# Chapter 2: Logical and Bit Operations and Integer Arithmetic

## Topics

- Logical Operations in C
  - &&, ||, and !
- Bit Shifting
  - << Left Shift
  - >> Right Shift (Logical and Arithmetic)
- Integer Representations
  - Unsigned and Two's Complement
- Integer Arithmetic and overflow

# Announcements

- Data Lab is due Friday Sept 12 by 11:55 pm
- Recitation exercises #1 due next Monday 9/8
  - hand in at your recitation
- Extra TA office hours have been announced by email
  - Thursdays 10:30-12:30 pm (Yogesh in CSEL)
  - Thursdays 3-5 pm  (Pate in ECCS 123)
  - Fridays 10-12 noon (Abhishek in CSEL)
- Essential that you read the textbook in detail & do the practice problems
  - Read Chapter 2 (2.1-2.3)
  - Chapter 3 next week (3.1-3.11 and 3.13-3.14)

# Recap…

- Byte ordering or Endianness – Little vs Big Endian
  - Ints
  - Pointers
  - Characters
  - Strings
- Boolean algebra on binary values
  - &, |, ~, ^
  - Properties at times like but distinct from integer algebra
  - Applied to N-bit vectors or words
  - Contrast to logical operators &&, ||, !

# Logical vs Bitwise Operations in C

- **Logical Operators**
  - `&& (AND), || (OR), ! (NOT or "bang")`
    - View 0 as "False"
    - Anything nonzero as "True"
    - Always return 0 or 1
    - Early termination
- **Example code:**
  - int x,y,z;

```
....
if (  !((x==0) && (x>y))  || (z<256) ) {
   ...
   z = ~(x & y) | z;
}
```

Each logical expression is either TRUE (1) or FALSE (0):

(x==0)

(x>y)

((x==0) && (x>y))

!((x==0) && (x>y))

(z<256)

!((x==0) && (x>y)) || (z<256)

Compare to the bit-wise logical operations, where input, intermediate, and final values don't have to be 0 or 1

By the way, parentheses are your friend, and good programming practice

# Logical vs Bitwise Operations in C (2)

- **Logical**
  - `!0x41 --> 0x00`
  - `!0x00 --> 0x01`
  - `!!0x41 --> 0x01`
  - `0x69 && 0x55 --> 0x01`
  - `0x69 || 0x55 --> 0x01`
  - `p && *p` (avoids null pointer access)
    - *p by itself could result in dereferencing a null pointer, causing an error, e.g. the line 'if (*p)' will cause an error if p has not been set yet
    - So use 'if (p && *p)', which first tests 'if p', and only if true (p is not null) will it proceed to the rest of the if test, i.e. try to dereference the pointer *p
- **Bitwise**
  - `~0x41 --> 0xBE`
    `~01000001₂ --> 10111110₂`
  - `~0x00 --> 0xFF`
    `~00000000₂ --> 11111111₂`
  - `0x69 & 0x55 --> 0x41`
    `01101001₂ & 01010101₂ --> 01000001₂`
  - `0x69 | 0x55 --> 0x7D`
    `01101001₂ | 01010101₂ --> 01111101₂`

# Logical vs Bitwise Operations in C (3)

- **Example code:**
  - int x,y;

    …

    if (x) { … do1(); …}

    if (!y) { … do2(); …}

This is shorthand for 'if (x!=0) {…}', So do1() is called only if x is non-zero,

- i.e. if x==1, then do1() is called,
- if x==137, do(1) is called, …
- if x==0, then do(1) is not called.

do2() is called only if y is zero, i.e. This is equivalent to 'if (y==0) {…}'

- i.e. if y==0, then do2() is called,
- if y==1, then do2() is not called,
- if y==137, then do2() is not called…

# Shift Operations

**Useful for multiplication and division by powers of 2**

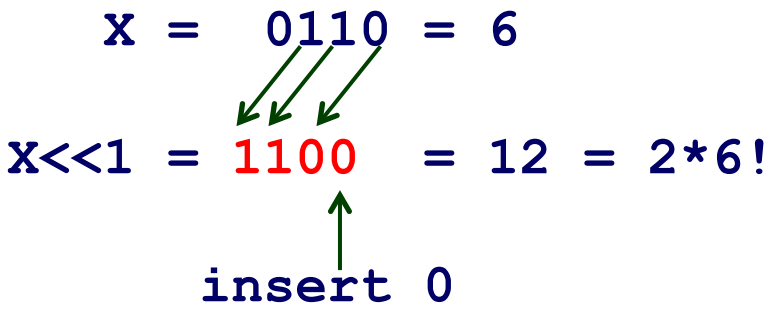
**Left Shift: `x << y`**

- Shift bit-vector `x` left `y` positions
  - Throw away extra bits on left
  - Fill with 0's on right

**Right Shift: `x >> y`**

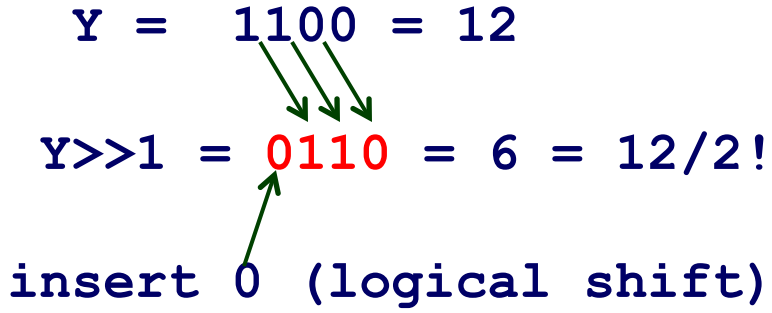
- Shift bit-vector `x` right `y` positions
  - Throw away extra bits on right
- Logical shift
  - Fill with 0's on left
- Arithmetic shift
  - Replicate most significant bit on left
  - Useful with two's complement integer representation

Example #1: x << 1, left-shift by 1 bit




*So x << 1 = Multiplication by two!*

Example #2: y >> 1, right-shift by 1 bit




*So y >> 1 = Division by two!*

# Shift Operations

## Useful for multiplication and division by powers of 2

## Left Shift: `x << y`

- Shift bit-vector `x` left `y` positions
  - Throw away extra bits on left
  - Fill with 0's on right

## Right Shift: `x >> y`

- Shift bit-vector `x` right `y` positions
  - Throw away extra bits on right
- Logical shift
  - Fill with 0's on left
- Arithmetic shift
  - Replicate most significant bit on left
  - Useful with two's complement integer representation

| Argument x | 01100010 |
|---|---|
| << 3 | 00010*000* |
| Log. >> 2 | *00*011000 |
| Arith. >> 2 | *00*011000 |

| Argument x | 10100010 |
|---|---|
| << 3 | 00010*000* |
| Log. >> 2 | *00*101000 |
| Arith. >> 2 | *11*101000 |

# Shift Operations in C

- If your word x is signed, then it is not defined in C whether x >> y will do an arithmetic or logical right shift on x
  - However, most implementations assume that a right shift on a signed quantity is an arithmetic right shift.
  - Example: If x is declared as an int, then ints are signed by default, so x >> y should do an arithmetic right shift on x by y bits
- If your word is unsigned, then x>>y will do a logical right shift on x
  - Example: If x is declared as an unsigned int, then x >> y will do a logical right shift on x by y bits.

# Encoding Integers

- Signed and unsigned integers
- Let's start with unsigned integers, assuming only 4 bits (16 values):

$$\text{Decimal} = \sum_{i=0}^{3} b_i * 2^i$$

bit in word corresponding to i'th power of 2

For a w-bit word,

$$\text{Decimal} = \sum_{i=0}^{w-1} b_i * 2^i$$

- Book uses B2U for "Binary To Unsigned integer"

| Unsigned 4-bit Integers | | | | Decimal |
|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 15 |
| 1 | 1 | 1 | 0 | 14 |
| 1 | 1 | 0 | 1 | 13 |
| 1 | 1 | 0 | 0 | 12 |
| 1 | 0 | 1 | 1 | 11 |
| 1 | 0 | 1 | 0 | 10 |
| 1 | 0 | 0 | 1 | 9 |
| 1 | 0 | 0 | 0 | 8 |
| 0 | 1 | 1 | 1 | 7 |
| 0 | 1 | 1 | 0 | 6 |
| 0 | 1 | 0 | 1 | 5 |
| 0 | 1 | 0 | 0 | 4 |
| 0 | 0 | 1 | 1 | 3 |
| 0 | 0 | 1 | 0 | 2 |
| 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 0 | 0 | 0 |
| $2^3$ | $2^2$ | $2^1$ | $2^0$ | |

# Representing Negative Integers?

- **Simplest approach is to use the most significant bit as the sign bit: '1' = negative, '0' = positive**
  - **e.g. 0100 = +4, while 1100 = -4**
  - **This is called Sign-Magnitude**
  - **Unfortunately, this means we have both**
    - **a Positive Zero (0000 = +0) and**
    - **a Negative Zero (1000 = -0) !!!**
  - **This complicates integer arithmetic**
- **Another encoding approach: One's complement**

  For a w-bit word,

$$\text{Decimal} = -b_{w-1} * (2^{w-1} - 1) + \sum_{i=0}^{w-2} b_i * 2^i$$

 Also called B2O for "Binary To One's Complement Integer"

# Two's Complement

- Two's complement encodes signed integers (positive and negative)
- Easy to implement integer arithmetic with two's complement – unique zero
- Most significant (MS) bit is sign bit with weight $-(2^{w-1})$, i.e. -8 when w=4 bits

**Example: '1101'**

**$= 1*(-2^3) + 1*2^2 + 0*2^1 + 1*2^0 = -3$**

'1000' = -8, '1111' = -1 (not 15), zero is unique & zero

| Signed 4-bit Integers | | | | Decimal |
|---|---|---|---|---|
| **0** | **1** | **1** | **1** | 7 |
| 0 | 1 | 1 | 0 | 6 |
| 0 | 1 | 0 | 1 | 5 |
| 0 | 1 | 0 | 0 | 4 |
| 0 | 0 | 1 | 1 | 3 |
| 0 | 0 | 1 | 0 | 2 |
| 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | -1 |
| 1 | 1 | 1 | 0 | -2 |
| 1 | 1 | 0 | 1 | -3 |
| 1 | 1 | 0 | 0 | -4 |
| 1 | 0 | 1 | 1 | -5 |
| 1 | 0 | 1 | 0 | -6 |
| 1 | 0 | 0 | 1 | -7 |
| 1 | 0 | 0 | 0 | -8 |
| **$sign*(-2^3)$** | **$2^2$** | **$2^1$** | **$2^0$** | |

# Two's Complement

- Asymmetric range from -8 to +7
  - one more negative # than positive # !
- MS bit indicates sign
  - 1=>negative, 0=>non-negative

For a w-bit word,

$$\text{Decimal} = -b_{w-1} * 2^{w-1} \quad \text{// MS sign Bit}$$
$$+ \sum_{i=0}^{w-2} b_i * 2^i \quad \text{// LS Bits}$$

| Signed 4-bit Integers | | | | Decimal |
|---|---|---|---|---|
| **0** | **1** | **1** | **1** | 7 |
| 0 | 1 | 1 | 0 | 6 |
| 0 | 1 | 0 | 1 | 5 |
| 0 | 1 | 0 | 0 | 4 |
| 0 | 0 | 1 | 1 | 3 |
| 0 | 0 | 1 | 0 | 2 |
| 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | -1 |
| 1 | 1 | 1 | 0 | -2 |
| 1 | 1 | 0 | 1 | -3 |
| 1 | 1 | 0 | 0 | -4 |
| 1 | 0 | 1 | 1 | -5 |
| 1 | 0 | 1 | 0 | -6 |
| 1 | 0 | 0 | 1 | -7 |
| 1 | 0 | 0 | 0 | -8 |
| **sign*(-$2^3$)** | **$2^2$** | **$2^1$** | **$2^0$** | |

Book uses B2T for "Binary To Two's Complement Integer"

# Encoding Integers

## Unsigned

$$B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

## Two's Complement

$$B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

```
short int x =  15213;
short int y = -15213;
```

Most Significant Bit is in essence the Sign Bit

- C `short` 2 bytes long

| | Decimal | Hex | Binary |
|---|---|---|---|
| x | 15213 | 3B 6D | 00111011 01101101 |
| y | -15213 | **C4 93** | 11000100 10010011 |

- For 2's complement, most significant bit indicates sign
  - 0 for nonnegative
  - 1 for negative
- Shortcut to get –x given x: complement x and add 1
  - i.e. -x = ~x + 1 in two's complement

# Encoding Example (Cont.)

```
x =      15213: 00111011 01101101
y =     -15213: 11000100 10010011
```

| Weight | 15213 | | -15213 | |
|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 |
| 2 | 0 | 0 | 1 | 2 |
| 4 | 1 | 4 | 0 | 0 |
| 8 | 1 | 8 | 0 | 0 |
| 16 | 0 | 0 | 1 | 16 |
| 32 | 1 | 32 | 0 | 0 |
| 64 | 1 | 64 | 0 | 0 |
| 128 | 0 | 0 | 1 | 128 |
| 256 | 1 | 256 | 0 | 0 |
| 512 | 1 | 512 | 0 | 0 |
| 1024 | 0 | 0 | 1 | 1024 |
| 2048 | 1 | 2048 | 0 | 0 |
| 4096 | 1 | 4096 | 0 | 0 |
| 8192 | 1 | 8192 | 0 | 0 |
| 16384 | 0 | 0 | 1 | 16384 |
| -32768 | 0 | 0 | 1 | -32768 |
| **Sum** | | **15213** | | **-15213** |

# Numeric Ranges

## Unsigned Values

- *UMin* = 0
  **000…0**
- *UMax* = $2^w - 1$
  **111…1**

## Two's Complement Values

- *TMin* = $-2^{w-1}$
  **100…0**
- *TMax* = $2^{w-1} - 1$
  **011…1**

## Other Values

- Minus 1
  **111…1**

## Values for *W* = 16

| | Decimal | Hex | Binary |
|---|---|---|---|
| UMax | 65535 | FF FF | 11111111 11111111 |
| TMax | 32767 | 7F FF | 01111111 11111111 |
| TMin | -32768 | 80 00 | 10000000 00000000 |
| -1 | -1 | FF FF | 11111111 11111111 |
| 0 | 0 | 00 00 | 00000000 00000000 |

unsigned

two's complement

both

# Values for Different Word Sizes

| | W | | | |
|---|---|---|---|---|
| | 8 | 16 | 32 | 64 |
| UMax | 255 | 65,535 | 4,294,967,295 | 18,446,744,073,709,551,615 |
| TMax | 127 | 32,767 | 2,147,483,647 | 9,223,372,036,854,775,807 |
| TMin | -128 | -32,768 | -2,147,483,648 | -9,223,372,036,854,775,808 |

## Observations

- $|TMin| = TMax + 1$
  - Asymmetric range
- $UMax = 2 * TMax + 1$

## C Programming

- `#include <limits.h>`
  - K&R App. B11
- Declares constants, e.g.,
  - `ULONG_MAX`
  - `LONG_MAX`
  - `LONG_MIN`
- Values platform-specific

# Unsigned Integer Addition

- **Example: 4-bit addition of 6 + 7**
  - Just as for decimal, there is a carry, e.g. 1 added to 1 causes a carry of a 1 to the next column

| Decimal | Unsigned Binary |
|---|---|
| *1* (carry)<br>06<br>+07<br>13 | *110* (carry)<br>0110<br>+0111<br>1101 |

- **Example: 13 + 5**
  - Note that the last carry = 1 which shows that there is an overflow

| Decimal | Unsigned Binary |
|---|---|
| *0* (carry)<br>13<br>+05<br>18 | *1101* (carry)<br>1101<br>+0101<br>10010 = 2!<br>(ignoring carry) |

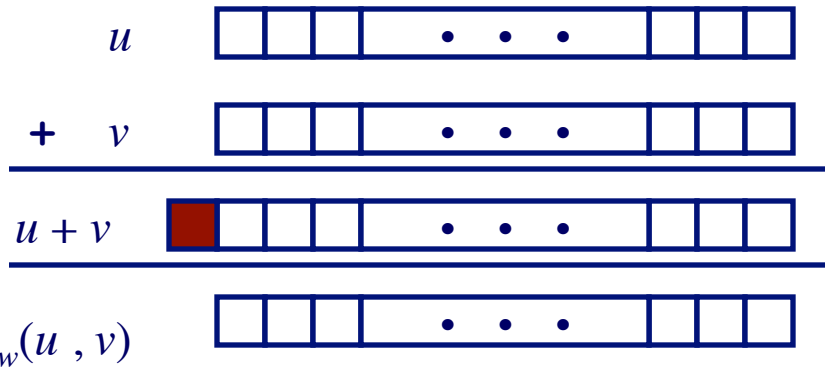
- **How to detect overflow?**
  - See if the MS/final carry bit = 1

# Unsigned Modular Addition

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits




$\text{UAdd}_w(u , v)$

## Standard Addition Function

- **Ignores carry output**

## Implements Modular Arithmetic

$s = \text{UAdd}_w(u , v) = (u + v) \bmod 2^w$

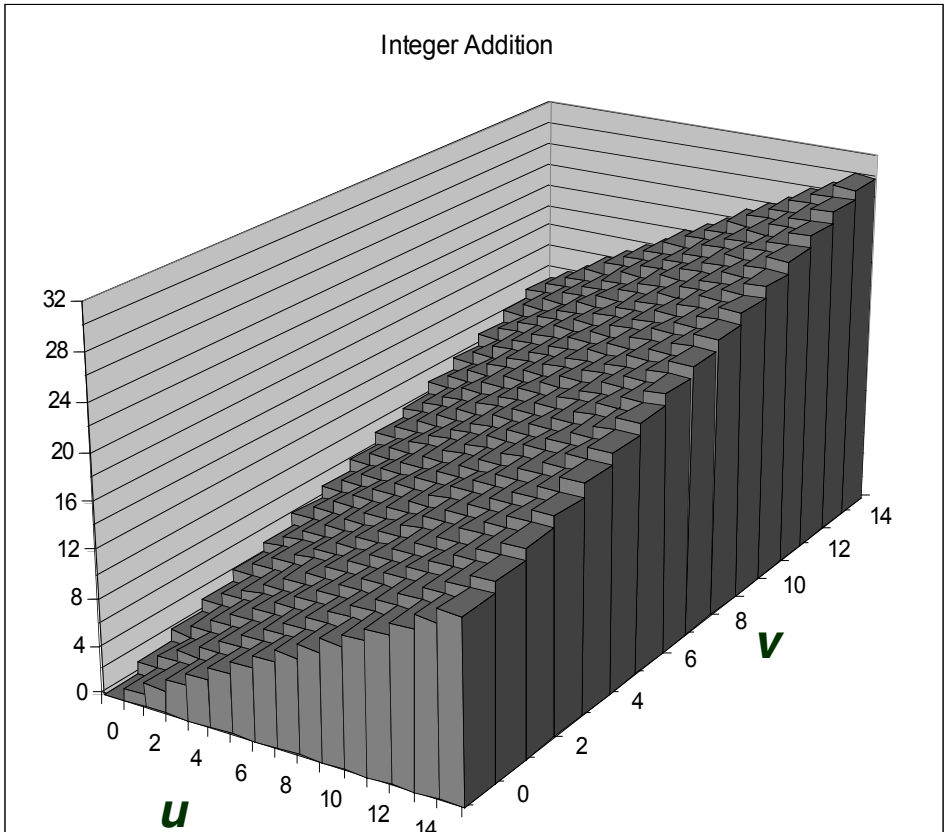
$$UAdd_w(u,v) = \begin{cases} u+v & u+v < 2^w \\ u+v-2^w & u+v \geq 2^w \end{cases}$$

# Visualizing Ideal Integer Addition

## Integer Addition

- 4-bit integers *u*, *v*
- Compute true sum $\text{Add}_4(u, v)$
- Values increase linearly with *u* and *v*
- Forms planar surface
- With 4-bit integers, the true sum could require 5 bits

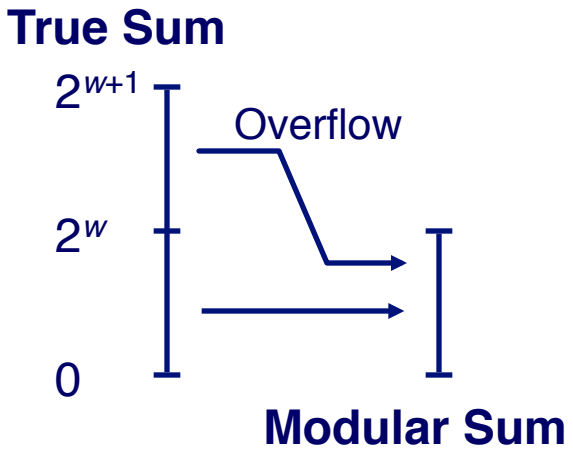
$\text{Add}_4(u, v)$

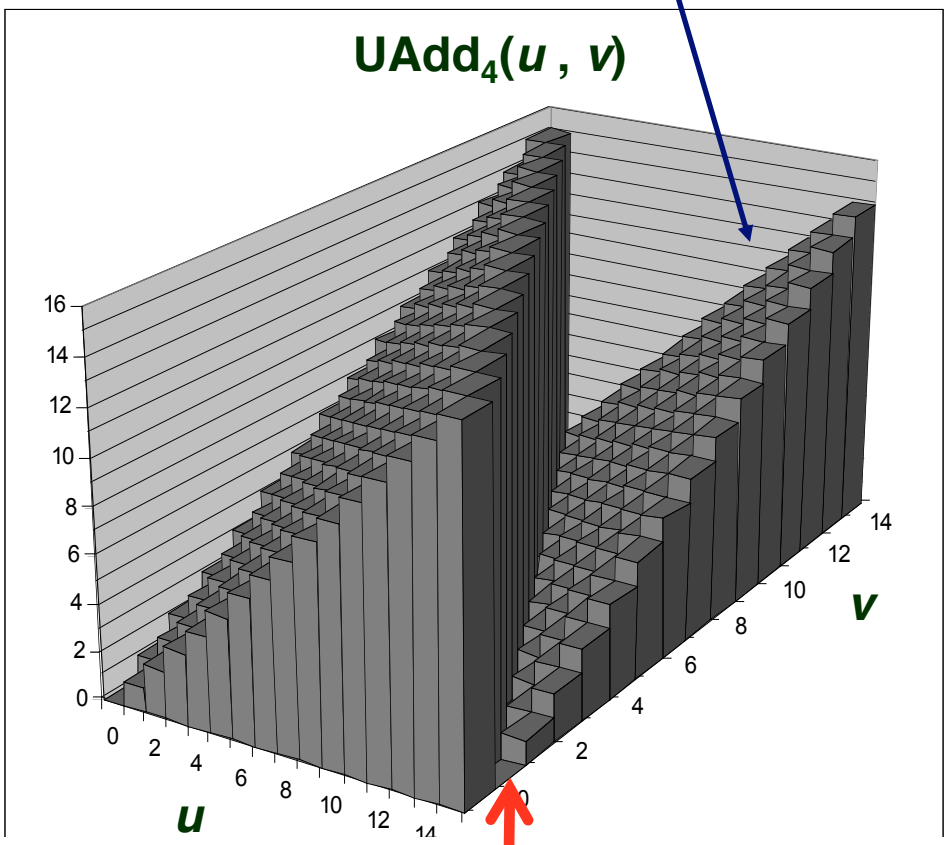


# Visualizing Unsigned Modular Addition

## Wraps Around

- If true sum ≥ $2^w$
- At most once




**Example: '1111' + '0001' = '0000'!**

# Two's Complement Addition

- Similar to unsigned addition with the carry rules
  - 1 added to 1 causes a carry of a 1 to the next column
- Note, here the final carry does not necessarily indicate overflow
  - In all examples, just the 4-bit result is correct, regardless of the carry
- In Two's complement, subtraction is same operation as addition!

| Decimal | Signed Binary |
|---|---|
| −2<br>+3<br>1 | *1110* (carry)<br>1110<br>+0011<br>0001 |

| Decimal | Signed Binary |
|---|---|
| −5<br>+3<br>−2 | *011* (carry)<br>1011<br>+0011<br>1110 |

| Decimal | Signed Binary |
|---|---|
| −4<br>−3<br>−7 | *1100* (carry)<br>1100<br>+1101<br>1001 |

# Two's Complement Overflow

- **Example: 6 + 3**
  - Overflows the maximum positive value for 4-bit two's complement (+7)
- **Example: -7 – 3**
  - Overflows the maximum negative value for 4-bit two's complement (-8)
- **How to detect overflow?**
  - If the sign of the operands are the same and the sum is the *opposite* sign, then there is overflow in two's complement signed addition
  - In both examples, both operands are + or -, but the sum is the opposite sign

| Decimal | Signed Binary |
|---|---|
| 6<br>+3<br>9 | *110* (carry)<br>0110<br>+0011<br>1001 =-7 (Two's C) |

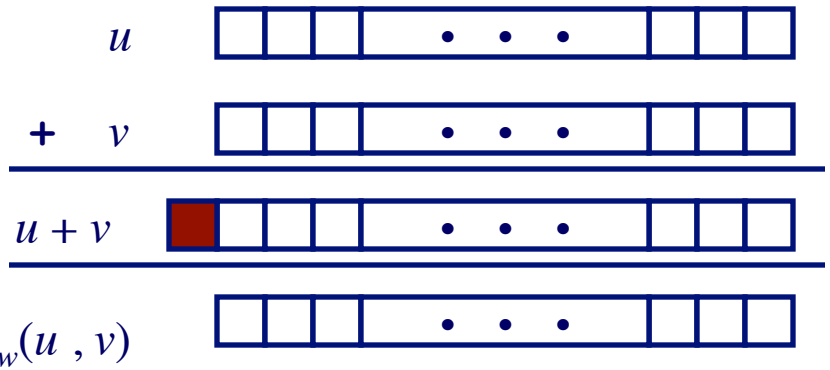
| Decimal | Signed Binary |
|---|---|
| -7<br>-3<br>-10 | *1001* (carry)<br>1001<br>+1101<br>0110 =+6 (Two's C) |

# Two's Complement Modular Addition

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits




## TAdd and UAdd have Identical Bit-Level Behavior

- **Signed vs. unsigned addition in C:**

```
int s, t, u, v;
s = (int) ((unsigned) u + (unsigned) v);
t = u + v
```

- **Will give `s == t`**

**Unsigned add UAdd**

**Signed add TAdd**

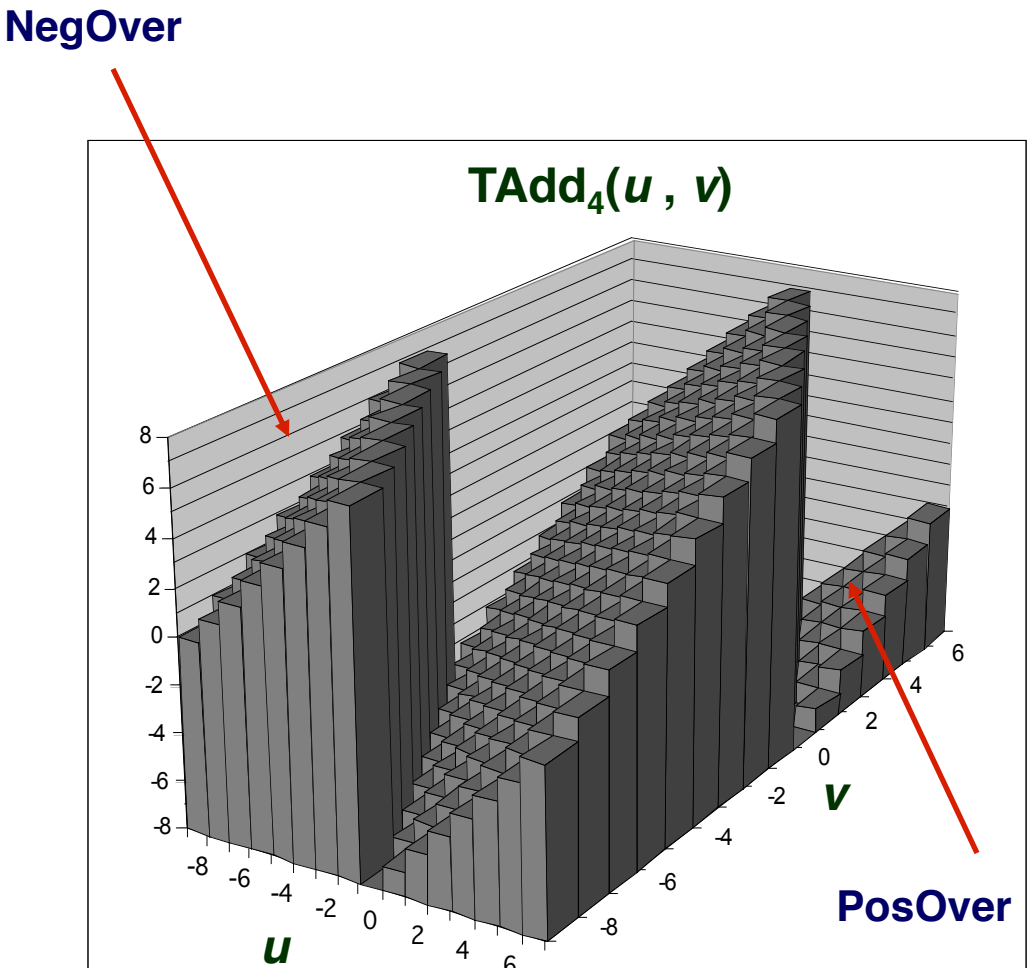
# Visualizing Two's Complement Signed Addition

## Values

- 4-bit two's comp.
- Range from -8 to +7

## Wraps Around

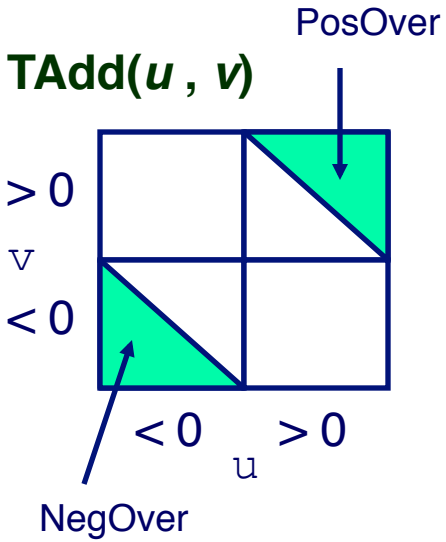
- If sum ≥ $2^{w-1}$
  - Becomes negative
  - At most once
- If sum < $-2^{w-1}$
  - Becomes positive
  - At most once

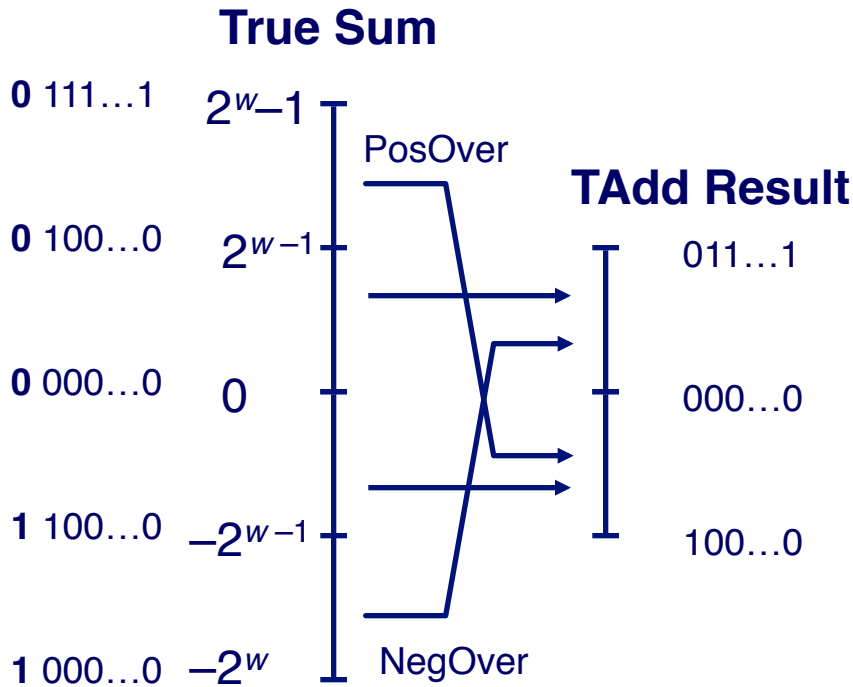


# Characterizing TAdd

## Functionality

- True sum requires $w$+1 bits
- Drop off carry bit
- Treat remaining bits as 2's comp. integer




$$TAdd_w(u,v) = \begin{cases} u+v+2^w & u+v < TMin_w \quad \textbf{(NegOver)} \\ u+v & TMin_w \le u+v \le TMax_w \\ u+v-2^w & TMax_w < u+v \quad \textbf{(PosOver)} \end{cases}$$

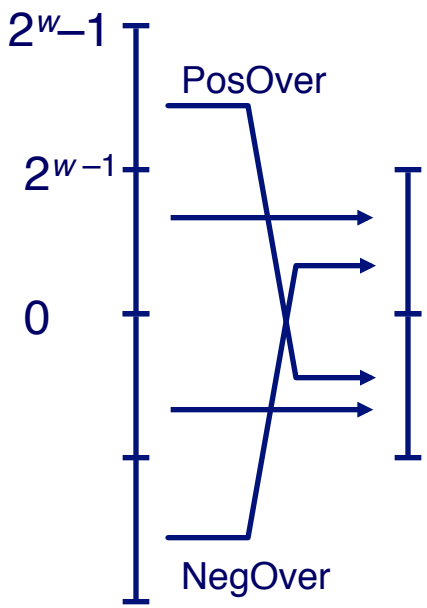
– 26 – Lesson: as sum reaches large #'s (>billion for 32-bit ints), get strange results

# Detecting 2's Comp. Overflow

## Task

- **Given** $s = \text{TAdd}_w(u, v)$
- **Determine if** $s = \text{Add}_w(u, v)$
- **Example**

```
int s, u, v;
s = u + v;
```




## Claim

- **Overflow iff either:**

  $u, v < 0, s \geq 0$ **(NegOver)**

  $u, v \geq 0, s < 0$ **(PosOver)**

# Supplementary Slides

# C Puzzles - practice

- Taken from old exams
- Assume machine with 32 bit word size, two's complement integers
- For each of the following C expressions, either:
  - Argue that is true for all argument values
  - Give example where not true

**Initialization**

```
int x = foo();
int y = bar();
unsigned ux = x;
unsigned uy = y;
```

- `x < 0` ⇒ `((x*2) < 0)`
- `ux >= 0`
- `x & 7 == 7` ⇒ `(x<<30) < 0`
- `ux > -1`
- `x > y` ⇒ `-x < -y`
- `x * x >= 0`
- `x > 0 && y > 0` ⇒ `x + y > 0`
- `x >= 0` ⇒ `-x <= 0`
- `x <= 0` ⇒ `-x >= 0`

# C Puzzle Answers

- Assume machine with 32 bit word size, two's comp. integers
- *TMin* makes a good counterexample in many cases

| | | | |
|---|---|---|---|
| `x < 0` | ⇒ | `((x*2) < 0)` | **False:** *TMin* |
| `ux >= 0` | | | **True:** 0 = *UMin* |
| `x & 7 == 7` | ⇒ | `(x<<30) < 0` | **True:** $x_1 = 1$ |
| `ux > -1` | | | **False:** 0 |
| `x > y` | ⇒ | `-x < -y` | **False:** `-1`, *TMin* |
| `x * x >= 0` | | | **False:** `30426` |
| `x > 0 && y > 0` | ⇒ | `x + y > 0` | **False:** *TMax*, *TMax* |
| `x >= 0` | ⇒ | `-x <= 0` | **True:** $-TMax < 0$ |
| `x <= 0` | ⇒ | `-x >= 0` | **False:** *TMin* |

# Two's Complement Shortcuts: Negation

## Claim: Following Holds for 2's Complement

```
~x + 1 == -x
```

## Complement

- **Observation:** `~x + x == 1111…11`$_2$ `== -1`

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| `x` | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 |
| `+ ~x` | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 |
| `-1` | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

## Increment

- `~x + x + (-x + 1)` `== -1 + (-x + 1)`
- `~x + 1` `==` `-x`

## Warning: Be cautious treating `int`'s as integers

- **OK here**

# Negation == Complement then Increment

x = 15213

| | Decimal | Hex | Binary |
|---|---|---|---|
| x | 15213 | 3B 6D | 00111011 01101101 |
| ~x | -15214 | C4 92 | 11000100 10010010 |
| ~x+1 | -15213 | C4 93 | 11000100 10010011 |
| y | -15213 | C4 93 | 11000100 10010011 |

0

| | Decimal | Hex | Binary |
|---|---|---|---|
| 0 | 0 | 00 00 | 00000000 00000000 |
| ~0 | -1 | FF FF | 11111111 11111111 |
| ~0+1 | 0 | 00 00 | 00000000 00000000 |

# Two's Complement Shortcuts: Negation (2)

- Another way to calculate –x from x in Two's Complement:
  - Write decimal x in binary.
  - start at the LS bit of x, copying all zeros to y until first 1 is reached; copy that 1, and then flip all the remaining bits and copy them to y. Then y=-x.
  - For example: 60 = $00111100_2$. To calculate -60, copy all zeros to the right of the first 1, so y = $100_2$. Then, flip all other bits and copy to y, so y = $11000100_2$ = -128 + 64 + 4 = -60.