Bits, Bytes & Transistors
Bitwise operations
Integers

Instructor: Alan Beadle

Department of Computer Science
University of Rochester

Action Items:
Get CSUG account
Make sure you have VPN setup!!!!
Sign up for Blackboard, Piazza
Announcement

Make sure you can access CSUG machines!!!

Programming assignment 1 will be posted tonight.

C language. Seek help from TAs. Ask questions on Piazza forum so everyone can benefit.
Announcement

You may work on assignments in pairs

Can use Piazza to find someone to work with

You have 3 slip days
Previously in 252...

• How is a human-readable program translated to a representation that computers can understand?

• How does a modern computer execute that program?

ISA is the contract between software and hardware.
Previously in 252...

C Program

```c
void add() {
    int a = 1;
    int b = 2;
    int c = a + b;
}
```

Assembly program

```assembly
movl $1, -4(%rbp)
movl $2, -8(%rbp)
movl -4(%rbp), %eax
addl -8(%rbp), %eax
```
Previously in 252…

**Assembly program**

```
movl $1, -4(%rbp)
movl $2, -8(%rbp)
movl -4(%rbp), %eax
%eax
addl -8(%rbp), %eax
```

**Executable Binary**

```
00011001 ...
01101010 ...
11010101 ...
01110001 ...
```

What’s the difference between an assembly program and an executable binary?
Previously in 252...

**Assembly program**

```
movl $1, -4(%rbp)
movl $2, -8(%rbp)
movl -4(%rbp), %eax
%eax
addl -8(%rbp), %eax
```

**Executable Binary**

```
00011001 ...
01101010 ...
11010101 ...
01110001 ...
```

What’s the difference between an assembly program and an executable binary?

They refer to the same thing — a list of instructions that the software asks the hardware to perform

They are just different representations
Previously in 252…

Assembly program

movl  $1, -4(%rbp)
movl  $2, -8(%rbp)
movl  -4(%rbp),
%eax
addl  -8(%rbp),
%eax

Executable Binary

00011001    ...
01101010    ...
11010101    ...
01110001    ...

What’s the difference between an assembly program and an executable binary?

They refer to the same thing — a list of instructions that the software asks the hardware to perform

They are just different representations

Instruction = Operator + Operand(s)
Bits Bytes & Transistors
Everything is bits

Each bit is 0 or 1. Bits are how programs talk to the hardware
Programs encode instructions in bits
Hardware then interprets the bits
Why bits? Electronic Implementation

Software is bits, data is bits, everything is bits in memory
Why Bits?

Use high voltage to represent 1
Use low voltage to represent 0

“Noise” is less likely to cause errors with only two possibilities
Two-value logic gates are easy/cheap to build and combine
Why Bits?

Processors are made of transistors, which are Metal Oxide Semiconductor (MOS)

two types: n-type and p-type

n-type (NMOS)
when Gate has high voltage, short circuit between #1 and #2 (switch closed)
when Gate has low voltage, open circuit between #1 and #2 (switch open)

Terminal #2 must be connected to GND (0V).
Why Bits?

\textit{p-type} is complementary to \textit{n-type} (PMOS)
when Gate has \textit{high} voltage, open circuit between #1 and #2 (switch open)
when Gate has \textit{low} voltage, short circuit between #1 and #2 (switch closed)

Terminal #1 must be connected to +1.2V
CMOS Circuit

Complementary MOS

Uses both *n-type* and *p-type* MOS transistors

- **p-type**
  - Attached to + voltage
  - Pulls output voltage UP when input is zero

- **n-type**
  - Attached to GND
  - Pulls output voltage DOWN when input is one
CMOS Inverter

<table>
<thead>
<tr>
<th>In</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Two cross coupled inverters store a single bit
Feedback path persists the value in the “cell”
4 transistors for storage
2 transistors for access
A “6T” cell
Transistors

Computers are made of transistors
Transistors have become smaller over the years
Not so much anymore...
Basic Logic Gates

- **NOT**: \( \overline{A} \)
- **OR**: \( A \lor B \)
- **NOR**: \( \overline{A \lor B} \)
- **AND**: \( A \land B \)
- **NAND**: \( \overline{A \land B} \)
Binary Notation

Base 2 Number Representation (Binary)

Weighted Positional Notation

Each bit has a weight depending on its position

\[ 1011_2 = 1 \times 2^0 + 1 \times 2^1 + 0 \times 2^2 + 1 \times 2^3 = 11_{10} \]
\[ b_3 b_2 b_1 b_0 = b^0 \times 2^0 + b^1 \times 2^1 + b^2 \times 2^2 + b^3 \times 2^3 \]

Binary Arithmetic:

\[
\begin{array}{c}
0110 \\
+ \quad 0101 \\
\hline
1011
\end{array}
\]

\[
\begin{array}{c}
6 \\
+ \quad 5 \\
\hline
11
\end{array}
\]
Binary Notation

For example,

What is $00010111_2$ in decimal?

$00010111$
Binary Notation

For example,

What is $00010111_2$ in decimal?

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

$0 \quad 0 \quad 0 \quad 1 \quad 0 \quad 1 \quad 1 \quad 1$

128  64  32  16  8  4  2  1
Binary Notation

For example,

What is $00010111_2$ in decimal?

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\
128 & 64 & 32 & 16 & 8 & 4 & 2 & 1 \\
\end{array}
\]

\[16 + 4 + 2 + 1 = 23\]
Hexadecimal Notation

Base 16 Number Representation

- Use characters ‘0’ to ‘9’ and ‘A’ to ‘F’
- Four bits per Hex digit

\[ 11111110_2 = \text{FE}_{16} \]

Two hex digits = 8 bits = 1 byte

Write \( \text{FA1D37B}_{16} \) in C as:

- 0xFA1D37B
- 0xFA1D37B
Bytes and Words

Byte = 8 bits
- Binary: 00000000₂ to 11111111₂; Decimal: 0₁₀ to 255₁₀; Hex: 00₁₆ to FF₁₆
- Least Significant Bit (LSb) vs. Most Significant Bit (MSb)

A word is one or bytes together, depending on the “word size” of the machine

Word = 4 Bytes (32-bit machine) / 8 Bytes (64-bit machine)
- Least Significant Byte (LSB) vs. Most Significant Byte (MSB)
Bitwise Operations
Bit-level manipulations

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

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

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

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

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

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></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></th>
<th>0</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
NOR (OR + NOT)

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Bit Vector Operations

Operate on Bit Vectors
Operations applied bitwise

\[
\begin{array}{c}
\text{and} & 01101001 \\
& \underline{\phantom{01101001}} 01010101 \\
& \underline{01000001} \\
\end{array}
\quad \begin{array}{c}
\text{or} & 01101001 \\
& \underline{\phantom{01101001}} 01010101 \\
& \underline{11111101} \\
\end{array}
\]

\[
\begin{array}{c}
\text{xor} & 01101001 \\
& \underline{\phantom{01101001}} 01010101 \\
& \underline{00111100} \\
\end{array}
\quad \begin{array}{c}
\text{not} & 01010101 \\
& \underline{\phantom{01010101}} \underline{\phantom{01010101}} \\
& \underline{10101010} \\
\end{array}
\]
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)

- \( \sim 0x41 \rightarrow 0xBE \)
  - \( \sim 01000001_2 \rightarrow 1011110_2 \)
- \( \sim 0x00 \rightarrow 0xFF \)
  - \( \sim 00000000_2 \rightarrow 11111111_2 \)
- \( 0x69 \& 0x55 \rightarrow 0x41 \)
  - \( 01101001_2 \& 01010101_2 \rightarrow 01000001_2 \)
- \( 0x69 \mid 0x55 \rightarrow 0x7D \)
  - \( 01101001_2 \mid 01010101_2 \rightarrow 01111101_2 \)
Contrast: Logic Operations in C

These are not bitwise, so don’t confuse them
&&, ||, !
View 0 as “False”
Anything nonzero as “True”
Always return 0 or 1
Early termination (e.g., 0 && 1 && 1)

Examples (char data type)
!0x41 → 0x00
!0x00 → 0x01
!!0x41 → 0x01

0x69 && 0x55 → 0x01
0x69 || 0x55 → 0x01
Shift Operations

Left Shift: $x \ll y$
Shift bit-vector $x$ left $y$ positions
Throw away extra bits on left
Fill with 0’s on right

<table>
<thead>
<tr>
<th>Argument $x$</th>
<th>01100010</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\ll 3$</td>
<td>00010000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Argument $x$</th>
<th>10100010</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\ll 3$</td>
<td>00010000</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Argument ( x )</th>
<th>( 01100010 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( &lt;&lt; \ 3 )</td>
<td>( 00010000 )</td>
</tr>
<tr>
<td>Log. ( &gt;&gt; \ 2 )</td>
<td>( 00011000 )</td>
</tr>
</tbody>
</table>

Right Shift: \( x >> y \)
- Shift bit-vector \( x \) right \( y \) positions
- Throw away extra bits on right

<table>
<thead>
<tr>
<th>Argument ( x )</th>
<th>( 10100010 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( &lt;&lt; \ 3 )</td>
<td>( 00010000 )</td>
</tr>
<tr>
<td>Log. ( &gt;&gt; \ 2 )</td>
<td>( 00101000 )</td>
</tr>
</tbody>
</table>

Logical shift:
- Fill with 0’s on left
Shift Operations

Left Shift:  \( x \ll y \)
Shift bit-vector \( x \) left \( y \) positions
  Throw away extra bits on left
  Fill with 0’s on right

Right Shift:  \( x \gg y \)
Shift bit-vector \( x \) right \( y \) positions
  Throw away extra bits on right

Logical shift:
  Fill with 0’s on left

Arithmetic shift
  Replicate most significant bit on left

<table>
<thead>
<tr>
<th>Argument ( x )</th>
<th>01100010</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \ll 3 )</td>
<td>00010000</td>
</tr>
<tr>
<td>Log. ( \gg 2 )</td>
<td>00011000</td>
</tr>
<tr>
<td>Arith. ( \gg 2 )</td>
<td>00011000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Argument ( x )</th>
<th>10100010</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \ll 3 )</td>
<td>00010000</td>
</tr>
<tr>
<td>Log. ( \gg 2 )</td>
<td>00101000</td>
</tr>
<tr>
<td>Arith. ( \gg 2 )</td>
<td>11101000</td>
</tr>
</tbody>
</table>
Shift Operations

Left Shift: \( x \ll y \)
- Shift bit-vector \( x \) left \( y \) positions
- Throw away extra bits on left
- Fill with 0’s on right

Right Shift: \( x \gg y \)
- Shift bit-vector \( x \) right \( y \) positions
- Throw away extra bits on right

Logical shift:
- Fill with 0’s on left

Arithmetic shift
- Replicate most significant bit on left

Undefined Behavior
- Shift amount < 0 or \( \geq \) total amount of bits
Integers
Representing Numbers in Binary

Different types of number
- Integer (Negative and Non-negative)
- Fractions
- Irrationals
Encoding Negative Numbers

So far we have been discussing non-negative numbers: (called unsigned integers)

How about negative numbers?
Encoding Negative Numbers

**Solution 1:** Sign-magnitude

First bit represents sign; 0 for positive; 1 for negative

The rest represents magnitude
Sign-Magnitude Implications

- Bits have different semantics
- Two zeros...

<table>
<thead>
<tr>
<th>Signed Value</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
</tr>
<tr>
<td>2</td>
<td>010</td>
</tr>
<tr>
<td>3</td>
<td>011</td>
</tr>
<tr>
<td>-0</td>
<td>100</td>
</tr>
<tr>
<td>-1</td>
<td>101</td>
</tr>
<tr>
<td>-2</td>
<td>110</td>
</tr>
<tr>
<td>-3</td>
<td>111</td>
</tr>
</tbody>
</table>
Sign-Magnitude Implications

- Bits have different semantics
- Two zeros...
- Normal arithmetic doesn’t work which would make hardware design harder

<table>
<thead>
<tr>
<th>Signed Value</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
</tr>
<tr>
<td>2</td>
<td>010</td>
</tr>
<tr>
<td>3</td>
<td>011</td>
</tr>
<tr>
<td>-0</td>
<td>100</td>
</tr>
<tr>
<td>-1</td>
<td>101</td>
</tr>
<tr>
<td>-2</td>
<td>110</td>
</tr>
<tr>
<td>-3</td>
<td>111</td>
</tr>
</tbody>
</table>

```
010
+) 101

111
```

```
2
+) -1

-3
```
Encoding Negative Numbers

**Solution 2:** Two’s Complement
Instead of the msb simply being a sign bit, just interpret that place as a negative value

<table>
<thead>
<tr>
<th>Signed Weight</th>
<th>Unsigned Weight</th>
<th>Bit Position</th>
<th>Signed</th>
<th>Unsigned</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^0$</td>
<td>$2^0$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>000</td>
</tr>
<tr>
<td>$2^1$</td>
<td>$2^1$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>001</td>
</tr>
<tr>
<td>$-2^2$</td>
<td>$2^2$</td>
<td>2</td>
<td>-4</td>
<td>4</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-3</td>
<td>5</td>
<td>101</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-2</td>
<td>6</td>
<td>110</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-1</td>
<td>7</td>
<td>111</td>
</tr>
</tbody>
</table>

$101_2 = 1*2^0 + 0*2^1 - 1*2^2 = -3_{10}$
Two’s Complement

For example,

What is $10010111_2$?

\[
\begin{array}{ccccccc}
1 & 0 & 0 & 0 & 1 & 0 & 1 1 1
\end{array}
\]
Two’s Complement

For example,

What is $10010111_2$?

\[
\begin{array}{ccccccc}
1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\
-128 & 64 & 32 & 16 & 8 & 4 & 2 & 1 \\
\end{array}
\]

\[-128 + 16 + 4 + 2 + 1 = -105\]
Two’s Complement

10010111_2 = -105

It’s very easy to invert a two’s complement number
Two’s Complement

\[ 10010111_2 = -105 \]

It’s very easy to invert a two’s complement number

Just flip all the bits

and add 1:
Two’s Complement

\[10010111_2 = -105\]

It’s very easy to invert a two’s complement number:

Just flip all the bits:

\[01101000\]

and add 1:

\[01101001_2 = 105\]
Two’s Complement

10010111₂ = -105

It’s very easy to invert a two’s complement number

Just flip all the bits

01101000

and add 1:

01101001₂ = 105

To go back, do the same thing (really!):

10010111₂ = -105
## Two-Complement Encoding Example

### Example Values

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

### Table

<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>0</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>0</td>
</tr>
<tr>
<td>32</td>
<td>1</td>
<td>32</td>
</tr>
<tr>
<td>64</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>128</td>
<td>0</td>
<td>0</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>
</tbody>
</table>

### Calculation

\[ \text{Sum} = 15213 + (-15213) \]

\[ = 00111011 01101101 + 11000100 10010011 \]

\[ = 01111111 01111111 \]

\[ = 15325 \]
Two’s-Complement Implications

Only 1 zero!

Usual arithmetic still works!

There is a bit that tells us the sign!

Most widely used in today’s machines

<table>
<thead>
<tr>
<th>Signed</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
</tr>
<tr>
<td>2</td>
<td>010</td>
</tr>
<tr>
<td>3</td>
<td>011</td>
</tr>
<tr>
<td>-4</td>
<td>100</td>
</tr>
<tr>
<td>-3</td>
<td>101</td>
</tr>
<tr>
<td>-2</td>
<td>110</td>
</tr>
<tr>
<td>-1</td>
<td>111</td>
</tr>
</tbody>
</table>

\[
\begin{array}{c}
010 \\
+ \quad 101 \\
\hline
111
\end{array}
\quad \begin{array}{c}
2 \\
+ \quad -3 \\
\hline
-1
\end{array}
Numeric Ranges

Unsigned Values

\[ UMin = 0 \]
\[ 000...0 \]
\[ UMax = 2^w - 1 \]
\[ 111...1 \]

Two’s Complement Values

\[ TMin = -2^{w-1} \]
\[ 100...0 \]
\[ TMax = 2^{w-1} - 1 \]
\[ 011...1 \]

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>
Data Representations in C (in Bytes)

By default variables are signed
Unless explicitly declared as unsigned (e.g., unsigned int)
Signed variables use two-complement encoding

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>32-bit</th>
<th>64-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>
# Data Representations in C (in Bytes)

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>32-bit</th>
<th>64-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>

## C Language

```c
#include <limits.h>

Declares constants, e.g.,
ULONG_MAX
LONG_MAX
LONG_MIN

Values platform specific
```