Distributive: $A\&(B|C) = (A\&B)|(A\&C)$

$A\&(B|C)$ $(A\&B)|(A\&C)$

Distributive: $A\&(B|C) = (A\&B)|(A\&C)$

$B|C$ $\quad$ $A\&(B|C)$ $\quad$ $(A\&B)|(A\&C)$

Distributive: $A\&(B|C) = (A\&B)|(A\&C)$

$B|C$ $\quad A\&B$ $\quad A\&C$ $\quad A\&(B|C)$ $\quad (A\&B)|(A\&C)$

Distributive: $A\&(B|C) = (A\&B)|(A\&C)$

A	B	C	$B|C$	$A\&B$	$A\&C$	$A\&(B|C)$	$(A\&B)|(A\&C)$

Distributive: $A\&(B|C) = (A\&B)|(A\&C)$

A	B	C	$B\|C$	$A\&B$	$A\&C$	$A\&(B\|C)$	$(A\&B)\|(A\&C)$
0	0	0					
0	0	1					
0	1	0					
0	1	1					
1	0	0					
1	0	1					
1	1	0					
1	1	1					

Distributive: $A\&(B|C) = (A\&B)|(A\&C)$

A	B	C	$B\|C$	$A\&B$	$A\&C$	$A\&(B\|C)$	$(A\&B)\|(A\&C)$
0	0	0	$(0\|0) = 0$	$(0\&0) = 0$	0		
0	0	1	$(0\|1) = 1$	$(0\&0) = 0$	0		
0	1	0	$(1\|0) = 1$	$(0\&1) = 0$	0		
0	1	1	$\ldots 1$	$\ldots 0$	0		
1	0	0	0	0	0		
1	0	1	1	0	1		
1	1	0	1	1	0		
1	1	1	1	1	1		

Distributive: $A\&(B|C) = (A\&B)|(A\&C)$

A	B	C	$B\|C$	$A\&B$	$A\&C$	$A\&(B\|C)$	$(A\&B)\|(A\&C)$
0	0	0	$(0\|0) = 0$	$(0\&0) = 0$	0	0	0
0	0	1	$(0\|1) = 1$	$(0\&0) = 0$	0	0	0
0	1	0	$(1\|0) = 1$	$(0\&1) = 0$	0	0	0
0	1	1	$\ldots 1$	$\ldots 0$	0	0	0
1	0	0	0	0	0	0	0
1	0	1	1	0	1	1	1
1	1	0	1	1	0	1	1
1	1	1	1	1	1	1	1

Distributive: $A\&(B|C) = (A\&B)|(A\&C)$

A	B	C	$B\|C$	$A\&B$	$A\&C$	$A\&(B\|C)$	$(A\&B)\|(A\&C)$
0	0	0	$(0\|0) = 0$	$(0\&0) = 0$	0	0	0
0	0	1	$(0\|1) = 1$	$(0\&0) = 0$	0	0	0
0	1	0	$(1\|0) = 1$	$(0\&1) = 0$	0	0	0
0	1	1	$\dots 1$	$\dots 0$	0	0	0
1	0	0	0	0	0	0	0
1	0	1	1	0	1	1	1
1	1	0	1	1	0	1	1
1	1	1	1	1	1	1	1

De Morgan's Laws: $\sim (A|B) = (\sim A)\&(\sim B)$ and $\sim (A\&B) = (\sim A)|(\sim B)$

Encoding Byte Values

- **Hexadecimal 00_{16} to FF_{16}**
  - Base 16 number representation
  - Use characters ‘0’ to ‘9’ and ‘A’ to ‘F’
- **Byte = 8 bits = 2 hex digits**
  - Binary 00000000_2 to 11111111_2
  - Decimal: 0_{10} to 255_{10}

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

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 left
- **Undefined Behavior**
  - Shift amount < 0 or ≥ word size

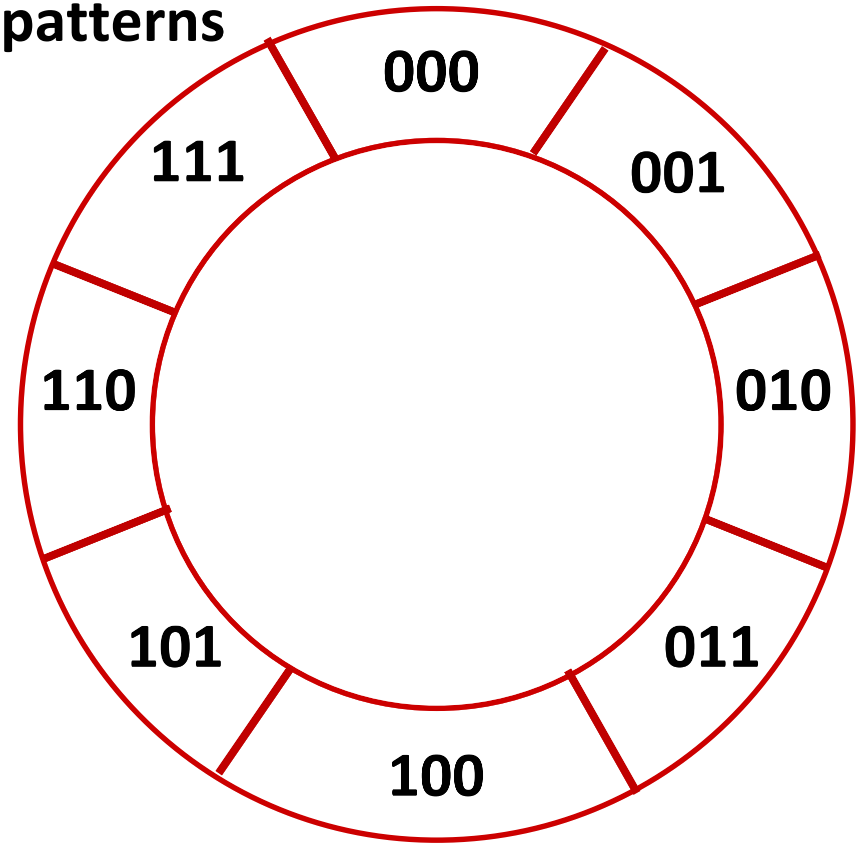
Argument x	01100010
<< 3	00010000
Log. >> 2	00011000
Arith. >> 2	00011000

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

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

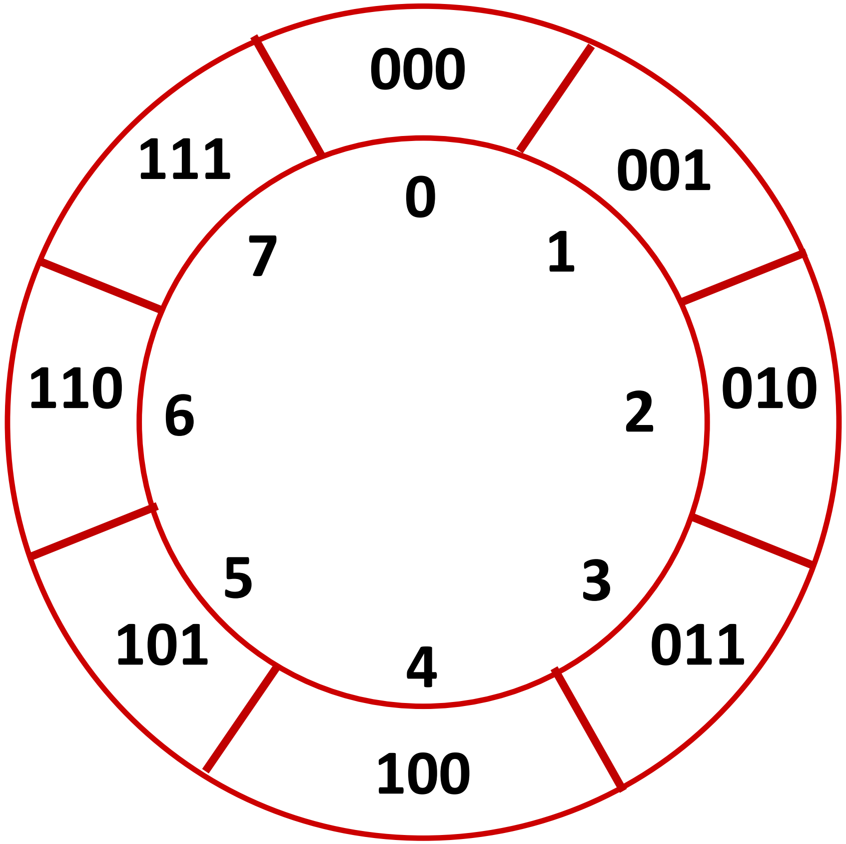
Signed vs. Unsigned numbers

- Using W bits to represent an integer gives 2^W different "states" or patterns



Signed vs. Unsigned numbers

- An unsigned representation gives $[0, 2^W-1]$



Unsigned Representation

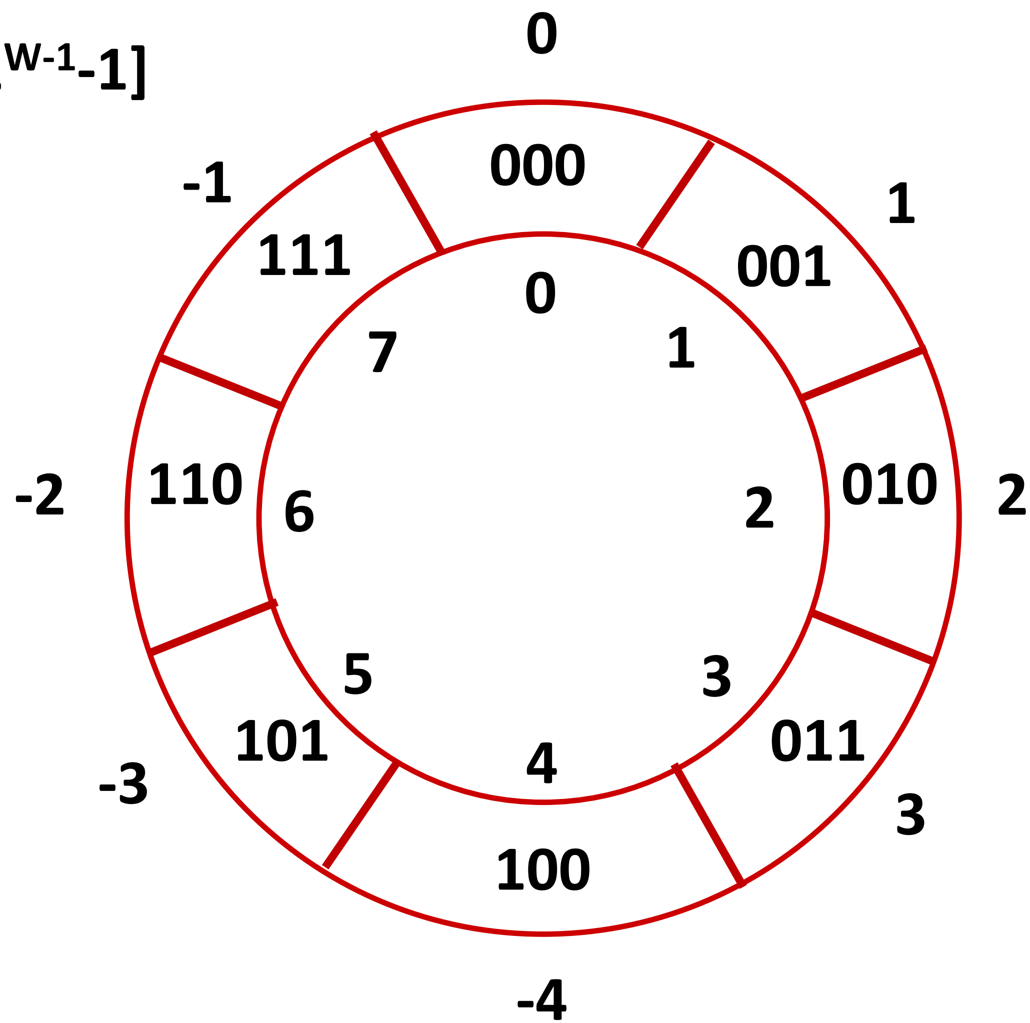
$$B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

$\boldsymbol{x_i}$ **:** $\boldsymbol{i^{\mathrm{th}}}$ **binary digit**

101 = 2^2 + 2^0 = 5

Signed vs. Unsigned numbers

- Signed gives $[-2^{W-1}, 2^{W-1}-1]$



Signed Representation

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

x_i : i^{th} binary digit

101 = $-2^2 + 2^0$ = -3

Two-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**

W = 4, Unsigned and Signed

	Unsigned	Signed
0000	0	0
1010	8+2 = 10	-8+2 = 6
0111	4+2+1 = 7	4+2+1 = 7
1000	-8	8
1111	8+4+2+1= 15	-8+4+2+1= -1

W = 4, Unsigned and Signed

	Unsigned	Signed
0000	UMin	
1010		
0111		TMax
1000		TMin
1111	UMax	

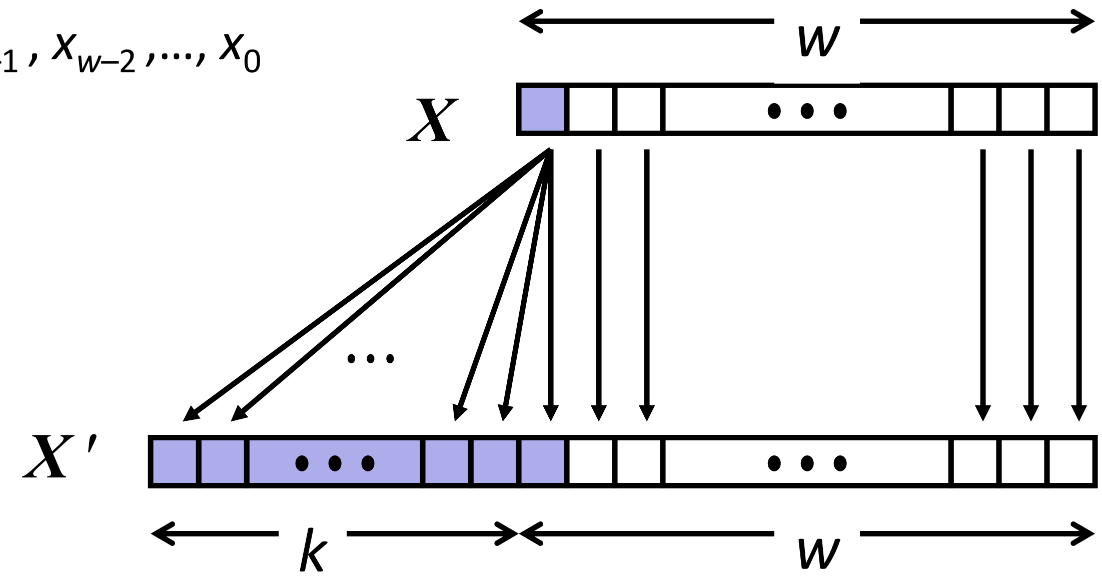
Casting Surprises

- **Expression Evaluation**
  - If there is a mix of unsigned and signed in single expression, ***signed values implicitly cast to unsigned***
  - Including comparison operations **<, >, ==, <=, >=**
  - Examples for ***W* = 32**: **TMIN = -2,147,483,648 ,** **TMAX = 2,147,483,647**

Constant$_1$	Constant$_2$	Relation	Evaluation
0	0U	==	unsigned
-1	0	<	signed
-1	0U	>	unsigned
2147483647	-2147483648	>	signed
2147483647U	-2147483648	<	unsigned
-1	-2	>	signed
(unsigned)-1	-2	>	unsigned
2147483647	2147483648U	<	unsigned
2147483647	(int) 2147483648U	>	signed

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:
  - $X' = \underbrace{x_{w-1}, \ldots, x_{w-1}}_{k \text{ copies of MSB}}, x_{w-1}, x_{w-2}, \ldots, x_0$



Sign Extension: w = 3 to w = 5

1 0 1 → 1 1 1 0 1

Sign Extension Example

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

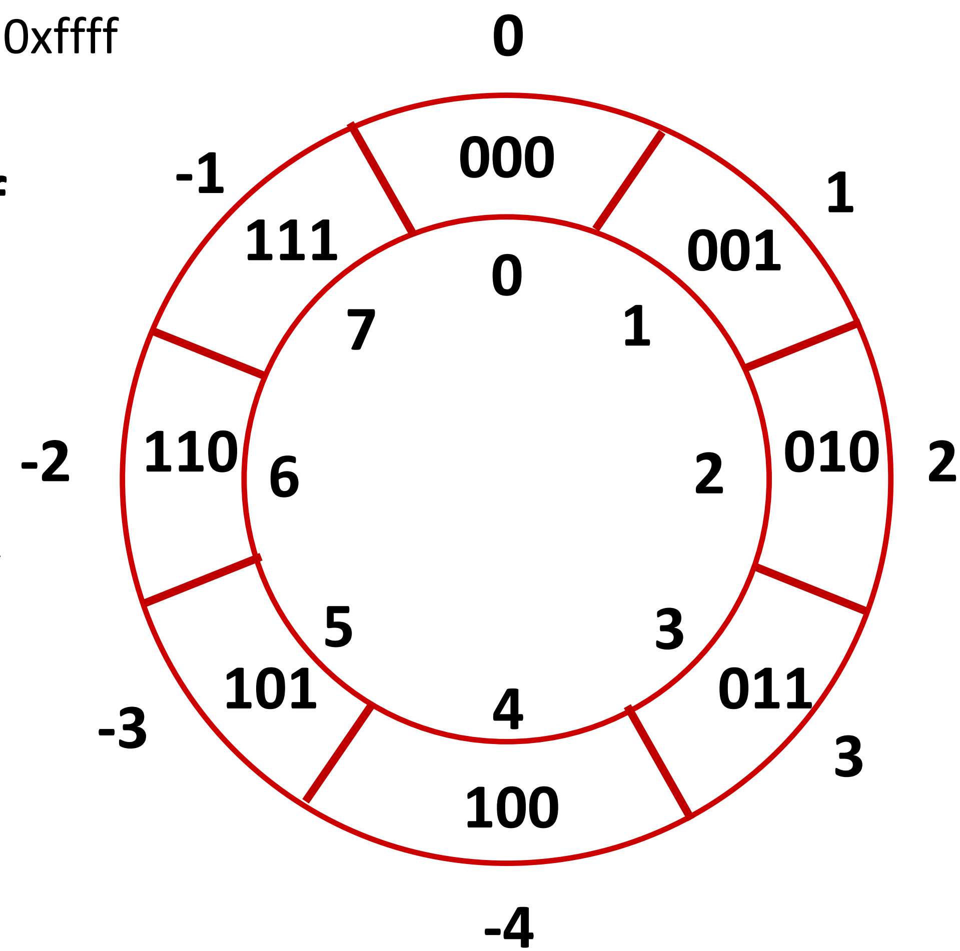
	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
ix	15213	00 00 3B 6D	00000000 00000000 00111011 01101101
y	-15213	C4 93	11000100 10010011
iy	-15213	FF FF C4 93	11111111 11111111 11000100 10010011

- **Converting from smaller to larger integer data type**
  - C automatically performs sign extension
- **When converting from larger to smaller, top is truncated**

Programmer Idioms

- **Signed value of -1 is "all ones"**
  - E.g. signed char x = -1 sets 'x' to 0xffff
  - (-1 << 8) = 0xff…f00
- **(1 << x)-1 produces "mask" of x-1 bits**
  - (1 << 5)-1 -> 0x10 - 1 -> 0x0f
- **Signed ~x+1 is the same as –x**
  - E.g. using 3 bit numbers,

~0x7 -> 0x0 + 1 -> 0x1
~0x2 -> 0x5 + 1 -> 0x6



Truncation: w = 5 to w = 3

1 0 0 1 1 → 0 1 1

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 Mathematical Operations And Memory Representations

Reading: CS:APP Chapter 2.3

Integer Mathematical Operations and Memory Representations

- **Unsigned addition**
- **Signed (2's complement) addition**
- **Unsigned multiplication**
- **Signed multiplication**
- **Power-of-two multiplication using shift**

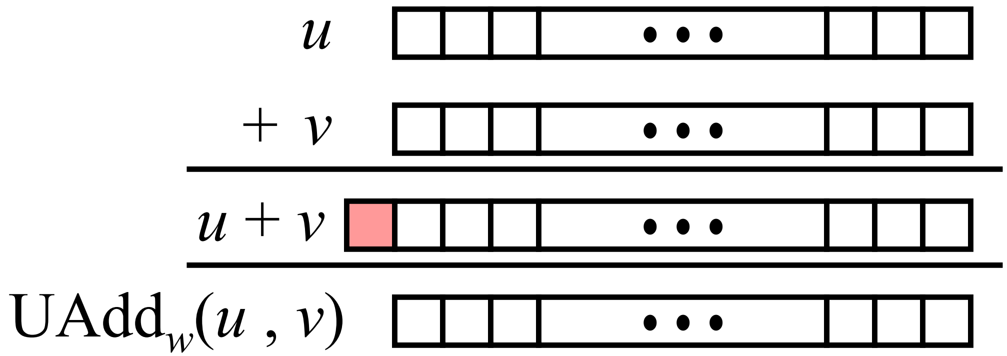
- **Representations in memory, pointers, strings**
- **Summary**

Unsigned Addition

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits



- **Standard Addition Function**
  - Ignores carry output
- **Implements Modular Arithmetic**

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

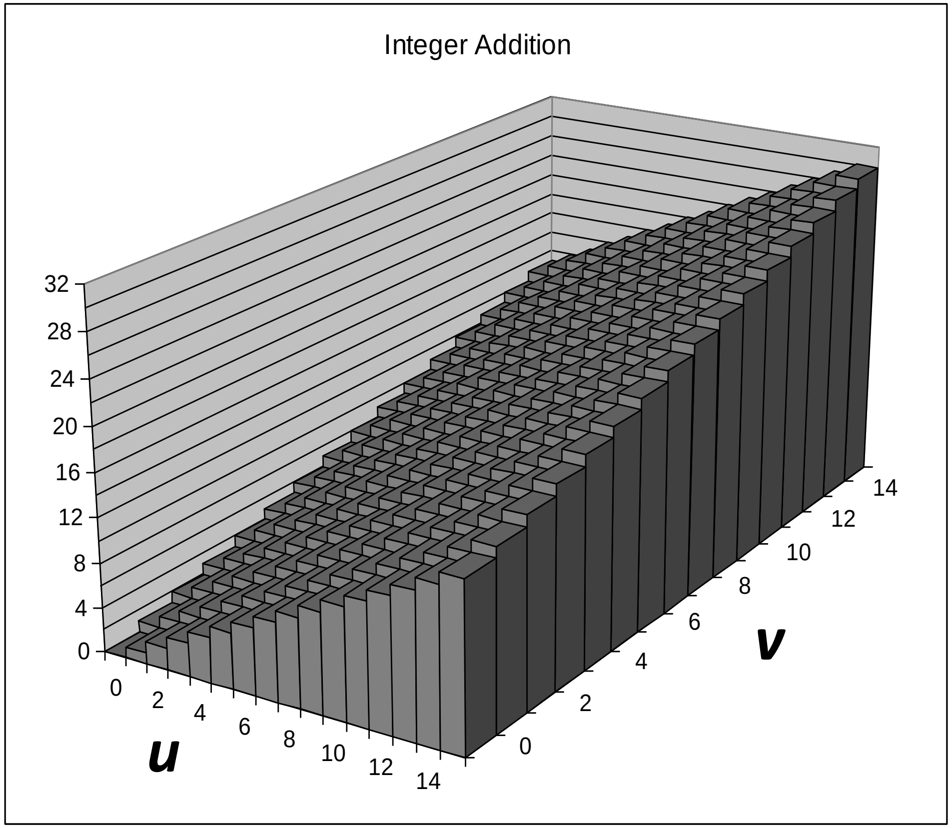
Unsigned Addition , W = 4

```
 0010   2            1010  10
+0110   6           +0110   6
 ----                ----

        8                   ?
```

Visualizing (Mathematical) Integer Addition

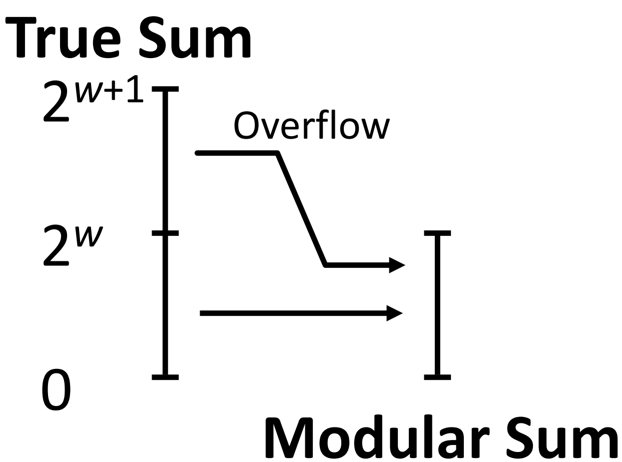
- **Integer Addition**
  - 4-bit integers *u*, *v*
  - Compute true sum $\text{Add}_4(u\ ,\ v)$
  - Values increase linearly with *u* and *v*
  - Forms planar surface
  - **but, >15 not possible! …**

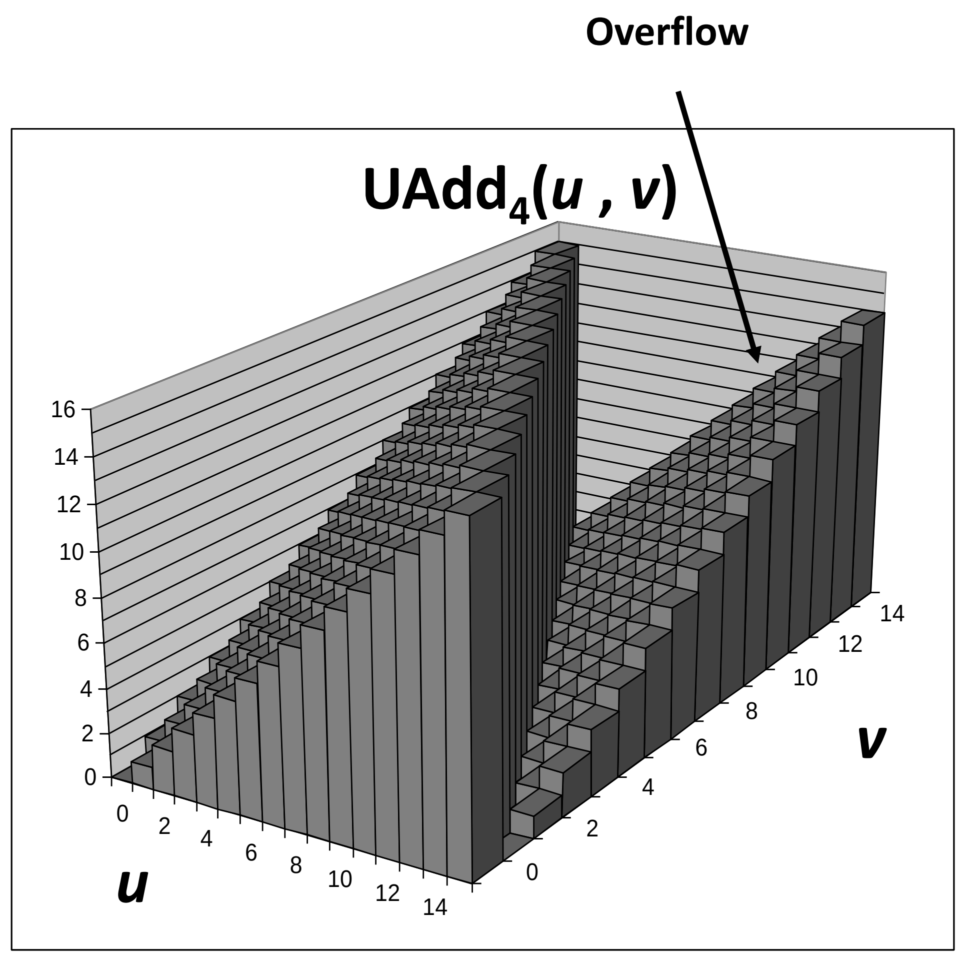
$\text{Add}_4(u\ ,\ v)$



Visualizing Unsigned Addition

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



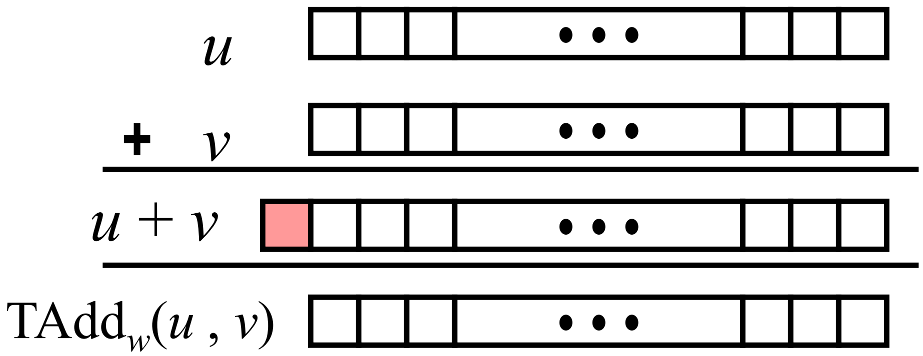


Two's Complement Addition

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`

2’s Complement (Signed) Addition , W = 4

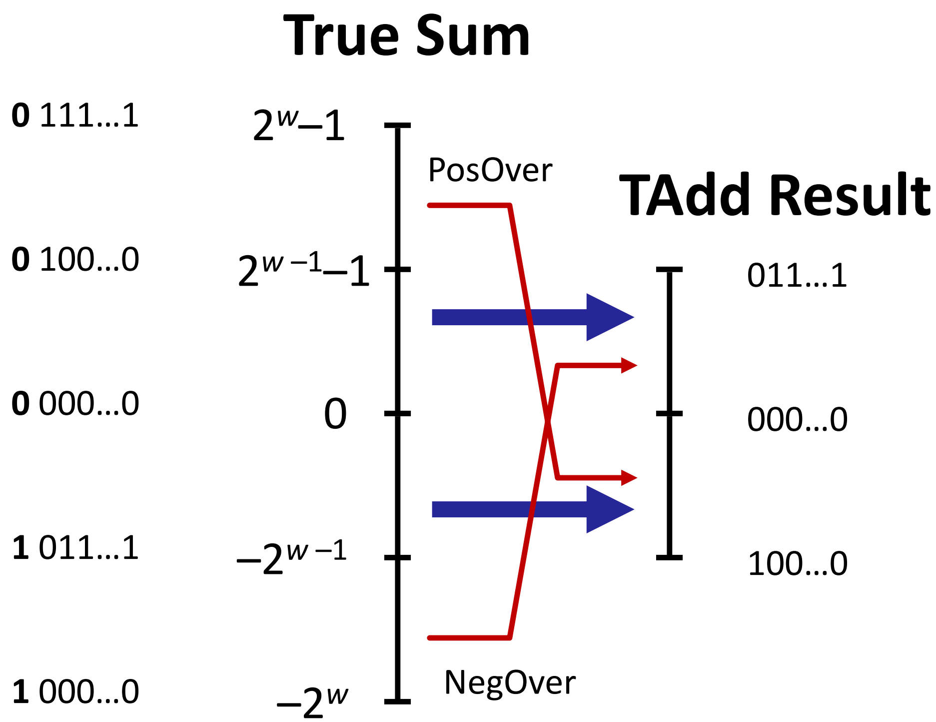
```
 0010   2        0111   7
+1010  -6       +1000  -8
 ----            ----

       -4              -1
```

2’s Complement (Signed) Addition , W = 4

```
 0111   7          1000  -8
+0001   1         +1000  -8
 ----              ----
        ?                 ?
```

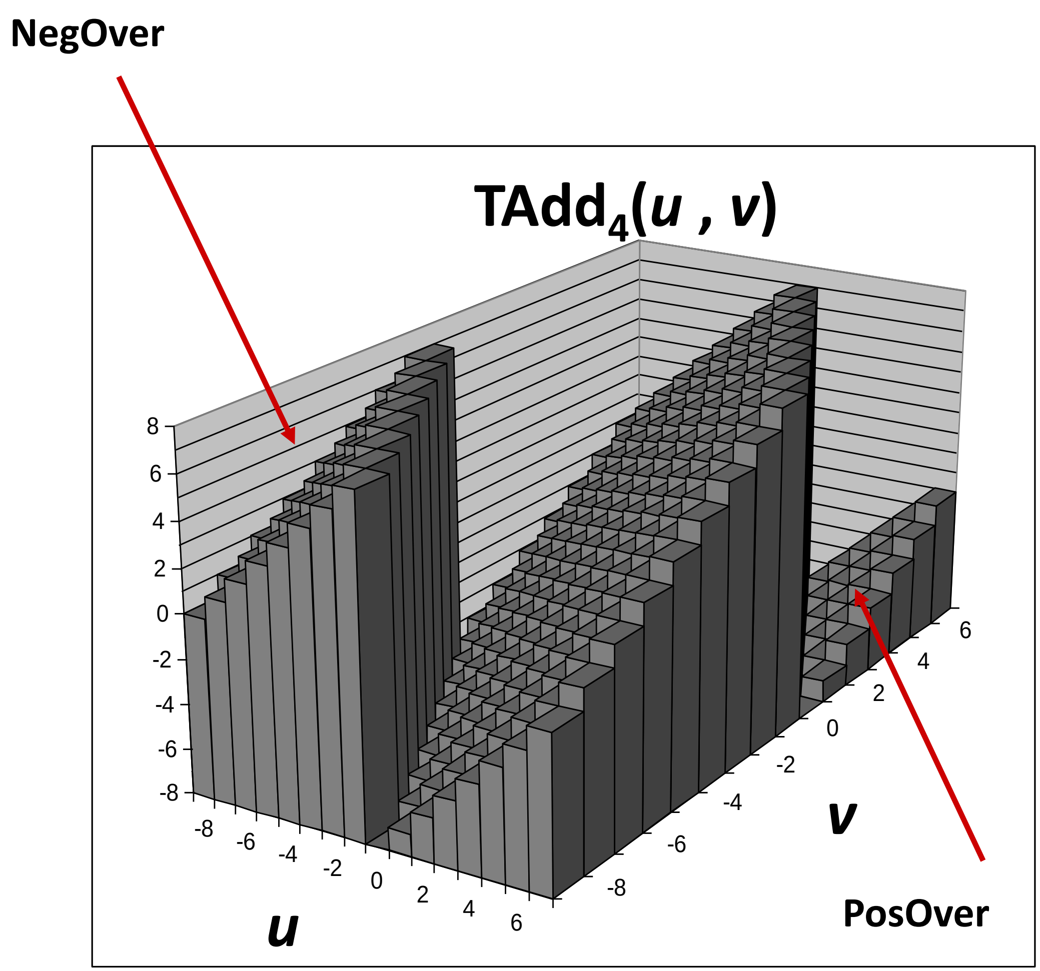
TAdd Overflow

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



Visualizing 2’s Complement 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



Multiplication, Unsigned

```
 0011     3
*0101    *5
 ----
```

- **Goal: Computing Product of *w*-bit numbers *x*, *y***
  - Either signed or unsigned

Multiplication

- **But, exact results can be bigger than *w* bits**
  - Unsigned: up to 2*w* 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 2*w*-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 2*w* bits, but only for $(TMin_w)^2$
    - Result range: $x * y \le (-2^{w-1})^{2} = 2^{2w-2}$

Multiplication

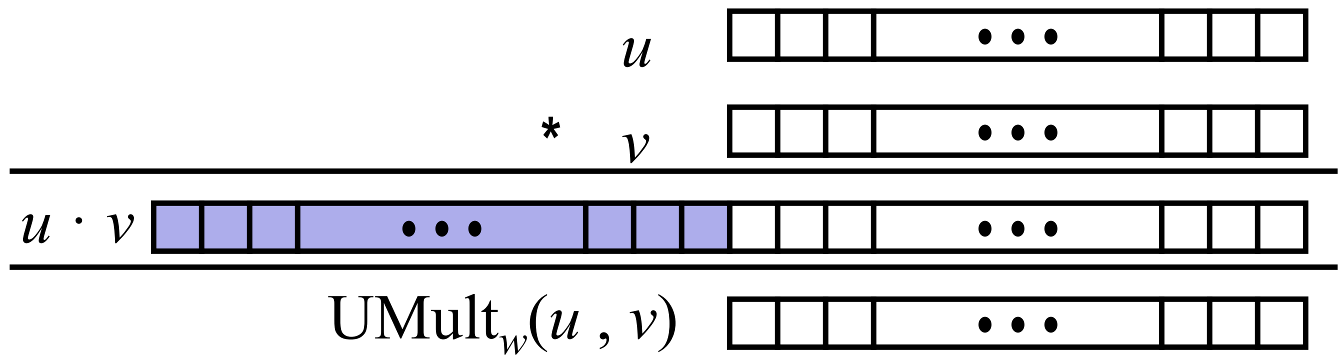
- **So, maintaining exact results…**
  - would need to keep expanding word size with each product computed
  - is done in software, if needed
    - e.g., by “arbitrary precision” arithmetic packages

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**

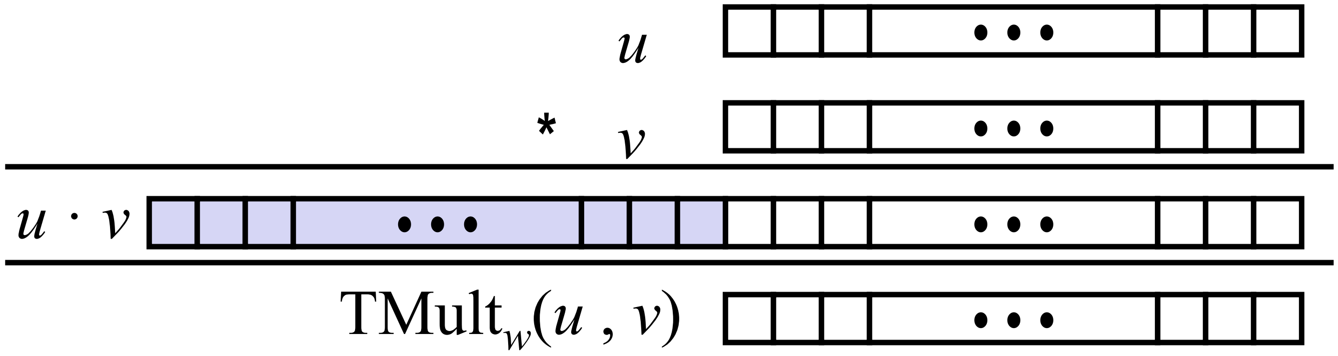
$\mathrm{UMult}_w(u, v) = u \cdot v \bmod 2^w$

Signed Multiplication in C

Operands: *w* bits

True Product: 2**w* bits

Discard *w* bits: *w* bits



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

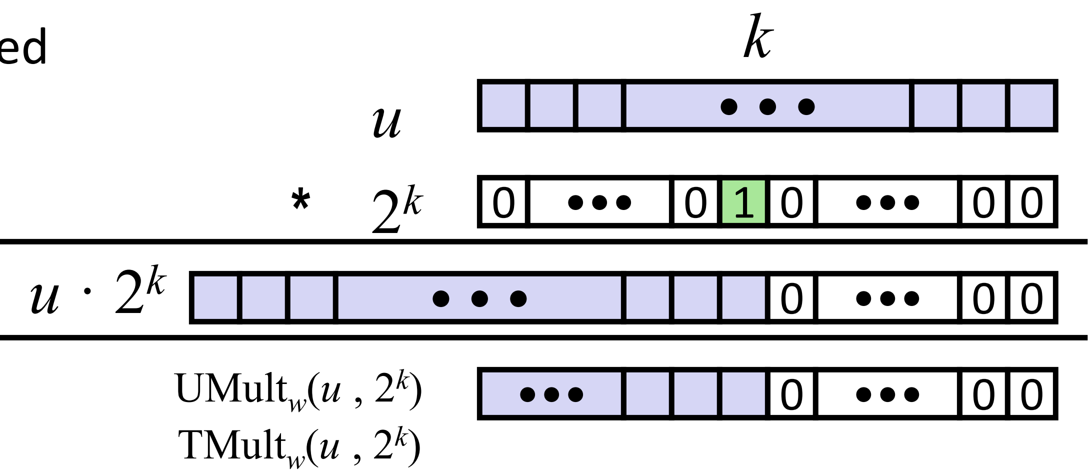
Power-of-2 Multiply with Shift

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

Operands: *w* bits

True Product: *w+k* bits

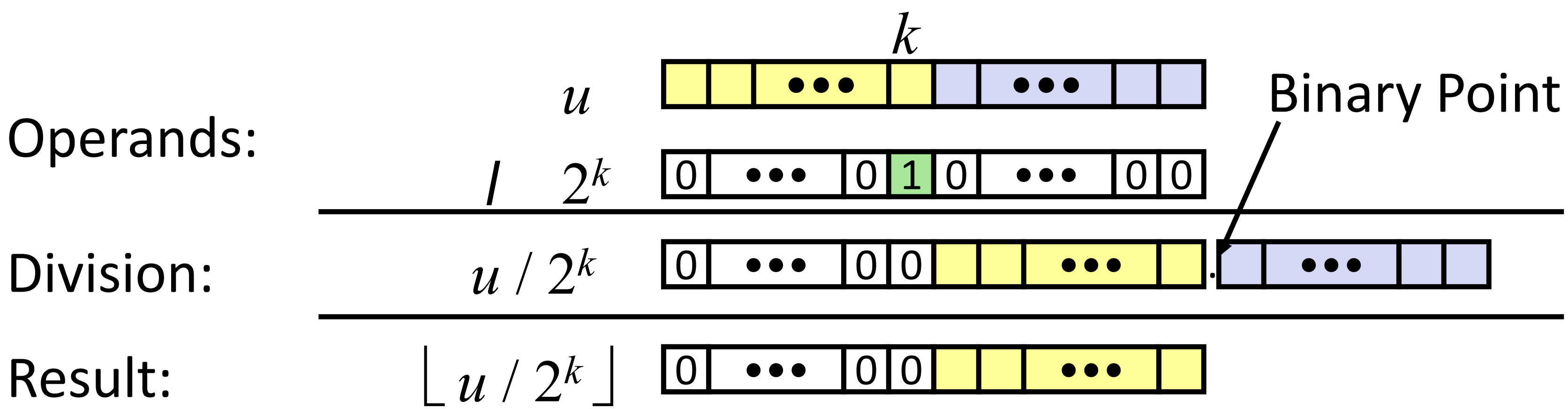
Discard *k* bits: *w* bits



- **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

- **Quotient of Unsigned by Power of 2**
  - `u >> k` gives $\lfloor$ `u / 2`^{k} $\rfloor$
  - Uses logical 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`

Compiled Signed Division Code

C Function

```
long idiv8(long x)
{
  return x/8;
}
```

- Use constants if it's constant
- Don't be clever it's built-in

Compiled Arithmetic Operations

```
  testq %rax, %rax
  js    L4
L3:
  sarq $3, %rax
  ret
L4:
  addq $7, %rax
  jmp  L3
```

Explanation

```
  if x < 0
    x += 7;
  # Arithmetic shift
  return x >> 3;
```

- Uses arithmetic shift for int
- For Java Users
  - Arith. shift written as >>

Integer Mathematical Operations and Memory Representations

- Unsigned addition
- Signed (2's complement) addition
- Unsigned multiplication
- Signed multiplication
- Power-of-two multiplication using shift
- **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

- **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? (cont.)

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