Encoding Byte Values

Byte = 8 bits

- Binary 00000000₂ to 11111111₂
- Decimal: 0₁₀ to 255₁₀
  - First digit must not be 0 in C
- Hexadecimal 00₁₆ to FF₁₆
  - Base 16 number representation
  - Use characters '0' to '9' and 'A' to 'F'
  - Write FA1D37B₁₆ in C as 0xFA1D37B
    - Or 0xFA1D37B

<table>
<thead>
<tr>
<th>Hex</th>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0010</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0011</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>0100</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>0110</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>0111</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>1000</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>1001</td>
</tr>
<tr>
<td>A</td>
<td>10</td>
<td>1010</td>
</tr>
<tr>
<td>B</td>
<td>11</td>
<td>1011</td>
</tr>
<tr>
<td>C</td>
<td>12</td>
<td>1100</td>
</tr>
<tr>
<td>D</td>
<td>13</td>
<td>1101</td>
</tr>
<tr>
<td>E</td>
<td>14</td>
<td>1110</td>
</tr>
<tr>
<td>F</td>
<td>15</td>
<td>1111</td>
</tr>
</tbody>
</table>
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 \( \approx 1.8 \times 10^{19} \) 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
  - int 4 4 4
  - long int 4 4 8
  - 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

Big Endian

<table>
<thead>
<tr>
<th>Address</th>
<th>0x100</th>
<th>0x101</th>
<th>0x102</th>
<th>0x103</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>01</td>
<td>23</td>
<td>45</td>
<td>67</td>
</tr>
</tbody>
</table>

Little Endian

<table>
<thead>
<tr>
<th>Address</th>
<th>0x100</th>
<th>0x101</th>
<th>0x102</th>
<th>0x103</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>67</td>
<td>45</td>
<td>23</td>
<td>01</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 32 bits: 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\t0x%.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):
```

```c
int a = 15213;
0x11ffffcb8 0x6d
0x11ffffcb9 0x3b
0x11ffffcba 0x00
0x11ffffcbb 0x00
```
Representing Integers

```c
int A = 15213;
int B = -15213;
long int C = 15213;
```

<table>
<thead>
<tr>
<th>Decimal: 15213</th>
<th>Binary: 0011 1011 0110 1101</th>
<th>Hex: 3 B 6 D</th>
</tr>
</thead>
</table>

Two's complement representation (Covered later)

Representing Pointers

```c
int B = -15213;
int *P = &B;
```

Different compilers & machines assign different locations to objects

Representing Strings

```c
char S[6] = "15213";
```

Strings in C
- 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

<table>
<thead>
<tr>
<th>And</th>
<th>Or</th>
</tr>
</thead>
<tbody>
<tr>
<td>A&amp;B = 1 when both A=1 and B=1</td>
<td>A</td>
</tr>
<tr>
<td>&amp; 0 1</td>
<td></td>
</tr>
<tr>
<td>0 0 0</td>
<td>0 0 1</td>
</tr>
<tr>
<td>1 0 1</td>
<td>1 1 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Not</th>
<th>Exclusive-Or (Xor)</th>
</tr>
</thead>
<tbody>
<tr>
<td>~A = 1 when A=0</td>
<td>A^B = 1 when either A=1 or B=1, but not both</td>
</tr>
<tr>
<td>~</td>
<td>^</td>
</tr>
<tr>
<td>0 1</td>
<td>0 1</td>
</tr>
<tr>
<td>1 0</td>
<td>0 1</td>
</tr>
<tr>
<td>1 1 0</td>
<td>1 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

\[ A \text{~} B \rightarrow \text{Connection when} \]
\[ A \text{~} B | \sim A \text{~} B \]
\[ \sim A \text{~} B = A ^ B \]

General Boolean Algebras

Operate on Bit Vectors

- Operations applied bitwise

| 01101001 | 01101001 | 01101001 |
| 01010101 | 01010101 | \sim 01010101 |
| 01000001 | 01111101 | 00111100 | 10101010 |

All of the Properties of Boolean Algebra Apply

Representing & Manipulating Sets

Representation

- Width \( w \) bit vector represents subsets of \( \{0, \ldots, w-1\} \)
- \( a_j = 1 \) 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 \rightarrow 0xBE
- ~0x00 \rightarrow 0xFF

| 0x69 \& 0x55 \rightarrow 0x41 |
| 01101001 \& 01010101 \rightarrow 01000001 |

| 0x69 | 0x55 \rightarrow 0x7D |
| 01101001 | 01010101 \rightarrow 01111010 |

15-213, S '08
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

Encoding Integers

unsigned \( x = 15213 \);  
short int \( y = -15213 \);

C short 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>3B 6D 00111011 01101101</td>
</tr>
<tr>
<td>y</td>
<td>-15213</td>
<td>C4 93 01100100 10010011</td>
</tr>
</tbody>
</table>

Signed Bit

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

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

Initialization

```c
int x = foo();
int y = bar();
unsigned ux = x;
unsigned uy = y;
```
Encoding Example (Cont.)

- \( x = 15213 \) \( \rightarrow \) \( 00111011 \ 01101101 \)
- \( y = -15213 \) \( \rightarrow \) \( 11000100 \ 10010011 \)

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

Sum: \( 15213 \) \( -15213 \)

Numeric Ranges

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

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

Other Values
- Minus 1 \( 111...1 \)

Values for \( w = 16 \)

<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 11111111 11111111</td>
</tr>
<tr>
<td>TMax</td>
<td>32767</td>
<td>7F FF 01111111 11111111</td>
</tr>
<tr>
<td>TMin</td>
<td>-32768</td>
<td>80 00 10000000 00000000</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>FF FF 11111111 11111111</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 00 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>9,223,720,368,547,755,807</td>
</tr>
<tr>
<td>TMin</td>
<td>-128</td>
<td>-32,768</td>
<td>-2,147,483,648</td>
<td>-9,223,720,368,547,755,808</td>
</tr>
</tbody>
</table>

Observations
- \( |TMin| = TMax + 1 \)
- Asymmetric range
- \( UMax = 2 \cdot TMax + 1 \)

C Programming
- \#include \( <limits.h> \)
- K&R App. B11
- Declares constants, e.g.,
  - ULONG_MAX
  - LONG_MAX
  - LONG_MIN
- Values platform-specific

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

⇒ Can Invert Mappings
- U2B(\( x \)) = B2U\(^{-1}\)(\( x \))
- T2B(\( x \)) = B2T\(^{-1}\)(\( x \))
Mapping Between Signed & Unsigned

- Two's Complement: $x \xrightarrow{T2U} \text{Unsigned}$
  - Maintain Same Bit Pattern

- 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>-8</td>
<td>-8</td>
</tr>
<tr>
<td>1000</td>
<td>-7</td>
<td>-7</td>
</tr>
<tr>
<td>1001</td>
<td>-6</td>
<td>-6</td>
</tr>
<tr>
<td>1010</td>
<td>-5</td>
<td>-5</td>
</tr>
<tr>
<td>1011</td>
<td>-4</td>
<td>-4</td>
</tr>
<tr>
<td>1100</td>
<td>-3</td>
<td>-3</td>
</tr>
<tr>
<td>1101</td>
<td>-2</td>
<td>-2</td>
</tr>
<tr>
<td>1110</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

- Define mappings been unsigned and two's complement numbers based on their bit-level representations

Mapping Signed ↔ Unsigned

- Signed: $x \xrightarrow{T2U} \text{Unsigned}$
  - $x \geq 0$
  - $x + 2^w$ for $x < 0$

- Relation between Signed & Unsigned
  - Large negative weight $\rightarrow$ Large positive weight

Maintain Same Bit Pattern
**Signed vs. Unsigned in C**

**Constants**
- By default are considered to be signed integers
- Unsigned if have “U” as suffix
  
  \( 0U, 4294967259U \)

**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 \(W = 32\)

<table>
<thead>
<tr>
<th>Constant, (x)</th>
<th>Constant, (y)</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>signed</td>
</tr>
<tr>
<td>(unsigned) -1</td>
<td>-2</td>
<td>&gt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>(int) 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>

---

**Explanation of Casting Surprises**

**2’s Comp. \(\rightarrow\) Unsigned**
- Ordering Inversion
- Negative \(\rightarrow\) 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:
- \(X’ = x_{w-1}, \ldots, x_{w-1}, x_{w-2}, \ldots, x_0\)
  ```
  \(k\) copies of MSB
  ```

**Diagram**
### Sign Extension Example

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

```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>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</td>
</tr>
<tr>
<td>iy</td>
<td>-15213</td>
<td>FF FF C4 93 11111111 11111111 11000100 10010011</td>
</tr>
</tbody>
</table>

### Why Should I Use Unsigned?

**Don’t Use Just Because Number Nonnegative**
- Easy to make mistakes
  - unsigned i;
  - for (i = 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

### Complement & Increment

**Claim: Following Holds for 2’s Complement**

- \(~x + 1 == -x\)

**Complement**
- Observation: \(\neg x + x == \text{111...1}_2 == -1\)

<table>
<thead>
<tr>
<th>x</th>
<th>(\neg\text{1001111010})</th>
<th>11111111 11111111</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>(\neg\text{01100010})</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td></td>
<td>(\neg\text{11111111})</td>
<td>00000000 00000000</td>
</tr>
</tbody>
</table>

**Increment**
- \(\neg x + x == -1\)
- \(\neg x + x + (\neg x + 1) == \neg x + (\neg x + 1)\)
- \(\neg x + 1 == -x\)

**Warning: Be cautious treating int’s as integers**
- OK here

### Comp. & Incr. Examples

```c
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>00111011 01101101</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>11000100 10100011</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00 00</td>
<td>00000000 00000000</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>FF FF</td>
<td>11111111 11111111</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>00 00</td>
<td>00000000 00000000</td>
<td></td>
</tr>
</tbody>
</table>

- 37 – 15-213, S ’08
- 38 – 15-213, S ’08
- 39 – 15-213, S ’08
- 40 – 15-213, S ’08
Unsigned Addition

Operands: w bits

True Sum: w+1 bits

Discard Carry: w bits

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 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

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
\[ u \]
\[ + \]
\[ v \]
True Sum: w+1 bits
\[ u + v \]
Discard Carry: w bits
\[ \text{TAdd}_w(u,v) \]

TAdd and UAdd have Identical Bit-Level Behavior
- Signed vs. unsigned addition in 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
\[
\begin{array}{c|c}
\text{TAdd Result} & \text{True Sum} \\
\hline
011...1 & 2^{w-1} \\
010...0 & 2^{w-1} - 1 \\
000...0 & 0 \\
1011...1 & -2^{w-1} - 1 \\
1000...0 & -2^w \\
\end{array}
\]

Visualizing 2’s Comp. Addition

Values
- 4-bit two’s comp.
- Range from -8 to +7

Wraps Around
- If \( \text{sum} \geq 2^{w-1} \)
  - Becomes negative
  - At most once
- If \( \text{sum} < -2^{w-1} \)
  - Becomes positive
  - At most once

Mathematical Properties of TAdd

Isomorphic Algebra to UAdd
- \( \text{TAdd}_w(u,v) = U2T(\text{UAdd}_w(T2U(u), T2U(v))) \)
  - 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

\[
\text{TComp}_w(u) = \begin{cases} 
-u & u \neq \text{TMin}_w \\
\text{TMin}_w & u = \text{TMin}_w 
\end{cases}
\]

- 45 – 15-213, S ‘08
- 46 – 15-213, S ‘08
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 $2w$ bits
- Two's complement min: $x \cdot y \geq (-2^{w-1})(2^{w-1} - 1) = -2^{2w-2} + 2^{w-1}$
  - Up to $2w-1$ bits
- Two's complement max: $x \cdot y \leq (-2^{w-1})^2 = 2^{2w-2}$
  - Up to $2w$ bits, but only for $(TMin_w)^2$

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

<table>
<thead>
<tr>
<th>$u$</th>
<th>$v$</th>
<th>$u \cdot v$</th>
</tr>
</thead>
</table>

Operands: $w$ bits

True Product: $2w$ bits

Discard $w$ bits: $w$ bits

Standard Multiplication Function

- Ignores high order $w$ bits

Implements Modular Arithmetic

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

Signed Multiplication in C

Operands: $w$ bits

<table>
<thead>
<tr>
<th>$u$</th>
<th>$v$</th>
<th>$u \cdot v$</th>
</tr>
</thead>
</table>

True Product: $2w$ 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 $u \cdot 2^k$
- Both signed and unsigned

Examples

- $u << 3$ $== u \cdot 8$
- $u << 5 - u << 3$ $== u \cdot 24$
- Most machines shift and add faster than multiply

Examples

- Compiler generates this code automatically
Compiled Multiplication Code

C Function

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

Compiled Arithmetic Operations

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

t <- x+x*2
return t << 2;

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

Compiled Unsigned Division Code

C Function

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

Compiled Arithmetic Operations

shrl $3, %eax

# Logical shift
return x >> 3;

- Uses logical shift for unsigned

For Java Users

# Logical shift written as >>>

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

Divison:

\[ \frac{u}{2^k} \]

Result:

\[ \lfloor \frac{u}{2^k} \rfloor \]

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 \( u < 0 \)

Divison:

\[ \frac{x}{2^k} \]

Result:

RoundDown(\( \frac{x}{2^k} \))
Correct Power-of-2 Divide (Cont.)

Case 2: Rounding

\[
\begin{array}{c|c|c|c|c}
\text{Dividend:} & x & \underbrace{\ddots}_{k} & \ddots & \ddots \\
+2^k-1 & \underbrace{0}_{k} & \ddots & \ddots & \ddots \\
\hline
\text{Divisor:} & 2^k & \\ \\
\text{\left\lceil x / 2^k \right\rceil} & \underbrace{1}_{k} & \ddots & \ddots & \ddots \\
\end{array}
\]

\[
\text{Binary Point}
\]

\text{Biasing adds 1 to final result}

\text{Incremented by 1}

\text{Incremented by 1}

Compiled Signed Division Code

C Function

\[
\text{int idiv8(int x) \{
    \text{return x/8; }
\}}
\]

Compiled Arithmetic Operations

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

Explanation

```
if x < 0
    x += 7;
# Arithmetic shift
    return x >> 3;
```

- 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.,
  \[ u > 0 \quad \Rightarrow \quad u + v > v \]
  \[ u > 0, v > 0 \quad \Rightarrow \quad u \cdot v > 0 \]
- These properties are not obeyed by two’s comp. arithmetic

\[ T_{Max} + 1 = T_{Min} \]
\[ -61 \quad 15213 \times 30426 = -10030 \text{ (16-bit words)} \]

Integer C Puzzles Revisited

- \( x < 0 \quad \Rightarrow \quad (x \times 2 < 0) \)
- \( u_x = 0 \)
- \( x \& 7 = 7 \quad \Rightarrow \quad (x < 30) < 0 \)
- \( u_x > -1 \)
- \( x > y \quad \Rightarrow \quad -x < -y \)
- \( x \times x > 0 \)
- \( x > 0 \quad \& \quad y > 0 \quad \Rightarrow \quad x + y > 0 \)
- \( x > 0 \quad \Rightarrow \quad -x <= 0 \)
- \( x <= 0 \quad \Rightarrow \quad -x >= 0 \)
- \( (x \mid -x) >> 31 = -1 \)
- \( u_x >> 3 = u_x / 8 \)
- \( x >> 3 = x / 8 \)
- \( x \& (x-1) != 0 \)

\[
\begin{align*}
\text{Initialization} \\
\text{int } x = \text{foo}(); \\
\text{int } y = \text{bar}(); \\
\text{unsigned } u_x = x; \\
\text{unsigned } u_y = y;
\end{align*}
\]