Roadmap

C:
```c
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:
```java
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg = c.getMPG();
```

Assembly language:
```assembly
get_mpg:
  pushq  %rbp
  movq   %rsp, %rbp
  ...
  popq   %rbp
  ret
```

Machine code:
```
0111010000011000
100011010000010000000010
100010011110000010
1100000111111110100001111
```

Computer system:

Data & addressing
Integers & floats
Machine code & C
x86 assembly
programming
Procedures & stacks
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C

OS:

Windows 8
Mac

Wireframes:
Integers

- Representation of integers: unsigned and signed
- Casting
- Arithmetic and shifting
- Sign extension
But before we get to integers....

- Encode a standard deck of playing cards.
- 52 cards in 4 suits
  - How do we encode suits, face cards?
- What operations do we want to make easy to implement?
  - Which is the higher value card?
  - Are they the same suit?
Two possible representations

- 52 cards – 52 bits with bit corresponding to card set to 1

- “One-hot” encoding

- Drawbacks:
  - Hard to compare values and suits
  - Large number of bits required
Two possible representations

- 52 cards – 52 bits with bit corresponding to card set to 1

  low-order 52 bits of 64-bit word

  - “One-hot” encoding
  - Drawbacks:
    - Hard to compare values and suits
    - Large number of bits required

- 4 bits for suit, 13 bits for card value – 17 bits with two set to 1

  - Pair of one-hot encoded values
  - Easier to compare suits and values
    - Still an excessive number of bits

Can we do better?
Two better representations

- Binary encoding of all 52 cards – only 6 bits needed
  - Fits in one byte
  - Smaller than one-hot encodings.
  - How can we make value and suit comparisons easier?

low-order 6 bits of a byte
Two better representations

- Binary encoding of all 52 cards – only 6 bits needed
  
  - Fits in one byte
  - Smaller than one-hot encodings.
  - How can we make value and suit comparisons easier?

- Binary encoding of suit (2 bits) and value (4 bits) separately
  
  - Also fits in one byte, and easy to do comparisons
Compare Card Suits

```c
#define SUIT_MASK  0x30

int sameSuitP(char card1, char card2) {
    return (! (card1 & SUIT_MASK) ^ (card2 & SUIT_MASK));
    //return (card1 & SUIT_MASK) == (card2 & SUIT_MASK);
}
```

- `SUITE_MASK = 0x30` is equivalent to `00110000`.

- `char hand[5];` represents a 5-card hand.
- `char card1, card2;` are two cards to compare.
- `card1 = hand[0];`
- `card2 = hand[1];`
- `if ( sameSuitP(card1, card2) ) { ... }`

**mask:** a bit vector that, when bitwise ANDed with another bit vector \( v \), turns all but the bits of interest in \( v \) to 0.
Compare Card Values

```c
#define VALUE_MASK 0x0F

int greaterValue(char card1, char card2) {
    return ((unsigned int)(card1 & VALUE_MASK) >
            (unsigned int)(card2 & VALUE_MASK));
}
```

VALUE_MASK = 0x0F = 0 0 0 0 1 1 1 1

- mask: a bit vector that, when bitwise ANDed with another bit vector \( v \), turns all but the bits of interest in \( v \) to 0.

- works even if value is stored in high bits

```c
char hand[5]; // represents a 5-card hand
char card1, card2; // two cards to compare
card1 = hand[0];
card2 = hand[1];
...
if ( greaterValue(card1, card2) ) { ... }
```
Announcements

- Everyone who’s registered turned in Lab 0, did well, and got credit!
- Let’s make sure everyone who thinks they’re registered is.
  - Auditors welcome!
- Section meeting Friday, July 5, 10:50am – 11:50am in CSE 303.
  - Right after lecture.
- Lab 1: trouble compiling with make? Jacob made a very small fix over the weekend.
Encoding Integers

- The hardware (and C) supports two flavors of integers:
  - *unsigned* – only the non-negatives
  - *signed* – both negatives and non-negatives

- There are only $2^W$ distinct bit patterns of $W$ bits, so...
  - Can not represent all the integers
  - **Unsigned values:** $0$ ... $2^{W-1}$
  - **Signed values:** $-2^{W-1}$ ... $2^{W-1}-1$

- Reminder: terminology for binary representations
  - “Most-significant” or “high-order” bit(s)
  - “Least-significant” or “low-order” bit(s)
  
  \[0110010110101001\]
Unsigned Integers

- Unsigned values are just what you expect
  - \[ b_7b_6b_5b_4b_3b_2b_1b_0 = b_72^7 + b_62^6 + b_52^5 + \ldots + b_12^1 + b_02^0 \]
  - Useful formula: \[ 1+2+4+8+\ldots+2^{N-1} = 2^N - 1 \]

- Add and subtract using the normal “carry” and “borrow” rules, just in binary.

- How would you make signed integers?
Signed Integers: Sign-and-Magnitude

- Let's do the natural thing for the positives
  - They correspond to the unsigned integers of the same value
    - Example (8 bits): 0x00 = 0, 0x01 = 1, ..., 0x7F = 127

- But, we need to let about half of them be negative
  - Use the **high-order bit** to indicate *negative*: call it the “**sign bit**”
    - Call this a “sign-and-magnitude” representation
  - Examples (8 bits):
    - 0x00 = 00000000₂ is non-negative, because the sign bit is 0
    - 0x7F = 01111111₂ is non-negative
    - 0x85 = 10000101₂ is negative
    - 0x80 = 10000000₂ is negative...
Signed Integers: Sign-and-Magnitude

- How should we represent -1 in binary?
  - $10000001_2$
  - Use the MSB for + or -, and the other bits to give magnitude.
Sign-and-Magnitude Negatives

- How should we represent -1 in binary?
  - $10000001_2$
    - Use the MSB for + or -, and the other bits to give magnitude.
    - (Unfortunate side effect: there are two representations of 0!)

![Diagram showing binary representations of numbers from -7 to +7]
Sign-and-Magnitude Negatives

How should we represent -1 in binary?

- \(10000001_2\)
  Use the MSB for + or -, and the other bits to give magnitude.
  (Unfortunate side effect: there are two representations of 0!)

- Another problem: \textit{arithmetic is cumbersome}.
  
  Example:
  \[4 - 3 \neq 4 + (-3)\]

How do we solve these problems?
Two’s Complement Negatives

- How should we represent -1 in binary?
Two’s Complement Negatives

How should we represent -1 in binary?

Rather than a sign bit, let MSB have same value, but negative weight.

\( b_{w-1} = 1 \) adds \(-2^{w-1}\) to the value.  for \( i < w-1 \): \( b_i = 1 \) adds \(+2^i\) to the value.
Two’s Complement Negatives

How should we represent -1 in binary?

Rather than a sign bit, let MSB have same value, but *negative weight*.

\[ b_{w-1} = 1 \text{ adds } -2^{w-1} \text{ to the value.} \]

\[ \text{for } i < w-1: \ b_i = 1 \text{ adds } +2^i \text{ to the value.} \]

\[ b_{w-1} b_{w-2} \ldots b_0 \]

e.g. *unsigned* $1010_2$:

\[ 1*2^3 + 0*2^2 + 1*2^1 + 0*2^0 = 10_{10} \]

2’s compl. $1010_2$:

\[ -1*2^3 + 0*2^2 + 1*2^1 + 0*2^0 = -6_{10} \]
Two’s Complement Negatives

- How should we represent -1 in binary?
  Rather than a sign bit, let MSB have same value, but \textit{negative weight}.
  \[ b_{w-1} = 1 \text{ adds } -2^{w-1} \text{ to the value.} \]
  \[ \text{for } i < w-1: \ b_i = 1 \text{ adds } +2^i \text{ to the value.} \]

- e.g. \textit{unsigned} 1010_2:
  \[ 1*2^3 + 0*2^2 + 1*2^1 + 0*2^0 = 10_{10} \]
  \textit{2’s compl.} 1010_2:
  \[ -1*2^3 + 0*2^2 + 1*2^1 + 0*2^0 = -6_{10} \]

- -1 is represented as 1111_2 = -2^3 + (2^3 - 1)
  All negative integers still have MSB = 1.

- \textbf{Advantages:} single zero, simple arithmetic

- \textbf{To get negative representation of any integer, take bitwise complement and then add one!}
  \[ \sim x + 1 == -x \]
Two’s Complement Arithmetic

- The same addition procedure works for both unsigned and two’s complement integers
  - Simplifies hardware: only one algorithm for addition
  - Algorithm: simple addition, discard the highest carry bit
    - Called “modular” addition: result is sum \( \text{modulo } 2^w \)

- Examples:

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0100</td>
<td>+ 3</td>
<td>4</td>
<td>0100</td>
<td>− 3</td>
<td>+ 1101</td>
</tr>
<tr>
<td></td>
<td></td>
<td>+ 0011</td>
<td></td>
<td></td>
<td>− 3</td>
<td>+ 0011</td>
</tr>
<tr>
<td>= 7</td>
<td>= 0111</td>
<td>= 1</td>
<td>drop carry</td>
<td>= 0001</td>
<td>= 1</td>
<td>= 1111</td>
</tr>
</tbody>
</table>

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>− 4</td>
<td>1100</td>
<td>+ 3</td>
<td>− 4</td>
<td>1100</td>
<td>+ 3</td>
<td>+ 0011</td>
</tr>
<tr>
<td></td>
<td></td>
<td>+ 0011</td>
<td></td>
<td></td>
<td>− 3</td>
<td>+ 0011</td>
</tr>
<tr>
<td>= 7</td>
<td>= 0111</td>
<td>= 1</td>
<td>drop carry</td>
<td>= 0001</td>
<td>= 1</td>
<td>= 1111</td>
</tr>
</tbody>
</table>
Two’s Complement

Why does it work?

- Put another way, for all positive integers $x$, we want:
  - $\text{bits}(x) + \text{bits}(−x) = 0$ (ignoring the carry-out bit)

- This turns out to be the \textit{bitwise complement plus one}
  - What should the 8-bit representation of -1 be?
    - $00000001$
    - $+????????$ (we want whichever bit string gives the right result)
    - $00000000$
    - $00000000$
    - $+????????$ $+????????$
    - $00000000$ $00000000$
Two’s Complement

- Why does it work?
  - Put another way, for all positive integers $x$, we want:
    - $\text{bits}(x) + \text{bits}(−x) = 0$ (ignoring the carry-out bit)
  - This turns out to be the bitwise complement plus one
    - What should the 8-bit representation of $-1$ be?
      00000001
      +11111111
      100000000

      00000010
      +????????
      00000000

      00000011
      +????????
      00000000
Two’s Complement

Why does it work?

- Put another way, for all positive integers $x$, we want:
  - $\text{bits}(x) + \text{bits}(-x) = 0$ (ignoring the carry-out bit)

- This turns out to be the *bitwise complement plus one*
  - What should the 8-bit representation of -1 be?
    - $00000001 + 11111111 = 100000000$ (we want whichever bit string gives the right result)
    - $00000010 + 11111110 = 100000000$
    - $00000011 + 11111101 = 100000000$
Unsigned & Signed Numeric Values

- Signed and unsigned integers have limits.
  - If you compute a number that is too big (positive), it wraps:
    \[ 6 + 4 = ? \quad 15U + 2U = ? \]
  - If you compute a number that is too small (negative), it wraps:
    \[ -7 - 3 = ? \quad 0U - 2U = ? \]
  - Answers are only correct mod \(2^b\)

- The CPU may be capable of “throwing an exception” for overflow on signed values.
  - It won't for unsigned.

- But C and Java just cruise along silently when overflow occurs... Oops.
Conversion Visualized

- **Two’s Complement → Unsigned**
  - Ordering Inversion
  - Negative → Big Positive

Diagram showing 2’s Complement Range and Unsigned Range with corresponding values like $T_{Max}$, $T_{Min}$, $U_{Max}$, $U_{Max} - 1$, $T_{Max} + 1$, and $T_{Max}$.
Values To Remember

- **Unsigned Values**
  - $\text{UMin} = 0$
    - $000...0$
  - $\text{UMax} = 2^w - 1$
    - $111...1$

- **Two’s Complement Values**
  - $\text{TMin} = -2^{w-1}$
    - $100...0$
  - $\text{TMax} = 2^{w-1} - 1$
    - $011...1$
  - Negative one
    - $111...1$  $0xF...F$

Values for $W = 32$

<table>
<thead>
<tr>
<th></th>
<th>Decimals</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>UMax</td>
<td>$4,294,967,296$</td>
<td>FF FF FF FF</td>
<td>$11111111 11111111 11111111 11111111$</td>
</tr>
<tr>
<td>Tmax</td>
<td>$2,147,483,647$</td>
<td>7F FF FF FF</td>
<td>$01111111 11111111 11111111 11111111$</td>
</tr>
<tr>
<td>TMin</td>
<td>$-2,147,483,648$</td>
<td>80 00 00 00</td>
<td>$10000000 00000000 00000000 00000000$</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>FF FF FF FF</td>
<td>$11111111 11111111 11111111 11111111$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 00 00 00</td>
<td>$00000000 00000000 00000000 00000000$</td>
</tr>
</tbody>
</table>
Signed vs. Unsigned in C

- **Constants**
  - By default are considered to be signed integers
  - Use “U” suffix to force unsigned:
    - \(0U, 4294967259U\)
Signed vs. Unsigned in C

Casting

- int tx, ty;
- unsigned ux, uy;

- Explicit casting between signed & unsigned:
  - tx = (int) ux;
  - uy = (unsigned) ty;

- Implicit casting also occurs via assignments and function calls:
  - tx = ux;
  - uy = ty;

- The gcc flag -Wsign-conversion produces warnings for implicit casts, but -Wall does not!

- How does casting between signed and unsigned work?
- What values are going to be produced?
Signed vs. Unsigned in C

- Casting
  - `int tx, ty;`
  - `unsigned ux, uy;`
  - Explicit casting between signed & unsigned:
    - `tx = (int) ux;`
    - `uy = (unsigned) ty;`
  - Implicit casting also occurs via assignments and function calls:
    - `tx = ux;`
    - `uy = ty;`
  - The gcc flag `-Wsign-conversion` produces warnings for implicit casts, but `-Wall` does not!

- How does casting between signed and unsigned work?
- What values are going to be produced?
  - *Bits are unchanged*, just interpreted differently!
Casting Surprises

Expression Evaluation

- If you mix unsigned and signed in a single expression, then
  *signed values are implicitly cast to unsigned.*

- Including comparison operations <, >, ==, <=, >=

- Examples for $W = 32$: $T_{MIN} = -2,147,483,648$ $T_{MAX} = 2,147,483,647$

<table>
<thead>
<tr>
<th>Constant_1</th>
<th>Constant_2</th>
<th>Relation</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0U</td>
<td>==</td>
<td>unsigned</td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
<td>&lt;</td>
<td>signed</td>
</tr>
<tr>
<td>-1</td>
<td>0U</td>
<td>&gt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>-2147483648</td>
<td>&gt;</td>
<td>signed</td>
</tr>
<tr>
<td>2147483647U</td>
<td>-2147483648</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>2147483648U</td>
<td>&lt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>(int) 2147483648U</td>
<td>&gt;</td>
<td>signed</td>
</tr>
</tbody>
</table>
Sign Extension

- What happens if you convert a 32-bit signed integer to a 64-bit signed integer?
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:
  - $X' = x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_0$

![Diagram showing sign extension process](diagram.png)
Sign Extension Example

- Converting from smaller to larger integer data type
- C automatically performs sign extension. (Java too)

```java
short int x = 12345;
int ix = (int) x;
short int y = -12345;
int iy = (int) y;
```

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>12345</td>
<td>30 39</td>
<td>00110000 01101101</td>
</tr>
<tr>
<td>ix</td>
<td>12345</td>
<td>00 00 30 39</td>
<td>00000000 00000000 00110000 01101101</td>
</tr>
<tr>
<td>y</td>
<td>-12345</td>
<td>CF C7</td>
<td>11001111 11000111</td>
</tr>
<tr>
<td>iy</td>
<td>-12345</td>
<td>FF FF CF C7</td>
<td>11111111 11111111 11001111 11000111</td>
</tr>
</tbody>
</table>
Shift Operations

- **Left shift:** \( x << y \)
  - Shift bit vector \( x \) left by \( y \) positions
    - Throw away extra bits on left
    - Fill with 0s on right

- **Right shift:** \( x >> y \)
  - Shift bit-vector \( x \) right by \( y \) positions
    - Throw away extra bits on right
  - Logical shift (for unsigned values)
    - Fill with 0s on left
  - Arithmetic shift (for signed values)
    - Replicate most significant bit on left
    - Maintains sign of \( x \)

<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>Logical &gt;&gt; 2</td>
<td>00011000</td>
</tr>
<tr>
<td>Arithmetic &gt;&gt; 2</td>
<td>00011000</td>
</tr>
</tbody>
</table>

<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>Logical &gt;&gt; 2</td>
<td>00101000</td>
</tr>
<tr>
<td>Arithmetic &gt;&gt; 2</td>
<td>11101000</td>
</tr>
</tbody>
</table>

The behavior of \( >> \) in C depends on the compiler! It is *arithmetic* shift right in GCC. Java: \( >>> \) is logical shift right; \( >> \) is arithmetic shift right.
Shift Operations

- **Left shift:**  \( x << y \)
  - Shift bit vector \( x \) left by \( y \) positions
    - Throw away extra bits on left
    - Fill with 0s on right

- **Right shift:**  \( x >> y \)
  - Shift bit-vector \( x \) right by \( y \) positions
    - Throw away extra bits on right
  - Logical shift (for unsigned values)
    - Fill with 0s on left
  - **Arithmetic shift** (for signed values)
    - Replicate most significant bit on left
    - Maintains sign of \( x \)
    - *Why is this useful?*

<table>
<thead>
<tr>
<th>Argument x</th>
<th>01100010</th>
</tr>
</thead>
<tbody>
<tr>
<td>( &lt;&lt; 3 )</td>
<td></td>
</tr>
<tr>
<td>Logical ( &gt;&gt; 2 )</td>
<td></td>
</tr>
<tr>
<td>Arithmetic ( &gt;&gt; 2 )</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Argument x</th>
<th>10100010</th>
</tr>
</thead>
<tbody>
<tr>
<td>( &lt;&lt; 3 )</td>
<td></td>
</tr>
<tr>
<td>Logical ( &gt;&gt; 2 )</td>
<td></td>
</tr>
<tr>
<td>Arithmetic ( &gt;&gt; 2 )</td>
<td></td>
</tr>
</tbody>
</table>

\( x >> 9? \)

The behavior of \( >> \) in C depends on the compiler! It is *arithmetic* shift right in GCC. Java: \( >>> \) is logical shift right; \( >> \) is arithmetic shift right.
What happens when...

- $x >> n$
- $x << m$
What happens when...

- x >> n: divide by $2^n$

- x << m: multiply by $2^m$

faster than general multiple or divide operations
Using Shifts and Masks

- Extract the 2nd most significant byte of an integer?

```
  x  01100001 01100010 01100011 01100100
```
Using Shifts and Masks

- Extract the 2nd most significant byte of an integer:
  - First shift, then mask: \(( x \gg 16 ) \& 0xFF\)

<table>
<thead>
<tr>
<th>x</th>
<th>01100001 01100010 01100011 01100100</th>
</tr>
</thead>
<tbody>
<tr>
<td>x &gt;&gt; 16</td>
<td>00000000 00000000 01100001 01100010</td>
</tr>
<tr>
<td>( x &gt;&gt; 16 ) &amp; 0xFF</td>
<td>00000000 00000000 00000000 11111111</td>
</tr>
<tr>
<td></td>
<td>00000000 00000000 00000000 01100010</td>
</tr>
</tbody>
</table>

- Extract the sign bit of a signed integer?
Using Shifts and Masks

- **Extract the 2nd most significant byte of an integer:**
  - First shift, then mask: \((x \gg 16) \& 0xFF\)

<table>
<thead>
<tr>
<th>(x)</th>
<th>01100001 01100010 01100011 01100100</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x \gg 16)</td>
<td>00000000 00000000 01100001 01100010</td>
</tr>
<tr>
<td>((x \gg 16) &amp; 0xFF)</td>
<td>00000000 00000000 00000000 11111111 00000000 00000000 01100010</td>
</tr>
</tbody>
</table>

- **Extract the sign bit of a signed integer:**
  - \((x \gg 31) \& 1\) - need the “& 1” to clear out all other bits except LSB

- **Conditionals as Boolean expressions** *(assuming \(x\) is 0 or 1)*
  - if \((x)\) \(a=y\) else \(a=z\); which is the same as \(a = x \? y : z\);
  - Can be re-written (assuming arithmetic right shift) as:
    \(a = (x << 31) \gg 31 \& y + (\neg x) << 31 \gg 31 \) & \(z\);
Multiplication

- What do you get when you multiply $9 \times 9$?

- What about $2^{30} \times 3$?

- $2^{30} \times 5$?

- $-2^{31} \times -2^{31}$?
Unsigned Multiplication in C

Operands: \( w \) bits

True Product: \( 2^w \) bits

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

- **Standard Multiplication Function**
  - Ignores high order \( w \) bits

- **Implements Modular Arithmetic**
  \[ \text{UMult}_w(u, v) = u \cdot v \mod 2^w \]
Power-of-2 Multiply with Shift

**Operation**

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

Operands: \( w \) bits

<table>
<thead>
<tr>
<th>u ( \times 2^k )</th>
<th>0 ( \cdots )</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>( \cdots )</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
</table>

True Product: \( w+k \) bits

<table>
<thead>
<tr>
<th>u ( \cdot 2^k )</th>
<th>0 ( \cdots )</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>( \cdots )</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
</table>

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

UMult\(_w\)(\( u \), \( 2^k \))
TMult\(_w\)(\( u \), \( 2^k \))

**Examples**

- \( u \ll 3 \) \( \equiv \) \( u \times 8 \)
- \( u \ll 5 - u \ll 3 \) \( \equiv \) \( u \times 24 \)
- Most machines shift and add faster than multiply
  - Compiler generates this code automatically
# Code Security Example

/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void* user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}

#define MSIZE 528

void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, MSIZE);
    printf("%s\n", mybuf);
}
Malicious Usage

/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void* user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}

#define MSIZE 528

void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, -MSIZE);
    ...
Floats! Later in the quarter...

- How do we represent fractional numbers?
- If you’re curious now, read the book, check out the videos.
- We’ll return to this topic later if we have time.