Integers II
CSE 351 Autumn 2019

Instructor: Justin Hsia

Teaching Assistants: Andrew Hu
Diya Joy
Antonio Castelli
Maurice Montag
Ivy Yu
Suraj Jagadeesh
Melissa Birchfield

http://xkcd.com/1953/
Administrivia

- hw4 due 10/7, hw5 due 10/9
- Lab 1a due Monday (10/7)
  - Submit `pointer.c` and `lab1Areflect.txt` to Gradescope
- Lab 1b released today, due 10/14
  - Bit puzzles on number representation
  - Can start after today’s lecture, but floating point will be introduced next week
  - Section worksheet from yesterday has helpful examples, too
  - Bonus slides at the end of today’s lecture have relevant examples
Extra Credit

- All labs starting with Lab 1b have extra credit portions
  - These are meant to be fun extensions to the labs

- Extra credit points *don't* affect your lab grades
  - From the course policies: “they will be accumulated over the course and will be used to bump up borderline grades at the end of the quarter.”
  - Make sure you finish the rest of the lab before attempting any extra credit
Integers

- Binary representation of integers
  - Unsigned and signed
  - Casting in C

- Consequences of finite width representations
  - Overflow, sign extension

- Shifting and arithmetic operations
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 \( \text{mod} \ 2^w \)

4-bit Examples:

<table>
<thead>
<tr>
<th>HW</th>
<th>TC</th>
<th>HW</th>
<th>TC</th>
<th>HW</th>
<th>TC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100</td>
<td>+4</td>
<td>1100</td>
<td>-4</td>
<td>0100</td>
<td>+4</td>
</tr>
<tr>
<td>+0011</td>
<td>+3</td>
<td>+0011</td>
<td>+3</td>
<td>+1101</td>
<td>-3</td>
</tr>
<tr>
<td>= 0111</td>
<td>+7</td>
<td>= 1111</td>
<td>-1</td>
<td>= 0001</td>
<td>+1</td>
</tr>
</tbody>
</table>
Why Does Two’s Complement Work?

- For all representable positive integers $x$, we want:
  \[
  \text{additive inverse of } x + \text{bit representation of } -x + 0\text{ (ignoring the carry-out bit)}
  \]

  What are the 8-bit negative encodings for the following?

  \[
  \begin{align*}
  00000001 + ???? & = 00000000 \\
  00000000 + ???? & = 00000000 \\
  11000011 + ???? & = 00000000
  \end{align*}
  \]
Why Does Two’s Complement Work?

- For all representable positive integers $x$, we want:

$$x + (\sim x) = 0 \pmod{2^n}$$

- What are the 8-bit negative encodings for the following?

<table>
<thead>
<tr>
<th>Bit Representation of $x$</th>
<th>Bit Representation of $-x$</th>
<th>Result of $x + (\sim x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000001</td>
<td>11111111</td>
<td>00000000</td>
</tr>
<tr>
<td>+ 11111111</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000000000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$-x = \sim x + 1$

These are the bitwise complement plus 1!
Signed/Unsigned Conversion Visualized

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

Two’s Complement Range:
- $2^{w-1} - 1 = 0b01...1$
  - $T_{Max}$

0/Unsigned Range:
- $0b10...0 = 2^{w-1}$
  - $0b10...0 = 2^{w-1}$

Signed/Unsigned Conversion:
- $\overline{U_{Max}} = 0b1...1 = 2^{w-1}$
  - $U_{Max} - 1$
- $T_{Max} + 1$
  - $T_{Max}$
- $0/\overline{U_{Min}}$
- $0b10...0 = 2^{w-1}$

- $-2^{w-1} = 0b10...0 = T_{Min}$
  - $T_{Min}$
Values To Remember

- **Unsigned Values**
  - UMin = 0b00...0
    = 0
  - UMax = 0b11...1
    = $2^w - 1$

- **Two’s Complement Values**
  - TMin = 0b10...0
    = $-2^{w-1}$
  - Tmax = 0b01...1
    = $2^{w-1} - 1$
  - -1 = 0b11...1

**Example:** Values for $w = 64$

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>UMax</td>
<td>18,446,744,073,709,551,615</td>
<td>FF FF FF FF FF FF FF FF FF</td>
</tr>
<tr>
<td>Tmax</td>
<td>9,223,372,036,854,775,807</td>
<td>7F FF FF FF FF FF FF FF FF</td>
</tr>
<tr>
<td>Tmin</td>
<td>-9,223,372,036,854,775,808</td>
<td>80 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>FF FF FF FF FF FF FF FF FF</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 00 00 00 00 00 00 00 00</td>
</tr>
</tbody>
</table>
In C: Signed vs. Unsigned

Casting

- Bits are unchanged, just interpreted differently!
  - \texttt{int} tx, ty;
  - \texttt{unsigned int} ux, uy;

Explicit casting
  - \texttt{tx} = (\texttt{int}) ux;
  - \texttt{uy} = (\texttt{unsigned int}) ty;

Implicit casting can occur during assignments or function calls
  - \texttt{tx} = \texttt{ux};
  - \texttt{uy} = \texttt{ty};

(new-type) expression

\textit{cast to target variable/parameter type}

(also implicitly occurs with printf format specifiers)
Casting Surprises

- Integer literals (constants)
  - By default, integer constants are considered *signed* integers
    - Hex constants already have an explicit binary representation
  - Use “U” (or “u”) suffix to explicitly force *unsigned*
    - Examples: 0U, 4294967259u

- Expression Evaluation
  - When you mixed unsigned and signed in a single expression, then *signed values are implicitly cast to unsigned*
  - Including comparison operators <, >, ==, <=, >=
Casting Surprises

- 32-bit examples:
  - TMin = -2,147,483,648, TMax = 2,147,483,647

<table>
<thead>
<tr>
<th>Left Constant</th>
<th>Order</th>
<th>Right Constant</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>&lt;=</td>
<td>0U</td>
<td>unsigned</td>
</tr>
<tr>
<td>0000 0000 0000 0000 0000 0000 0000 0000</td>
<td></td>
<td>0000 0000 0000 0000 0000 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>&lt;</td>
<td>0</td>
<td>signed</td>
</tr>
<tr>
<td>1111 1111 1111 1111 1111 1111 1111 1111</td>
<td></td>
<td>0000 0000 0000 0000 0000 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>&gt;</td>
<td>0U</td>
<td>unsigned</td>
</tr>
<tr>
<td>1111 1111 1111 1111 1111 1111 1111 1111</td>
<td></td>
<td>0000 0000 0000 0000 0000 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td>2147483647</td>
<td>&gt;</td>
<td>-2147483648</td>
<td>signed</td>
</tr>
<tr>
<td>0111 1111 1111 1111 1111 1111 1111 1111</td>
<td></td>
<td>1000 0000 0000 0000 0000 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td>2147483647U</td>
<td>&lt;</td>
<td>-2147483648</td>
<td>unsigned</td>
</tr>
<tr>
<td>0111 1111 1111 1111 1111 1111 1111 1111</td>
<td></td>
<td>1000 0000 0000 0000 0000 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>&gt;</td>
<td>-2</td>
<td>signed</td>
</tr>
<tr>
<td>1111 1111 1111 1111 1111 1111 1111 1111</td>
<td></td>
<td>1111 1111 1111 1111 1111 1111 1111 1111</td>
<td></td>
</tr>
<tr>
<td>(unsigned) -1</td>
<td>&gt;</td>
<td>-2</td>
<td>unsigned</td>
</tr>
<tr>
<td>1111 1111 1111 1111 1111 1111 1111 1111</td>
<td></td>
<td>1111 1111 1111 1111 1111 1111 1111 1111</td>
<td></td>
</tr>
<tr>
<td>2147483647</td>
<td>&lt;</td>
<td>2147483648U</td>
<td>unsigned</td>
</tr>
<tr>
<td>0111 1111 1111 1111 1111 1111 1111 1111</td>
<td></td>
<td>1000 0000 0000 0000 0000 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td>2147483647</td>
<td>&gt;</td>
<td>(int) 2147483648U</td>
<td>signed</td>
</tr>
<tr>
<td>0111 1111 1111 1111 1111 1111 1111 1111</td>
<td></td>
<td>1000 0000 0000 0000 0000 0000 0000 0000</td>
<td></td>
</tr>
</tbody>
</table>
Integers

- Binary representation of integers
  - Unsigned and signed
  - Casting in C
- Consequences of finite width representations
  - Overflow, sign extension
- Shifting and arithmetic operations
## Arithmetic Overflow

- When a calculation produces a result that can’t be represented in the current encoding scheme
  - Integer range limited by fixed width
  - Can occur in both the positive and negative directions

- C and Java ignore overflow exceptions
  - You end up with a bad value in your program and no warning/indication... oops!

<table>
<thead>
<tr>
<th>Bits</th>
<th>Unsigned</th>
<th>Signed</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0(^{\text{U Min}})</td>
<td>0(^{\text{Min}})</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(^{\text{T Max}})</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
<td>-8(^{\text{T Min}})</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(^{\text{U Max}})</td>
<td>-1</td>
</tr>
</tbody>
</table>
Overflow: Unsigned

- **Addition**: drop carry bit \((-2^N)\)

\[
\begin{array}{rcl}
15 & + & 2 \\
\hline
17 & + & 0010 \\
\hline
10001 \\
\end{array}
\]

- **Subtraction**: borrow \((+2^N)\)

\[
\begin{array}{rcl}
1 & - & 2 \\
\hline
-1 & - & 0010 \\
\hline
1111 \\
\end{array}
\]

\[\pm 2^N \text{ because of modular arithmetic}\]

\[2^4 = 16\]
Overflow: Two’s Complement

- **Addition**: \((+) + (+) = (−)\) result?

  \[
  \begin{array}{c}
  6 \\
  + 3 \\
  \hline
  9 \\
  \end{array}
  \quad
  \begin{array}{c}
  0110 \\
  + 0011 \\
  \hline
  1001 \\
  \end{array}
  \quad
  \begin{array}{c}
  \text{OK} \\
  \end{array}
  \]

  
  \[
  \begin{array}{c}
  -7 \\
  -3 \\
  \hline
  -10 \\
  \end{array}
  \quad
  \begin{array}{c}
  -7 \\
  + 0011 \\
  \hline
  0110 \\
  \end{array}
  \quad
  \begin{array}{c}
  6 \\
  \end{array}
  \]

- **Subtraction**: \((-) + (−) = (+)\)?

  \[
  \begin{array}{c}
  -7 \\
  -3 \\
  \hline
  -10 \\
  \end{array}
  \quad
  \begin{array}{c}
  1001 \\
  - 0011 \\
  \hline
  0110 \\
  \end{array}
  \quad
  \begin{array}{c}
  \text{For signed: overflow if operands have same sign and result’s sign is different} \\
  \end{array}
  \]
Sign Extension

- What happens if you convert a signed integral data type to a larger one?
  - e.g. char \rightarrow short \rightarrow int \rightarrow long

- **4-bit \rightarrow 8-bit Example:**
  - Positive Case
    - Add 0’s?
    - 4-bit: 0010 = +2
    - 8-bit: 00000010 = +2
  - Negative Case?
Polling Question

Which of the following 8-bit numbers has the same *signed* value as the 4-bit number 0b1100?

- Underlined digit = MSB

A. 0b 0000 1100 (add zeros)  
B. 0b 1000 1100 (add leading 1)  
C. 0b 1111 1100 (add ones)  
D. 0b 1100 1100 (duplicate)  
E. We’re lost...

\[-8^{4} 2^{3} 2^1 \]
\[-8 + 4 = -4\]
\[-x = 0 0 1 1 + 1\]
\[0 1 0 0 = 4 \Rightarrow x = -4\]

\[\text{positive!}\]
\[\text{much too negative: } -2^7 + 2^3 + 2^2 = -116\]
\[\text{correct! } -y = 0b 0000 0011 + 1 = 4, \ y = -4\]
\[-2^7 + 2^6 + 2^3 + 2^2 = -52\]
Sign Extension

- **Task:** Given a $w$-bit signed integer $X$, convert it to $w+k$-bit signed integer $X'$ *with the same value*

- **Rule:** Add $k$ copies of sign bit
  - Let $x_i$ be the $i$-th digit of $X$ in binary
  - $X' = x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_1, x_0$

![Sign Extension Diagram](image)
Sign Extension Example

- Convert from smaller to larger integral data types
- C automatically performs sign extension
  - Java too

\[
\begin{align*}
\text{short int } & \quad x = \quad 12345; \\
\text{int } & \quad ix = (\text{int}) x; \\
\text{short int } & \quad y = \quad -12345; \\
\text{int } & \quad iy = (\text{int}) y;
\end{align*}
\]

<table>
<thead>
<tr>
<th>Var</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 00111001</td>
</tr>
<tr>
<td>ix</td>
<td>12345</td>
<td>00 00 30 39</td>
<td>00000000 00000000 00110000 00111001</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>
Integers

- Binary representation of integers
  - Unsigned and signed
  - Casting in C

- Consequences of finite width representations
  - Overflow, sign extension

- **Shifting and arithmetic operations**
Shift Operations

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

- **Right shift** \((x >> n)\) bit-vector \(x\) by \(n\) positions
  - Throw away (drop) 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 \(x\)
Shift Operations

- **Left shift** \((x << n)\)
  - Fill with 0s on right

- **Right shift** \((x >> n)\)
  - **Logical shift** (for *unsigned* values)
    - Fill with 0s on left
  - **Arithmetic shift** (for *signed* values)
    - Replicate most significant bit on left

- **Notes:**
  - Shifts by \(n < 0\) or \(n \geq w\) (\(w\) is bit width of \(x\)) are *undefined*.
  - **In C:** behavior of \(>>\) is determined by compiler
    - In gcc / C lang, depends on data type of \(x\) (signed/unsigned)
  - **In Java:** logical shift is \(>>>\) and arithmetic shift is \(>>\)
Shifting Arithmetic?

- What are the following computing?
  - \( x \gg n \)
    - \( 0_{\text{b}} 0100 \gg 1 = 0_{\text{b}} 0010 \)
    - \( 0_{\text{b}} 0100 \gg 2 = 0_{\text{b}} 0001 \)
  - Divide by \( 2^n \)
  - \( x \ll n \)
    - \( 0_{\text{b}} 0001 \ll 1 = 0_{\text{b}} 0010 \)
    - \( 0_{\text{b}} 0001 \ll 2 = 0_{\text{b}} 0100 \)
  - Multiply by \( 2^n \)

- Shifting is faster than general multiply and divide operations
Left Shifting Arithmetic 8-bit Example

- No difference in left shift operation for unsigned and signed numbers (just manipulates bits)
  - Difference comes during interpretation: $x \times 2^n$?

<table>
<thead>
<tr>
<th>Signed</th>
<th>Unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = 25$; $00011001 =$</td>
<td>$25$</td>
</tr>
<tr>
<td>$L_1 = x &lt;&lt; 2$; $001100100 =$</td>
<td>$100$</td>
</tr>
<tr>
<td>$L_2 = x &lt;&lt; 3$; $0011001000 =$</td>
<td>$-56$</td>
</tr>
<tr>
<td>$L_3 = x &lt;&lt; 4$; $010010000 =$</td>
<td>$-112$</td>
</tr>
</tbody>
</table>
Right Shifting Arithmetic 8-bit Examples

**Reminder:** C operator `>>` does *logical* shift on unsigned values and *arithmetic* shift on signed values

- **Logical Shift:** $x / 2^n$

$xu = 240u; \ 11110000 \ = \ 240$

$R1u=xu\gg3; \ 00011110000 \ = \ 30$

$R2u=xu\gg5; \ 0000011110000 \ = \ 7$

(rounding (down))
Right Shifting Arithmetic 8-bit Examples

**Reminder:** C operator `>>` does *logical* shift on *unsigned* values and *arithmetic* shift on *signed* values

- **Arithmetic Shift:** $x / 2^n$?

$$xs = -16; \quad 11110000 \quad = \quad -16$$

$R1s = xu >> 3; \quad 11111110 \quad = \quad -2$ (rounding (down))

$R2s = xu >> 5; \quad 111111110000 \quad = \quad -1$ (rounding (down))
Practice Question

For the following expressions, find a value of signed char \( x \), if there exists one, that makes the expression TRUE. Compare with your neighbor(s)!

- Assume we are using 8-bit arithmetic:
  - \( x \equiv (\text{unsigned char}) \ x \)
  - \( x \geq 128U \)
  - \( x \neq (x>>2)\ll2 \)
  - \( x \equiv -x \)
    - Hint: there are two solutions
  - \( (x < 128U) \&\& (x > 0x3F) \)

\( U_{\text{Min}} = 0 \), \( U_{\text{Max}} = 255 \)
8 bits, so \( T_{\text{Min}} = -128 \), \( T_{\text{Max}} = 127 \)
Summary

- Sign and unsigned variables in C
  - Bit pattern remains the same, just interpreted differently
  - Strange things can happen with our arithmetic when we convert/cast between sign and unsigned numbers
    - Type of variables affects behavior of operators (shifting, comparison)

- We can only represent so many numbers in \( w \) bits
  - When we exceed the limits, arithmetic overflow occurs
  - Sign extension tries to preserve value when expanding

- Shifting is a useful bitwise operator
  - Right shifting can be arithmetic (sign) or logical (0)
  - Can be used in multiplication with constant or bit masking
Some examples of using shift operators in combination with bitmasks, which you may find helpful for Lab 1.

- Extract the 2\textsuperscript{nd} most significant byte of an \texttt{int}
- Extract the sign bit of a signed \texttt{int}
- Conditionals as Boolean expressions
Using Shifts and Masks

- Extract the 2\textsuperscript{nd} most significant \textit{byte} of an \texttt{int}:
  - First shift, then mask: \((x\gg 16) \& 0xFF\)
  - Or first mask, then shift: \((x \& 0xFF0000) \gg 16\)

<table>
<thead>
<tr>
<th></th>
<th>(x)</th>
<th>(x\gg 16)</th>
<th>0xFF</th>
<th>((x\gg 16) &amp; 0xFF)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x)</td>
<td>00000001 00000010 00000011 00000100</td>
<td>00000000 00000000 00000001 00000010</td>
<td>00000000 00000000 00000000 11111111</td>
<td>00000000 00000000 00000000 00000010</td>
</tr>
<tr>
<td>0xFF</td>
<td>00000000 00000000 00000000 00000000</td>
<td>00000000 00000000 00000000 00000000</td>
<td>00000000 00000000 00000000 00000000</td>
<td>00000000 00000000 00000000 00000010</td>
</tr>
</tbody>
</table>
Using Shifts and Masks

- Extract the *sign bit* of a signed `int`:
  - First shift, then mask: `(x>>31) & 0x1`
    - Assuming arithmetic shift here, but this works in either case
    - Need mask to clear 1s possibly shifted in

<table>
<thead>
<tr>
<th>x</th>
<th>00000001 00000010 00000011 00000100</th>
</tr>
</thead>
<tbody>
<tr>
<td>x&gt;&gt;31</td>
<td>00000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td>0x1</td>
<td>00000000 00000000 00000000 00000001</td>
</tr>
<tr>
<td>(x&gt;&gt;31) &amp; 0x1</td>
<td>00000000 00000000 00000000 00000000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x</th>
<th>10000001 00000010 00000011 00000100</th>
</tr>
</thead>
<tbody>
<tr>
<td>x&gt;&gt;31</td>
<td>11111111 11111111 11111111 11111111</td>
</tr>
<tr>
<td>0x1</td>
<td>00000000 00000000 00000000 00000001</td>
</tr>
<tr>
<td>(x&gt;&gt;31) &amp; 0x1</td>
<td>00000000 00000000 00000000 00000001</td>
</tr>
</tbody>
</table>
Using Shifts and Masks

- **Conditionals as Boolean expressions**
  - For `int x`, what does `(x<<31)>>31` do?

<table>
<thead>
<tr>
<th></th>
<th>00000000 00000000 00000000 00000000 00000001</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x=!123</code></td>
<td>00000000 00000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td><code>x&lt;&lt;31</code></td>
<td>10000000 00000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td><code>(x&lt;&lt;31)&gt;&gt;31</code></td>
<td>11111111 11111111 11111111 11111111</td>
</tr>
<tr>
<td><code>!x</code></td>
<td>00000000 00000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td><code>!x&lt;&lt;31</code></td>
<td>00000000 00000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td><code>(!!x&lt;&lt;31)&gt;&gt;31</code></td>
<td>00000000 00000000 00000000 00000000 00000000</td>
</tr>
</tbody>
</table>

- Can use in place of conditional:
  - In C: if(x) {a=y;} else {a=z;} equivalent to `a=x?y:z;`
  - `a=((x<<31)>>31)&y) | ((!(x<<31)>>31)&z);`