# CMPT 295

Unit - Data Representation

Lecture 4 – Representing integral numbers in memory – Arithmetic operations

# Warm up question

- What is the value of …
  - `TMin` (in hex) for `signed char` in C: ____________________
  - `TMax` (in hex) for `signed int` in C: ____________________
  - `TMin` (in hex) for `signed short` in C: ____________________

# Last Lecture

- Interpretation of bit pattern B into either unsigned value U or signed value T
  - *B2U(X)* and *U2B(X)* encoding schemes (conversion)
  - *B2T(X)* and *T2B(X)* encoding schemes (conversion)
    - Signed value expressed as two's complement => *T*
- Conversions from unsigned <-> signed values
  - *U2T(X)* and *T2U(X)* => adding or subtracting $2^w$
- Implication in C: when converting (implicitly via promotion and explicitly via casting):
  - Sign:
    - Unsigned <-> signed (of same size) -> Both have same bit pattern, however, this bit pattern may be interpreted differently
      - Can have unexpected effects -> producing a different value
  - Size:
    - Small -> large (for signed, e.g., `short` to `int` and for unsigned, e.g., `unsigned short` to `unsigned int`)
      - sign extension: For unsigned -> zeros extension, for signed -> sign bit extension
      - Both yield expected result –> resulting value unchanged
    - Large -> small (for signed, e.g., `int` to `short` and for unsigned, e.g., `unsigned int` to `unsigned short`)
      - truncation: Unsigned/signed -> most significant bits are truncated (discarded)
      - May not yield expected results -> original value may be altered
  - Both (sign and size): 1) **size** conversion is first done then 2) **sign** conversion is done

# Today's Menu

- Representing data in memory – Most of this is review
  - "Under the Hood" - Von Neumann architecture
  - Bits and bytes in memory
    - How to diagram memory -> Used in this course and other references
    - How to represent series of bits -> In binary, in hexadecimal (conversion)
    - What kind of information (data) do series of bits represent -> Encoding scheme
    - Order of bytes in memory -> Endian
  - Bit manipulation – bitwise operations
    - Boolean algebra + Shifting
- Representing integral numbers 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, …

Let's first illustrate what we covered last lecture with a demo!

# Demo – Looking at *size* and *sign* conversions in C

- What does the demo illustrate?
  - Size conversion:
    - Converting to a larger (wider) data type -> Converting `short` to `int`
    - Converting to a smaller (narrower) data type -> Converting `short` to `char`
  - Sign conversion:
    - Converting from signed to unsigned -> Converting `short` to `unsigned short`
    - Converting from unsigned to signed -> Converting `unsigned short` to `short`
  - Size and Sign conversion:
    - Converting from signed to unsigned larger (wider) data type -> Converting `short` to `unsigned int`
    - Converting from signed to unsigned smaller (narrower) data type -> Converting `short` to `unsigned char`
- This demo (code and results) posted on our course web site

# Integer addition (unlimited space)

- What happens when we add two decimal numbers?

  1 <- carry in

  $107_{10}$

  $+\ 938_{10}$

  carry out -> $1045_{10}$

- Same thing happens when we add two binary numbers:

  1 1 <- carry in

  $101100_2$

  $+\ 101110_2$

  carry out -> $1011010_2$

# Unsigned addition (limited space, i.e., fixed size in memory)

- What happens when we add two unsigned values:

$w = 8$

a) $$\begin{array}{r} 00111011_2 \\ +\ 01011010_2 \\ \hline \end{array} \qquad \begin{array}{r} 59_{10} \\ +\ 90_{10} \\ \hline 149_{10} \end{array}$$

b) $$\begin{array}{r} 10101110_2 \\ +\ 11001011_2 \\ \hline \end{array} \qquad \begin{array}{r} 174_{10} \\ +\ 203_{10} \\ \hline 377_{10} \end{array}$$

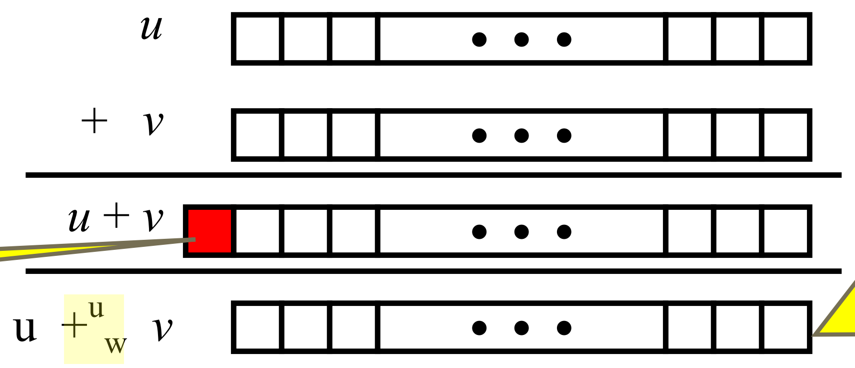
# Unsigned addition ($+^u_w$) and overflow

Would be the result of **integer addition** with unlimited space: expected sum

Operands: w bits

True Sum: w+1 bits

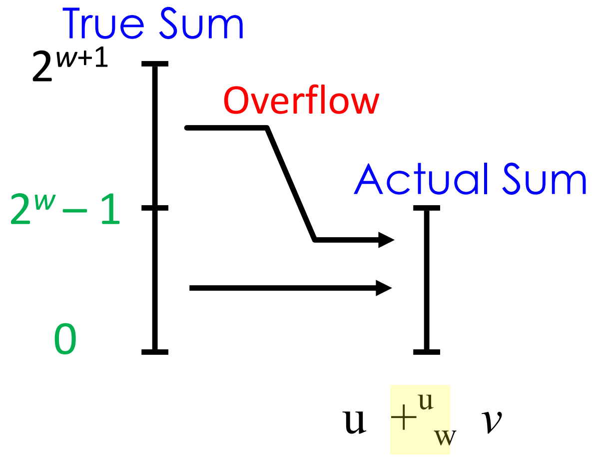
Discard carry out bit: w bits (overflow)




Result of **unsigned addition** with limited space: actual sum

- Discarding carry out bit has same effect as applying modular arithmetic

$$s = u +^u_w v = (u + v) \mod 2^w$$



# Closer look at unsigned addition overflow

w = 8 -> [0..255]

$255_{10} = 11111111_2$

$90_{10} = 01011010_2$

$45_{10} = 00101101_2$

carry in

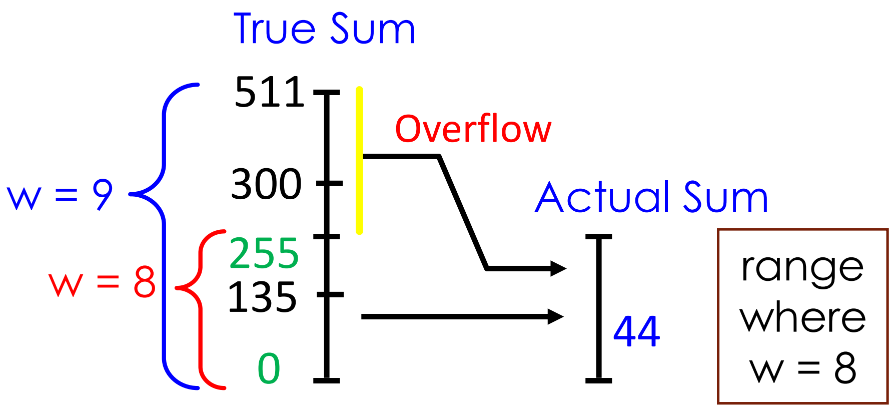
$$\begin{array}{rr} 90_{10} & 01011010_2 \\ +\ 45_{10} & +\ 00101101_2 \\ \hline 135_{10} & 10000111_2 \\ \hline \end{array}$$

carry out

carry in

$$\begin{array}{rr} 255_{10} & 11111111_2 \\ +\ 45_{10} & +\ 00101101_2 \\ \hline 300_{10} & 100101100_2 \\ \hline \end{array}$$

carry out

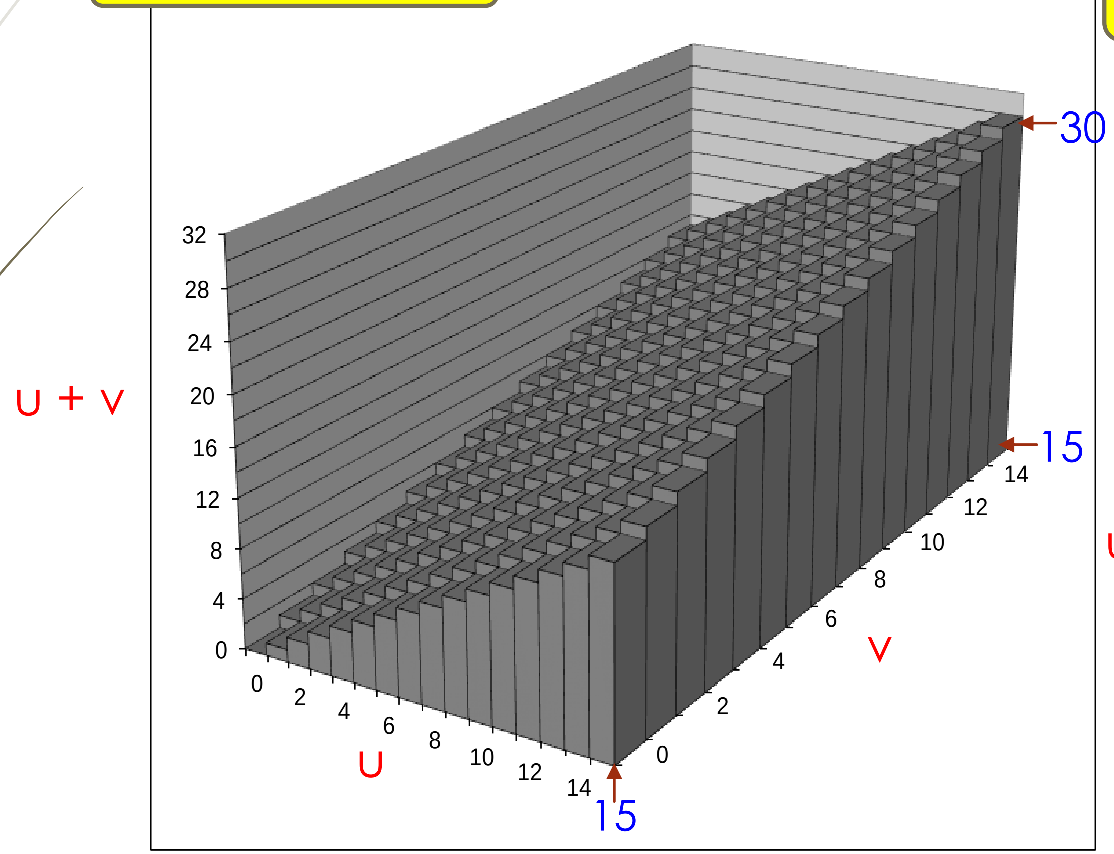


# Comparing integer addition with unsigned addition (w = 4)

**Overflow**: Effect of fixed size memory

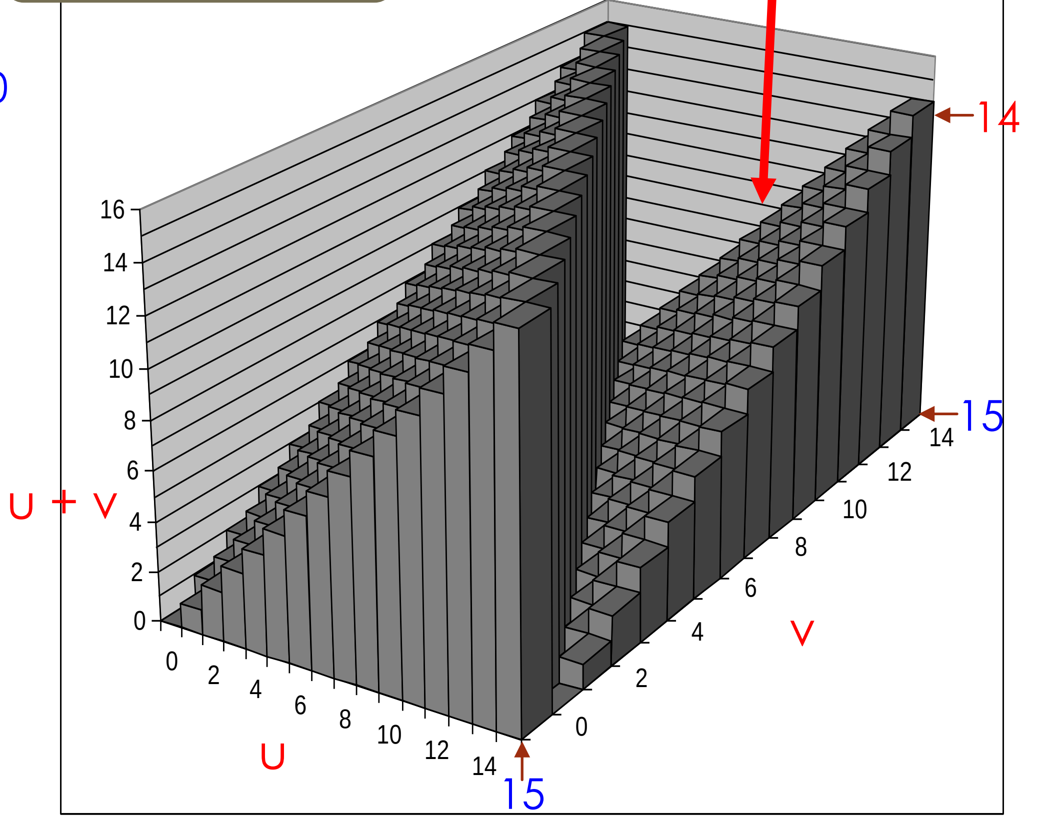
with unlimited space

Integer Addition




with limited space (fixed size memory)

Unsigned Integer Addition




An overflow occurs when there is a carry out

For example: 15 ($1111_2$) + 15 ($1111_2$) = 30 ($11110_2$ <- true sum) and = 14 ($\cancel{1}1110_2$ <- actual sum)

# Signed addition (limited space, i.e., fixed size in memory)

- What happens when we add two signed values:

w = 8

a) $00111011_2$ + $01011010_2$ — $59_{10}$ + $90_{10}$ = $149_{10}$

$$\begin{array}{r} 00111011_2 \\ +\ 01011010_2 \\ \hline \end{array} \qquad \begin{array}{r} 59_{10} \\ +\ 90_{10} \\ \hline 149_{10} \end{array}$$

b) 

$$\begin{array}{r} 10101110_2 \\ +\ 11001011_2 \\ \hline \end{array} \qquad \begin{array}{r} -82_{10} \\ +\ -53_{10} \\ \hline -135_{10} \end{array}$$

Observation: Unsigned and signed additions have identical behavior @ the bit level, i.e., their sum have the same bit-level representation, but their interpretation differs

# Signed addition ($+^{t}_{w}$) and overflow

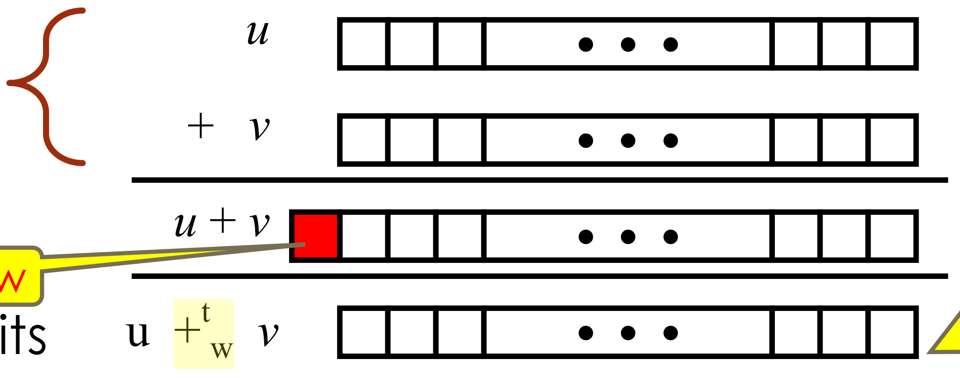
Would be the result of **integer addition** with unlimited space: expected sum

Operands: w bits

True Sum: w+1 bits

overflow

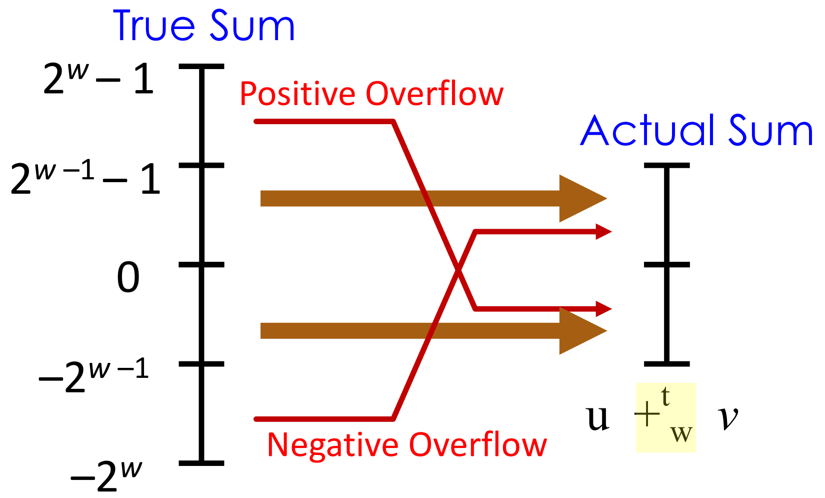
Discard carry out bit : w bits (overflow)




Result of **signed addition** with limited space: actual sum

- Discarding carry out bit has same effect as applying modular arithmetic

$$s = u +^{t}_{w} v = \mathrm{U2T}_{w}\,[(u + v) \bmod 2^{w}]$$



# Closer look at signed addition overflow

w = 8 -> [-128..127]

$$\begin{aligned}
90_{10} &= 01011010_2 \\
45_{10} &= 00101101_2 \\
-45_{10} &= 11010011_2 \\
-90_{10} &= 10100110_2
\end{aligned}$$

carry in

$$\begin{array}{r} 90_{10} \\ +\ 45_{10} \\ \hline 135_{10} \\ \hline \end{array} \qquad \begin{array}{r} 01011010_2 \\ +\ 00101101_2 \\ \hline 010000111_2 \\ \hline \end{array}$$

carry out

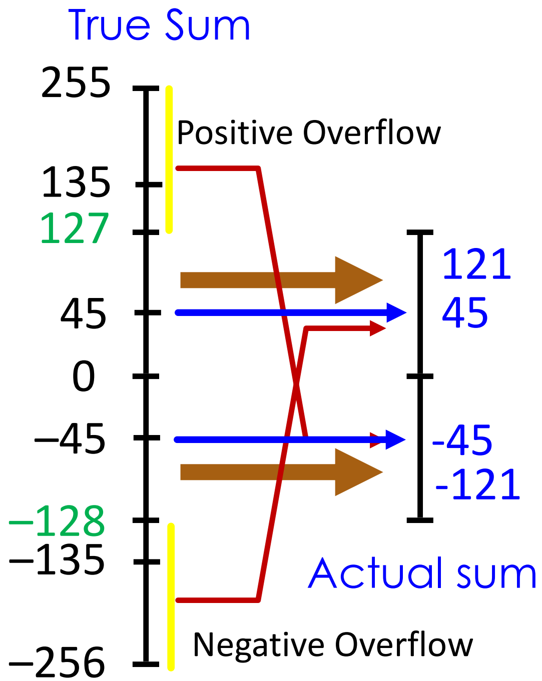
<= -121

carry in

$$\begin{array}{r} -90_{10} \\ +\ -45_{10} \\ \hline -135_{10} \\ \hline \end{array} \qquad \begin{array}{r} 10100110_2 \\ +\ 11010011_2 \\ \hline 101111001_2 \\ \hline \end{array}$$

carry out

<= 121




carry in

$$\begin{array}{r} -90_{10} \\ +\ 45_{10} \\ \hline -45_{10} \end{array} \qquad \begin{array}{r} 10100110_2 \\ +\ 00101101_2 \\ \hline 011010011_2 \end{array}$$

carry out

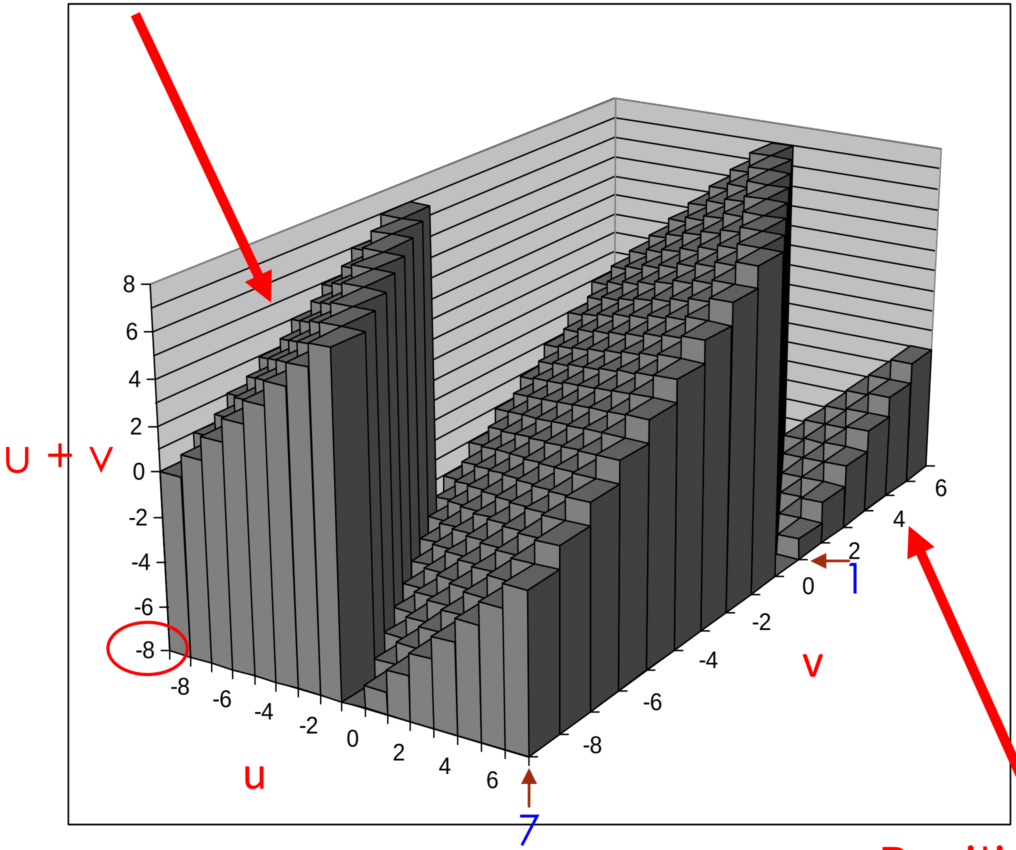
carry in

$$\begin{array}{r} 90_{10} \\ +\ -45_{10} \\ \hline 45_{10} \end{array} \qquad \begin{array}{r} 01011010_2 \\ +\ 11010011_2 \\ \hline 100101101_2 \end{array}$$

carry out

# Visualizing signed addition overflow ($w$ = 4)

Negative Overflow




Positive Overflow

For example: 7 ($0111_2$) + 1 ($0001_2$) = 8 ($1000_2$ <- true sum) and = -8 ($1000_2$ <- actual sum)

# What about subtraction? -> Addition

$x + (-x) = 0$

- Subtracting a number is equivalent to adding its additive inverse
  - Instead of subtracting a positive number, we could add its negative version:

$$\begin{array}{r} 107 \\ \underline{-\ 118} \\ -\ 11 \end{array} \quad \Rightarrow \quad \begin{array}{r} 107 \\ \underline{+\ (-118)} \end{array}$$

- Let 's try:

$$\begin{array}{rcrcr} 107_{10} & \rightarrow & 01101011_2 & \rightarrow & 01101011_2 \\ \underline{-\ 118_{10}} & \rightarrow & \underline{-\ \ 01110110_2} & \rightarrow & \underline{+\ \ 10001010_2} \\ -\ 11 & & & & 11110101_2 \Rightarrow -11_{10} \end{array}$$

$T2B(X)$ conversion: **(~(U2B(|$X$|)))+1**
**= (~(U2B(**|-118|**)))+1**
**= (~(U2B(**118**)))+1**
**= (~(**$01110110_2$**))+1**
**= (**$10001001_2$**)+1**
**=** $10001010_2$

CHECK: $-128+64+32+16+4+1 = -11_{10}$

# Multiplication ($*^u_w$ , $*^t_w$) and overflow

Would be the result of **integer multiplication** with unlimited space: expected product

Operands: w bits

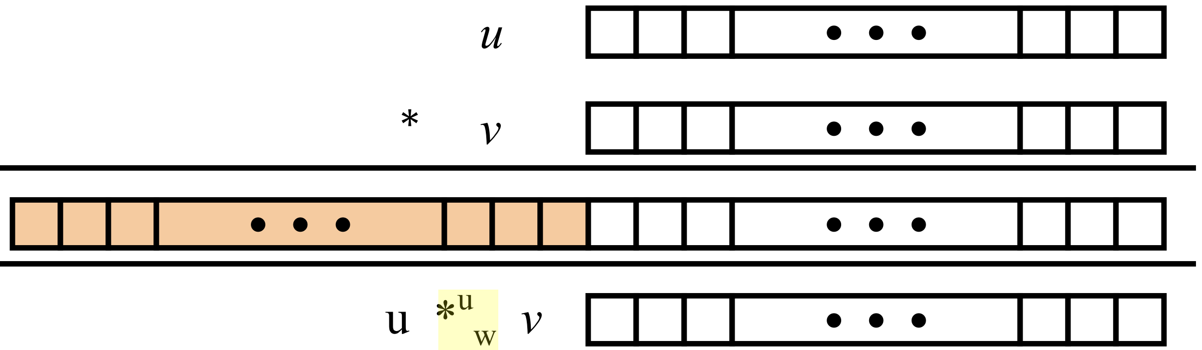
$u$

$* \quad v$

True Product: 2w bits $u \cdot v$

Discard: w bits

$u \; *^u_w \; v$



Result of **multiplication** with limited space: actual product

- Discarding high order w bits has same effect as applying modular arithmetic

$$p = u *^u_w v = (u * v) \mod 2^w$$

$$p = u *^t_w v = U2T_w[(u * v) \mod 2^w]$$

- Example: w = 4

$$\begin{array}{r} 5_{10} \\ \times \; 5_{10} \\ \hline 25_{10} \end{array} \qquad \begin{array}{r} 0101_2 \\ \times \; 0101_2 \\ \hline 0101_2 \\ 0000_2\phantom{0} \\ 0101_2\phantom{00} \\ 0000_2\phantom{000} \\ \hline \cancel{001}1001_2 \end{array}$$

# Multiplication with power-of-2 versus shifting

- If $x * y$ where $y = 2^k$ then $x << k$
  - For both signed and unsigned

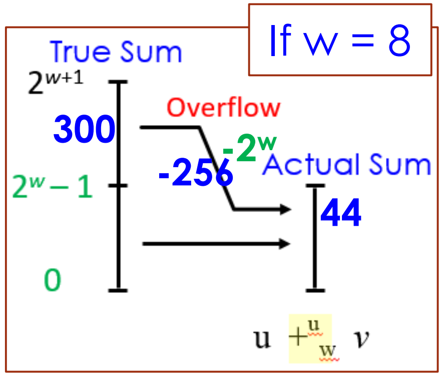
- Example:
  - $x * 8 \;=\; x * 2^3 \;\text{->}\; x << 3$
  - $x * 24 \;=\; (x * 2^5) – (x * 2^3) \;=\; (x * 32) – (x * 8) \;\text{->}\; (x << 5) – (x << 3)$

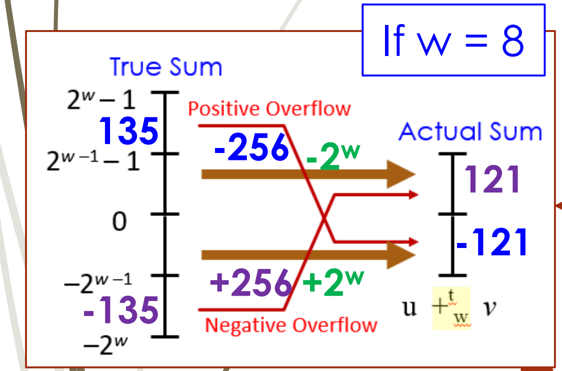
    (decompose 24 in powers of 2 => 32 – 8)

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

# Summary

- Demo of size and sign conversion in C: code and results posted!
- Addition:
  - Unsigned/signed:
    - Behave the same way at the bit level
    - Interpretation of resulting bit vector (sum) may differ
  - Unsigned addition -> may overflow, i.e., (w+1)$^{th}$ bit is set
    - If so, then actual sum obtained => $(x + y)$ mod $2^w$
  - Signed addition -> may overflow, i.e., (w+1)$^{th}$ bit is set
    - If so, then true sum may be too +ve -> positive overflow OR too –ve -> negative overflow
    - Then actual sum obtained => $U2T_w$ [$(x + y)$ mod $2^w$]




- Subtraction
  - Becomes an addition where negative operands are transformed into their additive inverse (in two's complement)
- Multiplication:
  - Unsigned: actual product obtained -> $(x * y)$ mod $2^w$
  - Signed: actual product obtained -> $U2T_w$ [$(x * y)$ mod $2^w$]
  - Can be replaced by additions and shifts

# Next lecture

- Representing data in memory – Most of this is review
  - "Under the Hood" - Von Neumann architecture
  - Bits and bytes in memory
    - How to diagram memory -> Used in this course and other references
    - How to represent series of bits -> In binary, in hexadecimal (conversion)
    - What kind of information (data) do series of bits represent -> Encoding scheme
    - Order of bytes in memory -> Endian
  - Bit manipulation – bitwise operations
    - Boolean algebra + Shifting
- Representing integral numbers 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, …

We'll illustrate what we covered today by having a demo!