
Lecture 5: Arithmetic

COS / ELE 375

Computer Architecture and Organization

Princeton University
Fall 2015

Prof. David August



Binary Representation of Integers

- Two physical states: call these 1 and 0
- Collections of these bits can represent numbers

A Familiar Assignment of Meaning



Binary	Decimal
0	0
1	1
10	2
11	3
100	4
101	5
110	6
111	7
1000	8
...	...
$1\{n\}$	2^{n-1}

Size and Overflow in Unsigned Integers

Bits	Integer Range
8	0 - 255
16	0 - 65,535
32	0 - 4,294,967,295
64	0 - 18,446,744,073,709,551,615

Binary	Decimal
0	0
1	1
10	2
...	...
$1\{n\}$	$2^n - 1$

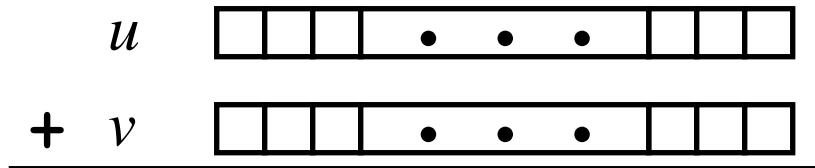
Number of bits determines **unsigned** integer range

Overflow:

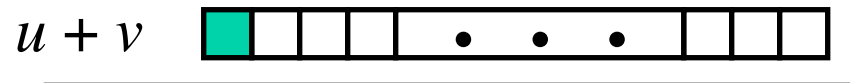
- 8-bit integer $\rightarrow 11111111_2 (= 255_{10})$
- Add 1
- What happens?

Overflow in Unsigned Addition

Operands: w bits



True Sum: $w + 1$ bits



Discard Carry: w bits $\text{UAdd}_w(u, v)$

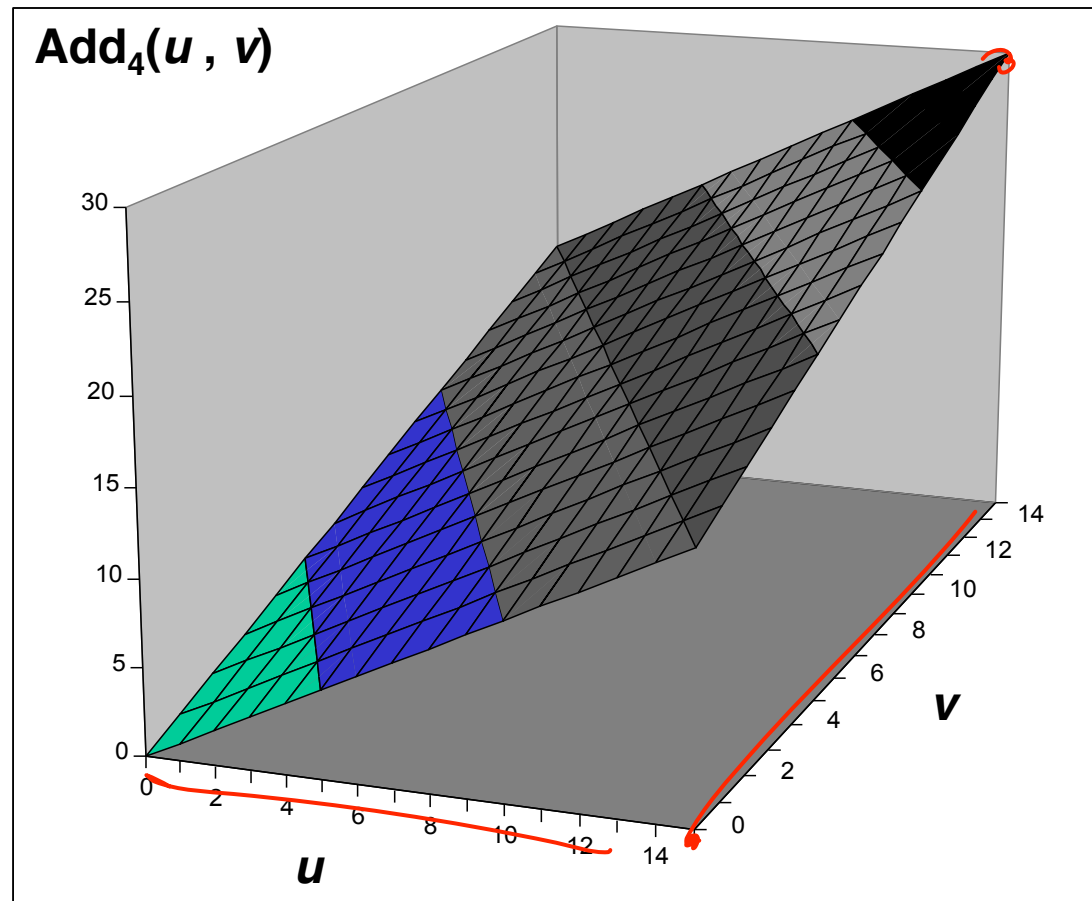


$$\text{UAdd}_w(u, v) = \begin{cases} u + v & u + v < 2^w \\ u + v - 2^w & u + v \geq 2^w \end{cases}$$

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

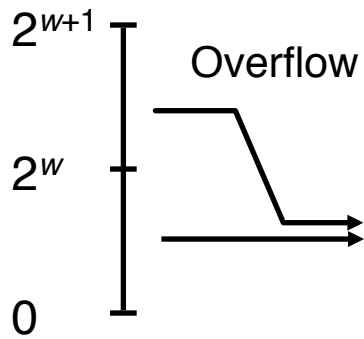
Visualization of Integer Addition

- Compute true sum $\text{Add}_4(u, v)$
- Values increase linearly with u and v
- Forms planar surface

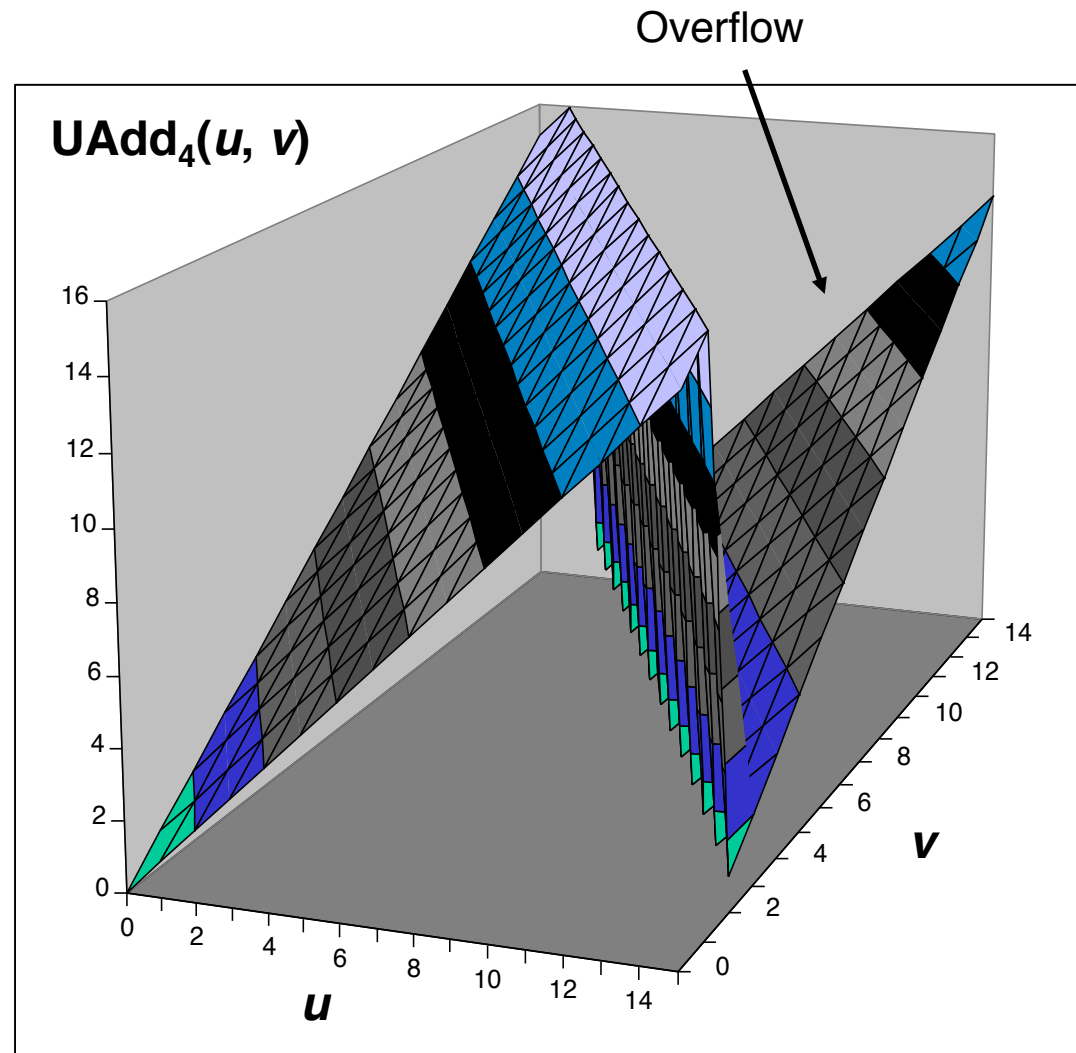


Visualization of Unsigned Addition

True Sum



Modulo Sum



Modulo Arithmetic

Modulo Addition Forms an Abelian Group

- Closed under addition
 - $0 \leq \text{UAdd}_w(u, v) \leq 2^w - 1$
- Commutative
 - $\text{UAdd}_w(u, v) = \text{UAdd}_w(v, u)$
- Associative
 - $\text{UAdd}_w(t, \text{UAdd}_w(u, v)) = \text{UAdd}_w(\text{UAdd}_w(t, u), v)$
- 0 is additive identity
 - $\text{UAdd}_w(u, 0) = u$
- Every element has additive inverse
 - Let $\text{UComp}_w(u) = 2^w - u$
 - $\text{UAdd}_w(u, \text{UComp}_w(u)) = 0$

Overflow Possibilities

1. Hardware does not detect overflow (MIPS: addu, addiu, subu)
 - Software checks if necessary
 - Reacts in manner specified
2. Hardware records overflow
 - Software decides whether or not to do anything
 - Software system / language specific
3. Hardware throws exception(interrupt) (MIPS: add, addi, sub)
 - Control jumps to predefined address for exception
 - Interrupted address is saved for possible resumption

Exception - an unscheduled procedure call

EPC - Exception PC records excepting instruction address

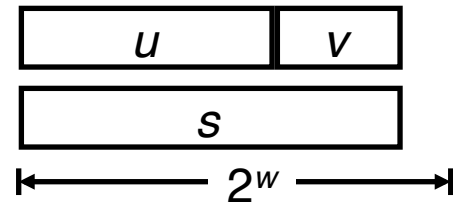
Detecting Unsigned Overflow

- Task:
 - Given $s = \text{UAdd}_w(u, v)$
 - Determine if $s = u + v$

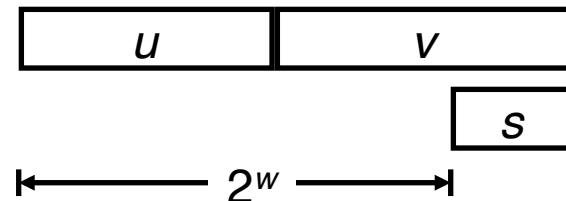
- Claim:
 - Overflow iff $s < u$
 - $\text{ovf} = (s < u)$
 - By symmetry iff $s < v$

- Proof:
 - $0 \leq v < 2^w$
 - No overflow $\Rightarrow s = u + v \geq u + 0 = u$
 - Overflow $\Rightarrow s = u + v - 2^w < u + 0 = u$

No Overflow:



Overflow:



19

KODAK 5053 TMY

20



19

19A

20

What about Negative Numbers?

Bits	Patterns
8	256
16	65,536
32	4,294,967,296
64	18,446,744,073,709,551,616

Binary	Pattern
0	1
1	2
10	3
...	...
$1\{n\}$	2^n

- We have been looking at **unsigned numbers**
- What about negative or **signed numbers**?
- Need new interpretation of bits
- Some patterns interpreted as negative numbers

Key Standard Pattern Assignments

Bit Pattern	Sign Magnitude	One's Complement	Two's Complement
000	+0	+0	0
001	+1	+1	+1
010	+2	+2	+2
011	+3	+3	+3
100	-0	-3	-4
101	-1	-2	-3
110	-2	-1	-2
111	-3	-0	-1

- Which one is best?
- Issues?
 - Balance
 - Zeros
 - Ease of operations

Most Common: Two's Complement

Bit Pattern	Two's Complement
000	0
001	+1
010	+2
011	+3
100	-4
101	-3
110	-2
111	-1

- “Invert and Add 1” to negate
- Sign Bit
- Zeros, Range
- What about arithmetic?

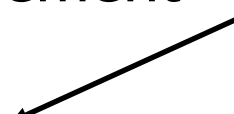
Unsigned and Two's Complement

- Unsigned Values

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

- **UMin** = 0
- **UMax** = $2^w - 1$

- Two's Complement **Sign Bit**

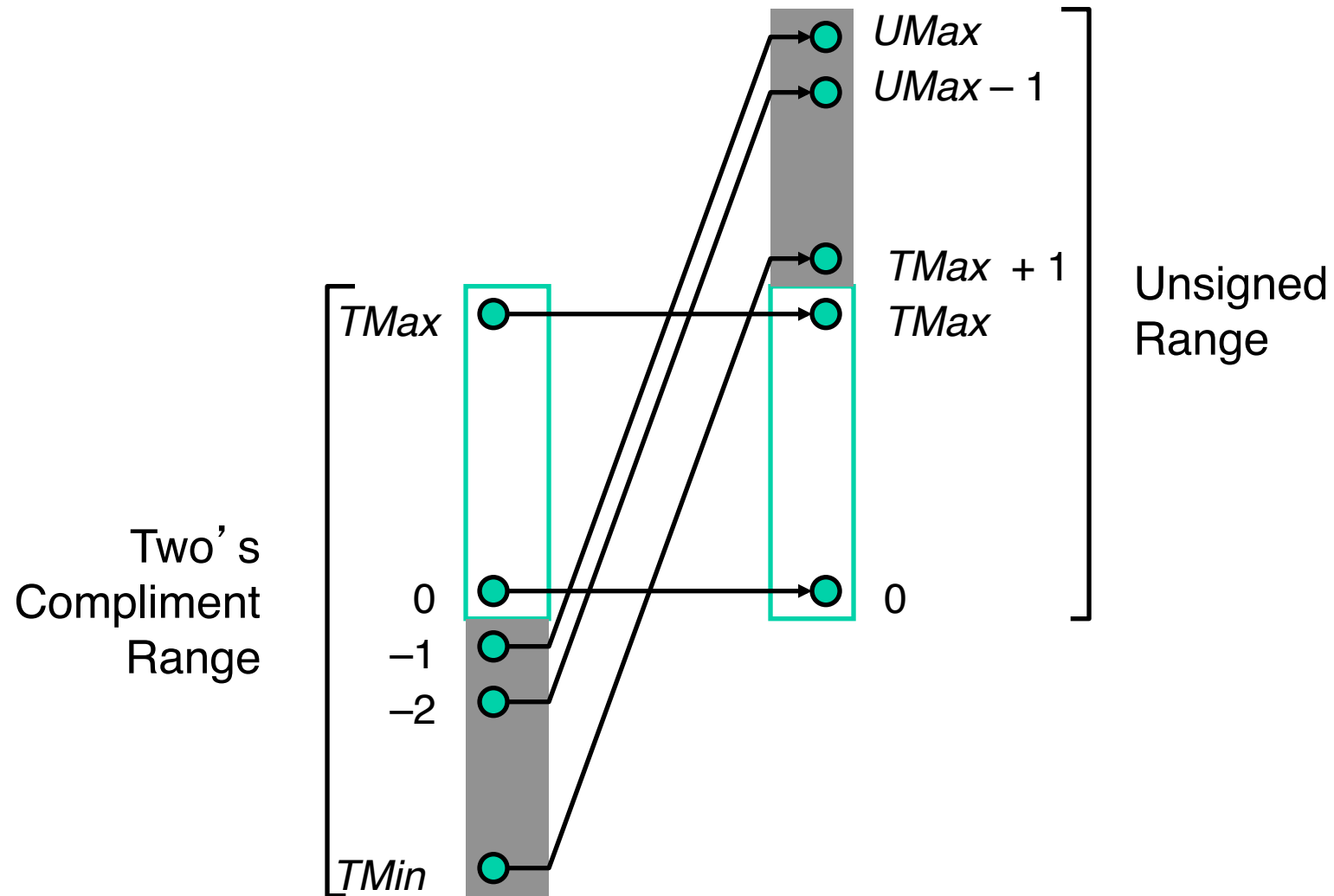

$$B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

- **TMin** = -2^{w-1}
- **TMax** = $2^{w-1} - 1$

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

Representation Relationship



Sizes and C Data Types

C Data Type	MIPS, x86	Alpha
char	8 bits	8 bits
short	16 bits	16 bits
int	32 bits	32 bits
long int	32 bits	64 bits

`char, short, int, long int`

- Refer to number of bits of integer
- Most machines: signed two's complement

`unsigned <type>`

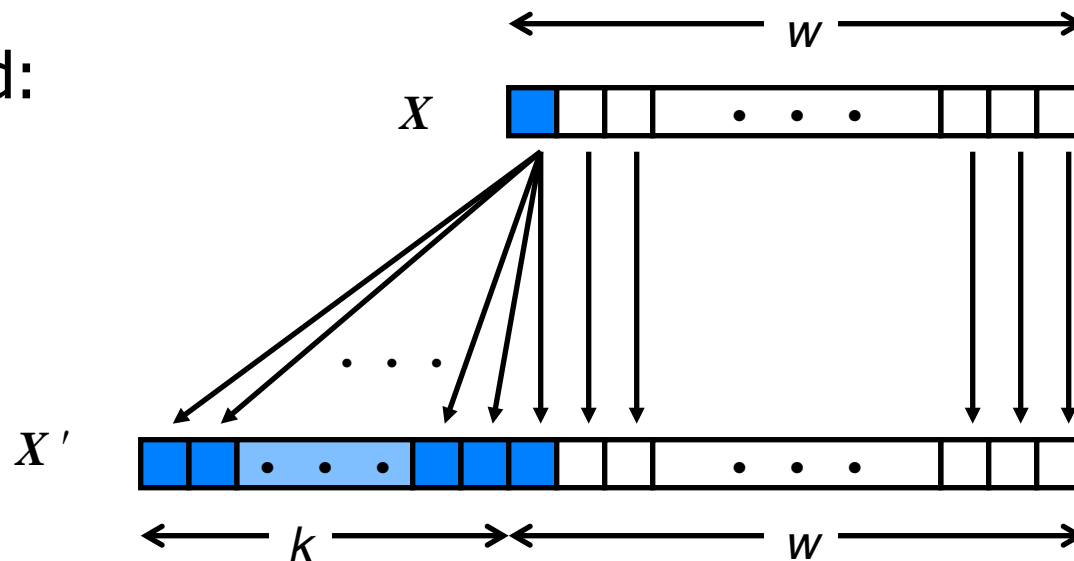
- Same number of bits as signed counterparts
- Unsigned integer

Sign Extension

```
char minusFour = -4;  
short moreBits;  
moreBits = (short)minusFour;
```

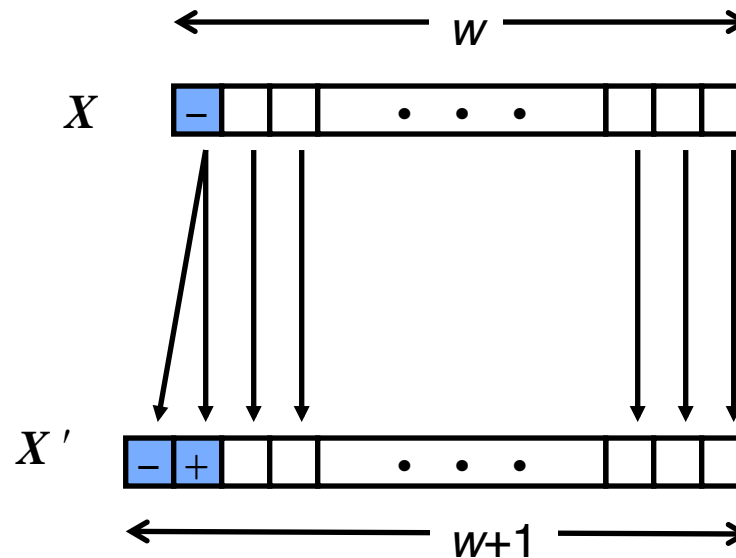
Given w bit signed integer, return equivalent $w+k$ bit signed integer

Sign Extend:



Sign Extension Proof of Correctness Outline

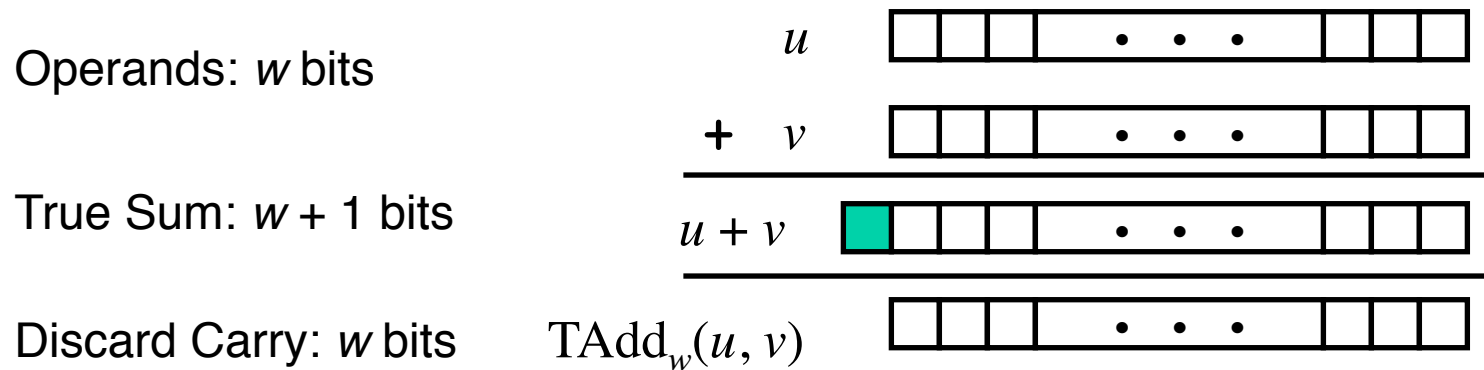
- Prove Correctness by Induction on k
- Induction Step: extending by single bit maintains value



lb, lbu - load byte (sign extend), load byte unsigned

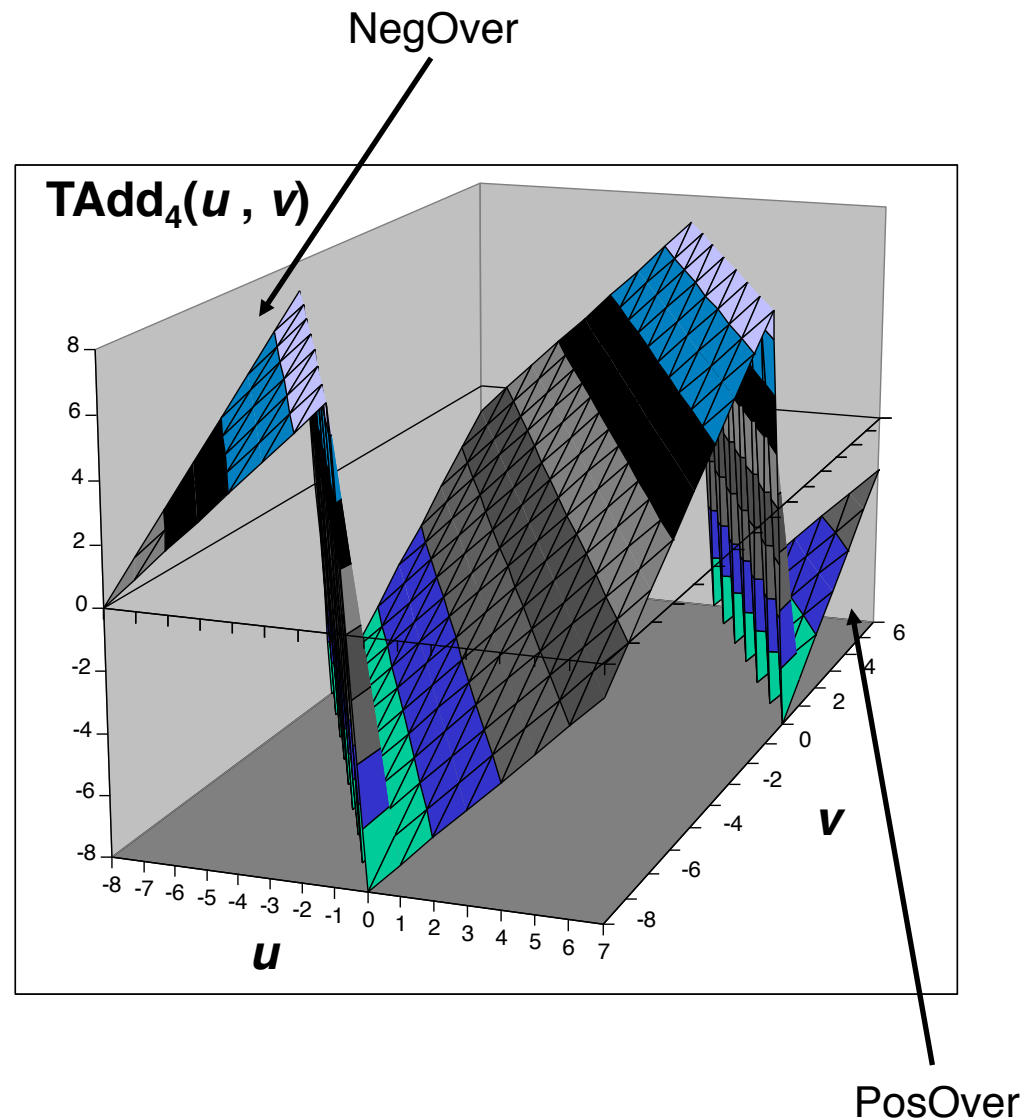
Two's Complement Addition

- TAdd and UAdd have identical Bit-Level Behavior!



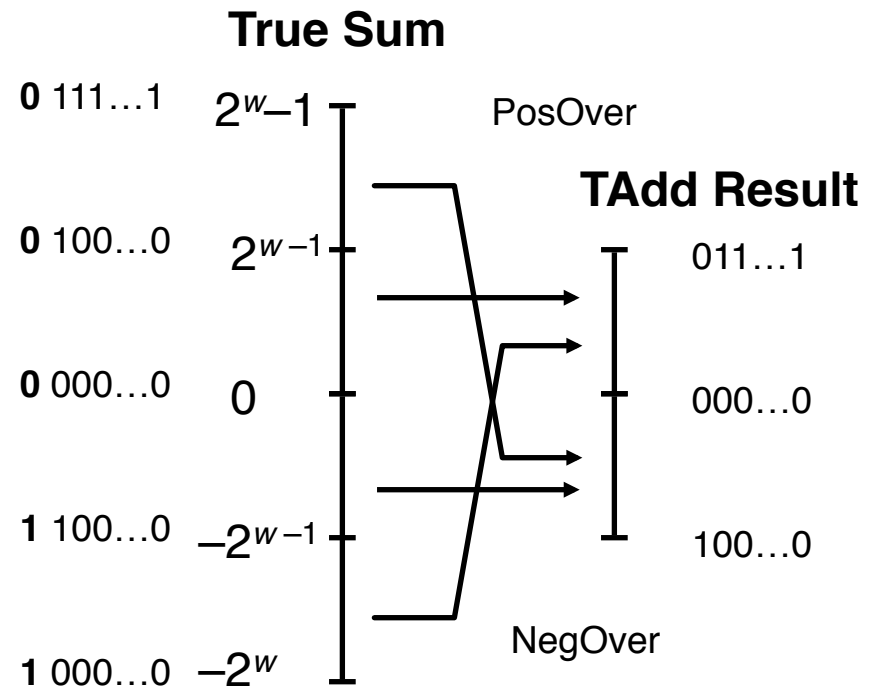
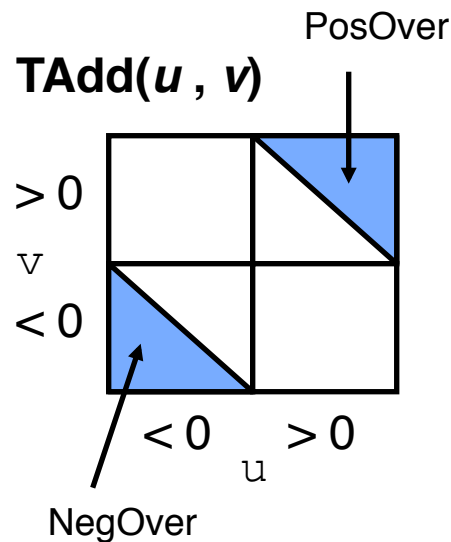
Visualizing Two's Complement Addition

- Range: -8 to +7
- If $\text{sum} \geq 2^w - 1$
 - Becomes negative
 - At most once
- If $\text{sum} < -2^w - 1$
 - Becomes positive
 - At most once



Characterizing TAdd

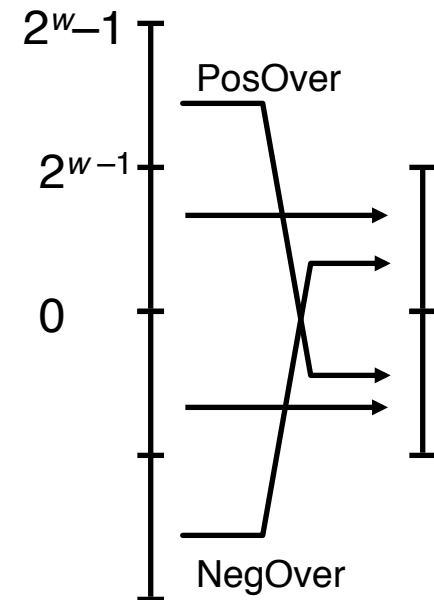
- True sum requires $w+1$ bits
- Drop MSB



$$TAdd_w(u, v) = \begin{cases} u + v + 2^{w-1} & u + v < TMin_w \text{ (NegOver)} \\ u + v & TMin_w \leq u + v \leq TMax_w \\ u + v - 2^{w-1} & TMax_w < u + v \text{ (PosOver)} \end{cases}$$

Detecting Two's Complement Overflow

- Task:
 - Given $s = \text{TAdd}_w(u, v)$
 - Determine if $s = \text{Add}_w(u, v)$
- Claim:
 - Overflow iff either:
 - $u, v < 0, s \geq 0$ (NegOver)
 - $u, v \geq 0, s < 0$ (PosOver)
 - $\text{ovf} = (u < 0 == v < 0) \ \&\& \ (u < 0 != s < 0);$
- Proof:
 - Obviously, if $u \geq 0$ and $v < 0$, then $\text{TMin}_w \leq u + v \leq \text{TMax}_w$
 - Symmetrically if $u < 0$ and $v \geq 0$
 - Other cases from analysis of TAdd



Negation, Inversion

Inversion:

- Flip all 0's to 1's and vice versa: $0011 \Rightarrow 1100$
- What does this do to the two's complement value?

Negation:

- Two's complement: invert all bits and add 1
- Example:

$$3_{10} = 0011$$

$$\text{invert}(0011) + 1 \rightarrow 1100 + 1 \rightarrow 1101$$

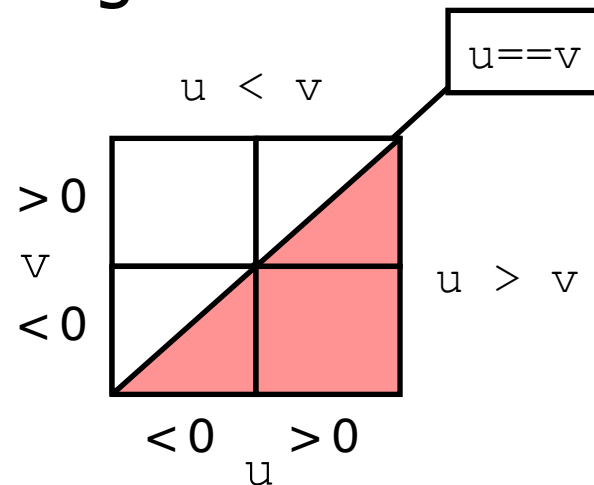
$$1101 = -3_{10}$$

Two's Complement Negation

- Mostly like Integer Negation
 - $\text{TComp}(u) = -u$
- TMin is Special Case
 - $\text{TComp}(\text{TMin}) = \text{TMin}$
 - Note Also: $\text{TComp}(0) = 0$
- Negation in C ($x = -x;$) is Actually TComp

Comparing Two's Complement Numbers

- Given signed numbers u, v
- Determine whether or not $u > v$
- Return true for shaded region:



- Bad Approach:
 - Test $(u - v) > 0$
 - Problem: Thrown off by Overflow

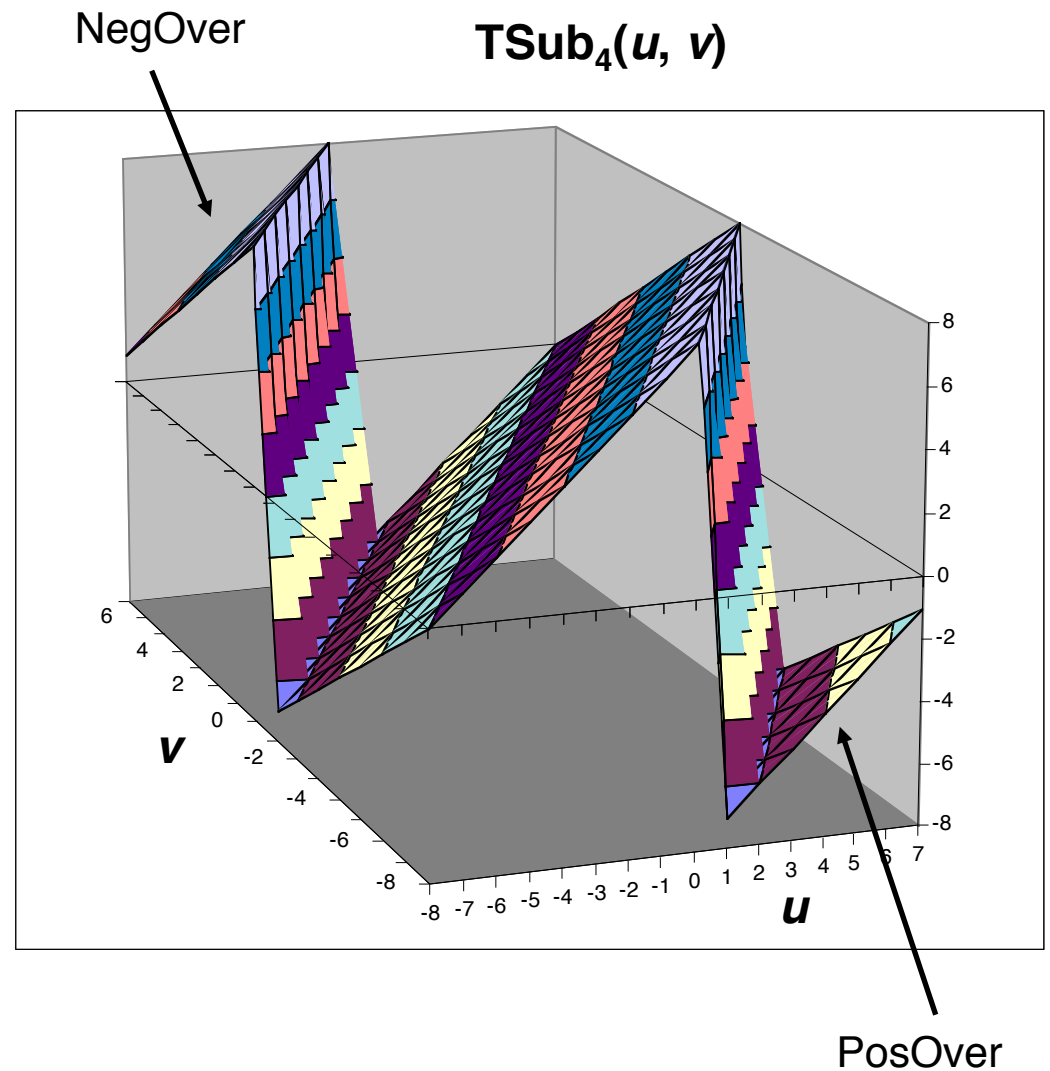
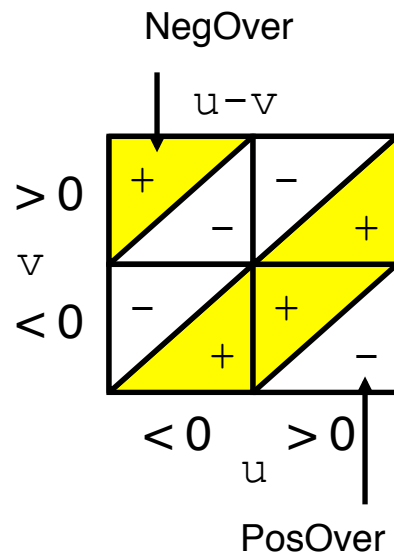
Comparing with TSub

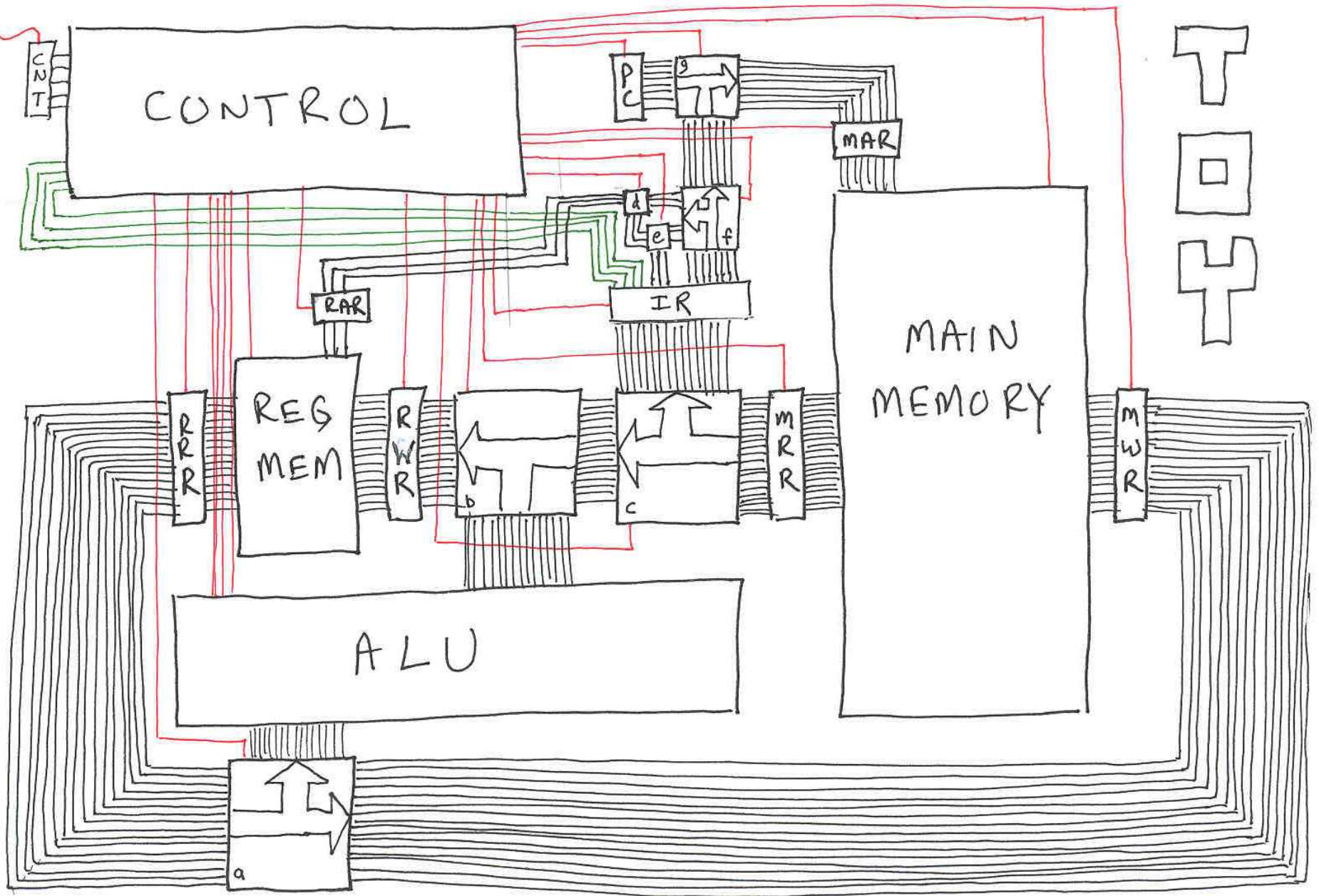
NegOver:

- $u < 0, v > 0$
- $TSub_4(u, v) > 0$

PosOver:

- $u > 0, v < 0$
- $TSub_4(u, v) < 0$

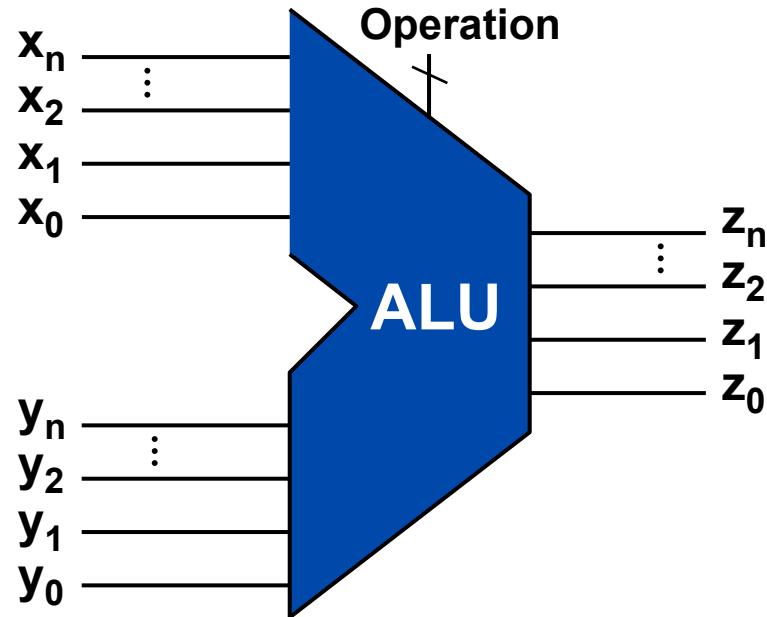




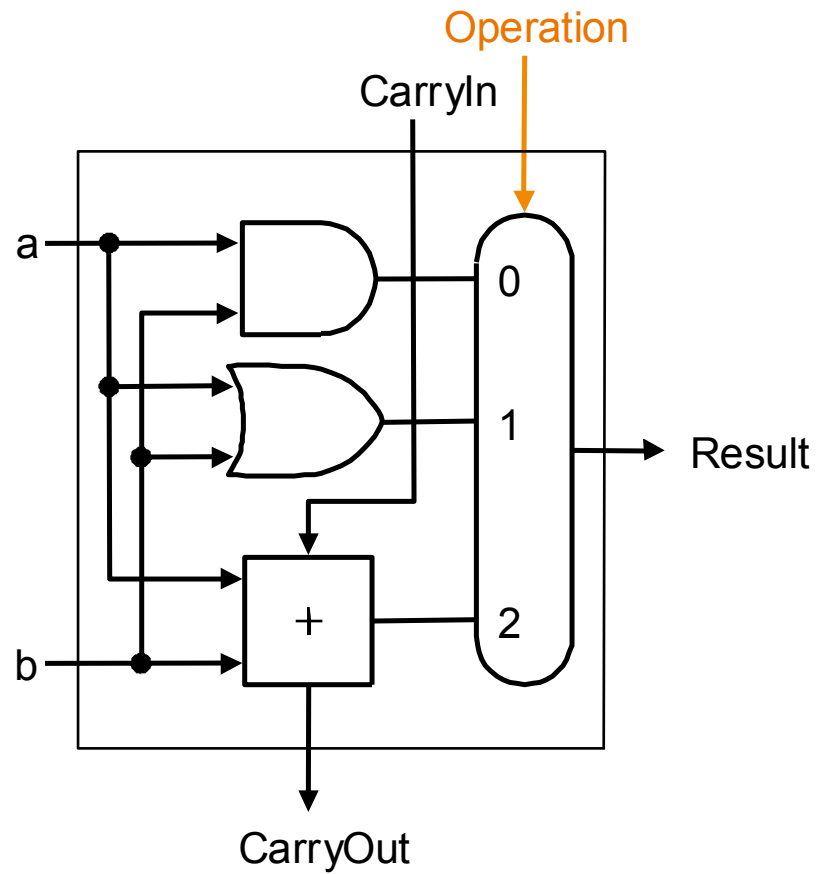
Building It!

The ALU

- ALU takes 2 N-bit numbers and operates upon them
- Let's start by supporting MIPS and, andi, or, and ori
- Outline: Build a 1 bit ALU, and use 32 of them



1-bit ALU for and, or, add



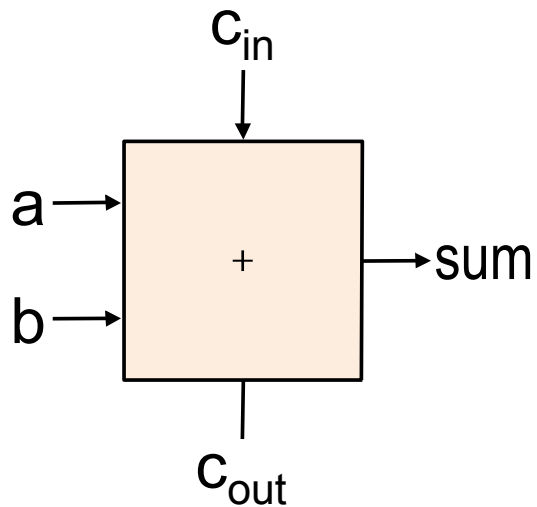
andi and ori Truth Table

Op	A_i	B_i	Result
0 (andi)	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1 (ori)	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

$$\text{Result} = A_i B_i \overline{(\text{Op})} + (A_i + B_i)(\text{Op})$$

Addition

- 1-bit ALU for addition:

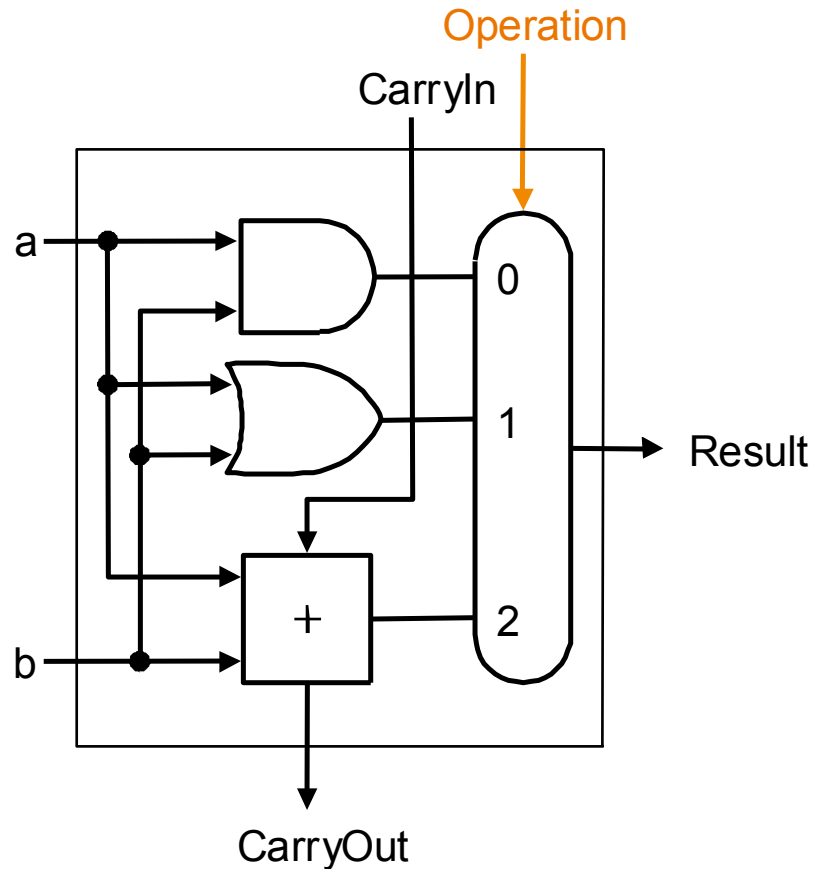


$$c_{out} = ab + ac_{in} + bc_{in}$$

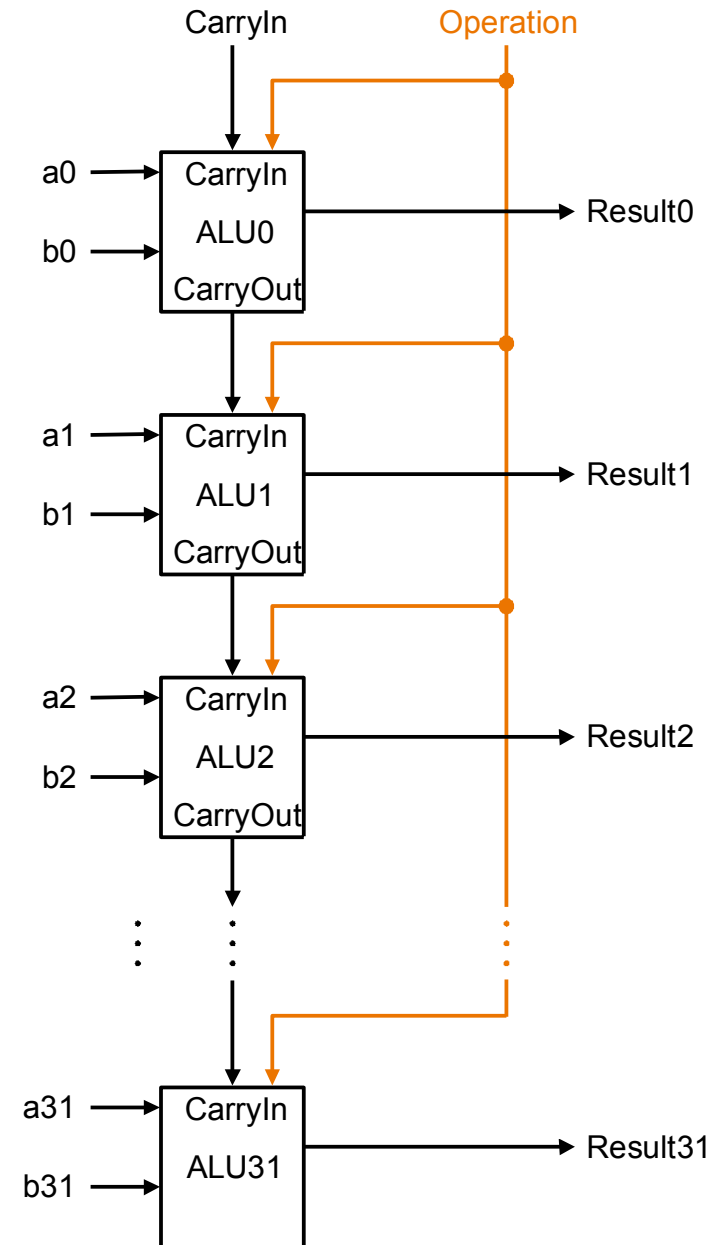
$$sum = a \text{ XOR } b \text{ XOR } c_{in}$$

- Carry is majority function (more ones than zeros?)
- Sum is a parity function (odd number of 1's?)

Building a 32 bit ALU for and, or, add

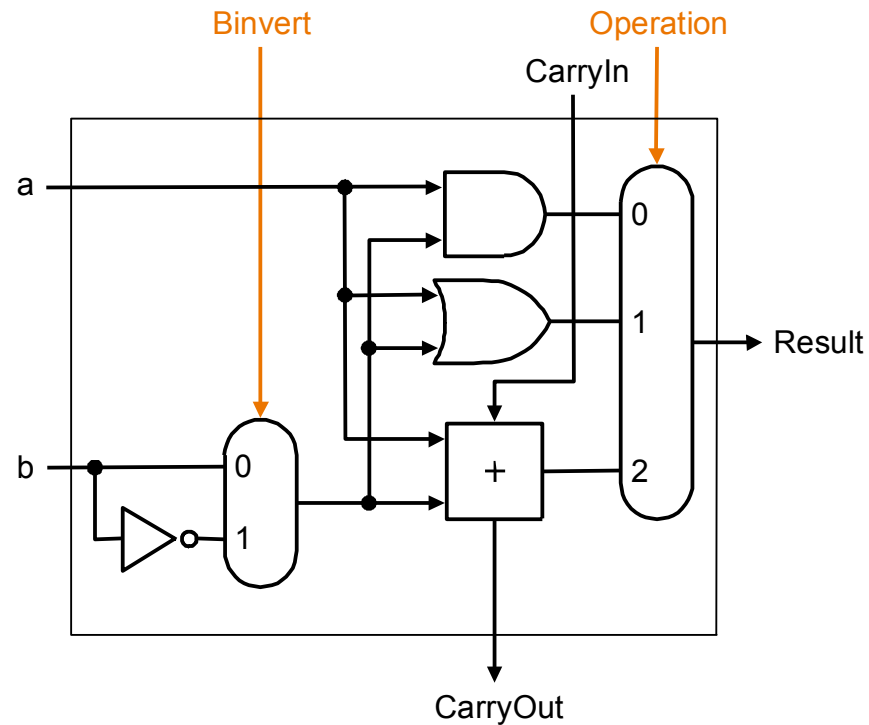


Forms a Ripple Carry Adder

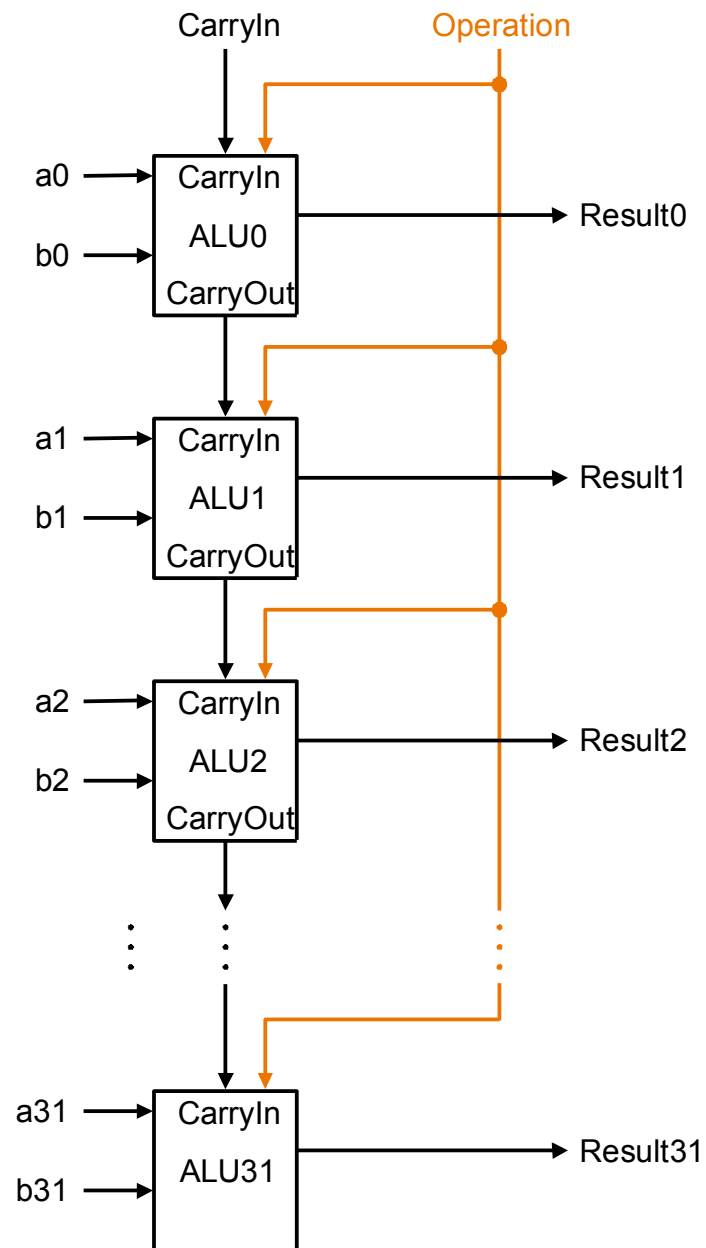


What about subtraction?

- Two's complement approach: just negate b and add.
- An elegant solution - use the carry in on bit 0:



Speed of 32 bit ALU for and, or, add, sub



A Performance Problem?

- Is a 32-bit ALU as fast as a 1-bit ALU?
- Can you see the ripple? How could you get rid of it?

$$c_1 = b_0c_0 + a_0c_0 + a_0b_0$$

$$c_2 = b_1c_1 + a_1c_1 + a_1b_1$$

- Other ways to do addition?
 - ripple carry adder - this approach, too slow!
 - complete sum-of-products - huge!
 - Somewhere in between...

Carry-Lookahead Adder (CLA)

Without carry in value, what can we do?

$$c_{i+1} = b_i c_i + a_i c_i + a_i b_i$$

$$c_{i+1} = (a_i b_i) + (a_i + b_i) c_i$$

When is the carry generated?

$$g_i = a_i b_i$$

When does the carry propagate?

$$p_i = a_i + b_i$$

$$c_{i+1} = (a_i b_i) + (a_i + b_i) c_i$$

$$c_{i+1} = g_i + p_i c_i$$

$$c_{i+1} = g_i + p_i c_i$$

Carry-Lookahead Adder (CLA)

$$C_1 = g_0 + p_0 C_0$$

$$C_2 = g_1 + p_1 C_1$$

$$C_3 = g_2 + p_2 C_2$$

$$C_4 = g_3 + p_3 C_3$$

$$C_4 = g_3 + p_3 g_2 + p_3 p_2 g_1 + \\ p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 C_0$$

Is this faster? Carry out = 2 gate delays, 4 bit adder

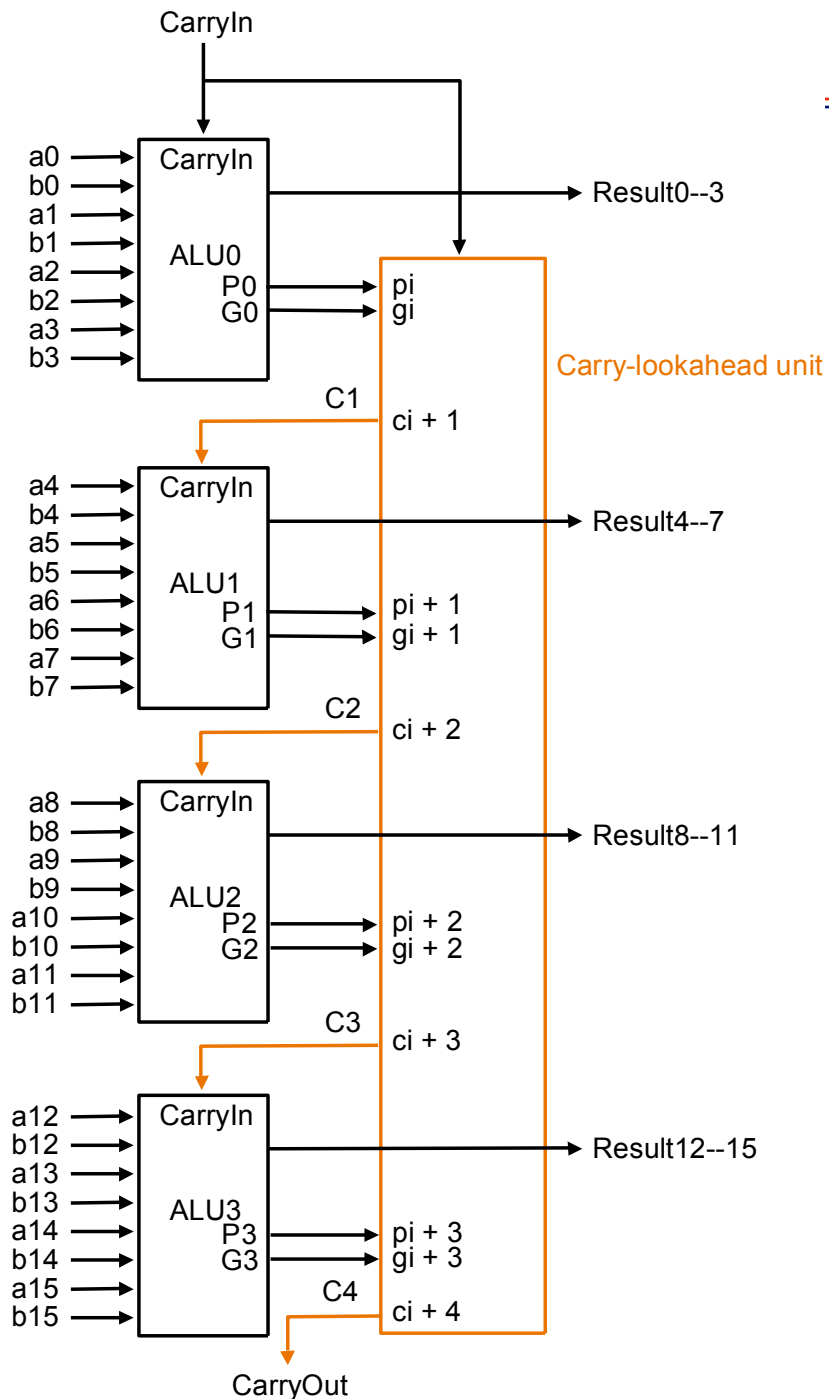
- Ripple Carry: $4 * 2 = 8$ gate delays
- CLA: 1 to compute all g/p + 2 for SOP = 3 gate delays

Use CLA principle to build bigger adders

16 bit CLA adder too big

Compose approaches:

- Use a set of 4-bit CLA adders
- Link them together:
 - Ripple carry between CLA blocks
 - Or, build another level of CLA



Carry-Lookahead Adder (CLA)

Hierarchy

$$C_4 = g_3 + p_3g_2 + p_3p_2g_1 + \\ p_3p_2p_1g_0 + p_3p_2p_1p_0c_0$$

$$G_0 = g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0$$

$$P_0 = p_3p_2p_1p_0$$

P_i and G_i are to a 4-bit adder unit
as

p_i and g_i are to a single bit adder

$$C_4 = G_3 + P_3G_2 + P_3P_2G_1 + \\ P_3P_2P_1G_0 + P_3P_2P_1P_0C_0$$

Summary

- Unsigned Numbers
- Signed Numbers: Signed Magnitude, One's Complement, Two's Complement
- Two's Complement
 - Addition
 - Inversion, Negation
 - Subtraction, Comparison
- ALU: and, or, add, sub
- Ripple Carry, Carry Lookahead

Reading:

- Pay particular attention to overflow detection logic