BITS, BYTES, AND INTEGERS

CS 045

Computer Organization and Architecture

Prof. Donald J. Patterson

Adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

WARM-UP

WORK IT OUT

WORK IT OUT

$$T2U_4(x)$$

-8
-3
-2
-1
0
5

WORK IT OUT

$$T2U_4(x)$$

-8	?
-3	?
-2	?
-1	?
0	?
5	?

WORK IT OUT

$$T2U_4(x)$$

-8	?
-3	?
-2	?
-1	?
0	?
5	?

WORK IT OUT

$$T2U_4(x)$$

-8	1000_2	
-3		?
-2		?
-1		?
0		?
5		?

WORK IT OUT

$$T2U_4(x)$$

-8	1000_2	8
-3		?
-2		?
-1		?
0		?
5		?

WORK IT OUT

$$T2U_4(x)$$

-8	1000_2	8
-3	1101_2	
-2		?
-1		?
0		?
5		?

WORK IT OUT

$$T2U_4(x)$$

-8	1000_2	$\boxed{8}$
-3	1101_2	$\boxed{13}$
-2		$\boxed{?}$
-1		$\boxed{?}$
0		$\boxed{?}$
5		$\boxed{?}$

WORK IT OUT

$$T2U_4(x)$$

-8	1000_2	8
-3	1101_2	13
-2	1110_2	
-1		?
0		?
5		?

WORK IT OUT

$$T2U_4(x)$$

-8	1000_2	$\boxed{8}$
-3	1101_2	$\boxed{13}$
-2	1110_2	$\boxed{14}$
-1		$\boxed{?}$
0		$\boxed{?}$
5		$\boxed{?}$

WORK IT OUT

$$T2U_4(x)$$

-8	1000_2	8
-3	1101_2	13
-2	1110_2	14
-1	1111_2	
0		?
5		?

WORK IT OUT

$$T2U_4(x)$$

-8	1000_2	8
-3	1101_2	13
-2	1110_2	14
-1	1111_2	15
0		?
5		?

WORK IT OUT

$$T2U_4(x)$$

-8	1000_2	8
-3	1101_2	13
-2	1110_2	14
-1	1111_2	15
0	0000_2	
5		?

WORK IT OUT

$$T2U_4(x)$$

-8	1000_2	8
-3	1101_2	13
-2	1110_2	14
-1	1111_2	15
0	0000_2	0
5		?

WORK IT OUT

$$T2U_4(x)$$

-8	1000_2	8
-3	1101_2	13
-2	1110_2	14
-1	1111_2	15
0	0000_2	0
5	0101_2	

WORK IT OUT

$$T2U_4(x)$$

-8	1000_2	$\boxed{8}$
-3	1101_2	$\boxed{13}$
-2	1110_2	$\boxed{14}$
-1	1111_2	$\boxed{15}$
0	0000_2	$\boxed{0}$
5	0101_2	$\boxed{5}$

WORK IT OUT

```c
#include <stdio.h>


int main(){

    char a = -8;
    char b = -3;
    char c = -2;
    char d = -1;
    char e = 0;
    char f = 5;
    printf("%2d signed -> 0x%02x -> %2u unsigned\n",a,a&0x0F,a&0x0F);
    printf("%2d signed -> 0x%02x -> %2u unsigned\n",b,b&0x0F,b&0x0F);
    printf("%2d signed -> 0x%02x -> %2u unsigned\n",c,c&0x0F,c&0x0F);
    printf("%2d signed -> 0x%02x -> %2u unsigned\n",d,d&0x0F,d&0x0F);
    printf("%2d signed -> 0x%02x -> %2u unsigned\n",e,e&0x0F,e&0x0F);
    printf("%2d signed -> 0x%02x -> %2u unsigned\n",f,f&0x0F,f&0x0F);


    return 0;
}
```

WORK IT OUT

```c
#include <stdio.h>


int main(){

    char a = -8;
    char b = -3;
    char c = -2;
    char d = -1;
    char e = 0;
    char f = 5;
    printf("%2d signed -> 0x%02x -> %2u unsigned\n",a,a&0x0F,a&0x0F);
    printf("%2d signed -> 0x%02x -> %2u unsigned\n",b,b&0x0F,b&0x0F);
    printf("%2d signed -> 0x%02x -> %2u unsigned\n",c,c&0x0F,c&0x0F);
    printf("%2d signed -> 0x%02x -> %2u unsigned\n",d,d&0x0F,d&0x0F);
    printf("%2d signed -> 0x%02x -> %2u unsigned\n",e,e&0x0F,e&0x0F);
    printf("%2d signed -> 0x%02x -> %2u unsigned\n",f,f&0x0F,f&0x0F);


    return 0;
}
```

```
-8 signed -> 0x08 ->  8 unsigned
-3 signed -> 0x0d -> 13 unsigned
-2 signed -> 0x0e -> 14 unsigned
-1 signed -> 0x0f -> 15 unsigned
 0 signed -> 0x00 ->  0 unsigned
 5 signed -> 0x05 ->  5 unsigned
```

EXPANDING AND TRUNCATING

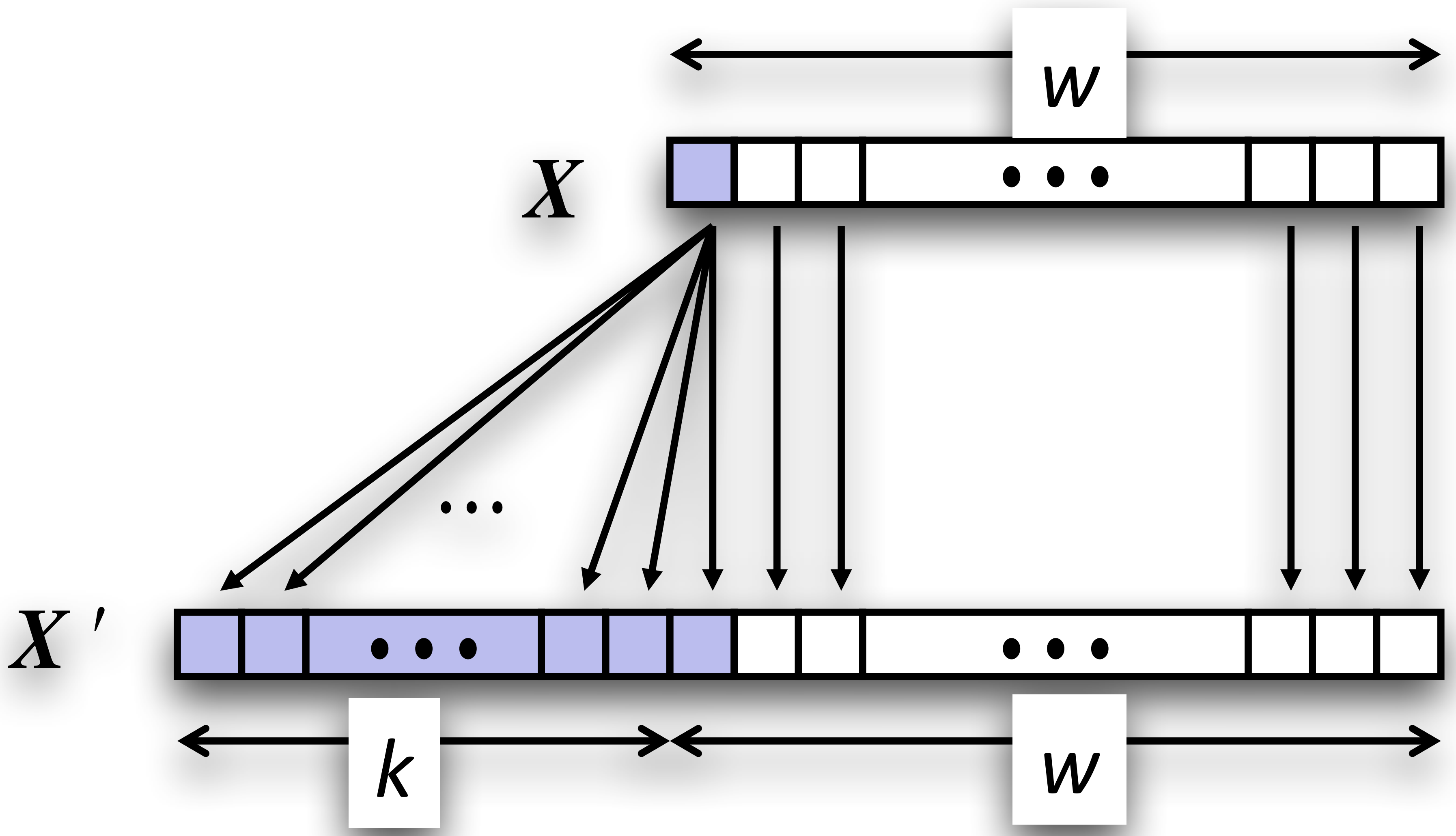
CONVERSION AND CASTING

SIGN EXTENSION

- Task:
  - Given w-bit signed integer x
  - Convert it to w+k-bit integer with same value
- Rule:
  - Make k copies of sign bit:
  - $X' = \underbrace{x_{w-1}, \ldots, x_{w-1}}_{k \text{ copies of MSB}}, x_{w-1}, x_{w-2}, \ldots, x_0$

CONVERSION AND CASTING

SIGN EXTENSION



CONVERSION AND CASTING

SIGN EXTENSION EXAMPLE

```
short int x =  15213;
int      ix = (int) x;
short int y = -15213;
int      iy = (int) y;
```

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
ix	15213	00 00 3B 6D	00000000 00000000 00111011 01101101
y	-15213	C4 93	11000100 10010011
iy	-15213	FF FF C4 93	11111111 11111111 11000100 10010011

- Converting from smaller to larger integer data type
- C automatically performs sign extension

CONVERSION AND CASTING

SUMMARY:
EXPANDING, TRUNCATING: BASIC RULES

- Expanding (e.g., short int to int)
  - Unsigned: zeros added
  - Signed: sign extension
  - Both yield expected result

- Truncating (e.g., unsigned to unsigned short)
  - Unsigned/signed: bits are truncated
  - Result reinterpreted
  - Unsigned: mod operation
  - Signed: similar to mod
  - For small numbers yields expected behavior

INTEGER ARITHMETIC

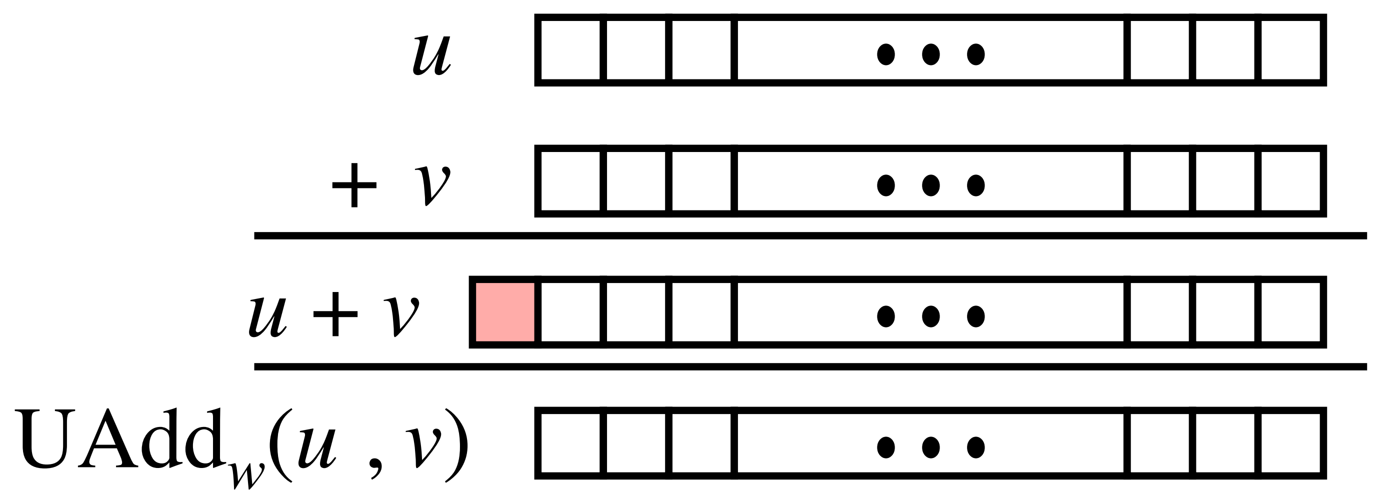
INTEGER ARITHMETIC

UNSIGNED ADDITION

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits

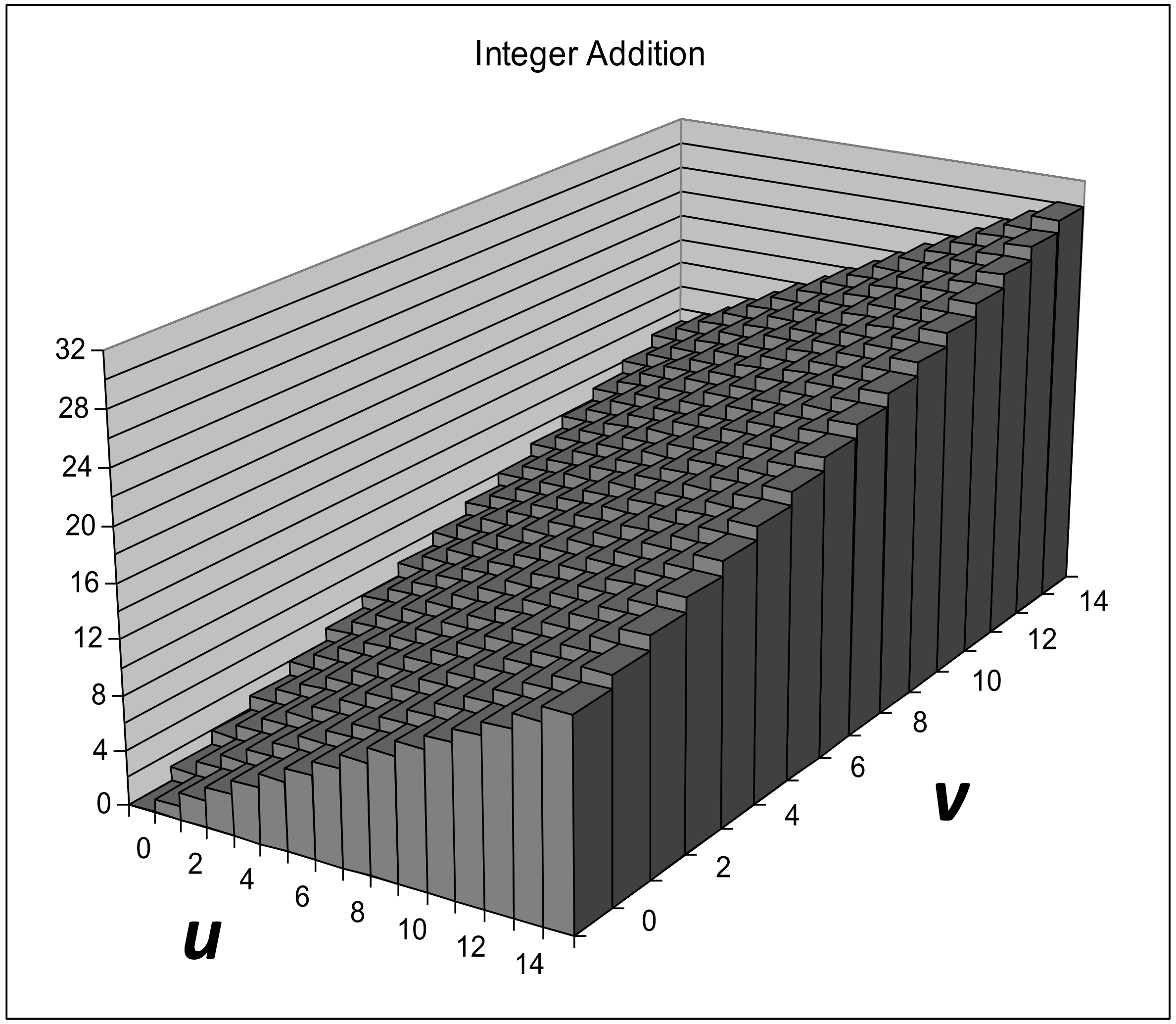


- Standard Addition Function
  - Ignores carry output
- Implements Modular Arithmetic
  - $s \quad = \quad \mathrm{UAdd}_w(u\ ,\ v) \quad = \quad (u + v) \mod 2^w$

VISUALIZING (MATHEMATICAL) INTEGER ADDITION

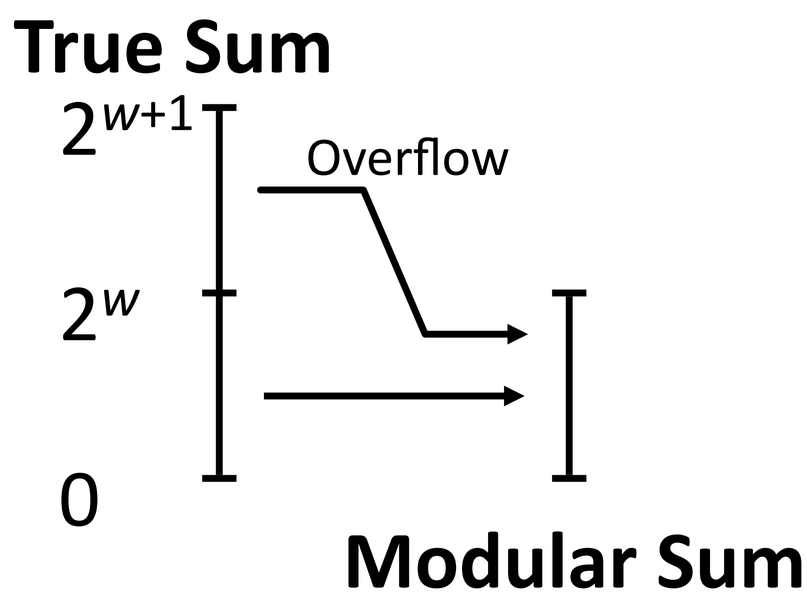
- Integer Addition
  - 4-bit integers u, v
  - Compute true sum $Add_4(u , v)$
  - Values increase linearly with u and v
  - Forms planar surface

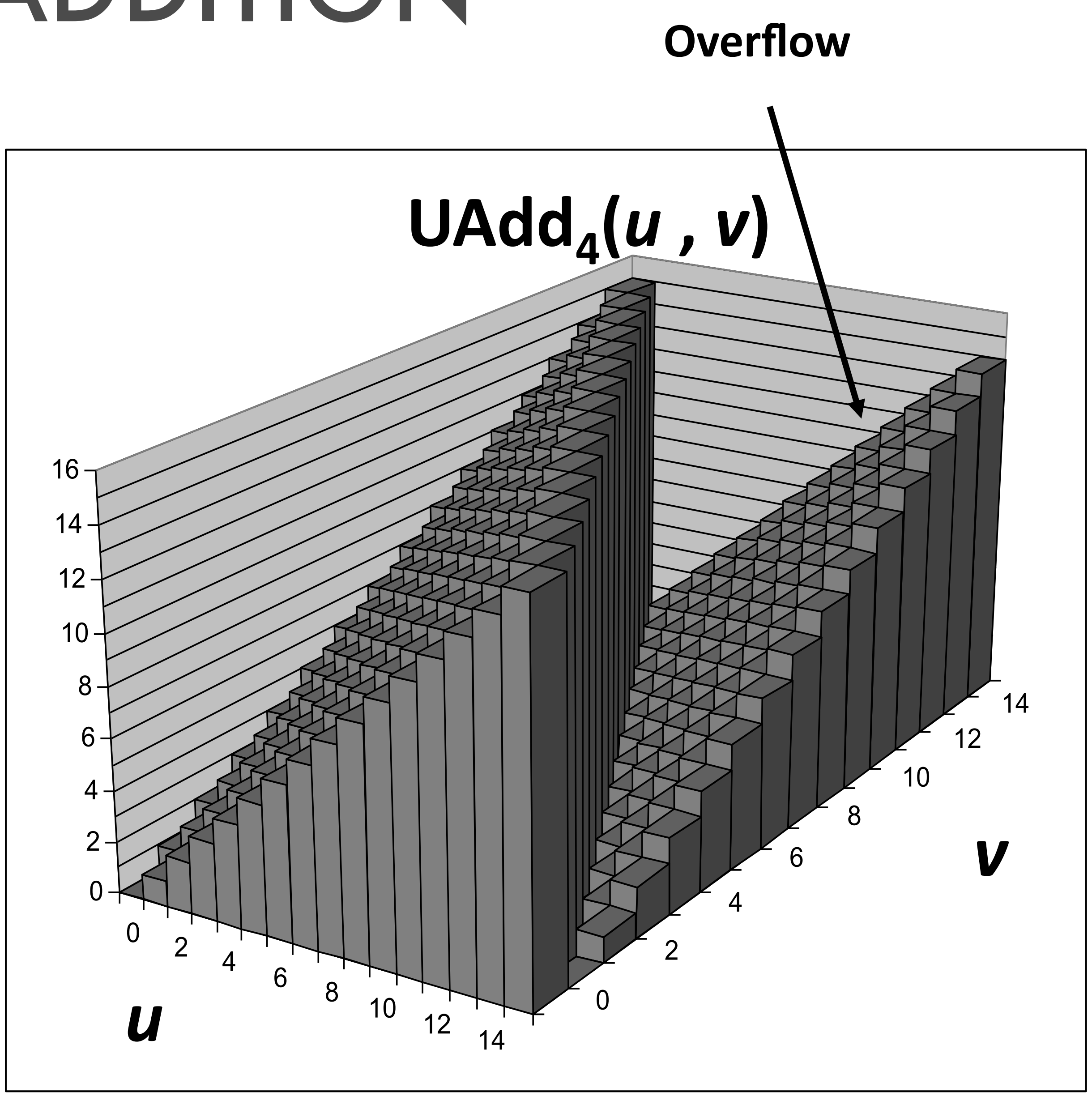
$Add_4(u , v)$



VISUALIZING UNSIGNED ADDITION

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





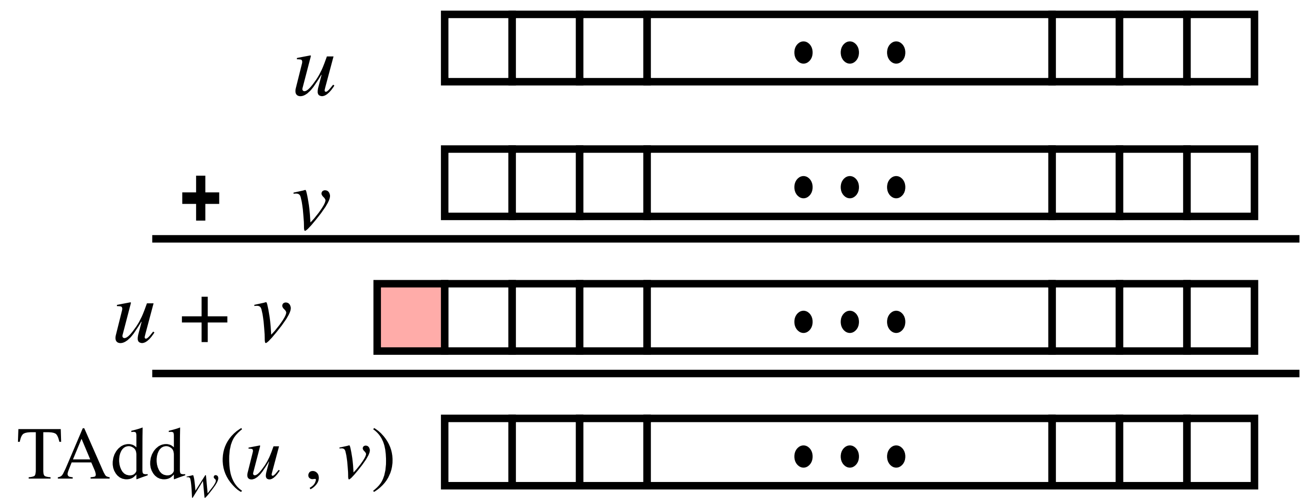
INTEGER ARITHMETIC

TWO'S COMPLEMENT 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`

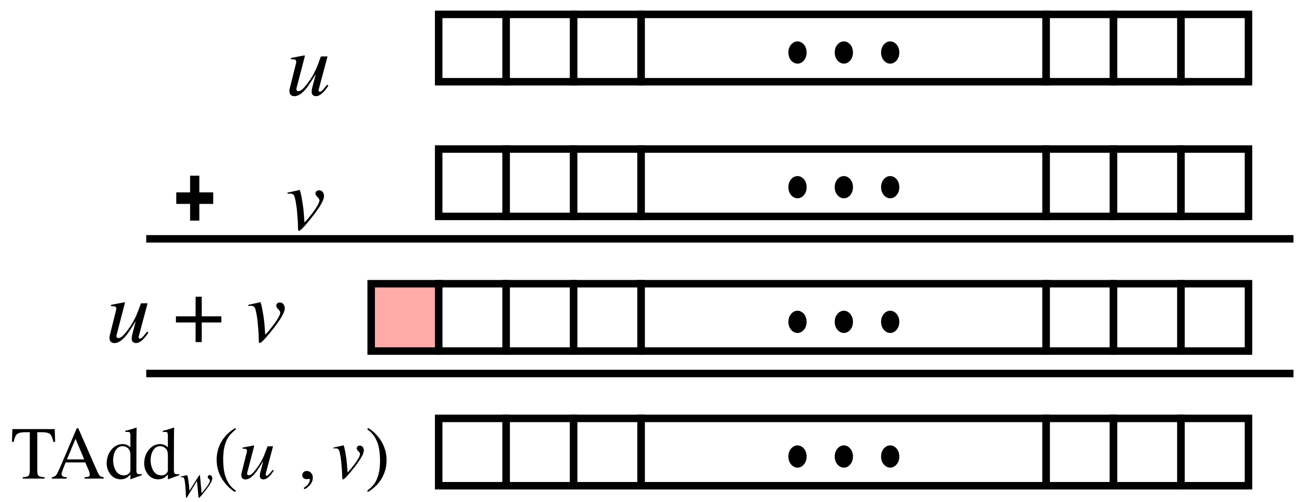
INTEGER ARITHMETIC

TWO'S COMPLEMENT ADDITION

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits



- TAdd and UAdd have Identical Bit-Level Beha
  - Signed vs. unsigned addition in C:
    - `int s, t, u, v;`
    - `s = (int) ((unsigned) u + (unsig`
    - `t = u + v`
  - Will give `s == t`

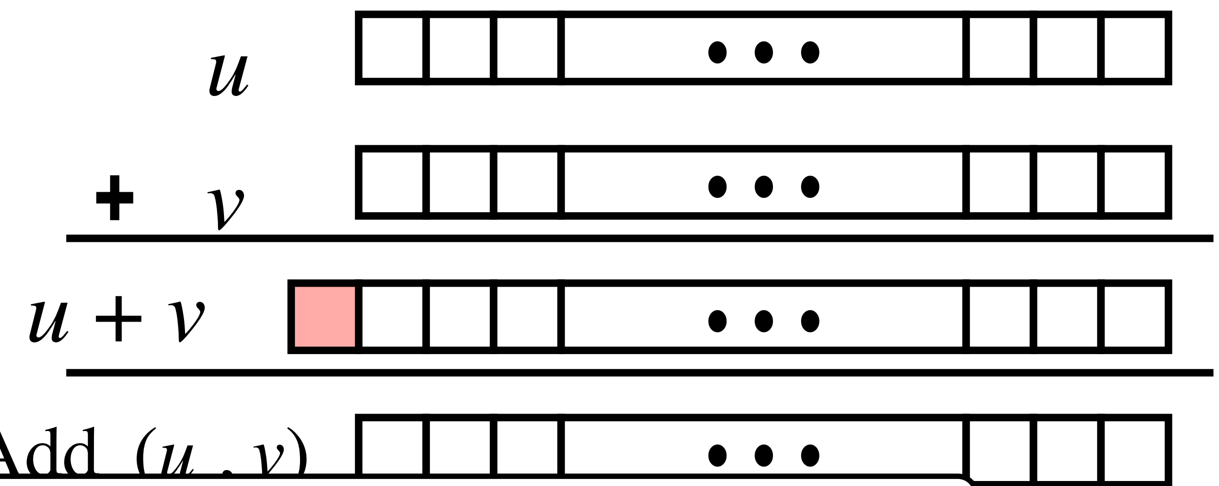
INTEGER ARITHMETIC

TWO'S COMPLEMENT ADDITION

Operands: *w* bits

True Sum: *w*+1 bits

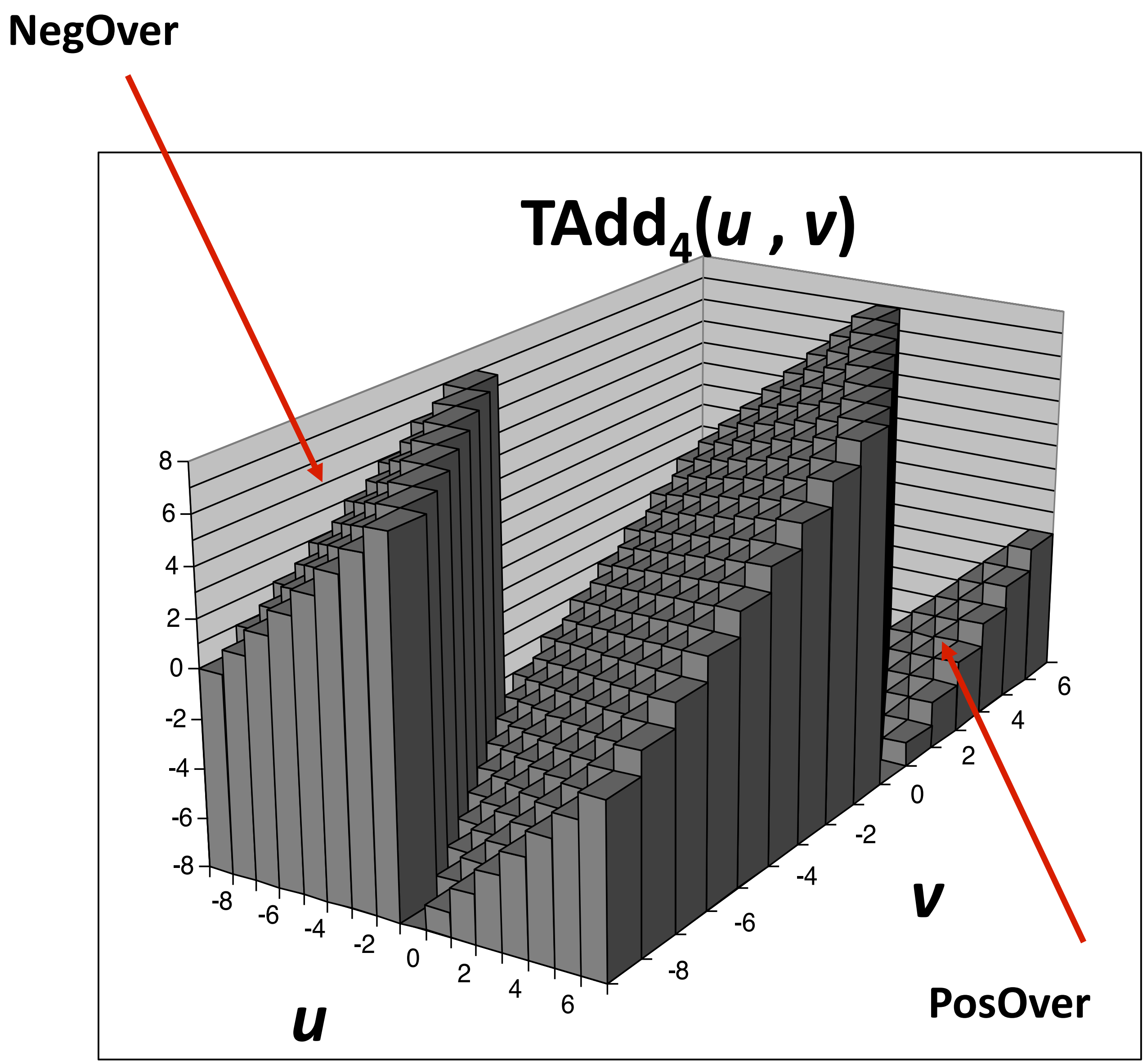
Discard Carry: *w* bits



- TAdd and UAdd hav
  - Signed vs. unsigned addition in C:
    - `int s, t, u, v;`
    - `s = (int) ((unsigned) u + (unsig`
    - `t = u + v`
  - Will give s == t

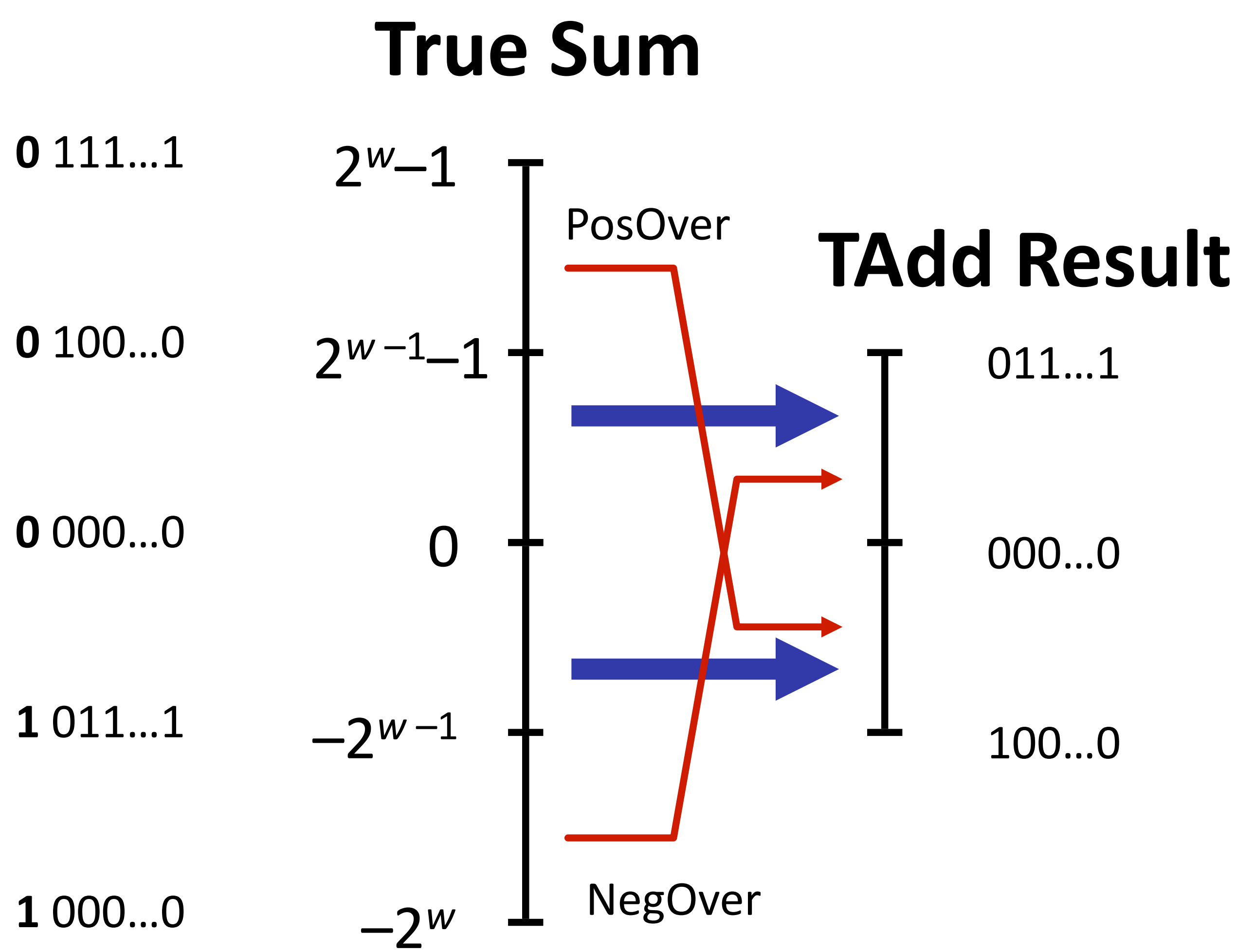
VISUALIZING 2'S COMPLEMENT ADDITION

- 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



TADD OVERFLOW

- Functionality
  - True sum requires w+1 bits
  - Drop off MSB
  - Treat remaining bits as 2's comp. integer



INTEGER MULTIPLICATION

MULTIPLICATION

- Goal: Computing Product of w-bit numbers x, y
  - Either signed or unsigned
- But, exact results can be bigger than w bits
  - Unsigned: up to 2w bits
    - Result range: $0 \le x * y \le (2^{w} - 1)\ 2 \;=\; 2^{2w} - 2^{w+1} + 1$
- Two's complement min (negative): Up to 2w-1 bits
  - Result range: $x * y \ \ge (-2^{w-1})*(2^{w-1}-1) \;=\; -2^{2w-2} + 2^{w-1}$
  - Two's complement max (positive): Up to 2w bits, but only for $(TMin_{w})^{2}$
  - Result range: $x * y \le (-2^{w-1})\ 2 \;=\; 2^{2w-2}$

MULTIPLICATION

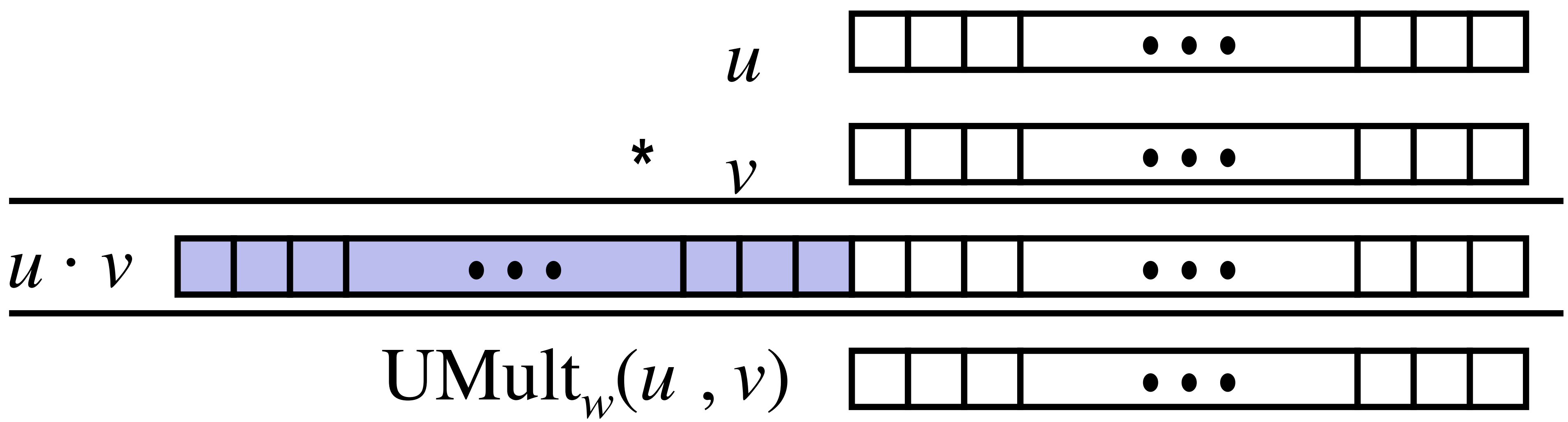
- So, maintaining exact results…
  - would need to keep expanding word size with each product computed
  - is done in software, if needed
    - e.g., by “arbitrary precision” arithmetic packages

UNSIGNED MULTIPLICATION IN C

Operands: *w* bits

True Product: 2**w* bits

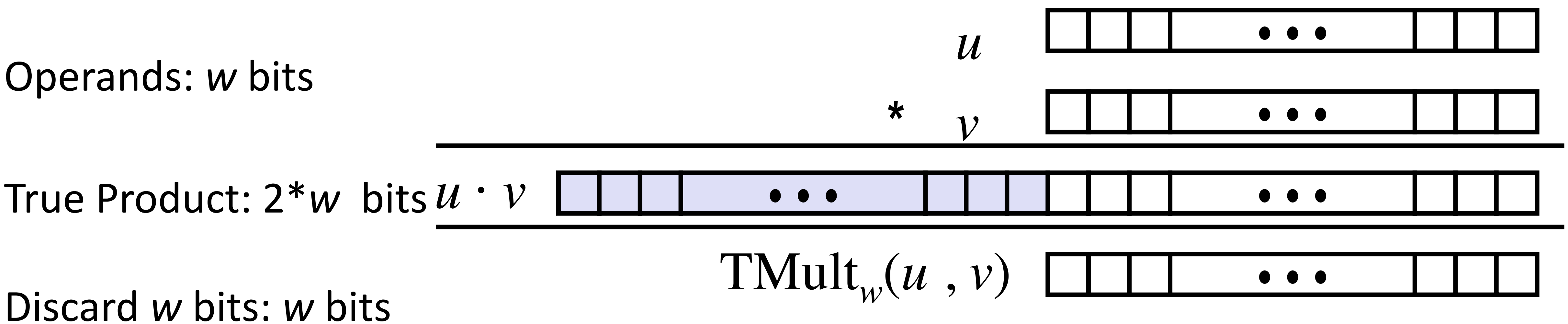
Discard *w* bits: *w* bits



- Standard Multiplication Function
  - Ignores high order w bits
- Implements Modular Arithmetic
  - $\mathrm{UMult}_w(u\ ,\ v) \ = \ u \cdot v \mod 2^w$

INTEGER ARITHMETIC

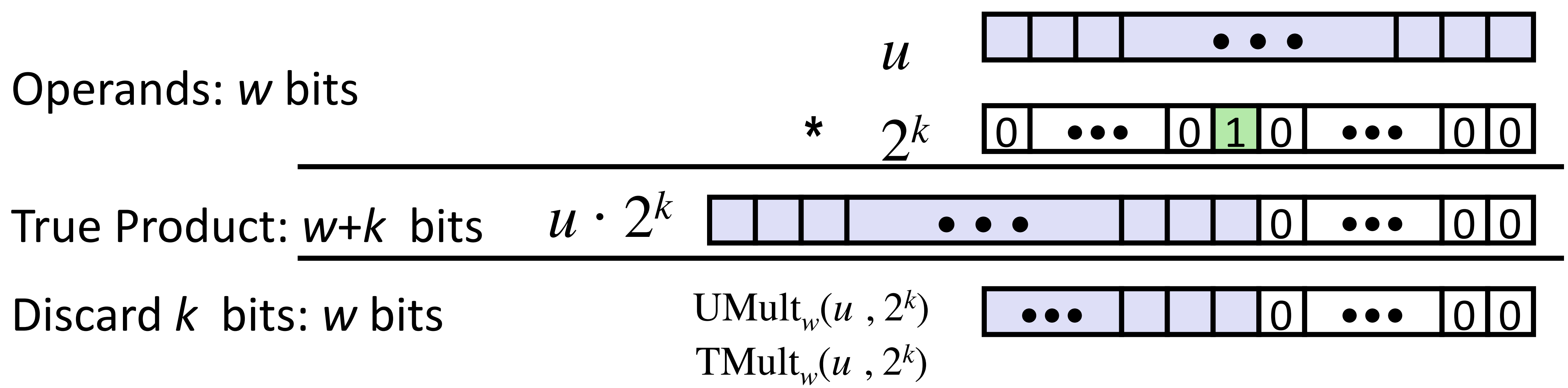
SIGNED MULTIPLICATION IN C



- Standard Multiplication Function
  - Ignores high order w bits
  - Some of which are different for signed vs. unsigned multiplication
  - Lower bits are the same

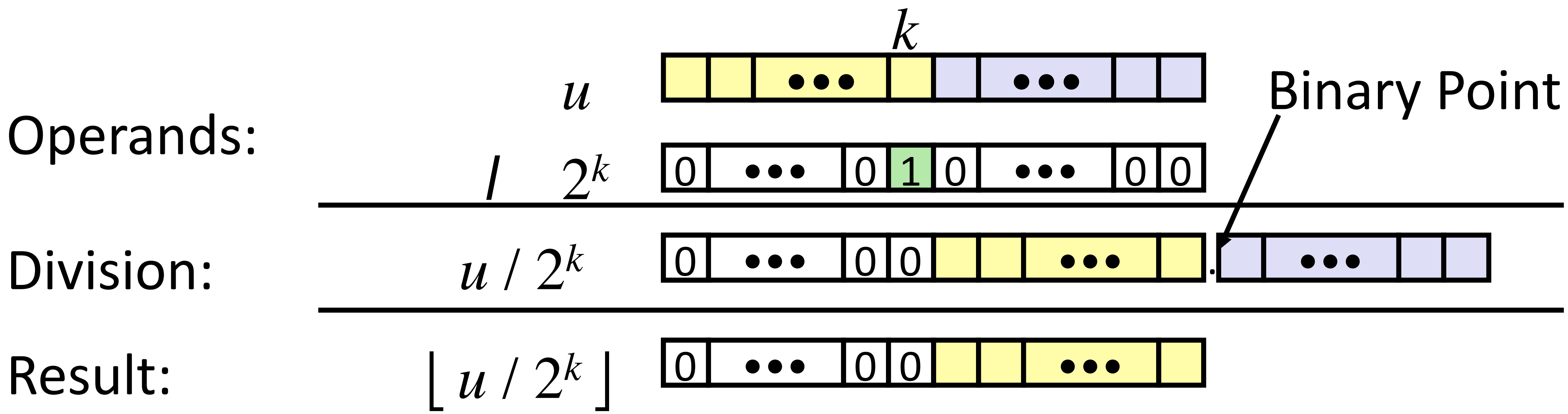
POWER-OF-2 MULTIPLY WITH SHIFT

- Operation
  - u << k gives u * 2^k
  - Both signed and unsigned



- Examples
  - u << 3 == u * 8
  - (u << 5) – (u << 3) == u * 24
- Most machines shift and add faster than multiply
  - Compiler generates this code automatically

UNSIGNED POWER-OF-2 DIVIDE WITH SHIFT



	Division	Computed	Hex	Binary
x	15213	15213	3B 6D	00111011 01101101
x >> 1	7606.5	7606	1D B6	00011101 10110110
x >> 4	950.8125	950	03 B6	00000011 10110110
x >> 8	59.4257813	59	00 3B	00000000 00111011

- Quotient of Unsigned by Power of 2
  - u >> k gives floor(u / 2k)
  - Uses logical shift

INTEGER ARITHMETIC SUMMARY

ARITHMETIC: BASIC RULES

- Addition:
  - Unsigned/signed: Normal addition followed by truncate, same operation on bit level
  - Unsigned: addition mod 2^w
    - Mathematical addition + possible subtraction of 2^w
  - Signed: modified addition mod 2^w (result in proper range)
    - Mathematical addition + possible addition or subtraction of 2^w

ARITHMETIC: BASIC RULES

- Multiplication:
  - Unsigned/signed: Normal multiplication followed by truncate, same operation on bit level
  - Unsigned: multiplication mod 2^w
    - Signed: modified multiplication mod 2^w (result in proper range)

WHY SHOULD I USE UNSIGNED?

- Don't use without understanding implications
  - Easy to make mistakes
    - `unsigned i;`
    - `for (i = cnt-2; i >= 0; i--)`
    - `  a[i] += a[i+1];`

  - Can be very subtle
    - `#define DELTA sizeof(int)`
    - `int i;`
    - `for (i = CNT; i-DELTA >= 0; i-= DELTA)`
    - `  . . .`

COUNTING DOWN WITH UNSIGNED

- Proper way to use unsigned as loop index
  - `unsigned i;`
  - `for (i = cnt-2; i < cnt; i--)`
  - `  a[i] += a[i+1];`
- See Robert Seacord, *Secure Coding in C and C++*
  - C Standard guarantees that unsigned addition will behave like modular arithmetic
    - 0 – 1 -> UMax

COUNTING DOWN WITH UNSIGNED

- Even better
  - size_t i;
  - for (i = cnt-2; i < cnt; i--)
    - a[i] += a[i+1];
  - Data type size_t defined as unsigned value with length = word size
  - Code will work even if cnt = UMax
  - What if cnt is signed and < 0?

INTEGER ARITHMETIC

WHY SHOULD I USE UNSIGNED? (CONT.)

- Do Use When Performing Modular Arithmetic
  - Multiprecision arithmetic
- Do Use When Using Bits to Represent Sets
  - Logical right shift, no sign extension

WORK IT OUT

WHAT ARE M AND N?

```
#define M       /* Mystery number 1 */
#define N       /* Mystery number 2 */
int arith(int x, int y) {
    int result = 0;
    result = x*M + y/N; /* M and N are mystery numbers. */
    return result;
}
```

Original code

```
/* Translation of assembly code for arith */
int optarith(int x, int y) {
    int t = x;
    x <<= 5;
    x -= t;
    if (y < 0) y += 7;
    y >>= 3;  /* Arithmetic shift */
    return x+y;
}
```

Optimized Code

WESTMONT INSPIRED

COMPUTING LAB