Chapter 3: Representing integers in C

252-0061-00 V Systems Programming and Computer Architecture

Goal:
Introduction to integer arithmetic on computers

- Recap of how numbers are represented
- Signed and unsigned values
- Integer ranges
- Integer addition and subtraction
    - In C
    - Mathematical properties
- Integer multiplication
    - In C
    - Mathematical properties
- Integer multiplication and division using shifts

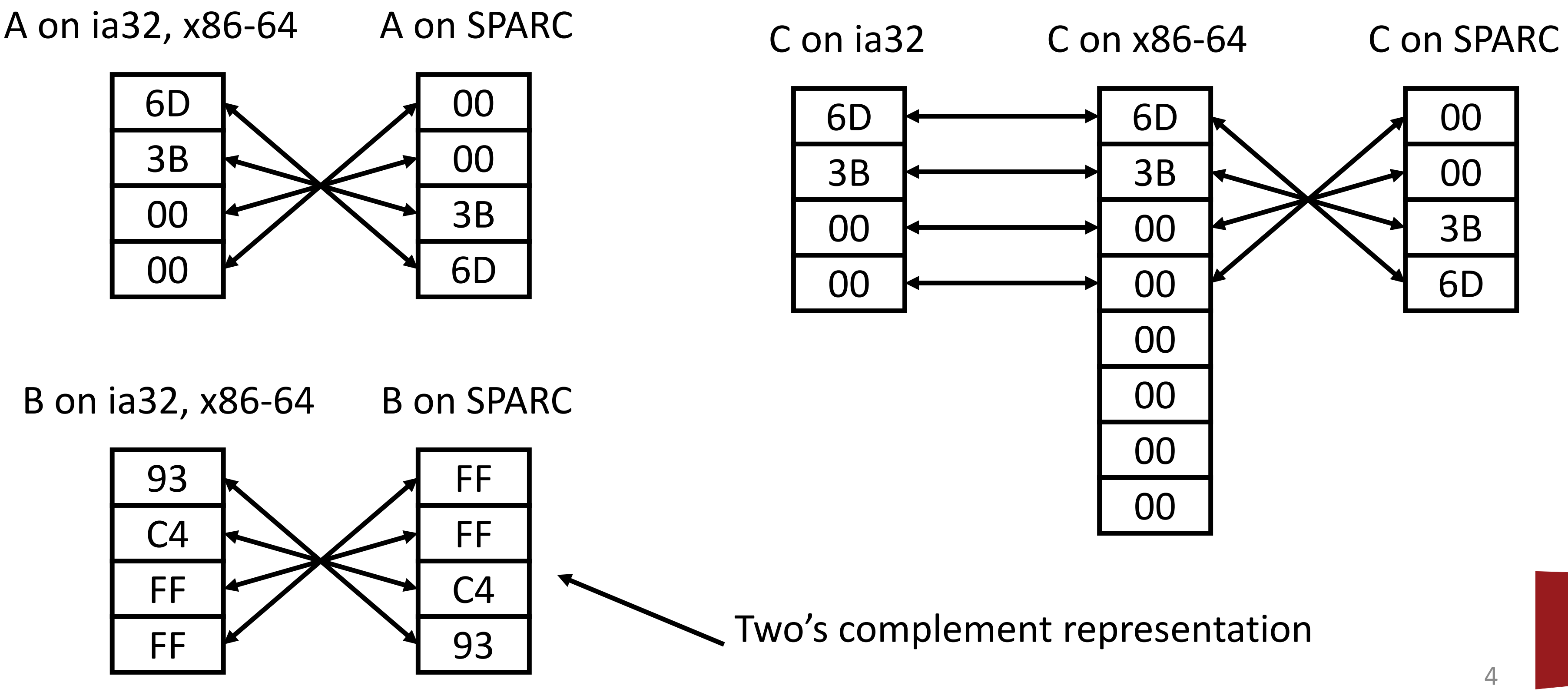
3.1: Recap: Encodings and operators

Computer Architecture and Systems Programming

Representing integers

```
int A = 15213;
int B = -15213;
long int C = 15213;
```

Decimal:	15213
Binary:	0011 1011 0110 1101
Hex:	3 B 6 D



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

&	Bitwise AND
|	Bitwise OR
~	Bitwise NOT
^	Bitwise XOR

- Examples (using `char` data type):
  - `~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

Systems@ETH Zürich

Contrast: Logic operations in C

- &&, ||, !
  - View 0 as "False"
  - Anything nonzero as "True"
  - Always return 0 or 1
  - Early termination

&&	Logical AND
||	Logical OR
!	Logical NOT

- Examples (`char` data type)
  - `!0x41` → `0x00`
  - `!0x00` → `0x01`
  - `!!0x41` → `0x01`

  - `0x69 && 0x55` → `0x01`
  - `0x69 || 0x55` → `0x01`

Representing & manipulating sets

- Width w bit vector represents subsets of {0, …, w–1}
- $a_j = 1$ if $j \in A$:

```
01101001      {0, 3, 5, 6 }
76543210

01010101      {0, 2, 4, 6 }
76543210
```

Operator	Operation	Result	Meaning
&	Intersection	01000001	{ 0, 6 }
|	Union	01111101	{ 2, 3, 4, 5, 6 }
^	Symmetric difference	00111100	{ 2, 3, 4, 5 }
~	Complement	10101010	{ 1, 3, 5, 7 }

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	00010*000*
Log. >> 2	*00*011000
Arith. >> 2	*00*011000

Argument x	10100010
<< 3	00010*000*
Log. >> 2	*00*101000
Arith. >> 2	*11*101000

Java writes this “>>>”.

Summary

- Integer encoding
- Bit-level operators
- Set representation
- Logic operators
- Shift operators

3.2: Integer ranges

Computer Architecture and Systems Programming

Encoding integers

Unsigned

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

Two's complement

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

Sign Bit

```
short int x =  15213;
short int y = -15213;
```

- A C `short` is 2 bytes long:

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
y	-15213	C4 93	11000100 10010011

- Sign bit
  - For 2's complement, most significant bit = 1 indicates negative

Numeric ranges

- Unsigned values
  - UMin = 0
    - 000…0
  - UMax = $2^w - 1$
    - 111…1
- Two's complement values
  - TMin = -2^{w-1}
    - 100…0
  - TMax = $2^{w-1} - 1$
    - 011…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

Values for different word sizes

	8	16	32	64
UMax	255	65,535	4,294,967,295	18,446,744,073,709,551,615
TMax	127	32,767	2,147,483,647	9,223,372,036,854,775,807
TMin	-128	-32,768	-2,147,483,648	-9,223,372,036,854,775,808

- Observations:
  - |TMin | = TMax + 1
    - Asymmetric range
  - UMax = 2 * TMax + 1

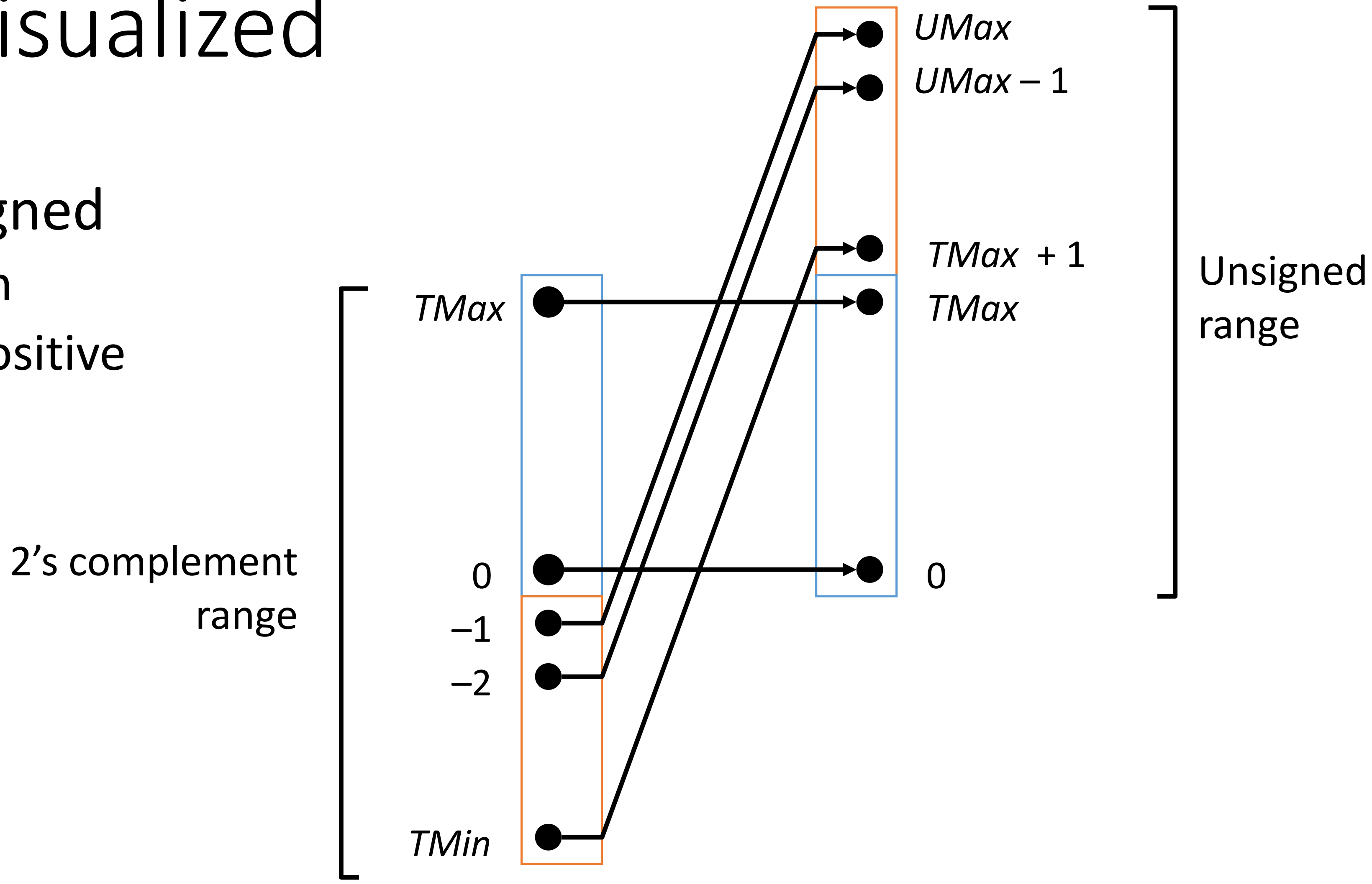
- C Programming
  - `#include <limits.h>`
  - Declares constants, e.g.,
    - `ULONG_MAX`
    - `LONG_MAX`
    - `LONG_MIN`
  - Values platform specific

Mapping signed ↔ unsigned

Bits	Signed		Unsigned
0000	0		0
0001	1		1
0010	2		2
0011	3	=	3
0100	4		4
0101	5		5
0110	6		6
0111	7		7
1000	-8		8
1001	-7		9
1010	-6	+16	10
1011	-5		11
1100	-4		12
1101	-3		13
1110	-2		14
1111	-1		15

Conversion visualized

- 2’s Comp. → Unsigned
  - Ordering inversion
  - Negative → big positive



Signed vs. unsigned in C

- Constants
  - By default are considered to be signed integers
  - Unsigned if have “U” as suffix:

```
0U
4294967259U
```

- Casting
  - Explicit casting between signed & unsigned same as U2T and T2U:

```
int tx, ty;
unsigned ux, uy;
tx = (int) ux;
uy = (unsigned) ty;
```

  - Implicit casting also occurs via assignments and procedure calls

```
tx = ux;
uy = ty;
```

Casting surprises in expression evaluation

- When mixing unsigned and signed in single expression, ***signed values implicitly cast to unsigned***
  - Including comparison operators! <, >, ==, <=, >=
- 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`	`-2147483647-1`	`>`	Signed
`2147483647U`	`-2147483647-1`	`<`	Unsigned
`-1`	`-2`	`>`	Signed
`(unsigned)-1`	`-2`	`>`	Unsigned
`2147483647`	`2147483648U`	`<`	Unsigned
`2147483647`	`(int) 2147483648U`	`>`	signed

Code security example from way back (but…)

```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

- Similar to code found in FreeBSD’s implementation of `getpeername`
- There are legions of smart people trying to find vulnerabilities in programs…

Malicious usage

```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

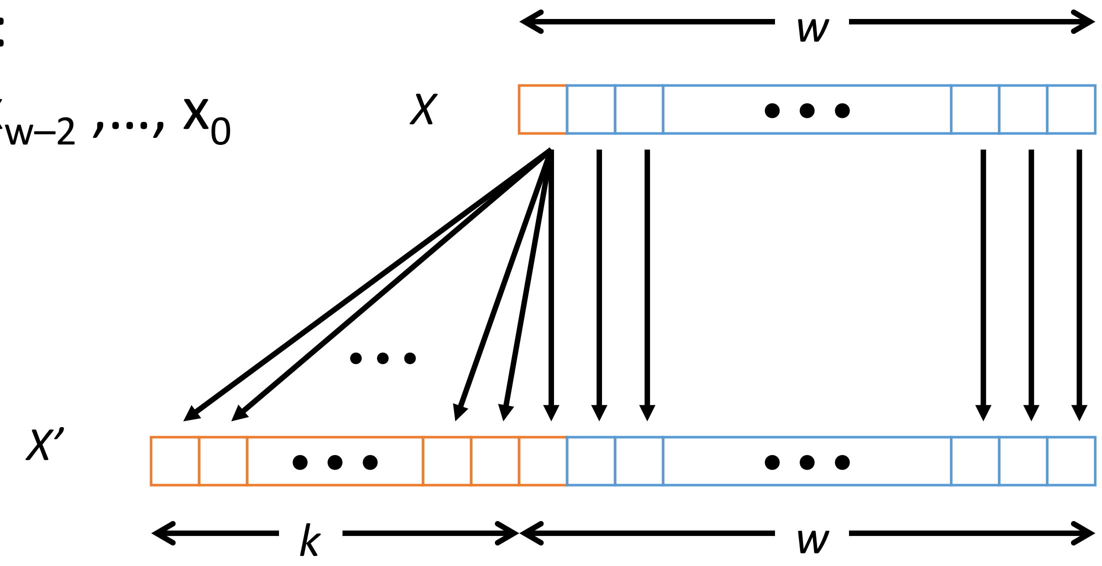
```
/* Declaration of library function memcpy */
void *memcpy(void *dest, void *src, size_t n);
```

```
#define MSIZE 528

void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, -MSIZE);
    . . .
}
```

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 example

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

```
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

Summary

- Casting signed ↔ unsigned
  - Bit pattern is maintained but reinterpreted
  - Can have unexpected effects: adding or subtracting 2^{w-1}
- Expression containing signed and unsigned `int`
  - `int` is cast to unsigned!
- Expanding (e.g., `short int` → `int`)
  - Unsigned: zeros added
  - Signed: sign extension
- Truncating (e.g., `unsigned` → `unsigned short`)
  - Unsigned: modulus operation
  - Signed: similar to modulus
  - For small numbers yields expected behaviour

3.3: Integer addition and subtraction in C

Computer Architecture and Systems Programming

Negation: complement & increment

- Recall the following holds for 2's complement:

  `~x + 1 == -x`

- Complement

  Observation: `~x + x == 1111…111 == -1`

`x`	1	0	0	1	1	1	0	1
`+ ~x`	0	1	1	0	0	0	1	0
`-1`	1	1	1	1	1	1	1	1

- Complete proof?

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

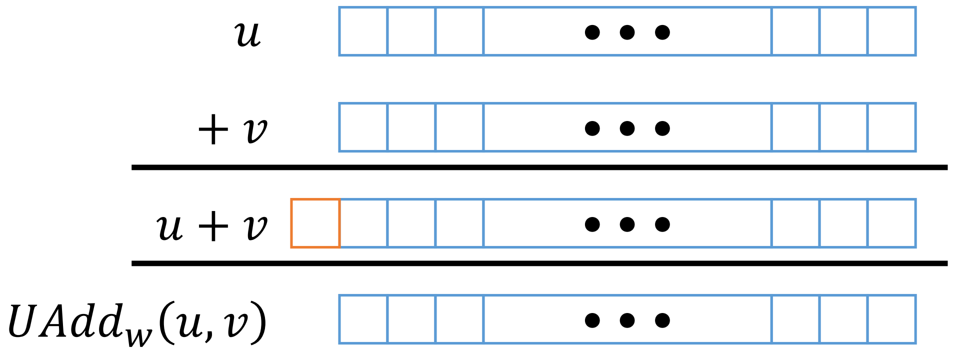
	Decimal	Hex	Binary
0	0	00 00	00000000 00000000
~0	-1	FF FF	11111111 11111111
~0+1	0	00 00	00000000 00000000

Unsigned addition

Operands: *w* bits

True sum: *w*+1 bits

Discard carry: *w* bits



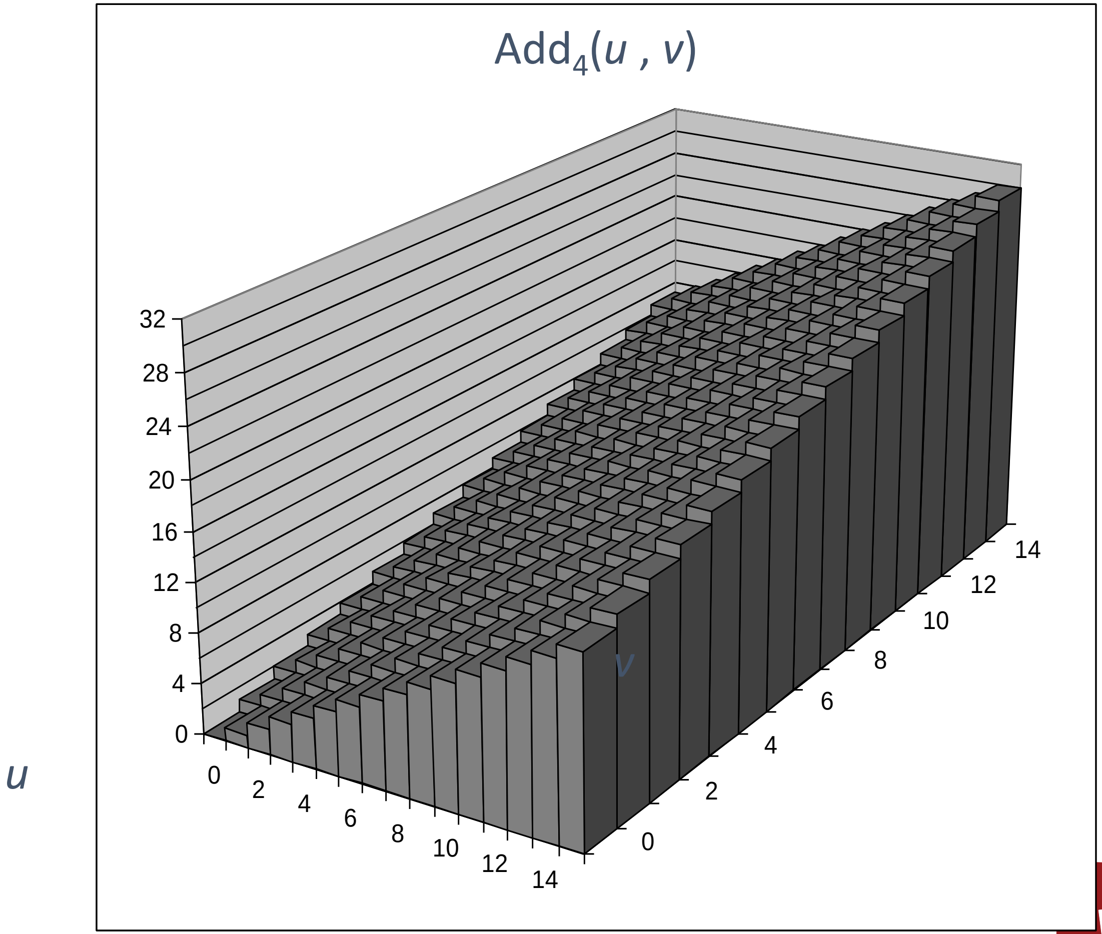
- Standard addition function
  - Ignores carry output
- Implements modular arithmetic

$$\boxed{s = UAdd_w(u,v) = u + v \mod 2^w}$$

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

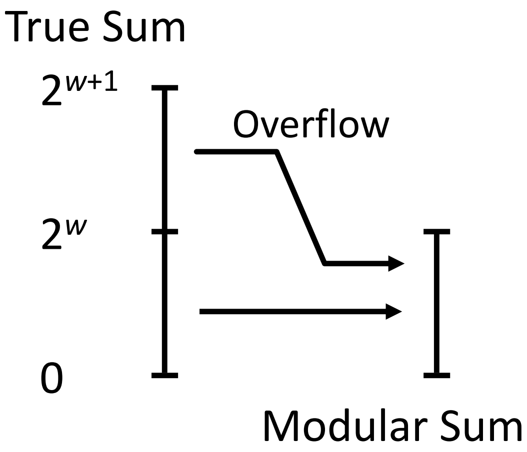
Visualizing (mathematical) integer addition

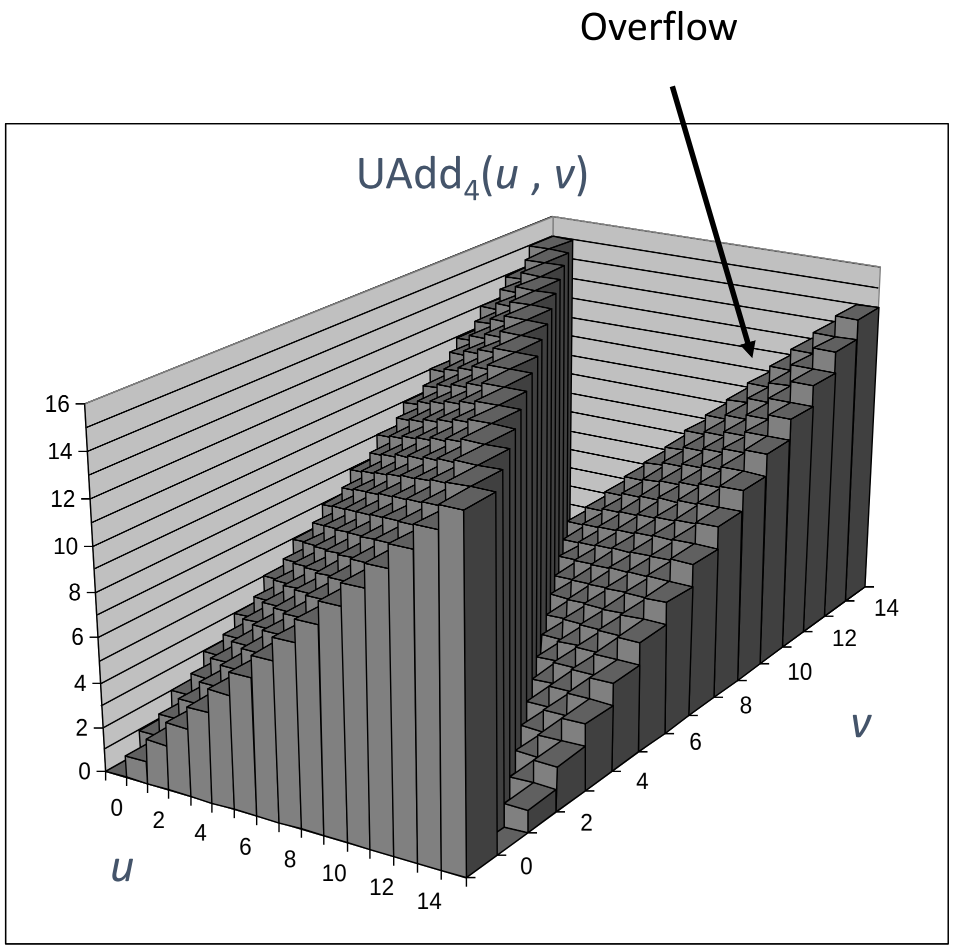
- 4-bit integers u, v
- Compute true sum $Add_4(u , v)$
- Values increase linearly with u and v
- Forms planar surface



Visualizing unsigned addition

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





Mathematical properties

- Modular addition forms an Abelian group
  - **Closed** under addition

$$0 \le UAdd_w(u,v) \le 2^w - 1$$

  - **Commutative**

$$UAdd_w(u,v) = UAdd_w(v,u)$$

  - **Associative**

$$UAdd_w\big(t, UAdd_w(u,v)\big) = UAdd_w(UAdd_w(t,u), v)$$

  - 0 is additive **identity**

$$UAdd_w(u,0) = u$$

  - Every element has additive **inverse**

Let: $UComp_w(u) = 2^w - u$

Then: $UAdd_w\big(u, UComp_w(u)\big) = 0$

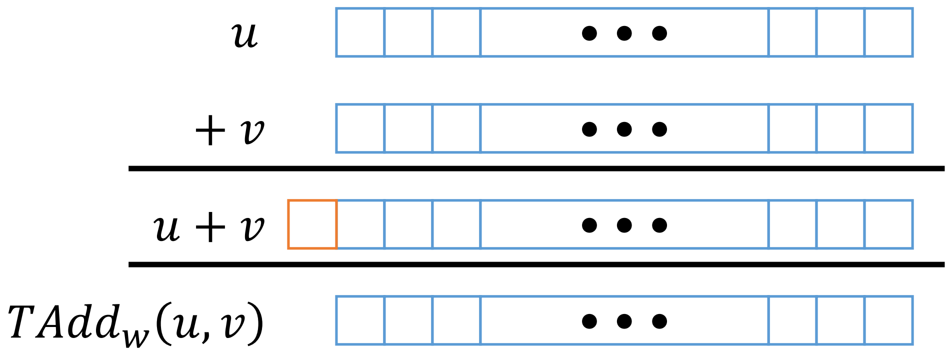
Systems@ETH Zürich

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:

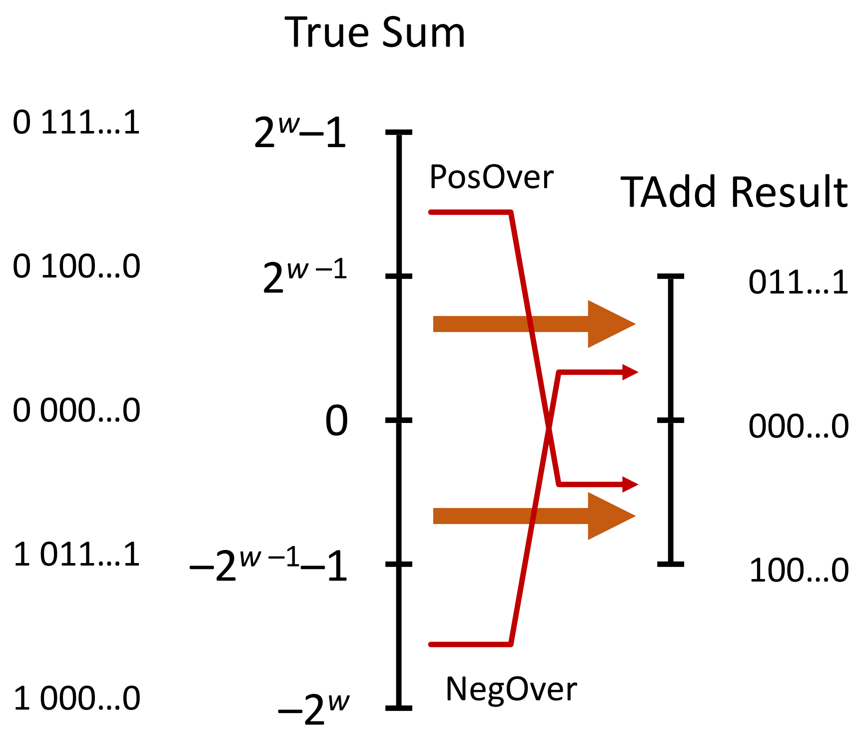
```
int s, t, u, v;
s = (int) ((unsigned) u + (unsigned) v);
t = u + v;
```

will give:

```
s == t
```

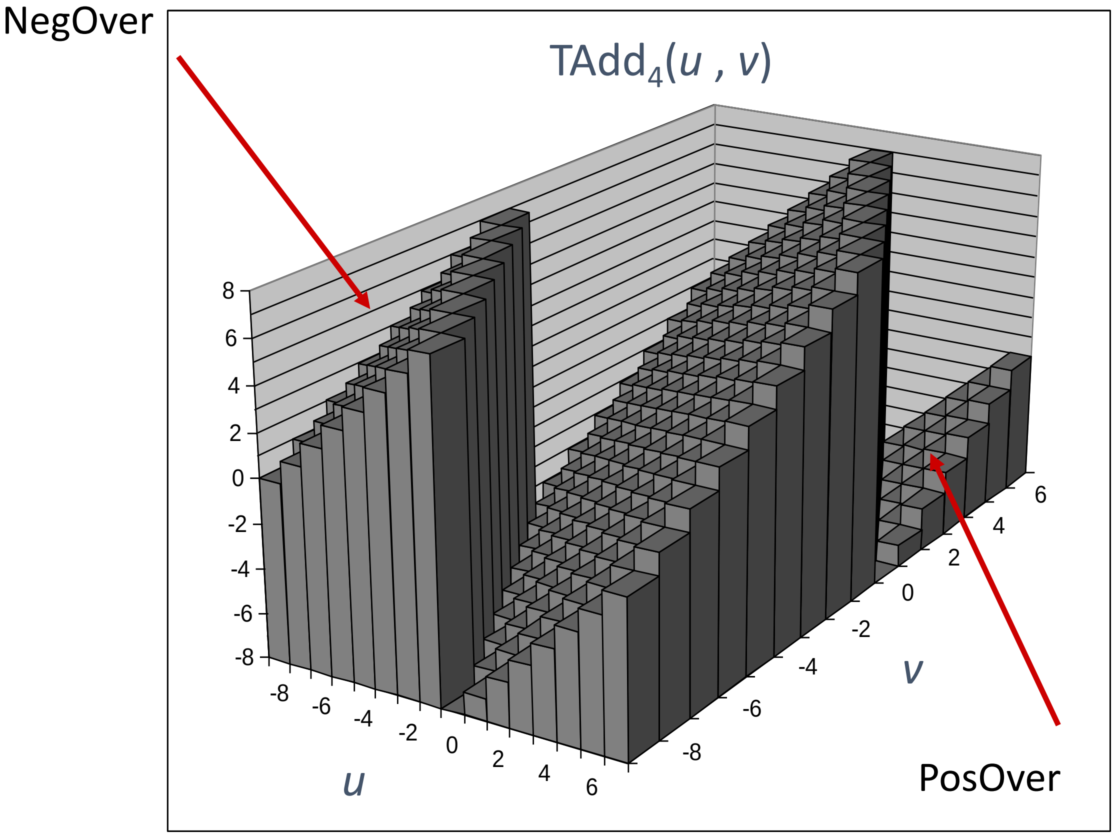
TAdd overflow

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



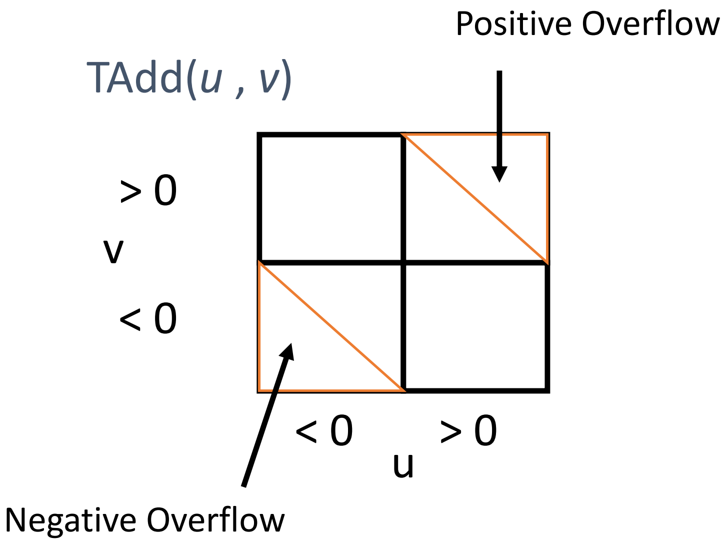
Visualizing 2's complement addition

- 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



Characterizing TAdd

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



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

(Neg. overflow)

(Pos. overflow)

Mathematical properties of TAdd

- Group isomorphic to unsigneds with UAdd
  - Since both have identical bit patterns

$$\boxed{TAdd_w(u,v) = U2T\left(UAdd_w\left(T2U(u), T2U(v)\right)\right)}$$

- 2's complement under TAdd forms a group
  - Closed, commutative, associative,
  - 0 is additive identity
  - Every element has additive inverse

$$\boxed{TComp_w(u) = \begin{cases} -u & u \neq TMin_w \\ TMin_w & u = TMin_w \end{cases}}$$

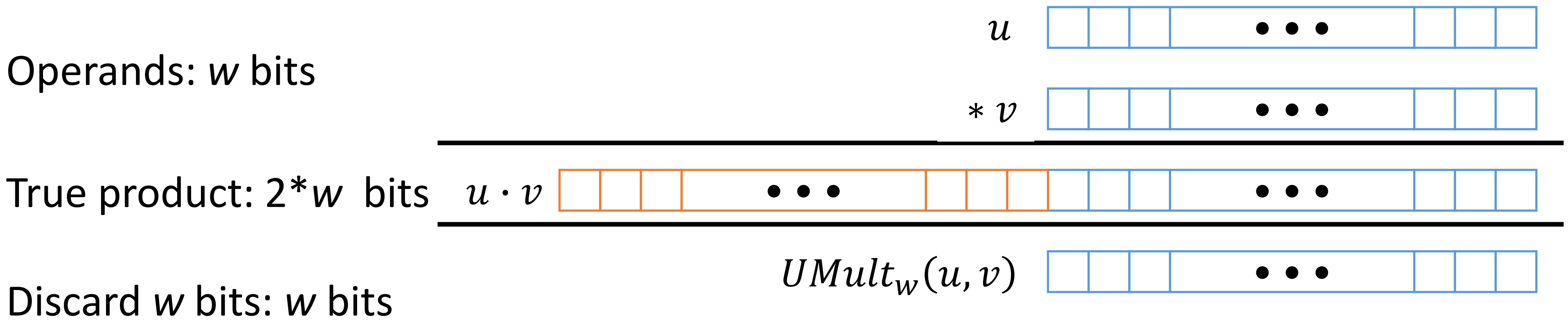
3.4: Integer multiplication in C

Computer Architecture and Systems Programming

Multiplication

- Computing exact product of w-bit numbers x, y
  - Either signed or unsigned
- Ranges
  - Unsigned (up to $2W$ bits):
$$0 \leq x * y \leq (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$$
  - Two's complement min (up to $2W - 1$ bits):
$$x * y \geq (-2^{w-1}) * (2^{w-1} - 1) = -2^{2w-2} + 2^{w-1}$$
  - Two's complement max (up to $2W - 2$ bits, but only for $(TMin_w)^2$):
$$x * y \leq (-2^{w-1})^2 = 2^{2w-2}$$
- Maintaining exact results
  - Would need to keep expanding word size with each product computed
  - Done in software by "arbitrary precision" arithmetic packages

Unsigned multiplication in C



- Standard multiplication function
  - Ignores high order w bits
- Implements modular arithmetic

$$\boxed{UMult_w(u, v) = u \cdot v \mod 2^w}$$

Unsigned multiplication with addition forms a commutative ring:

- Addition is a commutative group
- Closed under **multiplication**

$$0 \leq UMult_w(u, v) \leq 2^w - 1$$

- Multiplication is **commutative**

$$UMult_w(u, v) = UMult_w(v, u)$$

- Multiplication is **associative**

$$UMult_w(t, UMult_w(u, v)) = UMult_w(UMult_w(t, u), v)$$

- 1 is multiplicative **identity**

$$UMult_w(u, 1) = u$$

- Multiplication **distributes** over addition

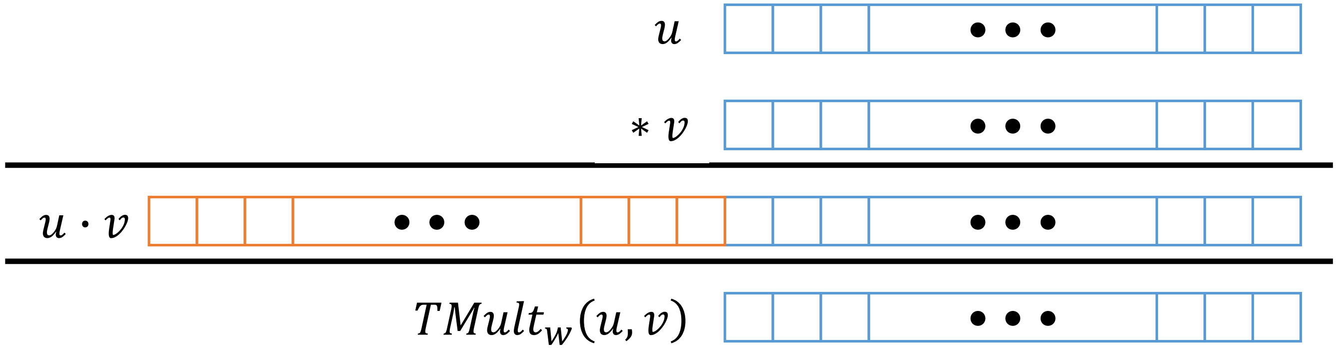
$$UMult_w(t, UAdd_w(u, v)) = UAdd_w(UMult_w(t, u), UMult_w(t, v))$$

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

Signed multiplication

- Isomorphic algebra to unsigned multiplication and addition
  - Both isomorphic to ring of integers mod 2^w

- Comparison to (mathematical) integer arithmetic
  - Both are rings
  - True integers obey ordering properties, e.g.,

$$u > 0 \Rightarrow u + v > v$$
$$u > 0, v > 0 \Rightarrow u \cdot v > 0$$

  - *Not the case* for two's complement arithmetic:

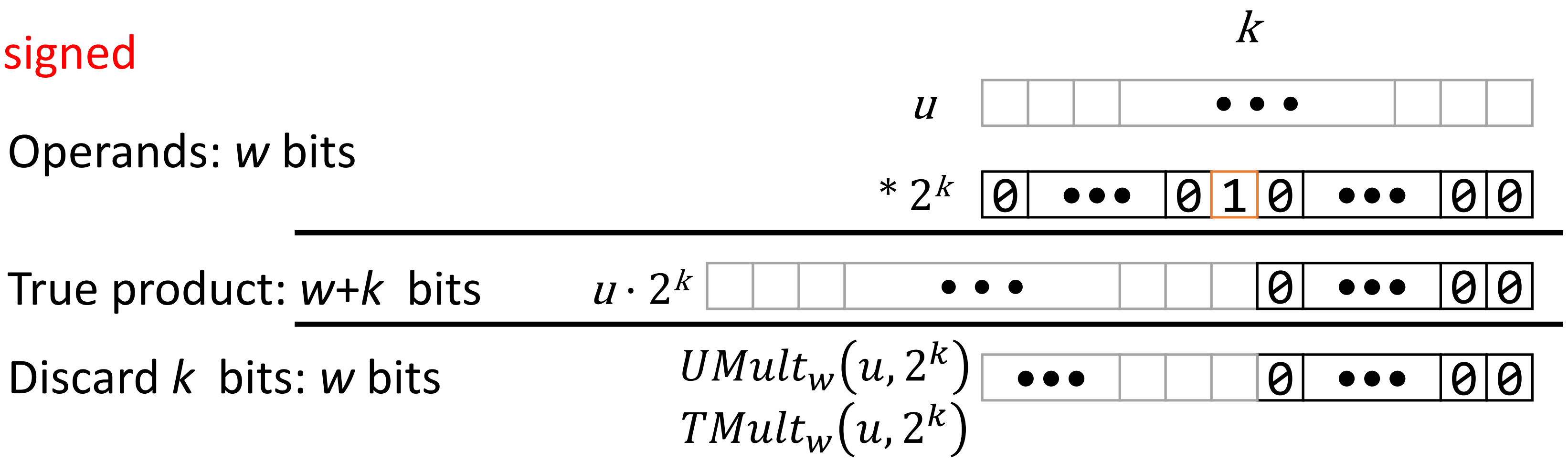
```
TMax + 1      == TMin
15213 * 30426 == -10030
```
(e.g. for 16-bit words)

3.5: Integer multiplication and division using shifts

Computer Architecture and Systems Programming

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

Compiled multiplication code

C Function

```
int mul12(int x)
{
  return x*12;
}
```

Compiled arithmetic operations

```
leal (%eax,%eax,2), %eax
sall $2, %eax
```

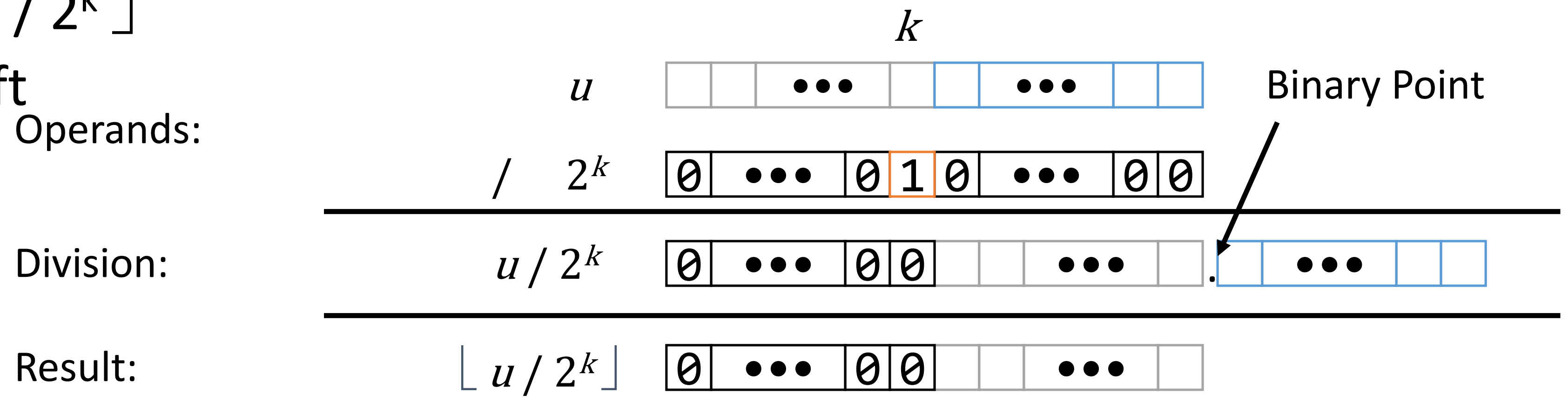
Explanation

```
t <- x+x*2
return t << 2;
```

- C compiler automatically generates shift/add code when multiplying by constant

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

Compiled unsigned division code

C Function

```
unsigned udiv8(unsigned x)
{
  return x/8;
}
```

Compiled Arithmetic Operations

```
  shrl  $3, %eax
```

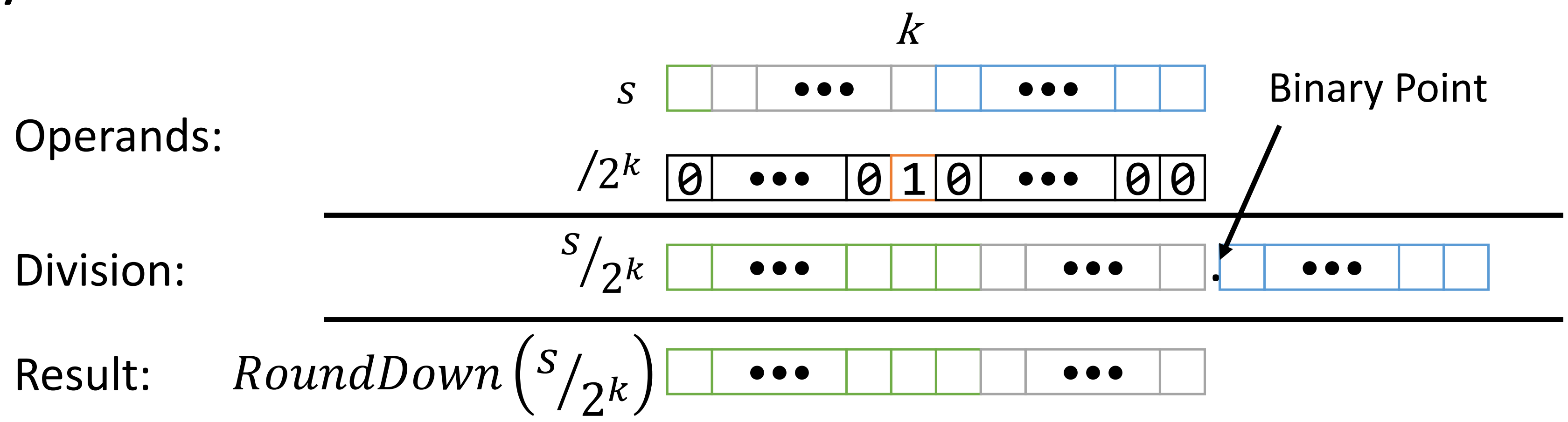
Explanation

```
  # Logical shift
  return x >> 3;
```

- Uses logical shift for unsigned
- For Java users: logical shift written as >>>

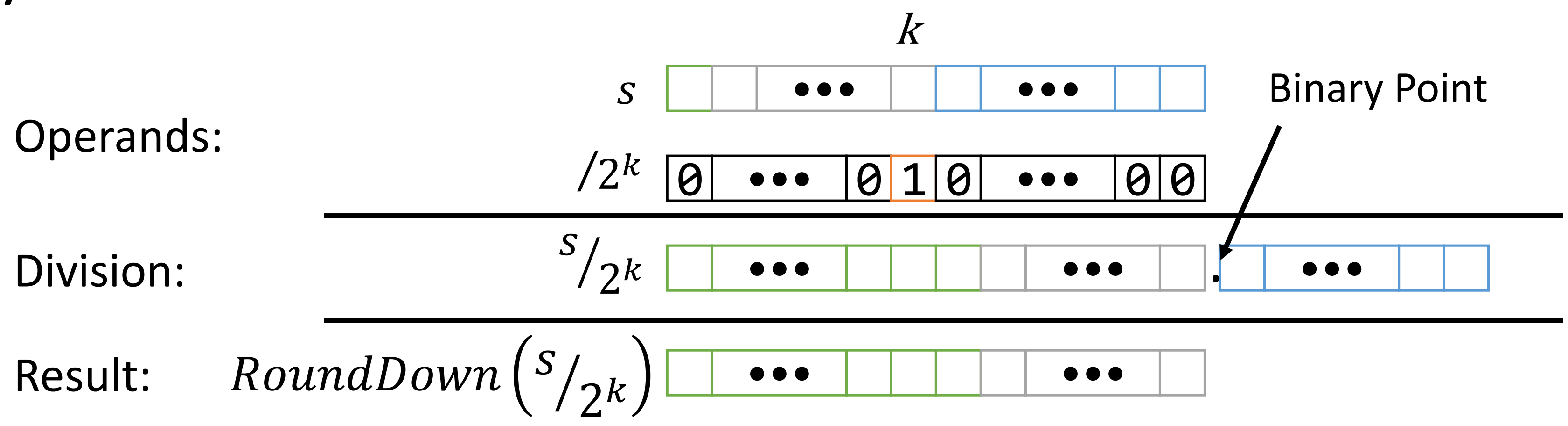
Signed power-of-2 divide w/ shift

- Quotient of Signed by Power of 2
  - s >> k gives ⌊ s / 2^k ⌋
  - Uses arithmetic shift



Signed power-of-2 divide w/ shift

- Quotient of Signed by Power of 2
  - s >> k gives ⌊ s / 2^k ⌋
  - Uses arithmetic shift
  - **Rounds wrong direction when s < 0.**



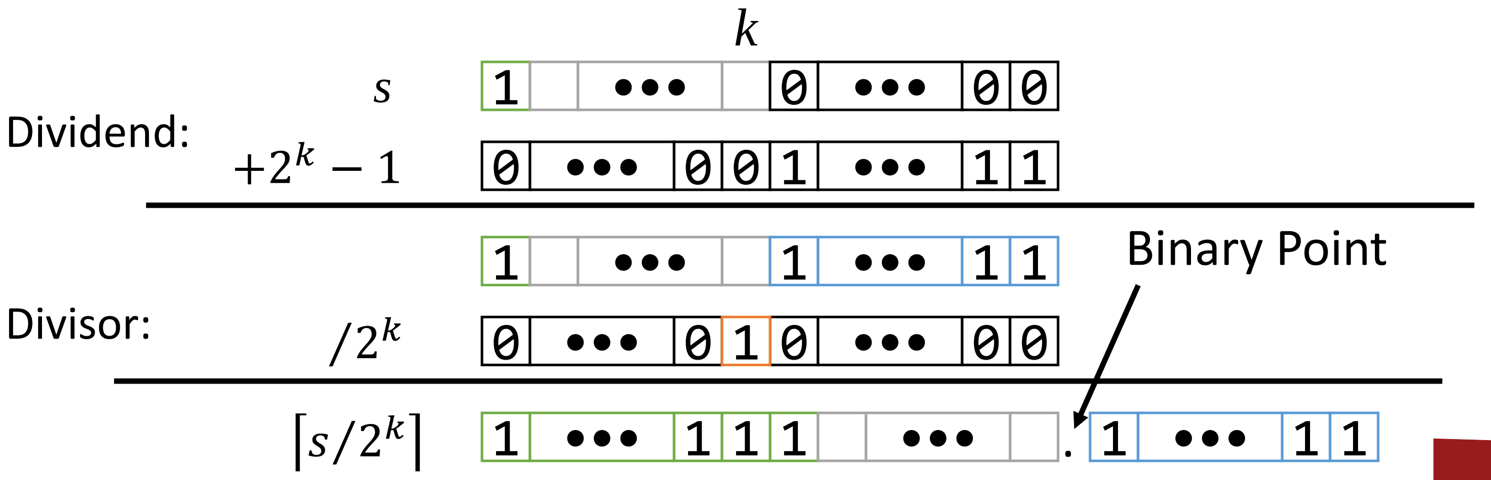
	Division	Computed	Hex	Binary
s	-15213	-15213	C4 93	11000100 10010011
s >> 1	-7606.5	-7607	E2 49	11100010 01001001
s >> 4	-950.8125	-951	FC 49	11111100 01001001
s >> 8	-59.4257813	-60	FF C4	11111111 11000100

Correct power-of-2 divide

- Quotient of negative number by power of 2
  - We want $\lceil s/2^k \rceil$ (round toward 0)
  - We compute it as $\lfloor (s + 2^k - 1)/2^k \rfloor$
    - In C: (s + (1<<k)-1) >> k
    - Biases the dividend toward 0

- Case 1: No rounding

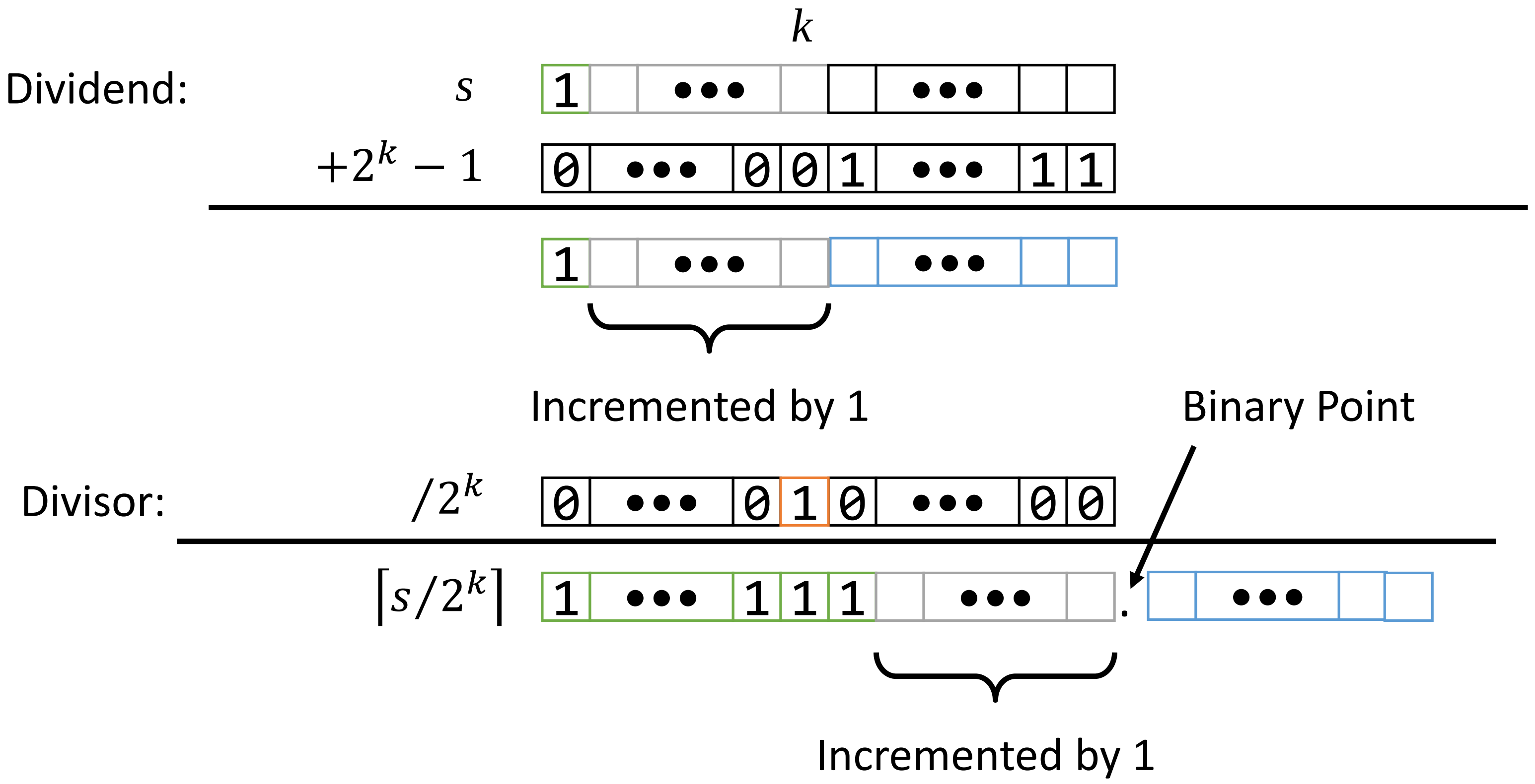
  Biasing has no effect



Correct power-of-2 divide (Cont.)

- Case 2: Rounding:

Biasing adds 1 to final result



Compiled signed division code

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

C function

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

Compiled arithmetic operations

```
  testl %eax, %eax
  js    L4
L3:
  sarl  $3, %eax
  ret
L4:
  addl  $7, %eax
  jmp   L3
```

Explanation

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

Summary

- Signed/unsigned multiply:

$$x * 2^k = x << k$$

- Unsigned divide:

$$u / 2^k = u >> k$$

Logical shift

- Signed divide:

$$s / 2^k = s >> k \qquad \text{for } s > 0$$

$$s / 2^k = s + (2^k - 1) >> k \qquad \text{for } s < 0$$

Arithmetic shift

3.6: C Integer puzzles

Computer Architecture and Systems Programming

Integer C Puzzles

- Assume 32-bit word size, two's complement integers
- Implementation defined by hardware
- For each of the following C expressions, either:
  - Argue that it is true for all argument values
  - Give an example where it is not true

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`