Lecture 4: Integer arithmetic

David Hovemeyer

September 7, 2022

601.229 Computer Systems Fundamentals

Integer arithmetic

Integer arithmetic

- Integer representations based on fixed-size machine words are *finite*
- I.e., only a finite number of possible values can be represented
    - For word with w bits, can represent 2^w possible values
- So, we should expect some (potentially) strange results when doing arithmetic using machine words
- These strange results can lead to surprising program behavior, including security vulnerabilities

Addition of unsigned values

Addition of unsigned values

- Same idea as what you learned in grade school
  - Start with least significant digit
  - As needed, carry excess into next-most-significant digit

Addition of unsigned values (no overflow)

Example: 0110 + 0111

$$\begin{array}{rr} & 0 \\ & 0110 \\ + & 0111 \\ \hline \end{array}$$

Addition of unsigned values (no overflow)

Example: 0110 + 0111

$$
\begin{array}{rr}
 & \mathbf{0}0 \\
 & 011\mathbf{0} \\
+ & 011\mathbf{1} \\
\hline
 & \mathbf{1}
\end{array}
$$

no carry

Addition of unsigned values (no overflow)

Example: 0110 + 0111

$$
\begin{array}{rr}
 & \mathbf{1}00 \\
 & 01\mathbf{1}0 \\
+ & 01\mathbf{1}1 \\
\hline
 & \mathbf{0}1
\end{array}
\quad \text{carry } 1
$$

Addition of unsigned values (no overflow)

Example: $0110 + 0111$

$$
\begin{array}{rr}
 & \mathbf{1}000 \\
 & 0\mathbf{1}10 \\
+ & 0\mathbf{1}11 \\
\hline
 & \mathbf{1}01 \quad \text{carry } 1
\end{array}
$$

Addition of unsigned values (no overflow)

Example: $0110 + 0111$

$$
\begin{array}{rr}
 & \mathbf{0}1000 \\
 & \mathbf{0}110 \\
+ & \mathbf{0}111 \\
\hline
 & \mathbf{1}101
\end{array}
\quad \text{no carry}
$$

Addition of unsigned values (no overflow)

Example: $0110 + 0111$

$$\begin{array}{rr} & 0110 \\ + & 0111 \\ \hline & 1101 \end{array} \quad \text{done}$$

Overflow

- If the sum of w-bit (unsigned) integer values is too large to represent using a w-bit word, *overflow* occurs
- Effective sum of w bit integers a and b is

$$(a + b) \bmod 2^w$$

Addition of unsigned values (overflow)

Example: $1110 + 0111$

$$\begin{array}{rr} & 0 \\ & 1110 \\ + & 0111 \\ \hline \end{array}$$

Addition of unsigned values (overflow)

Example: $1110 + 0111$

$$
\begin{array}{rr}
 & \mathbf{0}0 \\
 & 111\mathbf{0} \\
+ & 011\mathbf{1} \\
\hline
 & \mathbf{1}
\end{array}
$$

no carry

Addition of unsigned values (overflow)

Example: $1110 + 0111$

$$\begin{array}{rr} & \mathbf{1}00 \\ & 11\mathbf{1}0 \\ + & 01\mathbf{1}1 \\ \hline & \mathbf{0}1 \end{array} \quad \text{carry } 1$$

Addition of unsigned values (overflow)

Example: 1110 + 0111

$$
\begin{array}{rr}
 & \mathbf{1}100 \\
 & 11\mathbf{1}0 \\
+ & 0\mathbf{1}11 \\
\hline
 & \mathbf{1}01 \quad \text{carry } 1
\end{array}
$$

Addition of unsigned values (overflow)

Example: $1110 + 0111$

$$
\begin{array}{rr}
 & \mathbf{1}1100 \\
 & \mathbf{1}110 \\
+ & \mathbf{0}111 \\
\hline
 & \mathbf{0}101
\end{array}
\quad \text{carry } 1
$$

Addition of unsigned values (overflow)

Example: 1110 + 0111

$$\begin{array}{rr} & 11100 \\ & 1110 \\ + & 0111 \\ \hline & 1\underline{0101} \end{array}$$

True sum is 10101 (21), effective sum is 101 (5) (note 21 mod 16 = 5)

Clicker quiz

Clicker quiz omitted from public slides

Addition of signed values

Useful property of two's complement: addition is carried out *exactly the same way* for signed values as for unsigned values

Signed addition example

Example: 0101 (5) + 1110 (-2)

$$
\begin{array}{rr}
 & 0101 \\
+ & 1110 \\
\hline
\end{array}
$$

Signed addition example

Example: 0101 (5) + 1110 (-2)

$$\begin{array}{r} 0101 \\ +\quad 1110 \\ \hline 1\underline{0011} \end{array}$$

After truncating (discarding high bit of sum), effective sum is 0011 (3)

Signed overflow

What happens when sum of signed w-bit values can't be represented?

- If sum exceeds $2^{w-1} - 1$, it becomes negative (overflow)
- If sum is less than -2^{w-1}, it becomes positive (negative overflow)

Signed addition example (overflow)

Example: 0100 (4) + 0101 (5)

$$\begin{array}{rr} & 0100 \\ + & 0101 \\ \hline \end{array}$$

Signed addition example (overflow)

Example: 0100 (4) + 0101 (5)

$$\begin{array}{rr} & 0100 \\ + & 0101 \\ \hline & 1001 \end{array}$$

Result is -7 (-8 + 1)

Signed addition example (negative overflow)

Example: 1100 (-4) + 1011 (-5)

$$\begin{array}{r} 1100 \\ + \quad 1011 \\ \hline \end{array}$$

Signed addition example (negative overflow)

Example: 1100 (-4) + 1011 (-5)

$$\begin{array}{rr} & 1100 \\ + & 1011 \\ \hline & 1\underline{0111} \end{array}$$

Result (after truncating) is 7

Clicker quiz

Clicker quiz omitted from public slides

Two's complement negation and subtraction

- Negation: if x is a two-complement integer value, $-x$ can be computed by inverting bits of x, then adding 1

Two's complement negation and subtraction

- Negation: if x is a two-complement integer value, $-x$ can be computed by inverting bits of x, then adding 1
  - Why?

Two's complement negation and subtraction

- Negation: if x is a two-complement integer value, $-x$ can be computed by inverting bits of x, then adding 1
  - Why?
    - Inverting all bits is equivalent to subtracting from a bitstring consisting of all 1 bits

Two's complement negation and subtraction

- Negation: if x is a two-complement integer value, $-x$ can be computed by inverting bits of x, then adding 1
  - Why?
    - Inverting all bits is equivalent to subtracting from a bitstring consisting of all 1 bits
    - A bitstring of all 1 bits has the value -1

Two's complement negation and subtraction

- Negation: if x is a two-complement integer value, $-x$ can be computed by inverting bits of x, then adding 1
  - Why?
    - Inverting all bits is equivalent to subtracting from a bitstring consisting of all 1 bits
    - A bitstring of all 1 bits has the value -1
    - So, inverting the bits of x and adding 1 effectively means $-1 - x + 1 = 0 - x = -x$

Two's complement negation and subtraction

- Negation: if x is a two-complement integer value, $-x$ can be computed by inverting bits of x, then adding 1
  - Why?
    - Inverting all bits is equivalent to subtracting from a bitstring consisting of all 1 bits
    - A bitstring of all 1 bits has the value -1
    - So, inverting the bits of x and adding 1 effectively means $-1 - x + 1 = 0 - x = -x$
- Subtraction:

$$a - b = a + -b$$

I.e., to compute $a - b$, compute $-b$, then add $-b$ to a

Clicker quiz

Clicker quiz omitted from public slides

Integer arithmetic in C

Integer arithmetic in C

- C data types are "close to" the machine data types
- Understanding machine-level data representation will help you understand C
- But, there are traps for the unwary!
  - Certain operations in C are *undefined behavior*
    - Program could do anything (bad)
    - Compiler can (and often does) assume that undefined behavior will never occur, leading to surprising "optimizations"
  - Certain operations in C are *implementation defined*
    - The compiler will document what the code will do, but it can vary within a range of allowed behaviors

Shifts

- *Shifts* move the bits in a value some number of positions left or right
- Bits shifted out are discarded
- Bits shifted in could be 0 or 1 depending on operand type
- Can be used to multiply or divide a value by a power of 2
  - Left shift by 1 bit: multiply by 2
  - Right shift by 1 bit: divide by 2
- Typically faster than actual CPU integer multiply and divide instructions

Example unsigned shifts

Given declaration `uint16_t x = 0x0FFF;`

Expression	Dec	Hex	Binary
`x`	4095	0FFF	0000111111111111
`x << 1`	8190	1FFE	0001111111111110
`x << 5`	65504	FFE0	1111111111100000
`x >> 1`	2047	07FF	0000011111111111
`x >> 5`	127	007F	0000000001111111

Example signed shifts

Given declarations:

```
int16_t x = 0x0FFF;
int16_t y = 0x8000;
```

Expression	Dec[1]	Hex	Binary
`x`	4095	0FFF	0000111111111111
`x << 1`	8190	1FFE	0001111111111110
`x << 5`	*undefined*		
`x >> 1`	2047	07FF	0000011111111111
`x >> 5`	127	007F	0000000001111111
`y`	-32768	8000	1000000000000000
`y >> 1`	*implementation-defined*		

[1] Assuming two's complement

Gotchas with signed shifts

- Left shifts into or past the sign bit are *undefined*
  - Assuming 32-bit `int` values, `0x40000000 << 1` is undefined
  - Undefined behavior means *anything* could happen when the program attempts to perform this computation
- Right shifts could either replicate the sign bit (“arithmetic” shift) or shift in 0 bits (“logical” shift)
  - Assuming 32 bit `int` values, `0x80000000 >> 1` could yield either `0xC0000000` or `0x40000000`
  - This is *implementation-defined* behavior

Size conversions

What happens when integer values are converted to a different-sized representation:

- Unsigned small to large, 0 bits added (value preserved)
- Signed small to large, sign bit duplicated (value preserved)
- Unsigned large to small, truncation (value could change)
- Signed large to small, truncation (value could change)

Signed-ness conversions

When signed and unsigned values are used in an expression (a) the signed value is converted to unsigned (by reinterpreting its bits as an unsigned value), (b) the result is unsigned

This can lead to surprising results!

Overflows (unsigned)

Overflow for unsigned integer types is defined in terms of wrapping:

```
unsigned x = UINT_MAX;
x++;
printf("%u\n", x);
```

This code is guaranteed to print “0”

```
unsigned x = 0;
x--;
printf("%u\n", x);
```

This code is guaranteed to print the value of `UINT_MAX`

Overflows (signed)

Overflow for signed integer types is *undefined*!

That's really bad!