Integer Representation

CMPU 224 – Computer Organization
Jason Waterman

Review

- Bit-Level Operations in C
  - Operations & (AND), | (OR), ~ (NOT), ^ (XOR) available in C
  - Apply to any “integral” data type (long, int, short, and char)
  - View arguments as bit vectors
  - Arguments applied bit-wise
- Logical operators
  - &&, ||, ! (logical AND, OR, NOT)
- No Boolean type in C
  - View 0 as “False”
  - Anything nonzero as “True”
  - Logical operators always return 0 or 1 (for FALSE or TRUE)
  - Early termination

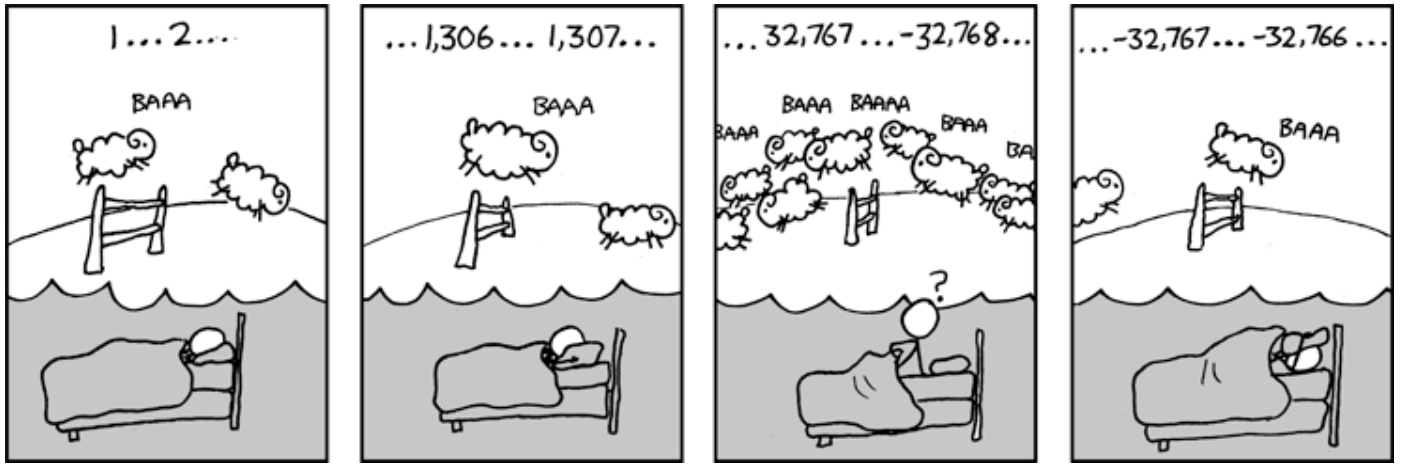
Shift Operations in C

- 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
- Undefined Behavior
  - Shift amount < 0 or ≥ data size

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

Today



Source: https://xkcd.com/571/

Representing integers

- Given *n* bits to store an integer, we can represent 2^n different values
- If we just care about non-negative (aka **unsigned**) integers, we can easily store the values

  0, 1, 2, …, 2^n-1
  - E.g., for 4 bits
    - 0x2 = 2
    - 0xB = 11
    - 0xF = 15 = 2^4-1

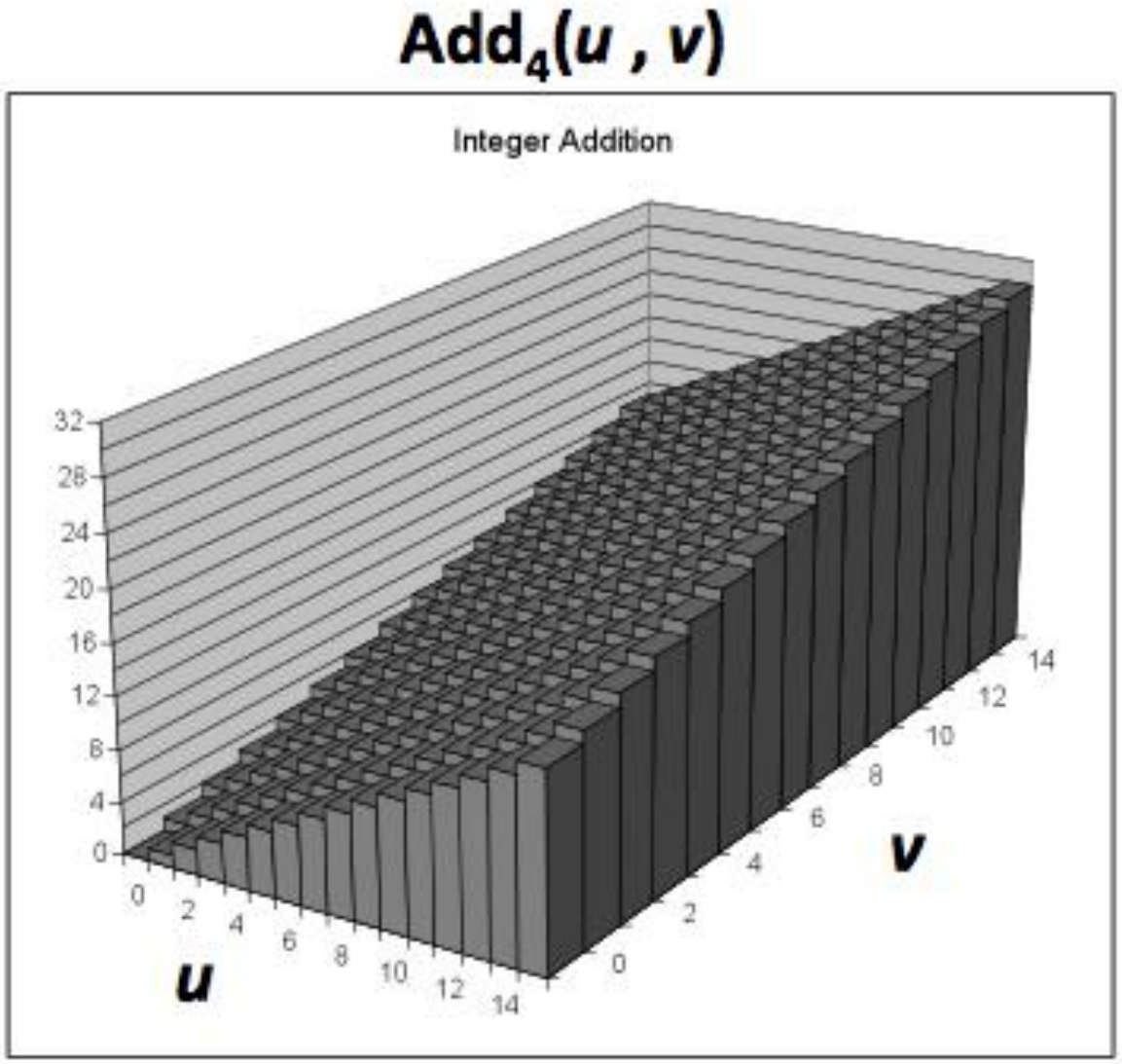
Integer overflow

- With n bits, we can represent values 0, 1, 2, …, 2n-1
- Overflow occurs when we have a result that doesn't fit in the n bits
  - E.g., using 4 bits: 0xF + 0x1

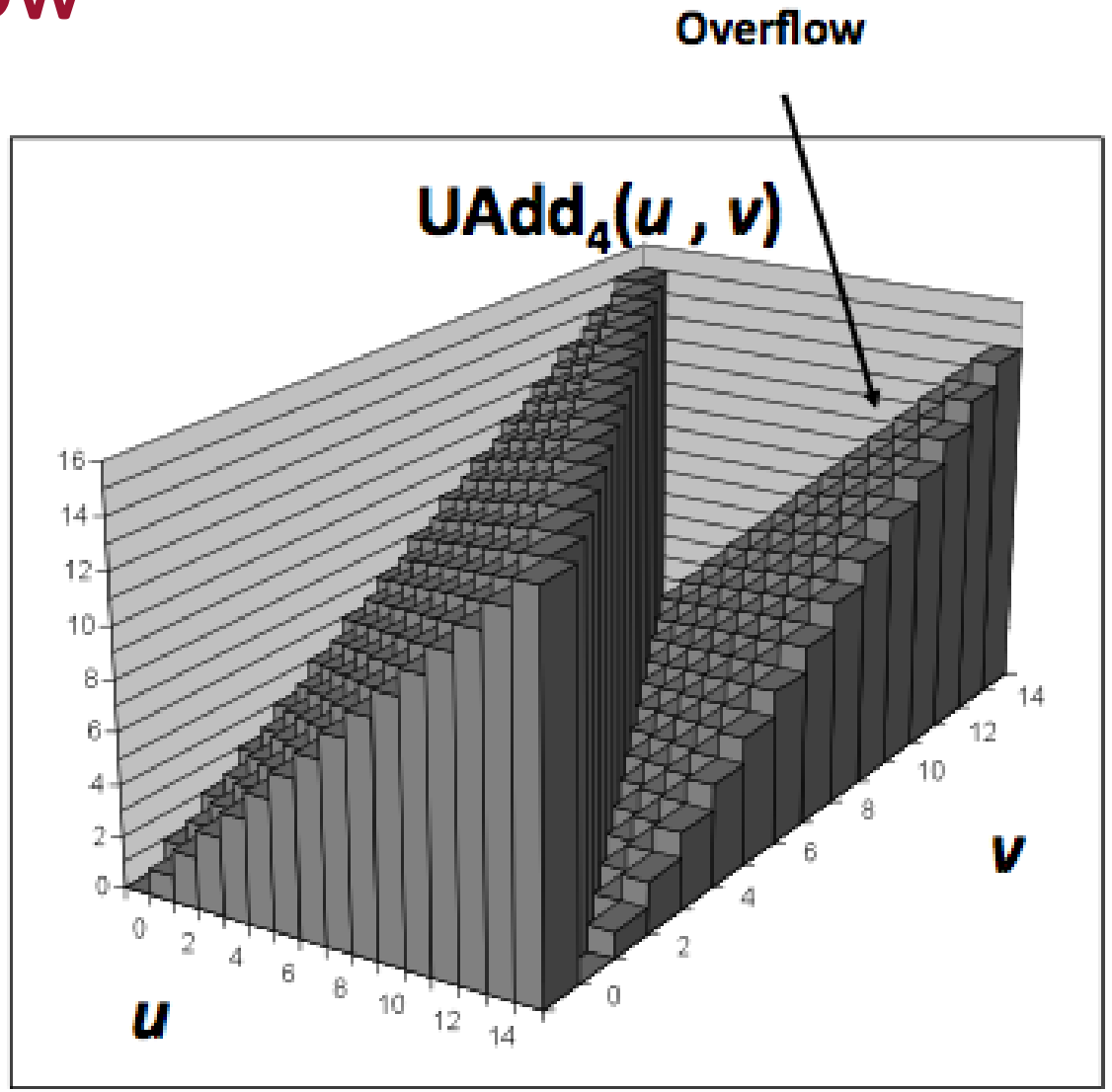
$$\begin{array}{rr} 0xF = & 1111 \\ 0x1 = & 0001 \\ \hline \end{array}$$

0xF + 0x1 = 0x0 Overflow!!

Integer overflow



Integer overflow

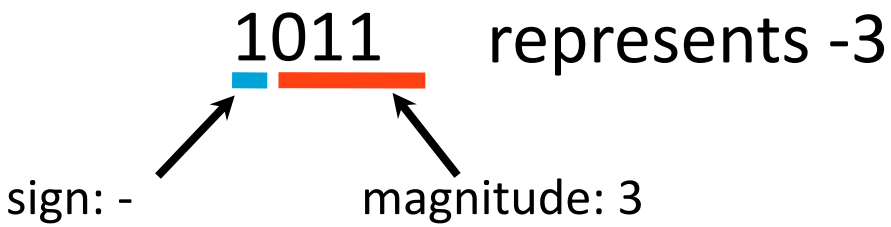


Representing negative integers

- We have seen how to represent **unsigned integers** (i.e., non-negative integers) as **unsigned binary** numbers
  - Addition works as we expect it to
  - Every number between 0 and 2^w-1 has a unique encoding as a w-bit value
- How do we represent negative integers?
- Three common encodings:
  - Sign and magnitude
  - Ones' complement
  - Two's complement

Sign and magnitude

- Use one bit to represent sign, remaining bits represent magnitude
- With *n* bits, have *n*-1 bits for magnitude
  - E.g., with 4 bits, can represent integers
    -7, -6, …, -1, 0, 1, …, 6, 7



Properties of sign and magnitude

- Straight-forward and intuitive
- Two different representations of zero!
  - E.g., using 4 bits, 1000 and 0000 both represent zero!
- Arithmetic operations need different implementations for signed and unsigned numbers
  - E.g., addition, using 4 bits
    - unsigned: 0001 + 1001 = 1 + 9 = 10 = 1010
    - sign and magnitude: 0001 + 1001 = 1 + -1 = 0, which is not 1010

Ones' complement

- Negation is performed by performing a bitwise not (~) on the number
  - In other words, flip all the bits in the number
  - For example, using 4 bit numbers:
    - 6 = 0110
    - -6 = 1001
- Like sign and magnitude, first bit indicates whether number is negative
  - If the msb (most significant bit) is 0, treat it like an unsigned binary number
  - If the msb is 1, the number is negative, flip all the bits to see its magnitude
- Using n bits, can represent numbers 2^n-1 values
  - E.g., using 4 bits, can represent integers
    -7, -6, …, -1, 0, 1, …, 6, 7

Properties of ones' complement

- We still have two different representations of zero!
  - E.g., using 4 bits, 1111 and 0000 both represent zero!
- Same implementation of arithmetic operations for signed and unsigned!
  - E.g., addition using 4 bits
    - unsigned: 0001 + 1001 = 1 + 9 = 10 = 1010
    - ones' complement: 0001 + 1001 = 1 + -6 = -5 = 1010

Two's complement

- Take the ones' complement of the number and add one
  - **Flip all the bits and add one to the number**
- Using n bits, can represent numbers 2^n values
  - E.g., using 4 bits, can represent integers
    -8, -7, …, -1, 0, 1, …, 6, 7
  - Like sign and magnitude and ones' complement, first bit indicates whether number is negative

Two's Compliment Comparison

	Unsigned	Ones' Complement	Two's Complement
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	8	-7	-8
1001	9	-6	-7
1010	10	-5	-6
1011	11	-4	-5
1100	12	-3	-4
1101	13	-2	-3
1110	14	-1	-2
1111	15	-0	-1

Properties of two's complement

- Same implementation of arithmetic operations for signed and unsigned
  - E.g., addition, using 4 bits
    - unsigned: 0001 + 1001 = 1 + 9 = 10 = 1010
    - two's complement: 0001 + 1001 = 1 + -7 = -6 = 1010
- Only one representation of zero!
  - Simpler to implement operations
- Not symmetric around zero
  - Can represent one more negative number than positive numbers
- Most common representation of negative integers in computers

Converting to and from two's complement

- To encode a negative number in two's complement in n bits:
  - Compute out the binary notation for the absolute value using **all n bits**
  - Invert the bits and add 1
  - E.g., to encode -5 using 8 bits
    - 5 = 00000101 using all 8 bits
    - Invert the bits: 11111010
    - Add one: 11111010 + 1 = 11111011
    - -5 encoded in two's complement using 8 bits is 11111011
- To decode two's complement:
  - If the first bit is 0 then number is positive
  - If the first bit is 1, then number is negative:
    - invert bits and add 1
    - Gives you the magnitude of the number

Two's Complement Interpretation

- You can interpret a two's complement number as having a "negative weighting" in the MSB

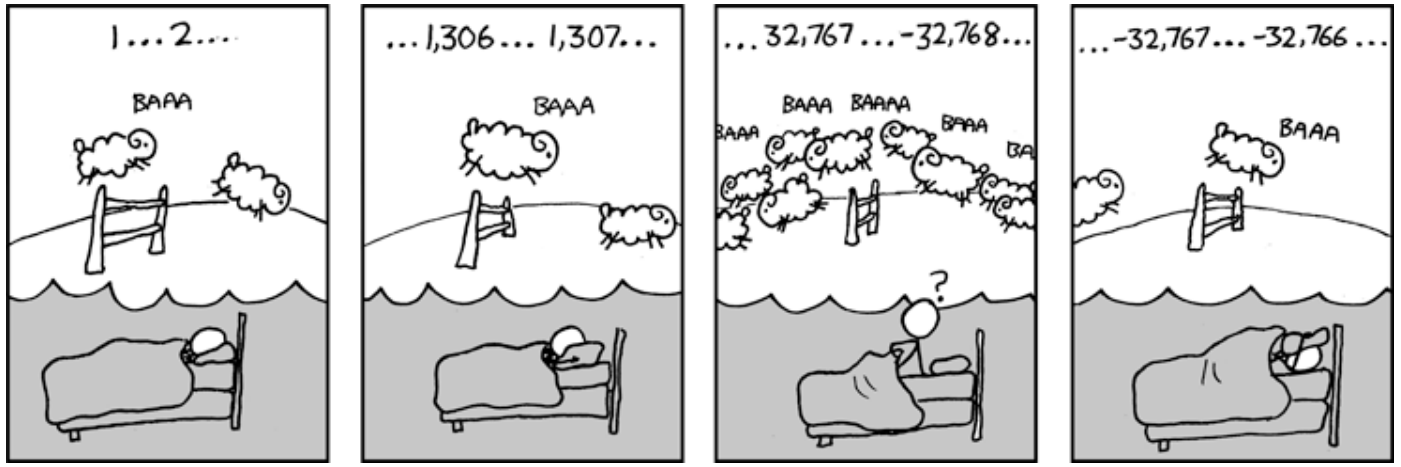
$-2^3 = -8$	$2^2 = 4$	$2^1 = 2$	$2^0 = 1$	
1	**1**	**0**	**1**	
-8	+4	+0	+1	= -3

- One cool trick for converting to two's complement
  - Flip all the bits, then starting from the LSB, flip all the ones you see (to zero) until you get to a zero. Flip that zero (to a one) and stop.
  - 00100 (4)
  - 11011 (flip all the bits)
  - 11100 (flip the rightmost 1's and the first 0)

Different representations of 4-bit numbers

Binary	Unsigned	Sign&Mag	Ones' Comp	Two's Comp
0000	0	0	0	0
0001	1	1	1	1
0010	2	2	2	2
0011	3	3	3	3
0100	4	4	4	4
0101	5	5	5	5
0110	6	6	6	6
0111	7	7	7	7
1000	8	-0	-7	-8
1001	9	-1	-6	-7
1010	10	-2	-5	-6
1011	11	-3	-4	-5
1100	12	-4	-3	-4
1101	13	-5	-2	-3
1110	14	-6	-1	-2
1111	15	-7	-0	-1

What is going on with this cartoon?



They are counting sheep using a 16-bit two's complement number!