Arithmetic and Bitwise Operations on Binary Data

CSCI 224 / ECE 317: Computer Architecture

Instructor:
Prof. Jason Fritts

Slides adapted from Bryant & O'Hallaron's slides

Boolean Algebra

- **Developed by George Boole in 19th Century**
  - Algebraic representation of logic
    - Encode “True” as 1 and “False” as 0

And

- **A&B = 1 when both A=1 and B=1**

&	0	1
0	0	0
1	0	1

Or

- **A|B = 1 when either A=1 or B=1**

|	0	1
0	0	1
1	1	1

Not

- **~A = 1 when A=0**

~	
0	1
1	0

Exclusive-Or (Xor)

- **A^B = 1 when either A=1 or B=1, but not both**

^	0	1
0	0	1
1	1	0

General Boolean Algebras

- **Operate on Bit Vectors**
  - Operations applied bitwise
  - Bitwise-AND operator: &
  - Bitwise-NOR operator: |
  - Bitwise-XOR operator: ^
  - Bitwise-NOT operator: ~

```
  01101001     01101001     01101001
& 01010101   | 01010101   ^ 01010101   ~ 01010101
  --------     --------     --------     --------
  01000001     01111101     00111100     10101010
```

- **All of the Properties of Boolean Algebra Apply**

Bit-Level Operations in C

- **Operations &, |, ~, ^ Available in C**
  - Apply to any "integral" data type
    - long, int, short, char, unsigned
  - View arguments as bit vectors
  - Arguments applied bit-wise

- **Examples (`char` data type):**

	in hexadecimal			***in binary***	
▪	~0x41 →	0xBE	//	$\sim 01000001_2$ →	10111110_2
▪	~0x00 →	0xFF	//	$\sim 00000000_2$ →	11111111_2
▪	0x69 & 0x55 →	0x41	//	01101001_2 & 01010101_2 →	01000001_2
▪	0x69 | 0x55 →	0x7D	//	01101001_2 | 01010101_2 →	01111101_2

Contrast: Logic Operations in C

- **Contrast to Logical Operators**
  - &&, ||, !
    - View 0 as “False”
    - Anything nonzero as “True”
    - Always return 0 or 1
    - Early termination
- **Examples (`char` data type):**
  - !0x41 → 0x00
  - !0x00 → 0x01
  - !!0x41 → 0x01
  - 0x69 && 0x55 → 0x01
  - 0x69 || 0x55 → 0x01
  - p && *p // avoids null pointer access

Shift Operations

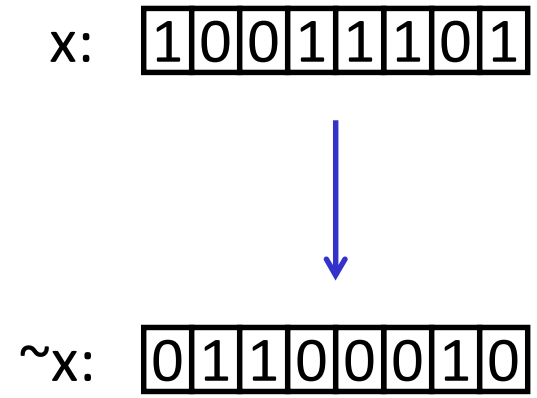
- **Left Shift: x << y**
  - Shift bit-vector x left y positions
    - Throw away extra bits on left
    - Fill with 0's on right
- **Right Shift: x >> y**
  - Shift bit-vector x right y positions
    - Throw away extra bits on right
  - Logical shift
    - Fill with 0's on left
  - Arithmetic shift
    - Replicate most significant bit on right
- **Undefined Behavior**
  - Shift amount < 0 or ≥ word size

Argument x	01100010
<< 3	00010000
Log. >> 2	00011000
Arith. >> 2	00011000

Argument x	10100010
<< 3	00010000
Log. >> 2	00101000
Arith. >> 2	11101000

Bitwise-NOT: One's Complement

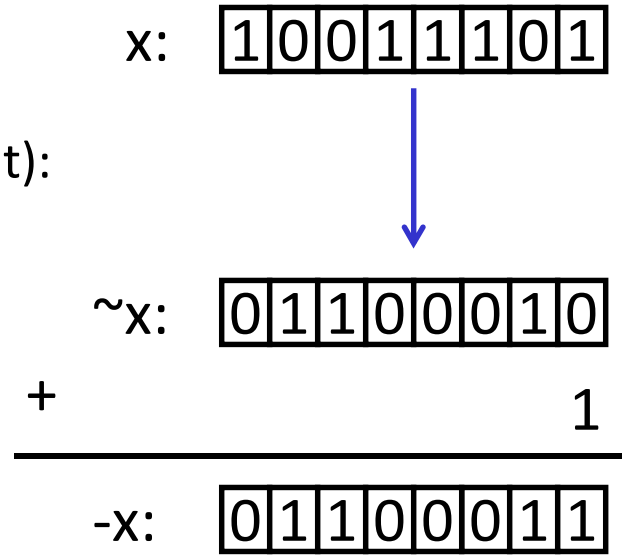
- **Bitwise-NOT operation: `~`**
  - Bitwise-NOT of **`x`** is **`~x`**
  - Flip all bits of **`x`** to compute **`~x`**
    - flip each 1 to 0
    - flip each 0 to 1
- **Complement**
  - Given **`x == 10011101`**
  - Flip bits (one's complement):



Negation: Two's Complement

- **Negate a number by taking 2's Complement**
  - Flip bits (one's complement) and add 1

    `~x + 1 == -x`
- **Negation (Two's Complement):**
  - Given `x == 10011101`
  - Flip bits (one's complement):
  - Add 1:



Complement & Increment Examples

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

x = 0

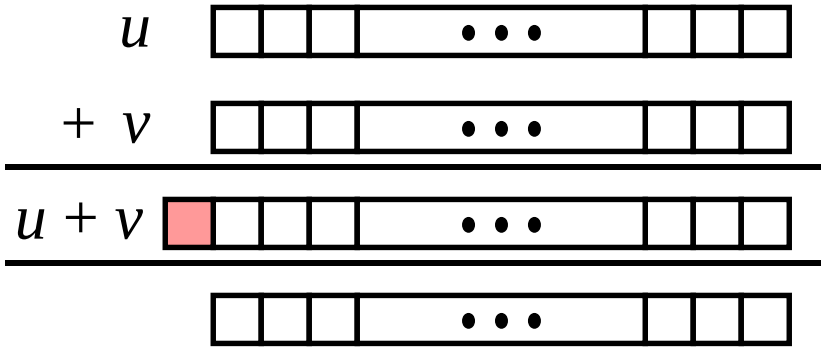
	Decimal	Hex	Binary
0	0	00 00	00000000 00000000
~0	-1	FF FF	11111111 11111111
~0+1	0	00 00	00000000 00000000

Unsigned Addition

Operands: w bits

True Sum: w+1 bits

Discard Carry: w bits



- **Addition Operation**
  - Carry output dropped at end of addition
  - Valid ONLY if true sum is within w-bit range
- **Example #1:**

		1					1		
		0	1	1	0	0	0	1	0
+		0	1	0	0	1	0	1	0
	0	1	0	1	0	1	1	0	0

98_{10}

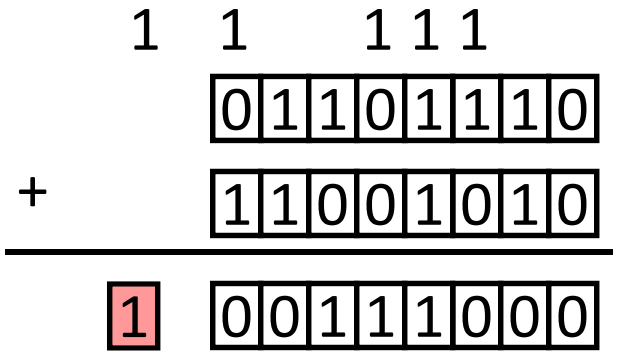
74_{10}

172_{10}

***Valid** in 8-bit unsigned range*

Unsigned Addition

- **Example #2:**



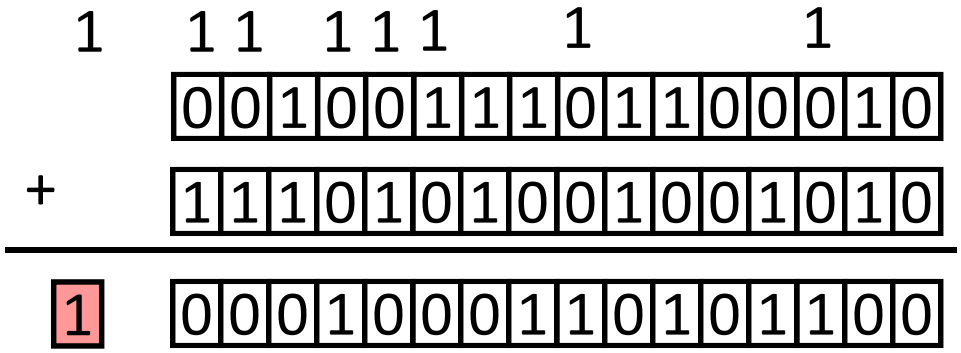
110_{10}

202_{10}

~~56_{10}~~

***Not Valid** in 8-bit unsigned range (312 is > 255)*

- **Example #3:**



10082_{10}

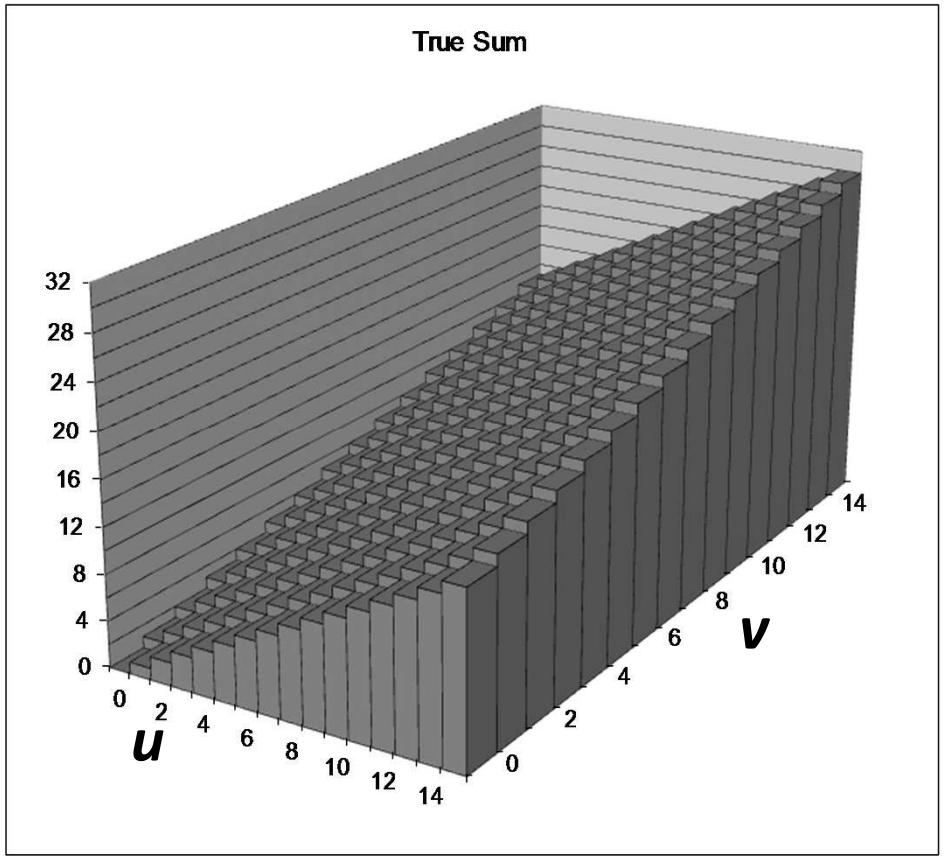
59978_{10}

~~4524_{10}~~

***Not Valid** in 16-bit unsigned range (70060 is > 65535)*

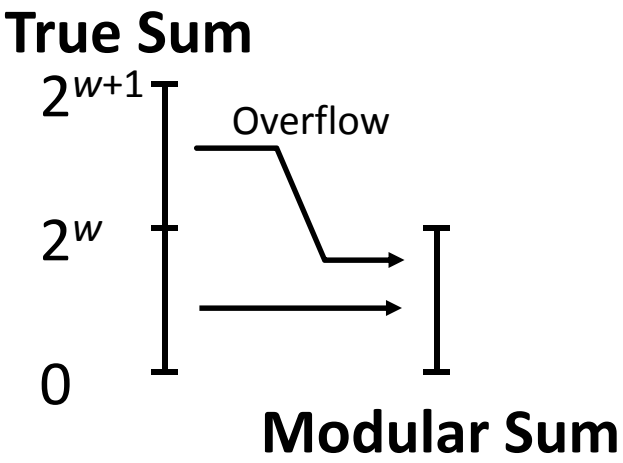
Visualizing (Mathematical) Integer Addition

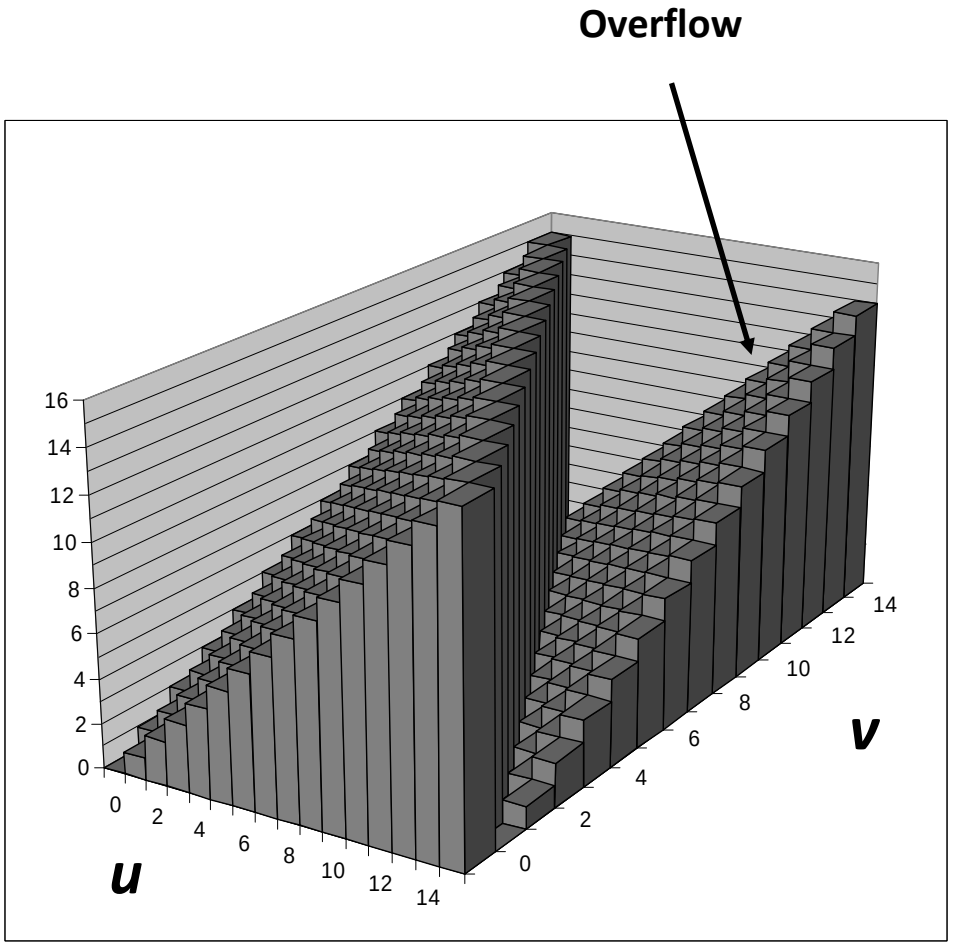
- **Integer Addition**
  - 4-bit integers *u*, *v*
  - Compute true sum
  - Values increase linearly with *u* and *v*
  - Forms planar surface



Visualizing Unsigned Addition

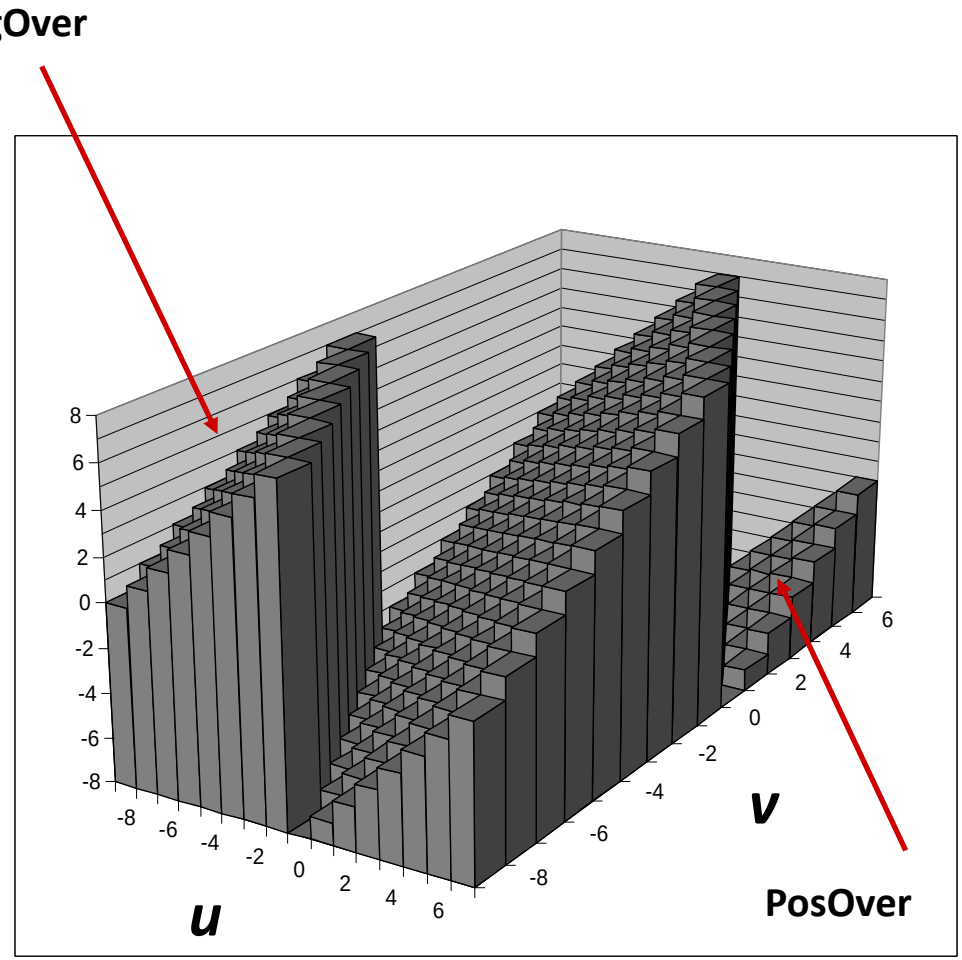
- **Wraps Around**
  - If true sum ≥ 2^w
  - At most once





Visualizing Signed Addition

- **Values**
  - 4-bit two's comp.
  - Range from -8 to +7
- **Wraps Around**
  - If sum $\geq 2^{w-1}$
    - Becomes negative
    - At most once
  - If sum $< -2^{w-1}$
    - Becomes positive
    - At most once



Signed Addition

Note: Same bytes as for Ex #1 and Ex #2 in unsigned integer addition, but now interpreted as 8-bit signed integers

- **Example #1:**

	1 1		
	0 1 1 0 0 0 1 0	98_{10}	
+	0 1 0 0 1 0 1 0	74_{10}	
0	1 0 1 0 1 1 0 0	~~-84_{10}~~	***Not Valid*** *in 8-bit signed range (172 > 127)*

- **Example #2:**

	1 1 1 1 1		
	0 1 1 0 1 1 1 0	110_{10}	
+	1 1 0 0 1 0 1 0	-54_{10}	
1	0 0 1 1 1 0 0 0	56_{10}	***Valid*** *in 8-bit signed range (-128 < 56 < 127)*

Arithmetic: Basic Rules

- **Unsigned ints, 2's complement ints are isomorphic rings: isomorphism = casting**

- **Left shift**
  - Unsigned/signed: multiplication by 2^k
  - Always logical shift

- **Right shift**
  - Unsigned: logical shift, div (division + round to zero) by 2^k
  - Signed: arithmetic shift
    - Positive numbers: div (division + round to zero) by 2^k
    - Negative numbers: div (division + round away from zero) by 2^k
      Use biasing to fix

Integer C Puzzles

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
- (x|-x)>>31 == -1
- ux >> 3 == ux/8
- x >> 3 == x/8
- x & (x-1) != 0