr>
<th>\oplus</th>
<th>0 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 1</td>
</tr>
<tr>
<td>1</td>
<td>1 0</td>
</tr>
</tbody>
</table>
Application of Boolean Algebra

Applied to Digital Systems by Claude Shannon

- 1937 MIT Master's Thesis
- Reason about networks of relay switches
  - Encode closed switch as 1, open switch as 0

\[ A \& \neg B \quad \neg A \& B = A^B \]

Connection when

\[ A \& B \mid \neg A \& B \]

= A^B

General Boolean Algebras

Operate on Bit Vectors

- Operations applied bitwise

| 01101001 | 01101001 | 01101001 |
| 01010101 | 01010101 | ^ 01010101 | ~ 01010101 |

01000001 01111101 00111100 10101010

All of the Properties of Boolean Algebra Apply

Representing & Manipulating Sets

Representation

- Width \( w \) bit vector represents subsets of \{0, ..., \( w-1 \}\)
- \( a_j = 1 \) if \( j \in A \)

| 01101001 | 0, 3, 5, 6 |
| 76543210 |

| 01010101 | 0, 2, 4, 6 |
| 76543210 |

Operations

- \& Intersection
- | Union
- ^ Symmetric difference
- ~ Complement

| 01000001 | \{0, 6\} |
| 01111101 | \{0, 2, 3, 4, 5, 6\} |
| 00111100 | \{2, 3, 4, 5\} |
| 10101010 | \{1, 3, 5, 7\} |

Bit-Level Operations in C

Operations &, |, ~, ^ Available in C

- Apply to any "integral" data type
  - long, int, short, char, unsigned
- View arguments as bit vectors
- Arguments applied bit-wise

Examples (Char data type)

| ~0x41 --> 0xBE |
| ~010000012 --> 101111012 |
| ~0x00 --> 0xFF |
| ~000000002 --> 111111112 |
| 0x69 & 0x55 --> 0x41 |
| 011010012 & 010101012 --> 010000012 |
| 0x69 | 0x55 --> 0x7D |
| 011010012 | 010101012 --> 011111012 |
Contrast: Logic Operations in C

Contrast to Logical Operators
- &&, ||, !
  - View 0 as "False"
  - Anything nonzero as "True"
  - Always return 0 or 1
  - Early termination

Examples (char data type)
- !0x41 --> 0x00
- !0x00 --> 0x01
- !!0x41 --> 0x01
- 0x69 && 0x55 --> 0x01
- 0x69 || 0x55 --> 0x01
- p && *p (avoids null pointer access)

Shift Operations
Left Shift: $x << y$
- Shift bit-vector $x$ left $y$ positions
  - Throw away extra bits on left
  - Fill with 0’s on right
Right Shift: $x >> y$
- Shift bit-vector $x$ right $y$ positions
  - Logical shift
    - Fill with 0’s on left
  - Arithmetic shift
    - Replicate most significant bit on right

Strange Behavior
- Shift amount > word size

Integer C Puzzles
- Assume 32-bit word size, two’s complement integers
- For each of the following C expressions, either:
  - Argue that is true for all argument values
  - Give example where not true
- x < 0
- x >= 0
- x & 7 == 7
- x * x >= 0
- x > y
- x * x >= 0
- x > 0 && y > 0
- x >= 0
- x <= 0
- (x|-x)>>31 == -1
- ux >> 3 == ux/8
- x >> 3 == x/8
- x & (x-1) != 0

Encoding Integers

Unsigned
$$B2U(X) = \sum_{i=0}^{n-1} x_i \cdot 2^i$$

Two’s Complement
$$B2T(X) = -x_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} x_i \cdot 2^i$$

C short 2 bytes long

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimal</td>
<td>Hex</td>
<td>Binary</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>15213</td>
<td>3B 6D 00111011 01101101</td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>-15213</td>
<td>C4 93 11000100 10010011</td>
<td></td>
</tr>
</tbody>
</table>

Sign Bit
- For 2’s complement, most significant bit indicates sign
  - 0 for nonnegative
  - 1 for negative
Encoding Example (Cont.)

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

Weight

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

Sum

<table>
<thead>
<tr>
<th>Sum</th>
<th>15213</th>
<th>-15213</th>
</tr>
</thead>
</table>

Numeric Ranges

Unsigned Values
- \(U_{\text{Min}} = 0\)
- \(U_{\text{Max}} = 2^w - 1\)

Two’s Complement Values
- \(T_{\text{Min}} = -2^{\text{w-1}} - 1\)
- \(T_{\text{Max}} = 2^{\text{w-1}} - 1\)

Other Values
- Minus 1

Values for \(w = 16\)

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>65535</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>32767</td>
<td>7F FF</td>
<td>01111111 11111111</td>
</tr>
<tr>
<td>8000</td>
<td>80 00</td>
<td>10000000 00000000</td>
</tr>
<tr>
<td>-1</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>0</td>
<td>00 00</td>
<td>00000000 00000000</td>
</tr>
</tbody>
</table>

Values for Different Word Sizes

<table>
<thead>
<tr>
<th>(W)</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>

C Programming

- #include <limits.h>
- K&R App. B11
- Declares constants, e.g.,
  - UMAX
  - LONG_MAX
  - LONG_MIN
- Values platform-specific

Observations
- \(|T_{\text{Min}}| = T_{\text{Max}} + 1\)
  - Asymmetric range
- \(U_{\text{Max}} = 2 \times T_{\text{Max}} + 1\)

Unsigned & Signed Numeric Values

<table>
<thead>
<tr>
<th>(X)</th>
<th>(B2U(X))</th>
<th>(B2T(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

\(\Rightarrow\) 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
Relation between Signed & Unsigned

Two's Complement

Maintain Same Bit Pattern

Large negative weight → Large positive weight

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 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;

Casting Surprises

Expression Evaluation

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

Examples for W = 32

Constant1, Constant2 Relation Evaluation
0 0U == unsigned
0 -1 < signed
0U -1 > unsigned
2147483647 -2147483648 > signed
2147483647U -2147483648 < unsigned
-1 -2 > signed
(unsigned) -1 -2 > unsigned
2147483647 2147483648U < unsigned
-31 2147483647 (int) 2147483648U > signed

Explanation of Casting Surprises

2’s Comp. → Unsigned

- Ordering Inversion
- Negative → Big Positive
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

<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 3B 6D 00000000 00000000 00111011 01101101</td>
</tr>
<tr>
<td>( y )</td>
<td>-15213</td>
<td>C4 93 11000100 10010011</td>
</tr>
<tr>
<td>( iy )</td>
<td>-15213</td>
<td>FF FF C4 93 11111111 11111111 11000100 10010011</td>
</tr>
</tbody>
</table>

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

Why Should I Use Unsigned?

*Don’t Use Just Because Number Nonzero*
- Easy to make mistakes
  ```c
signed 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 Need Extra Bit’s Worth of Range*
- Working right up to limit of word size

Negating with Complement & Increment

Claim: Following Holds for 2’s Complement
\[ \sim x + 1 = -x \]

Complement
- Observation: \( \sim x + x = 1111...112 = -1 \)

\[ x \quad 100111101 
+ \quad \sim x \quad 01100010 
\]
\[ -1 \quad 11111111 \]

Increment
- \( \sim x + \sim (\sim x + 1) = \sim (\sim x + 1) \)
- \( \sim x + 1 = -x \)

Warning: Be cautious treating \texttt{int’s} as integers
- OK here
Comp. & Incr. 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 00111011 01101101</td>
</tr>
<tr>
<td>(-x)</td>
<td>-15214</td>
<td>C4 92 11000100 10010010</td>
</tr>
<tr>
<td>(-x+1)</td>
<td>-15213</td>
<td>C4 93 11000100 10010011</td>
</tr>
<tr>
<td>( y )</td>
<td>-15213</td>
<td>C4 93 11000100 10010011</td>
</tr>
</tbody>
</table>

Unsigned Addition

Operands: \( w \) bits

\[
\begin{array}{c}
\text{True Sum: } w+1 \text{ bits} \\
\text{Discard Carry: } w \text{ bits}
\end{array}
\]

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

Visualizing 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

\[ \text{Add}_4(u, v) \]

Visualizing Unsigned Addition

Wraps Around
- If true sum \( \geq 2^w \)
- At most once

\[ \text{UAdd}_4(u, v) \]
Mathematical Properties

Modular Addition Forms an Abelian Group
- Closed under addition
  \[ 0 \leq \text{UAdd}_w(u, v) \leq 2^w - 1 \]
- Commutative
  \[ \text{UAdd}_w(u, v) = \text{UAdd}_w(v, u) \]
- Associative
  \[ \text{UAdd}_w(t, \text{UAdd}_w(u, v)) = \text{UAdd}_w(\text{UAdd}_w(t, u), v) \]
- 0 is additive identity
  \[ \text{UAdd}_w(u, 0) = u \]
- Every element has additive inverse
  - Let \[ \text{UComp}_w(u) = 2^w - u \]
  - \[ \text{UAdd}_w(u, \text{UComp}_w(u)) = 0 \]

Two’s Complement Addition

Operands: \( w \) bits

True Sum: \( w + 1 \) bits

Discard Carry: \( w \) bits

\[ \text{TAdd}_w(u, v) \]

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
  ```
  - Will give \( s == t \)

Characterizing TAdd

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

True Sum

\[ \text{TAdd Result} \]

\[ \text{TAdd}_w(u, v) \]

\[ u + v < 2^{w-1} \] (NegOver)
\[ u + v \geq 2^{w-1} \] (PosOver)

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 \( \leq -2^{w-1} \)
  - Becomes positive
  - At most once

Visualizing 2’s Comp. Addition

NegOver

PosOver

TAdd\(_w(u, v)\)
Mathematical Properties of TAdd

Isomorphic Algebra to UAdd

- TAdd\(_w\)(u, v) = U2T(UAdd\(_w\)(T2U(u), T2U(v)))
  - Since both have identical bit patterns

Two’s Complement Under TAdd Forms a Group

- Closed, Commutative, Associative, 0 is additive identity
- Every element has additive inverse

\[
T\text{Comp}_w (u) = \begin{cases} 
-u & u \neq T\text{Min}_w \\
T\text{Min}_w & u = T\text{Min}_w 
\end{cases}
\]

Multiplication

Computing Exact Product of \(w\)-bit numbers \(x, y\)

- Either signed or unsigned

Ranges

- **Unsigned:** \(0 \leq x \cdot y \leq (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1\)
  - Up to 2\(w\) bits
- **Two’s complement min:** \(x \cdot y \geq (-2^{w-1})^2(2^{w-1}-1) = -2^{2w-2} + 2^{w-1}\)
  - Up to 2\(w-1\) bits
- **Two’s complement max:** \(x \cdot y \leq (-2^{w-1})^2 = 2^{2w-2}\)
  - Up to 2\(w\) bits, but only for \((T\text{Min}_w)^2\)

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

Operands: \(w\) bits

\[
\begin{array}{c}
u \\
n \times \ v \\
\hline
u \cdot v \\
\text{UMult}_w (u, v)
\end{array}
\]

True Product: \(2^w\) bits

Discard \(w\) bits: \(w\) bits

Standard Multiplication Function

- Ignores high order \(w\) bits

Implements Modular Arithmetic

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

Signed Multiplication in C

Operands: \(w\) bits

\[
\begin{array}{c}
u \\
n \times \ v \\
\hline
u \cdot v \\
\text{TMult}_w (u, v)
\end{array}
\]

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 \( u \times 2^k \)
- Both signed and unsigned

Operands: \( w \) bits

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

Examples
- \( u << 3 = u \times 8 \)
- \( u << 5 - u << 3 = u \times 24 \)
- Most machines shift and add faster than multiply
  - Compiler generates this code automatically

Compiled Multiplication Code

C Function

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

Compiled Arithmetic Operations

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

Explanation
- C compiler automatically generates shift/add code when multiplying by constant

Unsigned Power-of-2 Divide with Shift

Quotient of Unsigned by Power of 2
- \( u >> k \) gives \( \left\lfloor \frac{u}{2^k} \right\rfloor \)
- Uses logical shift

Operands:

\[
\begin{array}{c}
\text{binary point: }
\end{array}
\]

Division:

\[
\begin{array}{c}
\text{result: }
\end{array}
\]

Result:

\[
\begin{array}{c}
\text{computed: }
\end{array}
\]

<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>3B 6D</td>
<td>00111011 01101101</td>
</tr>
<tr>
<td>x &gt;&gt; 1</td>
<td>7606.5</td>
<td>7606</td>
<td>1D B6          00111010 10110110</td>
</tr>
<tr>
<td>x &gt;&gt; 4</td>
<td>950.8125</td>
<td>950</td>
<td>03 B6          00000011 10110110</td>
</tr>
<tr>
<td>x &gt;&gt; 8</td>
<td>59.4257813</td>
<td>59</td>
<td>00 3B          00000000 00111011</td>
</tr>
</tbody>
</table>

Compiled Unsigned Division Code

C Function

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

Compiled Arithmetic Operations

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

Explanation
- Uses logical shift for unsigned

For Java Users
- Logical shift written as `>>>`
## 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 \( u < 0 \)

\[
x \gg k\quad \downarrow\quad \lfloor x / 2^k \rfloor
\]

<table>
<thead>
<tr>
<th>Operands:</th>
<th>Divisor:</th>
<th>Division:</th>
<th>Result:</th>
<th>Quotient</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x )</td>
<td>( 2^k )</td>
<td>( x / 2^k )</td>
<td>RoundDown(( x / 2^k ))</td>
<td></td>
</tr>
</tbody>
</table>

### Correct Power-of-2 Divide

**Quotient of Negative Number by Power of 2**
- Want \( \lfloor x / 2^k \rfloor \) (Round Toward 0)
- Compute as \( \lfloor (x+2^k-1)/2^k \rfloor \)
  - In C: \( (x + (1<<k)-1) \gg k \)
  - Biases dividend toward 0

### Case 1: No rounding

<table>
<thead>
<tr>
<th>Dividend:</th>
<th>( u )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y \gg 1 )</td>
<td>-15213</td>
</tr>
<tr>
<td>( y \gg 4 )</td>
<td>-950.8125</td>
</tr>
<tr>
<td>( y \gg 8 )</td>
<td>-59.4257813</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Divisor:</th>
<th>( 2^k )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y \gg 1 )</td>
<td>-15213</td>
</tr>
<tr>
<td>( y \gg 4 )</td>
<td>-950.8125</td>
</tr>
<tr>
<td>( y \gg 8 )</td>
<td>-59.4257813</td>
</tr>
</tbody>
</table>

### Case 2: Rounding

<table>
<thead>
<tr>
<th>Dividend:</th>
<th>( x )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y \gg 1 )</td>
<td>-15213</td>
</tr>
<tr>
<td>( y \gg 4 )</td>
<td>-950.8125</td>
</tr>
<tr>
<td>( y \gg 8 )</td>
<td>-59.4257813</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Divisor:</th>
<th>( 2^k )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y \gg 1 )</td>
<td>-15213</td>
</tr>
<tr>
<td>( y \gg 4 )</td>
<td>-950.8125</td>
</tr>
<tr>
<td>( y \gg 8 )</td>
<td>-59.4257813</td>
</tr>
</tbody>
</table>

### Compiled Signed Division Code

#### C Function

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

#### Compiled Arithmetic Operations

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

#### Explanation
- Uses arithmetic shift for int
- For Java Users
  - Arith. shift written as >>
**Properties of Unsigned Arithmetic**

**Unsigned Multiplication with Addition Forms**

- Commutative Ring
  - Addition is commutative group
  - Closed under multiplication
    \[ 0 \leq \text{UMult}_w(u, v) \leq 2^w - 1 \]
  - Multiplication Commutative
    \[ \text{UMult}_w(u, v) = \text{UMult}_w(v, u) \]
  - Multiplication is Associative
    \[ \text{UMult}_w(t, \text{UMult}_w(u, v)) = \text{UMult}_w(\text{UMult}_w(t, u), v) \]
  - 1 is multiplicative identity
    \[ \text{UMult}_w(u, 1) = u \]
  - Multiplication distributes over addition
    \[ \text{UMult}_w(t, \text{UAdd}_w(u, v)) = \text{UAdd}_w(\text{UMult}_w(t, u), \text{UMult}_w(t, v)) \]

**Properties of Two’s Comp. 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 Integer Arithmetic**

- Both are rings
- Integers obey ordering properties, e.g.,
  \[ u > 0 \implies u + v > v \]
  \[ u > 0, v > 0 \implies u \cdot v > 0 \]

- These properties are not obeyed by two’s comp. arithmetic

\[ \text{TMax} + 1 == \text{TMin} \]
\[ 15213 \times 30426 == -10030 \text{ (16-bit words)} \]

**Integer C Puzzles Revisited**

- \( x < 0 \implies (x \times 2) < 0 \)
- \( u \times u \geq 0 \)
- \( x \& 7 == 7 \implies (x \ll 30) < 0 \)
- \( u \times u > -1 \)
- \( x > y \implies -x < -y \)
- \( x \times y \geq 0 \)
- \( x > 0 \& y > 0 \implies x + y > 0 \)
- \( x > 0 \implies -x <= 0 \)
- \( x <= 0 \implies -x >= 0 \)
- \( (x|\neg x)\gg 31 == -1 \)
- \( u \times u \gg 3 == u \times u / 8 \)
- \( x \gg 3 == x / 8 \)
- \( x \& (x-1) != 0 \)

**Initialization**

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