Computer Systems Principles

Data Representation in C
Today

- Overflow and underflow
- Data representation in C
  - basic data types
  - cast
- Bit Manipulation
  - bitwise and, or, exclusive-or, and not, shift
OVERFLOW AND UNDERFLOW
Ariane 5

Exploded 37 seconds after lift-off with cargo worth 500 million

Why..

• Computed horizontal velocity as 64-bit floating-point number
• Converted to 16-bit integer
• Worked for Ariane 4
• Overflowed for Ariane 5
Two’s Complement Overflow & Underflow

- **Overflow** is caused by a value near the upper limit of the range, while **an underflow** is caused by values near the lower limit of the range.
Overflow: Example

Consider the 8-bit two’s complement addition:

\[
\begin{array}{cccccccc}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 2 & 7 & 0 & 1 & 1 & 1 & 1 & 1 \\
+ & 1 & & & & & & + & 1 \\
1 & 2 & 8 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}
\]

• The result should be +128, but the leftmost bit is 1, therefore the result is -128!
• This is an overflow: an arithmetic operation that should be positive gives a negative result.
Underflow: Example

Consider the 8-bit two’s complement addition:

\[
\begin{array}{cccccccccccc}
- & 1 & 2 & 8 & & & & & & & & \\
- & 1 & - & & & & & & & & & \\
\hline
- & 1 & 2 & 9 & & & & & & & & \\
\end{array}
\]

\[
\begin{array}{cccccccccccc}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
& 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\hline
1 & & & & & & & & & & & \\
\end{array}
\]

• The result should be -129, but the leftmost bit is 0, therefore the result is +127!
• This is an underflow: as an arithmetic operation that should be negative gives a positive result.
Is this dynamic ram??

Source: http://xkcd.com/571/
DATA TYPES IN C
Data types in C

```c
int x;
```

– first IBM PC: `int` [16 bits]
– today’s PC: `int` [32 bits]
  (even on 64-bit PCs – but be careful!)
## Data types in C (for gcc)

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>Typical 32-bit</th>
<th>Intel IA 32</th>
<th>x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>longlong</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>longdouble</td>
<td>8</td>
<td>10/12</td>
<td>10/16</td>
</tr>
<tr>
<td>pointer</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>

**Code Portability?**

Notice that `long` and `pointer` data types are different on different processors (and maybe compilers).
A simple to print data type size

```c
#include <stdio.h> // This is needed to run printf()
int main()
{
    int a;
    short int b;
    unsigned int c;
    char d;
    // size-of displays the size of the data type
    printf("Size of int=%d bytes\n",sizeof(a));
    printf("Size of short int=%d bytes\n",sizeof(b));
    printf("Size of unsigned int=%d bytes\n",sizeof(c));
    printf("Size of char=%d bytes\n",sizeof(d));
    return 0;
}
```
C allows conversions between signed (two’s complement) and unsigned.

```
unsigned short int ux = 15213;
short int x           = (short int) ux;
short int y           = -15213;
unsigned short int uy = (unsigned short) y;
```

Resulting Value

- No change in bit representation!
- Results reinterpreted
Signed vs. Unsigned in C

• Declaration for two signed and unsigned integers
  
  ```
  int tx, ty; // signed
  unsigned ux, uy; // unsigned
  ```

• Explicit casting between signed & unsigned
  
  ```
  tx = (int) ux;
  uy = (unsigned) ty;
  ```

• Implicit casting also occurs via assignments and procedure calls
  
  ```
  tx = ux;
  uy = ty;
  ```
Expanding Bit representation: Sign Extension

- Given $w$-bit signed integer $x$:
  - Convert it to $w+k$-bit integer with same value
  - Make $k$ copies of sign bit: $X = x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_0$

$$X' = \underbrace{x_{w-1}, \ldots, x_{w-1}}_{k \text{ copies of MSB}}$$

![Diagram](image-url)
Converting from smaller to larger integer data type

C automatically performs sign- or zero-extension

```
short int sx = -12345;
int x = sx;
unsigned short int usx = sx;
unsigned int ux = usx;
```

<table>
<thead>
<tr>
<th>Variables</th>
<th>Value</th>
<th>Hexadecimal representation</th>
<th>Binary representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>sx</td>
<td>-12345</td>
<td>cf c7</td>
<td>11001111 11000111</td>
</tr>
<tr>
<td>usx</td>
<td>53191</td>
<td>cf c7</td>
<td>11001111 11000111</td>
</tr>
<tr>
<td>x</td>
<td>-12345</td>
<td>ff ff cf c7</td>
<td>11111111 11111111 11001111 11000111</td>
</tr>
<tr>
<td>ux</td>
<td>53191</td>
<td>00 00 cf c7</td>
<td>00000000 00000000 11001111 11000111</td>
</tr>
</tbody>
</table>
Expanding Bit representation: Sign Extension

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

```
short int sx = -12345;
int x = sx;
unsigned short int usx = sx;
unsigned int ux = usx;
```

<table>
<thead>
<tr>
<th>Variables</th>
<th>Value</th>
<th>Hexadecimal representation</th>
<th>Binary representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>sx</td>
<td>-12345</td>
<td>cf c7</td>
<td>11001111 11000111</td>
</tr>
<tr>
<td>usx</td>
<td>53191</td>
<td>cf c7</td>
<td>11001111 11000111</td>
</tr>
<tr>
<td>x</td>
<td>-12345</td>
<td>ff ff cf c7</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>ux</td>
<td>53191</td>
<td>00 00 cf c7</td>
<td>00000000 00000000</td>
</tr>
</tbody>
</table>

Expanding:

- Unsigned: zero added
- Signed: sign extension
Signed and unsigned integer constants in C

• By default an integer is assumed to be signed integer
  – Example: 3

• An integer constant may be suffixed by the letter u or U, to specify that it is unsigned.
  – Example: 3u
Casting Surprises: Expression evaluation

• If there is a mix of unsigned and signed in a single expression, signed values are implicitly cast to unsigned!
Casting Surprises: Expression evaluation

- If there is a mix of unsigned and signed in a single expression, signed values are implicitly cast to unsigned!
- Including comparison operations <,>,==,<=,>=
- E.g.: $W = 32$ TMIN = -2,147,483,648 ($2^{31}$) TMAX = 2,147,483,647 ($2^{31}-1$)

<table>
<thead>
<tr>
<th>Constant-1</th>
<th>Relation</th>
<th>Constant-2</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>0u</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td></td>
<td>0u</td>
<td></td>
</tr>
<tr>
<td>2147483647</td>
<td></td>
<td>-2147483648</td>
<td></td>
</tr>
<tr>
<td>2147483647u</td>
<td></td>
<td>-2147483648</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td></td>
<td>-2</td>
<td></td>
</tr>
<tr>
<td>(unsigned) -1</td>
<td></td>
<td>-2</td>
<td></td>
</tr>
<tr>
<td>2147483647</td>
<td></td>
<td>2147483648u</td>
<td></td>
</tr>
<tr>
<td>2147483647</td>
<td></td>
<td>(int) 2147483648u</td>
<td></td>
</tr>
</tbody>
</table>
Casting Surprises: Expression evaluation

- If there is a mix of unsigned and signed in a single expression, signed values are implicitly cast to unsigned!
- Including comparison operations <,>,==,<=,>=
- E.g.: \( W = 32 \) TMIN = \(-2,147,483,648\) \((2^{31})\) TMAX = \(2,147,483,647\) \((2^{31}-1)\)

<table>
<thead>
<tr>
<th>Constant-1</th>
<th>Relation</th>
<th>Constant-2</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>==</td>
<td>0u</td>
<td>unsigned</td>
</tr>
<tr>
<td>-1</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td></td>
<td>0u</td>
<td></td>
</tr>
<tr>
<td>2147483647</td>
<td></td>
<td>-2147483648</td>
<td></td>
</tr>
<tr>
<td>2147483647u</td>
<td></td>
<td>-2147483648</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td></td>
<td>-2</td>
<td></td>
</tr>
<tr>
<td>(unsigned) -1</td>
<td></td>
<td>-2</td>
<td></td>
</tr>
<tr>
<td>2147483647</td>
<td></td>
<td>2147483648u</td>
<td></td>
</tr>
<tr>
<td>2147483647</td>
<td></td>
<td>(int) 2147483648u</td>
<td></td>
</tr>
</tbody>
</table>
Casting Surprises: Expression evaluation

- If there is a mix of unsigned and signed in a single expression, signed values are implicitly cast to unsigned!
- Including comparison operations <, >, ==, <=, >=
- E.g.: \( W = 32 \) TMIN = \(-2,147,483,648\) (\(2^{31}\)) TMAX = \(2,147,483,647\) (\(2^{31}-1\))

<table>
<thead>
<tr>
<th>Constant-1</th>
<th>Relation</th>
<th>Constant-2</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>==</td>
<td>0u</td>
<td>unsigned</td>
</tr>
<tr>
<td>-1</td>
<td>&lt;</td>
<td>0</td>
<td>signed</td>
</tr>
<tr>
<td>-1</td>
<td></td>
<td>0u</td>
<td></td>
</tr>
<tr>
<td>2147483647</td>
<td></td>
<td>-2147483648</td>
<td></td>
</tr>
<tr>
<td>2147483647u</td>
<td></td>
<td>-2147483648</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td></td>
<td>-2</td>
<td></td>
</tr>
<tr>
<td>(unsigned) -1</td>
<td></td>
<td>-2</td>
<td></td>
</tr>
<tr>
<td>2147483647</td>
<td></td>
<td>2147483648u</td>
<td></td>
</tr>
<tr>
<td>2147483647</td>
<td></td>
<td>(int) 2147483648u</td>
<td></td>
</tr>
</tbody>
</table>
Casting Surprises: Expression evaluation

- If there is a mix of unsigned and signed in a single expression, signed values are implicitly cast to unsigned!
- Including comparison operations <, >, ==, <=, >=
- E.g.: \( W = 32 \) TMIN = \(-2,147,483,648\) (2^31) TMAX = \(2,147,483,647\) (2^31-1)

<table>
<thead>
<tr>
<th>Constant-1</th>
<th>Relation</th>
<th>Constant-2</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>==</td>
<td>0u</td>
<td>unsigned</td>
</tr>
<tr>
<td>-1</td>
<td>&lt;</td>
<td>0</td>
<td>signed</td>
</tr>
<tr>
<td>-1</td>
<td>&gt;</td>
<td>0u</td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td></td>
<td>-2147483648</td>
<td></td>
</tr>
<tr>
<td>2147483647u</td>
<td></td>
<td>-2147483648</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td></td>
<td>-2</td>
<td></td>
</tr>
<tr>
<td>(unsigned) -1</td>
<td></td>
<td>-2</td>
<td></td>
</tr>
<tr>
<td>2147483647</td>
<td></td>
<td>2147483648u</td>
<td></td>
</tr>
<tr>
<td>2147483647</td>
<td></td>
<td>(int) 2147483648u</td>
<td></td>
</tr>
</tbody>
</table>
Casting Surprises: Expression evaluation

• If there is a mix of unsigned and signed in a single expression, signed values are implicitly cast to unsigned!

• Including comparison operations <,>,==,<=,>=

• E.g.: \( W = 32 \) TMIN = \(-2,147,483,648\) (\(2^{31}\)) TMAX = \(2,147,483,647\) (\(2^{31}-1\))

<table>
<thead>
<tr>
<th>Constant-1</th>
<th>Relation</th>
<th>Constant-2</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>==</td>
<td>0u</td>
<td>unsigned</td>
</tr>
<tr>
<td>-1</td>
<td>&lt;</td>
<td>0</td>
<td>signed</td>
</tr>
<tr>
<td>-1</td>
<td>&gt;</td>
<td>0u</td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>&gt;</td>
<td>-2147483648</td>
<td>signed</td>
</tr>
<tr>
<td>2147483647u</td>
<td></td>
<td>-2147483648</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td></td>
<td>-2</td>
<td></td>
</tr>
<tr>
<td>(unsigned) -1</td>
<td></td>
<td>-2</td>
<td></td>
</tr>
<tr>
<td>2147483647</td>
<td></td>
<td>2147483648u</td>
<td></td>
</tr>
<tr>
<td>2147483647</td>
<td>(int)</td>
<td>2147483648u</td>
<td></td>
</tr>
</tbody>
</table>
Casting Surprises: Expression evaluation

• If there is a mix of unsigned and signed in a single expression, signed values are implicitly cast to unsigned!
• Including comparison operations <,>,==,<=,>=
• E.g.: \( W = 32 \) TMIN = \(-2,147,483,648\) (\(2^{31}\)) TMAX = \(2,147,483,647\) (\(2^{31}-1\))

<table>
<thead>
<tr>
<th>Constant-1</th>
<th>Relation</th>
<th>Constant-2</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>==</td>
<td>0u</td>
<td>unsigned</td>
</tr>
<tr>
<td>-1</td>
<td>&lt;</td>
<td>0</td>
<td>signed</td>
</tr>
<tr>
<td>-1</td>
<td>&gt;</td>
<td>0u</td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>&gt;</td>
<td>-2147483648</td>
<td>signed</td>
</tr>
<tr>
<td>2147483647u</td>
<td>&lt;</td>
<td>-2147483648</td>
<td>unsigned</td>
</tr>
<tr>
<td>(unsigned) -1</td>
<td></td>
<td>-2</td>
<td></td>
</tr>
<tr>
<td>2147483647</td>
<td></td>
<td>2147483648u</td>
<td></td>
</tr>
<tr>
<td>2147483647</td>
<td></td>
<td>(int) 2147483648u</td>
<td></td>
</tr>
</tbody>
</table>
Casting Surprises: Expression evaluation

• If there is a mix of unsigned and signed in a single expression, signed values are implicitly cast to unsigned!
• Including comparison operations <,>,==,<=,>=
• E.g.: $W = 32$ $TMIN = -2,147,483,648$ $(2^{31})$ $TMAX = 2,147,483,647$ $(2^{31}-1)$

<table>
<thead>
<tr>
<th>Constant-1</th>
<th>Relation</th>
<th>Constant-2</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>==</td>
<td>0u</td>
<td>unsigned</td>
</tr>
<tr>
<td>-1</td>
<td>&lt;</td>
<td>0</td>
<td>signed</td>
</tr>
<tr>
<td>-1</td>
<td>&gt;</td>
<td>0u</td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>&gt;</td>
<td>-2147483648</td>
<td>signed</td>
</tr>
<tr>
<td>2147483647u</td>
<td>&lt;</td>
<td>-2147483648</td>
<td>unsigned</td>
</tr>
<tr>
<td>-1</td>
<td>&gt;</td>
<td>-2</td>
<td>signed</td>
</tr>
<tr>
<td>(unsigned) -1</td>
<td></td>
<td>-2</td>
<td></td>
</tr>
<tr>
<td>2147483647</td>
<td></td>
<td>2147483648u</td>
<td></td>
</tr>
<tr>
<td>2147483647</td>
<td></td>
<td>(int) 2147483648u</td>
<td></td>
</tr>
</tbody>
</table>
Casting Surprises: Expression evaluation

• If there is a mix of unsigned and signed in a single expression, signed values are implicitly cast to unsigned!
• Including comparison operations <,>,==,<=,>=
• E.g.: \( W = 32 \) TMIN = \(-2,147,483,648\) \( (2^{31}) \) TMAX = \(2,147,483,647\) \( (2^{31}-1)\)

<table>
<thead>
<tr>
<th>Constant-1</th>
<th>Relation</th>
<th>Constant-2</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>==</td>
<td>0u</td>
<td>unsigned</td>
</tr>
<tr>
<td>-1</td>
<td>&lt;</td>
<td>0</td>
<td>signed</td>
</tr>
<tr>
<td>-1</td>
<td>&gt;</td>
<td>0u</td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>&gt;</td>
<td>-2147483648</td>
<td>signed</td>
</tr>
<tr>
<td>2147483647u</td>
<td>&lt;</td>
<td>-2147483648</td>
<td>unsigned</td>
</tr>
<tr>
<td>-1</td>
<td>&gt;</td>
<td>-2</td>
<td>signed</td>
</tr>
<tr>
<td>(unsigned) -1</td>
<td>&gt;</td>
<td>-2</td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td></td>
<td>2147483648u</td>
<td></td>
</tr>
<tr>
<td>2147483647</td>
<td></td>
<td>(int) 2147483648u</td>
<td></td>
</tr>
</tbody>
</table>
What is the relationship between two integer 2147483647 and 2147483648u?
Assume each integer has 32 bit. MINIMUM = -2,147,483,648 (2^31) MAXIMUM = 2,147,483,647 (2^31-1)

A. ==
B. <
C. >
i-clicker question

What is the relationship between two integer 2147483647 and (int) 2147483648u? Asssume each integer has 32 bit. MINIMUM = \(-2,147,483,648\) \((2^{31})\) MAXIMUM = \(2,147,483,647\) \((2^{31}-1)\)

A. ==
B. <
C. >
BOOLEAN ALGEBRA
Boolean Algebra

• Developed by George Boole in the 19th Century and applied to Digital Systems by Claude Shannon

“Laws of Thought”
Bit-Manipulations

Boolean Algebra:
• Developed by George Boole in the 19th Century and applied to Digital Systems by Claude Shannon
• Encode “True”/“On”/“Yes” as 1 and “False”/“Off”/“No” as 0

“Laws of Thought”
Bit-Manipulations

Not (~A)

<table>
<thead>
<tr>
<th>~</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
Bit-Manipulations

<table>
<thead>
<tr>
<th></th>
<th>Not (~A)</th>
<th>And (A &amp; B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>~</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0 0 0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1 0 1</td>
</tr>
</tbody>
</table>
Bit-Manipulations

|       | Not (¬A) | And (A & B) | Or (A|B) |
|-------|----------|-------------|---------|
| ~     | 0 1      | 0 0 0       | 0 0 1   |
| 0     | 1 0      | 1 0 1       | 1 1 1   |
| 1     | 0 0      | 1 1 1       | 1 1 1   |
Bit-Manipulations

| Not (~A) | And (A & B) | Or (A|B) | Xor A^ B |
|---------|-------------|----------|---------|
| ~       | &           | | ^      |
| 0       | 0           | 0        | 0       |
| 1       | 1           | 1        | 1       |
| 1       | 0           | 1        | 0       |
| 0       | 1           | 0        | 1       |
### Bit-Manipulations

Boolean operations are applied *bitwise* on the bit sequences (i.e., by columns)

| Not (~A) | And (A & B) | Or (A|B) | Xor (A^ B) |
|---------|-------------|---------|------------|
| ~1 0 1 0 | & 1 0 1 0  | l 1 0 1 0 | ^ 1 0 1 0  |
| 0 1 0 1 | 0 0 1 0  | 1 1 1 0 | 1 1 0 0 |
Bit Manipulations

Boolean algebra obeys some of the properties of integer algebra.. but not all!

<table>
<thead>
<tr>
<th>Boolean</th>
<th>Boolean</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sum and product identities</td>
<td>A(</td>
<td>0 = A)</td>
</tr>
<tr>
<td></td>
<td>A&amp;1 = A</td>
<td>A*1 = A</td>
</tr>
<tr>
<td>Zero is product annihilator</td>
<td>A &amp; 0 = 0</td>
<td>A *0 = 0</td>
</tr>
<tr>
<td>Cancellation of negation</td>
<td>(~(\sim A) = A)</td>
<td>(-(-A) = A)</td>
</tr>
<tr>
<td>Laws of Complements</td>
<td>A \mid \sim A = 1</td>
<td>A + -A \neq 1</td>
</tr>
<tr>
<td>Every element has an additive inverse</td>
<td>A \mid \sim A \neq 0</td>
<td>A + -A = 0</td>
</tr>
</tbody>
</table>
SHIFT OPERATION
Shift unsigned integers

• Left shift $x \ll y$
  – Discard bits on the left
  – Fill with 0s on the right
  – $00000010 \ll 2 = 00001000$

• Right shift $x \gg y$
  – Discard bits on the right
  – Fill with 0s on the left
  – $10000010 \gg 3 = 00010000$
Shift unsigned integers

• Left shift \( x<<<y \)
  – Discard bits on the left
  – Fill with 0s on the right
  – \( 00000010<<<2 = 00001000 \)

• Right shift \( x>>y \)
  – Discard bits on the right
  – Fill with 0s on the left
  – \( 10000010>>3 = 00010000 \)
Shift unsigned integers

• Left shift $x << y$
  – Discard bits on the left
  – Fill with 0s on the right
  – $00000010 << 2 = 00001000$

• Right shift $x >> y$
  – Discard bits on the right
  – Fill with 0s on the left
  – $10000010 >> 3 = 00010000$
Shift unsigned integers

• **Left shift** \( x << y \)
  – Discard bits on the left
  – Fill with 0s on the right
  – \( 00000010 << 2 = 00001000 \)

• **Right shift** \( x >> y \)
  – Discard bits on the right
  – Fill with 0s on the left
  – \( 10000010 >> 3 = 00010000 \)
Shift unsigned integers

- Left shift $x << y$
  - Discard bits on the left
  - Fill with 0s on the right
  - $00000100 << 2 = 00001000$

- Right shift $x >> y$
  - Discard bits on the right
  - Fill with 0s on the left
  - $10000010 >> 3 = 00010000$

- Left shift $y$ equivalent to multiplying by $2^y$
Shift unsigned integers

• Left shift $x\ll y$
  – Discard bits on the left
  – Fill with 0s on the right
  – $00000010\ll2=00001000$

• Right shift $x\gg y$
  – Discard bits on the right
  – Fill with 0s on the left
  – $10000010\gg3=00010000$

• Left shift $y$ equivalent to multiplying by $2^y$
• Right shift $y$ equivalent to dividing by $2^y$
Shift signed integers

- Left shift: \( x \ll k \): Shift bit-vector \( x \) left by \( k \) positions
  - Throw away extra bits on the left
  - Fill with 0’s on the right.
  - \( x \ll k \) is equivalent to \( x \times 2^k \)
Bit Manipulations: shift operators

• Right shift: \( x >> k \) : Shift bit-vector \( x \) right \( k \) positions.
  
  – Throw away extra bits on the right

TWO KINDS:

• Logical Shift: Fill with 0’s on the left.

• Arithmetic Shift : Replicate with most significant bit on the left.
  
  – Copies the sign bit
  
  – Arithmetic shift is equivalent to logical shift for positive numbers
Bit Manipulations: shift operators

• Right shift: $x >> k$: Shift bit-vector $x$ right $k$ positions.
  – Throw away extra bits on the right

TWO KINDS:

• Logical Shift: Fill with 0’s on the left.

• Arithmetic Shift: Replicate with most significant bit on the left.
  – Copies the sign bit
  – Arithmetic shift is equivalent to logical shift for positive numbers
  – 0100 1000 $>>$ 2 = 0001 0010
Bit Manipulations: shift operators

• Right shift: \( x >> k \) : Shift bit-vector \( x \) right \( k \) positions.
  – Throw away extra bits on the right

TWO KINDS:

• Logical Shift: Fill with 0’s on the left.

• Arithmetic Shift : Replicate with most significant bit on the left.
  – Copies the sign bit
  – Arithmetic shift is equivalent to logical shift for positive numbers
  – \( 0100\ 1000 >> 2 = 0001\ 0010 \)
  – \( 1001\ 0001 >> 3 = 1111\ 0010 \)
  – \( x >> k \) corresponds to \( x/2^k \) for rounding.
    • \( 1001\ 0001 >> 3 \) in decimal: \((-111) / 2^3 = -13.875\)
    • \( 1111\ 0010 \) in decimal: -14
Bit Manipulations: shift operators

- Right shift: $x >> k$ : Shift bit-vector $x$ right $k$ positions.
  - Throw away extra bits on the right

**TWO KINDS:**

- Logical Shift: Fill with 0’s on the left.
- Arithmetic Shift : Replicate with most significant bit on the left.
  - Copies the sign bit
  - Arithmetic shift is equivalent to logical shift for positive numbers
  - $0100\ 1000 >> 2 = 0001\ 0010$
  - $1001\ 0001 >> 3 = 1111\ 0010$
  - $x >> k$ corresponds to $x/2^k$ for rounding.
    - $1001\ 0001 >> 3$ in decimal: $(-111) / 2^3 = -13.875$
    - $1111\ 0010$ in decimal: -14
Comparison with shifting in Java

Both use $\ll$ to shift left
Comparison with shifting in Java

Both use << to shift left

C:
• Has signed and unsigned integer

Java:
• Has only signed integer
Comparison with shifting in Java

Both use << to shift left

C:
• Has signed and unsigned integer
• Shift operator (>>) is implementation defined
• In our Virtual Machine, >> operates according to the type of the operand
  – When shifting an unsigned value, >> is a logical shift.
  – When shifting a signed value, >> is an arithmetic shift.

Java:
• Has only signed integer
• >> is arithmetic shift, >>> is logical shift
iClicker Question

Compute this arithmetic right shift:
1001 0001 >> 2

a) 1111 0010
b) 1110 0100
c) 1110 0101
d) 0010 0100
APPLICATION OF SHIFT OPERATION
Ranges of bits

Sometimes you will encounter situations where multiple smaller numbers are packed into a single larger one. Some reasons for this are:

• To save space in memory
• To fit a quantity into a single register
• Because some hardware was designed that way and you have to talk to it
• To save sending bits over a network or to/from a device (such as a disk)

This leads to requirements to extract ranges of bits in a number, and to update ranges of bits.
Extracting a range of bits: Method 1

- Number bits starting from 0 on the right:
  bit7 bit6 bit5 bit4 bit3 bit2 bit1 bit0
- Suppose you want bits 2 through 4
- Step 1: Isolate the bits using a mask:
  Bitwise And (&):
  \[ \begin{array}{cccccccc}
  b7 & b6 & b5 & b4 & b3 & b2 & b1 & b0 \\
  \end{array} \]
  With \[ \begin{array}{cccccccc}
  0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\
  \end{array} \]
  - The mask has 1 bits for the positions you want
    - And 0 elsewhere
- Step 2: Shift masked value right to get rid of unwanted 0 bits on the right.
  - In this case, \( \gg 2 \)
Extracting bits L through R

• Mask has L-R+1 bits that are 1
  • Can form by: \((1 \ll (L-R+1)) - 1\)
    • ... or by writing it out
  • For \(L == 4\) and \(R == 2\), we have \((1 \ll 3) - 1\)
    • In binary: \(1 \ll 3\) is 00001000
    • Subtract 1 and you get: 00000111
  • It is shifted left by R bits
    • In this case \(((1 \ll 3) - 1) \ll 2\)
    • In binary, shift 00000111 left by 2: 00011100
Whole sequence in C

```c
int mask = ((1 << 3) - 1) << 2;
// or: int mask = 0b00011100;
// (...b is a gcc extension to C)
int range =
  (unsigned)(((full & mask)) >> 2)
```
Whole sequence in C

```c
int mask = ((1 << 3) - 1) << 2;
// or: int mask = 0b00011100;
// (...b is a gcc extension to C)
int range =
    (unsigned)((full & mask)) >> 2
```

Why do we cast signed int to unsigned int?
Extracting a range of bits: Method 2

- Number bits starting from 0 on the right:
  bit7 bit6 bit5 bit4 bit3 bit2 bit1 bit0
- Suppose you want bits 2 through 4
- **Step 1**: Shift left, eliminating unwanted high bits:
  \[
  \begin{array}{cccccccc}
  b7 & b6 & b5 & b4 & b3 & b2 & b1 & b0 \\
  \end{array}
  \ll 3
  \begin{array}{cccccccc}
  b4 & b3 & b2 & b1 & b0 & 0 & 0 & 0 \\
  \end{array}
  
- **Step 2**: Shift right logical to get desired bits.
  \[
  \begin{array}{cccccccc}
  b4 & b3 & b2 & b1 & b0 & 0 & 0 & 0 \\
  \end{array}
  \ll 5
  \begin{array}{cccccccc}
  0 & 0 & 0 & 0 & 0 & b4 & b3 & b2 \\
  \end{array}
  
Desired field may be signed, or unsigned; cast to the right type *before* shifting right.
Extracting bits L through R

- Shift left by $n-(L+1)$ bits
- Shift right by $n-(L-R+1)$ bits
- $n$ is the number of bits in the data type (8, 16, 32, 64)
- For $L == 4$ and $R == 2$, with $n == 8$, we have $(x << 3) >> 5$
Extracting bits L through R

- Shift left by $n-(L+1)$ bits
- Shift right by $n-(L-R+1)$ bits
- $n$ is the number of bits in the data type (8, 16, 32, 64)
- For $L == 4$ and $R == 2$, with $n == 8$, we have $(x << 3) >> 5$

Both methods are equally good
Updating bits L through R of N with M

You can assume that the bits L through R of N have enough space to fit all of M.

For example, you are given a 11-bit number N=10000000000 and a 5-bit number M=10011, update N such that M starts at bit L=2 and ends at bit R=6.
Output:
N = 10001001100
**Updating** bits L through R of N with M

Steps:
- Clear the bits L through R in N
- Shift M so that it lines up with bits L through R
- merge M and N
**Updating** bits L through R of N with M

Steps:
- Clear the bits L through R in N
- Shift M so that it lines up with bits L through R
- merge M and N
Updating bits L through R of N with M

Steps:
• Clear the bits L through R in N
• Shift M so that it lines up with bits L through R
• merge M and N
• In C:

```c
int mask = ((1 << (L-R+1) - 1) << R);
int newN = N & (~mask);
int newM = M << R;
int result = newN | newM;
```
Updating bits L through R of N with M

Steps:

• Clear the bits L through R in N
• Shift M so that it lines up with bits L through R
• merge M and N
• In C:

```c
int mask = ((1 << (L-R+1) - 1) << R);
int newN = N & (~mask);
int newM = M << R;
int result = newN | newM;
```
Updating bits L through R of N with M

Steps:
• Clear the bits L through R in N
• Shift M so that it lines up with bits L through R
• merge M and N
• In C:

```c
int mask = ((1 << (L-R+1) - 1) << R);
int newN = N & (~mask);
int newM = M << R;
int result = newN | newM;
```
Updating bits L through R of N with M

- Let \( L = 4 \), \( R = 2 \), \( M = 0b101 \)
- Let \( N = 0b\underline{011} 011\underline{01} \)

```cpp
int mask = ((1 << (L-R+1)) - 1) << R);
// mask == 0b00011100
int newN = N & (~mask);
// newN == 0b01100001
int newM = M << R;
// M << R == 0b10100
int result = newM | newN;
// result == 0b01110101
```
Summary

- Bit representation and manipulation is extremely useful in a wide variety of applications like compiler analyses, network programming, cryptography and many more!
- The same binary sequence can be used to represent ASCII characters, unsigned binary, and two’s complement integers. Their interpretation is based on the context in which they are defined!
- C has different data types to store integers and floating point numbers that have different memory sizes on different operating systems.
- Typecasting operations between two different data types can be explicit or implicit.
  - Casting surprises when changing between data types can change the numeric value.
  - Casting surprises also occur if we use arithmetic and relational operators on two different data types.
- Boolean algebra includes {not, and, or and x-or} operations and left and right shifts.
  - Not to be confused with conditional operators!
- Using &, |, <<, and >> you can extract and replace ranges of bits using masks
- Next class we will cover more programming in C!