CMPT 295
Unit - Data Representation
Lectures 4 and 5 – Representing integers in memory - Arithmetic operations
Summary

- Interpretation of bit patterns into either unsigned or signed values
  - $B2U(X)$ and $U2B(X)$ encoding schemes
- Conversions from unsigned $\leftrightarrow$ signed values - $B2T(X)$ and $T2B(X)$
- Signed value expressed as two's complement
- Implication in C: when converting (implicitly and explicitly via casting):
  - Sign:
    - Unsigned $\leftrightarrow$ signed (of same size) $\rightarrow$ Bit pattern is maintained, but reinterpreted
    - Can have unexpected effects: adding or subtracting $2^w$
  - Size:
    - Small $\rightarrow$ large (both signed or both unsigned - e.g., short to int)
      - sign extension: Unsigned $\rightarrow$ zeros extension, signed $\rightarrow$ sign bit extension
      - Both yield expected result
    - Large $\rightarrow$ small (e.g., unsigned to unsigned short)
      - truncation: Unsigned/signed $\rightarrow$ most significant bits are truncated
      - May alter original value
  - Both (sign and size): 1) size then 2) sign
Today’s Menu

- Representing information as bits
  - “Under the Hood” - A look at memory
    - Bits in memory
    - Encoding scheme
    - Endian
  - Bit manipulation
    - Boolean algebra + Shifting

- Representing integers in memory
  - Unsigned and signed
  - Converting, expanding and truncating
  - Arithmetic operations

- Representing real numbers in memory
  - IEEE floating point representation
  - Floating point in C – casting, rounding, addition, …
What happens when we add two decimal numbers?

\[
\begin{array}{c}
107 \\
+ \ 938 \\
\hline
1045
\end{array}
\]

Same thing happens when we add two binary numbers:

\[
\begin{array}{c}
101100_2 \\
+ \ 101110_2 \\
\hline
1011110_2
\end{array}
\]
Unsigned addition (limited space, i.e., fixed size memory)

What happens when we add two unsigned values:

\( w = 8 \)

a) \( \begin{array}{c}
00111011_2 \\
+ 01011010_2 \\
\hline
01110101_2
\end{array} \)

b) \( \begin{array}{c}
10101110_2 \\
+ 11001011_2 \\
\hline
10100001_2
\end{array} \)
Unsigned addition \( (+_w^u) \) and overflow

- Operands: \( w \) bits
- True Sum: \( w+1 \) bits
- Discard carry out bit: \( w \) bits (overflow)

- Discarding carry out bit has same effect as applying modular arithmetic
  \[ s = u +^w u v = (u + v) \mod 2^w \]

Would be the result of **normal integer addition** with unlimited space

Result of **unsigned addition** with limited space

Overflow

True Sum

\[ 2^{w+1} \]

\[ 2^w - 1 \]

\[ 0 \]

\[ u +^u w v \]
Closer look at unsigned addition overflow

\[ w = 8 \]

\[
\begin{array}{c}
255_{10} = 11111111_2 \\
90_{10} = 01011010_2 \\
45_{10} = 00101101_2 \\
\end{array}
\]

\[
\begin{array}{c}
\phantom{+}90_{10} \\
+ 45_{10} \\
\hline
135_{10}
\end{array}
\]

\[
\begin{array}{c}
01011010_2 \\
+ 00101101_2 \\
\hline
\phantom{+}11011111_2
\end{array}
\]

\[
\begin{array}{c}
255_{10} \\
+ 45_{10} \\
\hline
300_{10}
\end{array}
\]

\[
\begin{array}{c}
\phantom{+}01011010_2 \\
+ 00101101_2 \\
\hline
\phantom{+}11011111_2
\end{array}
\]

Actual sum

True Sum

Overflow
Comparing integer addition with unsigned addition ($w = 4$)

**Overflow:** Effect of fixed size memory

An overflow occurs when there is a carry out
Signed addition (limited space, i.e., fixed size memory)

- What happens when we add two signed values:

\[
\begin{align*}
\text{w} &= 8 \\
a) & \quad 00111011_2 \\
& \quad + \ 01011010_2 \\
b) & \quad 10111010_2 \\
& \quad + \ 11001011_2 \\
\end{align*}
\]

Observation:
Signed addition \(+^t_w\) and overflow

Operands: \(w\) bits
True Sum: \(w+1\) bits

Discard carry out bit: \(w\) bits (overflow)

- Discarding carry out bit has same effect as applying modular arithmetic
  \[ s = u +^t_w v = U2T_w [(u + v) \mod 2^w] \]

True Sum

\[
\begin{array}{c}
2^w - 1 \\
2^w - 1 - 1 \\
0 \\
-2^w - 1 \\
-2^w \\
\end{array}
\]

- Positive Overflow
- Negative Overflow

Result of signed addition with limited space

Would be the result of normal integer addition with unlimited space
Closer look at signed addition overflow

\[ w = 8 \]

\[
\begin{array}{c}
90_{10} = 01011010_2 \\
45_{10} = 00101101_2 \\
-45_{10} = 11010011_2 \\
-90_{10} = 10100110_2
\end{array}
\]

\[
\begin{array}{c}
90_{10} + 45_{10} = 135_{10} \\
90_{10} + 00101101_2 = 10101101_2 \\
-90_{10} + -45_{10} = -135_{10} \\
-90_{10} + 11010011_2 = 01011010_2
\end{array}
\]

True Sum

Positive Overflow

Negative Overflow

Actual sum
Visualizing signed addition overflow ($w = 4$)

Negative Overflow

Positive Overflow
What about subtraction? -> Addition

- 107
- 118
=> + (-118)

- Subtracting a number is equivalent to adding its additive inverse
- Let’s try: 107 -> 01101011 -> 01101011
  - 118 -> -01110110 -> +
Multiplication ($\ast \! _w^u$, $\ast \! _w^t$) and overflow

Operands: $w$ bits

True Product: $2w$ bits

Discard: $w$ bits

- Discarding high order $w$ bits has same effect as applying modular arithmetic
  
  \[ p = u * \! _w^u v = (u * v) \mod 2^w \]
  
  \[ p = u * \! _w^t v = U2T_w [(u * v) \mod 2^w] \]

Would be the result of normal integer multiplication with unlimited space

Result of multiplication with limited space

Example:
Multiplication with power-of-2 versus shifting

- If \( x \times y \) where \( y = 2^k \) then \( x \ll k \)
  - For both signed and unsigned

- Example:
  - \( x \times 8 = x \times 2^3 \Rightarrow x \ll 3 \)
  - \( x \times 24 = (x \times 2^5) - (x \times 2^3) = (x \times 32) - (x \times 8) \Rightarrow (x \ll 5) - (x \ll 3) \)
    (decompose 24 in powers of 2: \( 32 - 8 \))

- Most machines shift and add faster than multiply
  - We’ll soon see that compiler generates this code automatically
Summary

- Demo (code and results) posted!
- Addition:
  - Unsigned/signed:
    - Behave the same way at the bit level
    - Interpretation of resulting bit vector (sum) may differ
  - Unsigned: \((x + y) \mod 2^w\) (sum in proper range)
    - Integer addition with possible subtraction of \(2^w\)
  - Signed: \(\text{U2T}_w [(x + y) \mod 2^w]\) (sum in proper range)
    - Integer addition with possible addition or subtraction of \(2^w\)
- Subtraction
  - Becomes an addition where negative operands are transformed into their additive inverse
- Multiplication:
  - Unsigned: \((x \times y) \mod 2^w\) (product in proper range)
  - Signed: \(\text{U2T}_w [(x \times y) \mod 2^w]\) (product in proper range)
  - Can be replaced by additions and shifts
Next lecture

- Representing information as bits
- Representing integers in memory
  - Unsigned and signed
  - Converting, expanding and truncating
  - Arithmetic operations
- Representing real numbers in memory
  - IEEE floating point representation
  - Floating point in C – casting, rounding, addition, ...