Integer arithmetic

Arquitectura de Computadores
Departamento de Engenharia Informática
Instituto Superior de Engenharia do Porto

Luís Nogueira (lmn@isep.ipp.pt)

Today: Integer arithmetic

- Encoding integers
- Addition
- Negation
- Multiplication
- Shifting

Integer arithmetic

- **Many beginning programmers are surprised to find that…**
  - adding two positive numbers can yield a negative result
  - the comparison $x < y$ can yield a different result than the comparison $x - y < 0$

- **These properties are artifacts of the finite nature of computer arithmetic**
  - As we will see, the integer arithmetic performed by computers is really a form of modular arithmetic

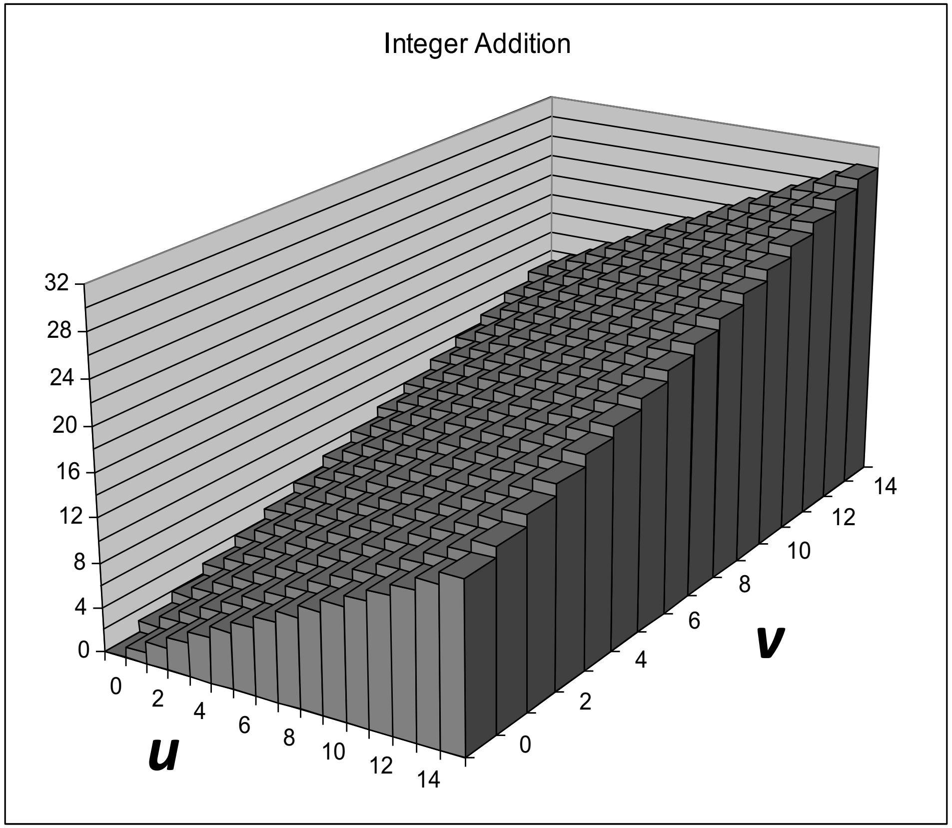
- **Understanding the nuances of computer arithmetic can help programmers write more reliable code**

Today: Integer arithmetic

- **Addition**
- Negation
- Multiplication
- Shifting

Visualizing (mathematical) integer addition

- **Example:**
  - 4-bit integers *u*, *v*
  - Compute true sum with $\mathrm{Add}_4(u\ ,\ v)$
- **Values increase linearly with *u* and *v***
  - Forms planar surface



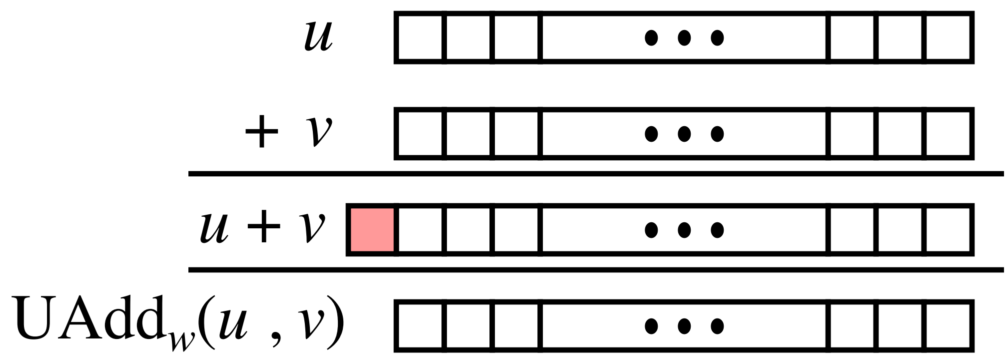
$$\mathbf{Add_4}(\boldsymbol{u}\ ,\ \boldsymbol{v}) = \mathbf{u + v}$$

Unsigned addition in C

Operands: *w* bits

True sum: *w*+1 bits

Discard carry: *w* bits



- **Unsigned addition ignores carry output**

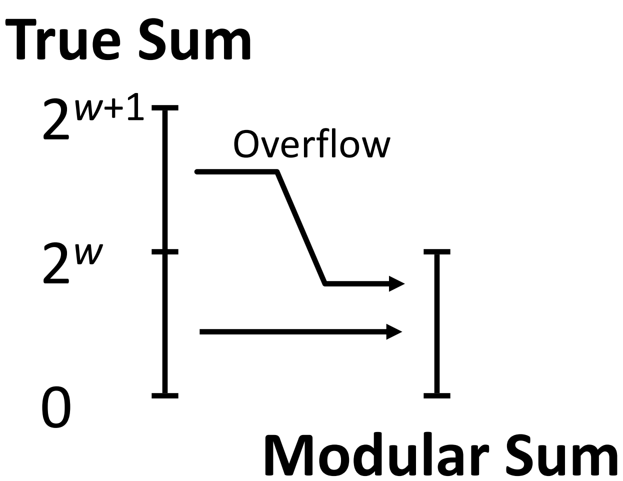
- **Implements modular arithmetic**

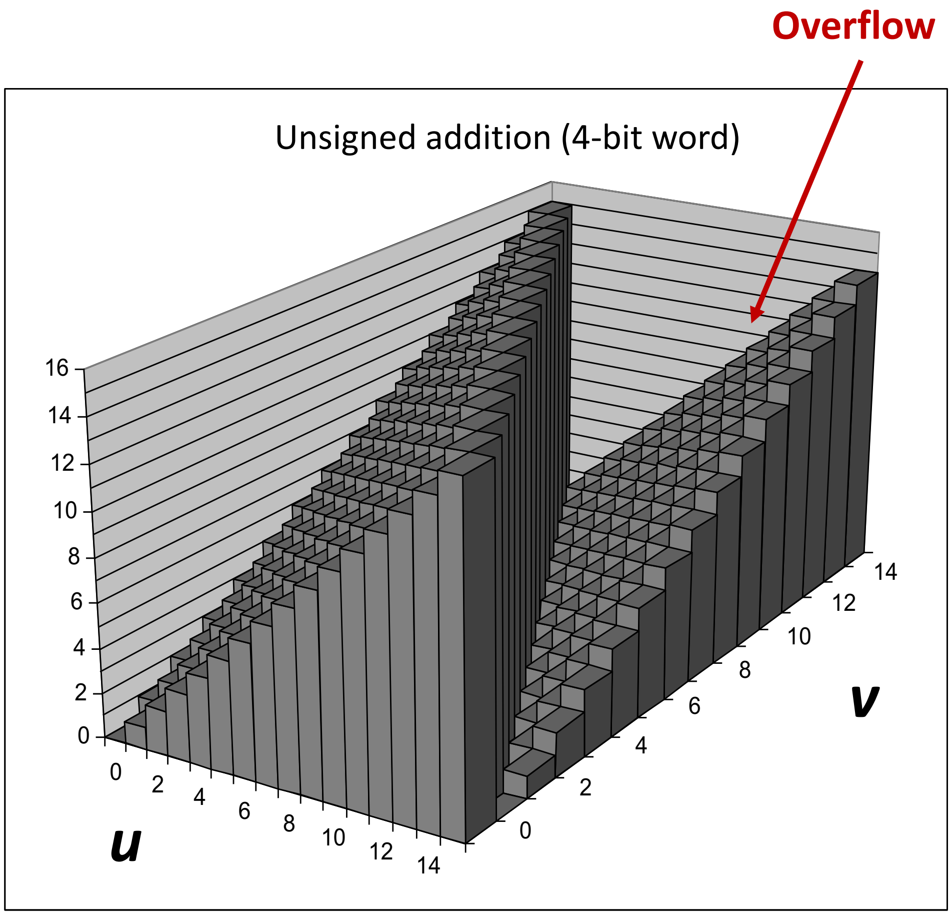
  $s \;=\; \mathrm{UAdd}_w(u\,,v) \;=\; (u+v) \bmod 2^w$

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

Visualizing unsigned addition

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





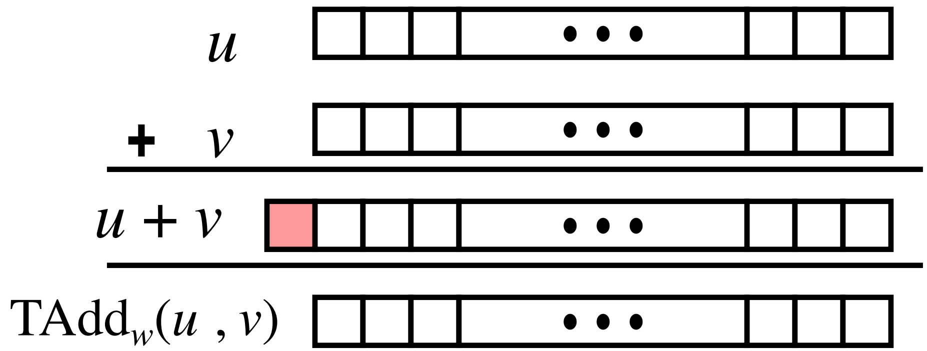
$\mathrm{UAdd}_4(u\ ,\ v) = (u + v) \bmod 16$

Two's complement addition in C

Operands: *w* bits

True sum: *w*+1 bits

Discard carry: *w* bits

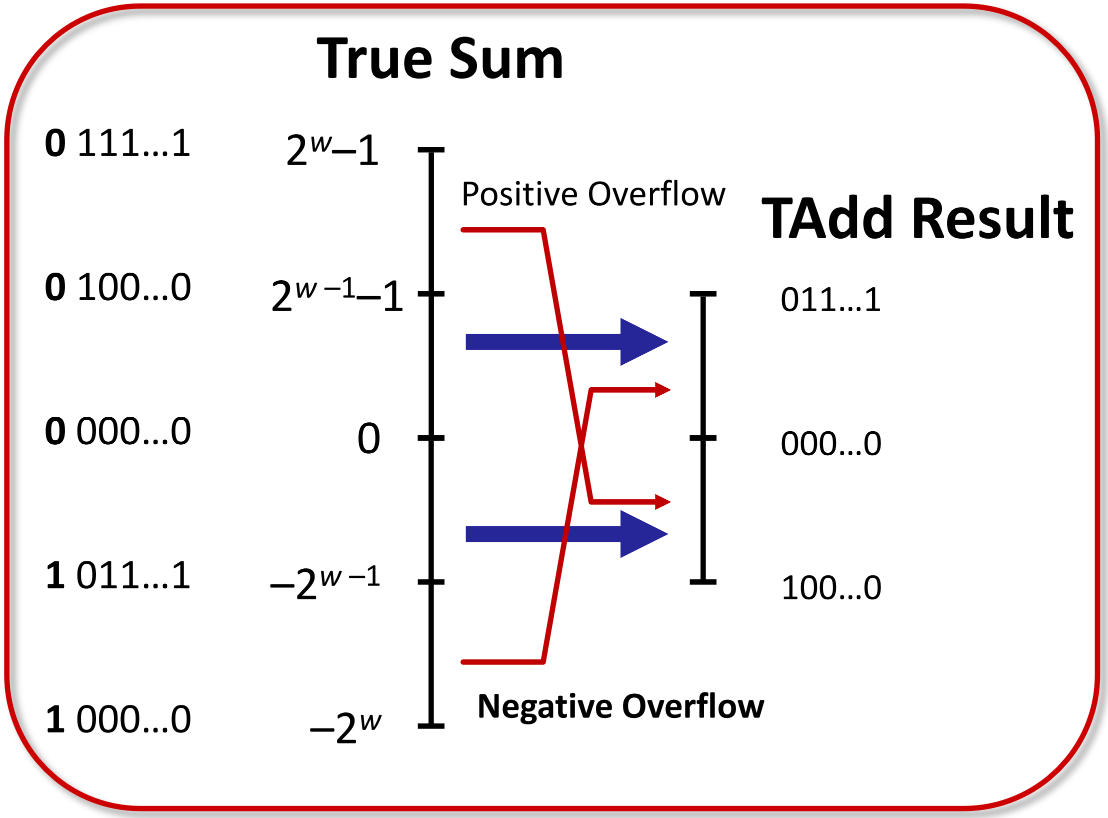


- **As before, we avoid ever-expanding data sizes by truncating the representation to *w* bits**

- **However, we must decide what to do when the result is either too large (positive) or too small (negative) to represent**

Two's complement addition overflow

- Drop off most significant bit
- Treat remaining bits as **two's complement integer**



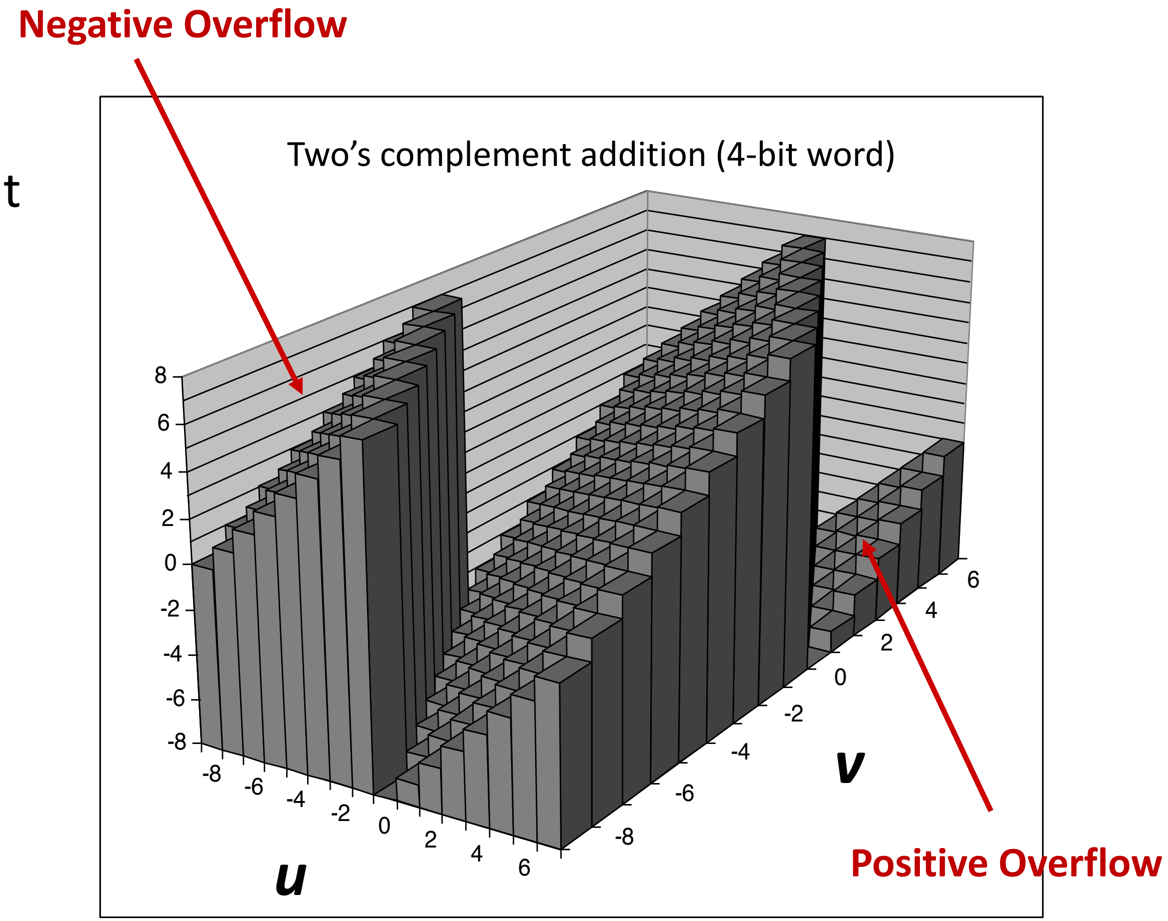
$$TAdd_w = \begin{cases} u+v+2^{w-1}, & u+v < TMin_w \\ u+v, & TMin_w \le u+v \le TMax_w \\ u+v-2^{w-1}, & TMax_w < u+v \end{cases}$$

(Negative Overflow)

(Positive Overflow)

Visualizing two's complement addition

- **Example:**
  - 4-bit two's complement
  - 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



$TAdd_4(u\ ,\ v) = U2T_4[(x + y) \bmod 16]$

Practice problem

- **Write a function with the following prototype:**

```
int add_ok(int x, int y);
```

- **This function should return 1 if arguments x and y can be added without causing overflow**

Practice problem

```
int add_ok(int x, int y) {
  int sum = x+y;
  int neg_over = x < 0 && y < 0 && sum >= 0;
  int pos_over = x >= 0 && y >= 0 && sum < 0;

  return !neg_over && !pos_over;
}
```

- **In C, overflows are not signaled as errors**

- **We can check if an overflow has occurred on `x + y` by seeing, if and only if,:**
  - sum < x (or equivalently, sum < y) for unsigned addition
  - The above conditions for signed addition

Summary: Signed vs Unsigned addition

- **TAdd and UAdd have identical bit-level behavior**
  - Most CPUs use the same machine instruction to perform either unsigned or signed addition

- **Signed vs. Unsigned addition in C**

```
int s, t, u, v;

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

Will give s == t

Today: Integer arithmetic

- Addition
- **Negation**
- Multiplication
- Shifting

Two's complement negation

- **Find the corresponding negative/positive number with the same absolute value**

- **Every number *x* in the range $-2^{w-1} \le x < 2^{w-1}$ has an additive inverse**
  - For *x* != 2^{w-1}, we can see that its additive inverse is simply $-x$

- **For $x = -2^{w-1} = TMin_w$, $-x = 2^w$ which cannot be represented as a *w*-bit number**

$$x = \begin{cases} -2^{w-1}, & x = -2^{w-1} \\ -x, & x > -2^{w-1} \end{cases}$$

Two's complement negation

- **Solution: complement and increment**

  `~x + 1 == -x`

- **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

Examples: Complement & Increment

x = 15213

	Decimal	Hex	Binary
x	15 213	3B 6D	00111011 01101101
~x	-15 214	C4 92	11000100 10010010
~x+1	-15 213	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

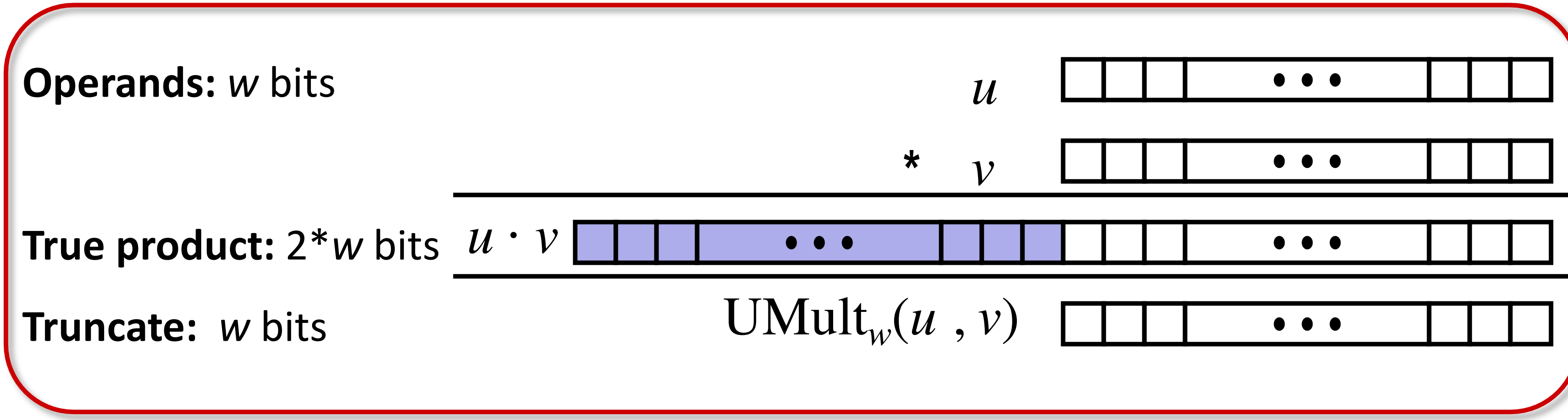
Today: Integer arithmetic

- Addition
- Negation
- **Multiplication**
- Shifting

Multiplication

- **Computing product of *w*-bit numbers *x* and *y***
  - Either signed or unsigned
- **Exact results can require as many as 2****w*** **bits to represent**
  - Most cases would fit into 2*w − 1 bits, but the special case of 2^{2w-2} requires the full 2*w bits (to include a sign bit of 0)
- **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)

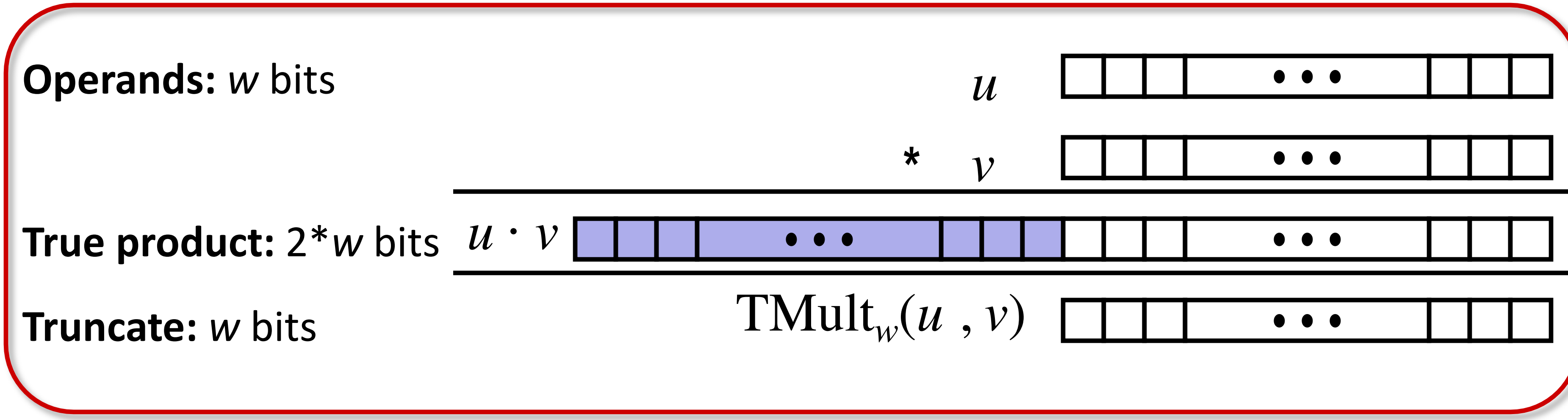
Unsigned multiplication in C



- **Standard multiplication algorithm**
  - Truncate result to w-bit number
- **Implements modular arithmetic**

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

Two’s complement multiplication in C



- **Compute exact product and ignores high order bits**
  - Some of which are different for signed vs. unsigned multiplication

- **Lower bits are the same of unsigned multiplication**
  - Treat them as a two’s complement integer

Signed vs Unsigned multiplication

- **TMul and UMul have identical bit-level behavior for the low-order *w* bits, even though the full 2w-bit differ**
  - Separate instructions are provided in IA32 for signed and unsigned multiplication

- **Multiplication in C is performed by truncating the 2w-bit product to w bits**

```
int x, y;
unsigned ux = (unsigned) x;
unsigned uy = (unsigned) y;

unsigned up = ux * uy;
int p = x * y;
```

up == (unsigned) p

Today: Integer arithmetic

- Addition
- Negation
- Multiplication
- **Shifting**

Multiplying by constants

- **Most machines shift and add faster than multiply**
  - The integer multiply instruction is fairly slow, requiring 10 or more clock cycles
  - Other integer operations—such as addition, subtraction, bit-level operations, and shifting—require only 1 clock cycle

- **Compiler optimization**
  - Replace multiplications by constant factors with combinations of shift and addition/subtraction operations

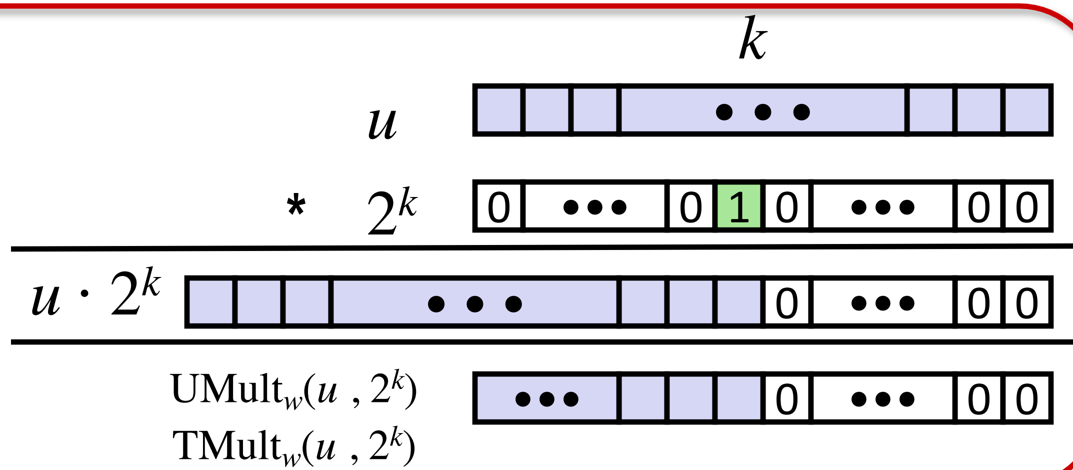
- **Examples**
  - `u << 3  ==  u * 8`
  - `(u << 5) - (u << 3) ==  u * 24`

Power-of-2 multiply with shift

Operands: *w* bits

True product: *w*+*k* bits

Truncate: *w* bits



- **Operation**
  - `u << k` gives `u *` ***2^k***
- **Yield the same result**
  - Both signed and unsigned
  - Even in overflow

Compiled multiplication code

C function

```
long int mul_12(long int x){
   return x*12;
}
```

Compiled arithmetic operations

```
shll $3, %eax
shll $2, %ebx
addl %ebx, %eax
```

Explanation

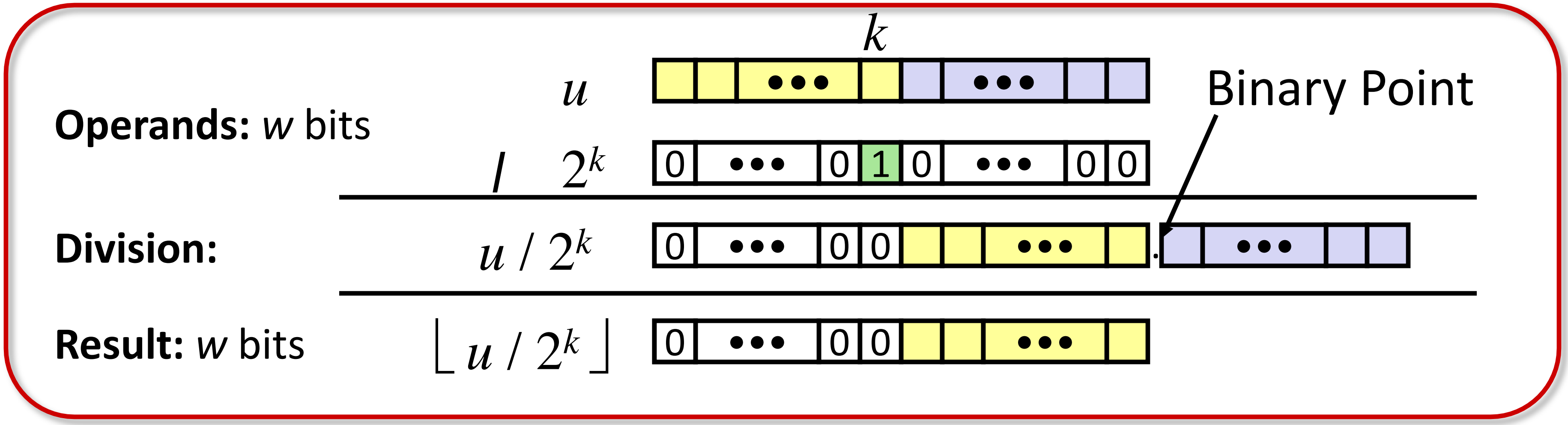
```
t1 <- x * 8;
t2 <- x * 4;
return t1 + t2;
```

- **C compiler automatically generates shift/add code when multiplying by constant**
  - Always uses logical shift

Power-of-2 divide with shift

- **Integer division on most machines is even slower than integer multiplication**
  - Requiring 30 or more clock cycles
- **Dividing by a power of 2 can also be performed using shift operations**
  - Uses a right shift rather than a left shift
- **Unsigned vs Two's complement**
  - Unsigned: logical shift
  - Two's complement: arithmetic shift

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**

Unsigned power-of-2 divide with shift

	Division	Computed	Hex	Binary
x	15 213	15 213	3B 6D	00111011 01101101
x >> 1	7 606.5	7 606	1D B6	00011101 10110110
x >> 4	950.8125	950	03 B6	00000011 10110110
x >> 8	59.4257813	59	00 3B	00000000 00111011

- **The result of shifting consistently rounds toward zero**
  - As is the convention for integer division

Compiled unsigned division code

C function

```
unsigned long int udiv_8(unsigned long int x){
  return x/8;
}
```

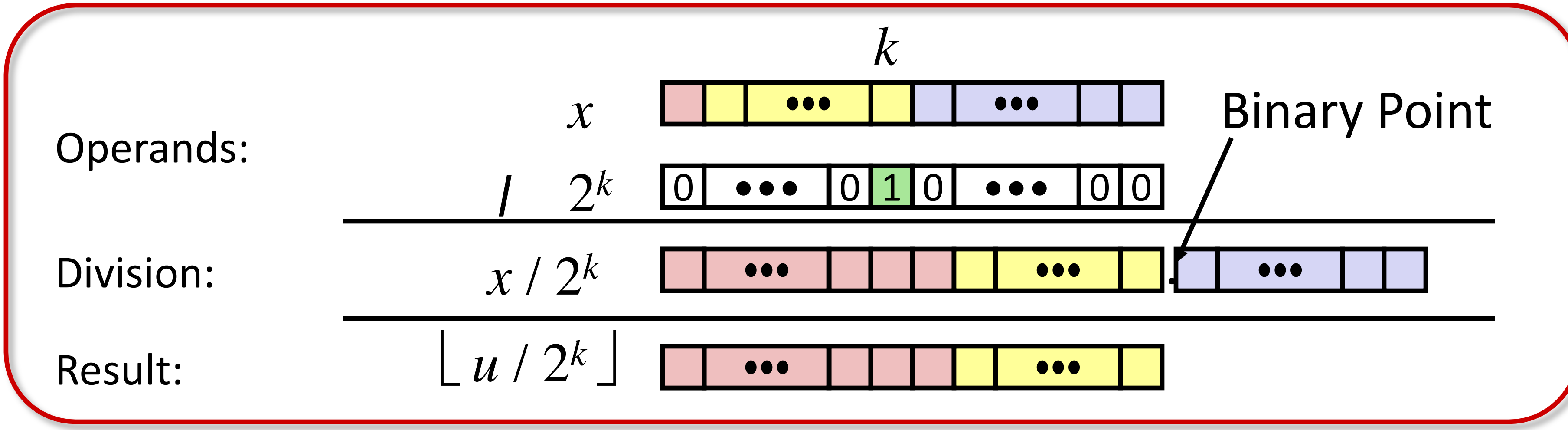
Compiled arithmetic operations

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

Explanation

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

Signed power-of-2 divide with shift



- **Quotient of signed by power-of-2**
  - `x >> k` gives $\lfloor$ **x / 2^k** $\rfloor$

- **Uses** **arithmetic** **shift**

Signed power-of-2 divide with shift

	Division	Computed	Hex	Binary
`y`	-15 213	-15 213	`C4 93`	`11000100 10010011`
`y >> 1`	-7 606.5	-7 607	`E2 49`	`11100010 01001001`
`y >> 4`	-950.8125	-951	`FC 49`	`11111100 01001001`
`y >> 8`	-59.4257813	-60	`FF C4`	`11111111 11000100`

- **For a negative number, arithmetic right shift rounds down rather than toward zero**
  - For a positive number, we have 0 as the most significant bit, and so the effect is the same as for a logical right shift

Correct power-of-2 divide with shift

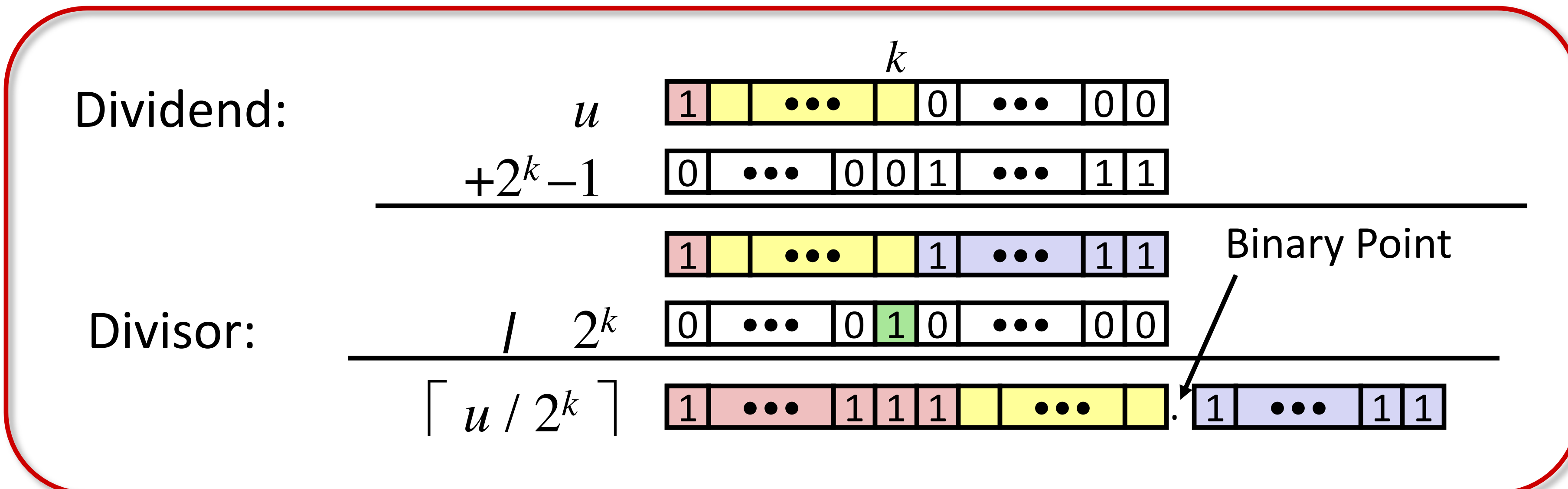
- **We can correct for this improper rounding by "biasing" the value before shifting**

- **Quotient of negative number by power-of-2**
  - Want $\lceil$ **x / 2**^{k} $\rceil$ (Round toward 0 – convention for integer division)
  - Compute as $\lfloor$ **(x+2**^{k}**–1)/ 2**^{k} $\rfloor$
    - In C: `(x + (1<<k)-1) >> k`

- **This technique exploits the property that $\lceil$`x/y`$\rceil$ = $\lfloor$`(x + y - 1)/y`$\rfloor$ for integers `x` and `y` such that `y > 0`**

Correct power-of-2 divide with shift

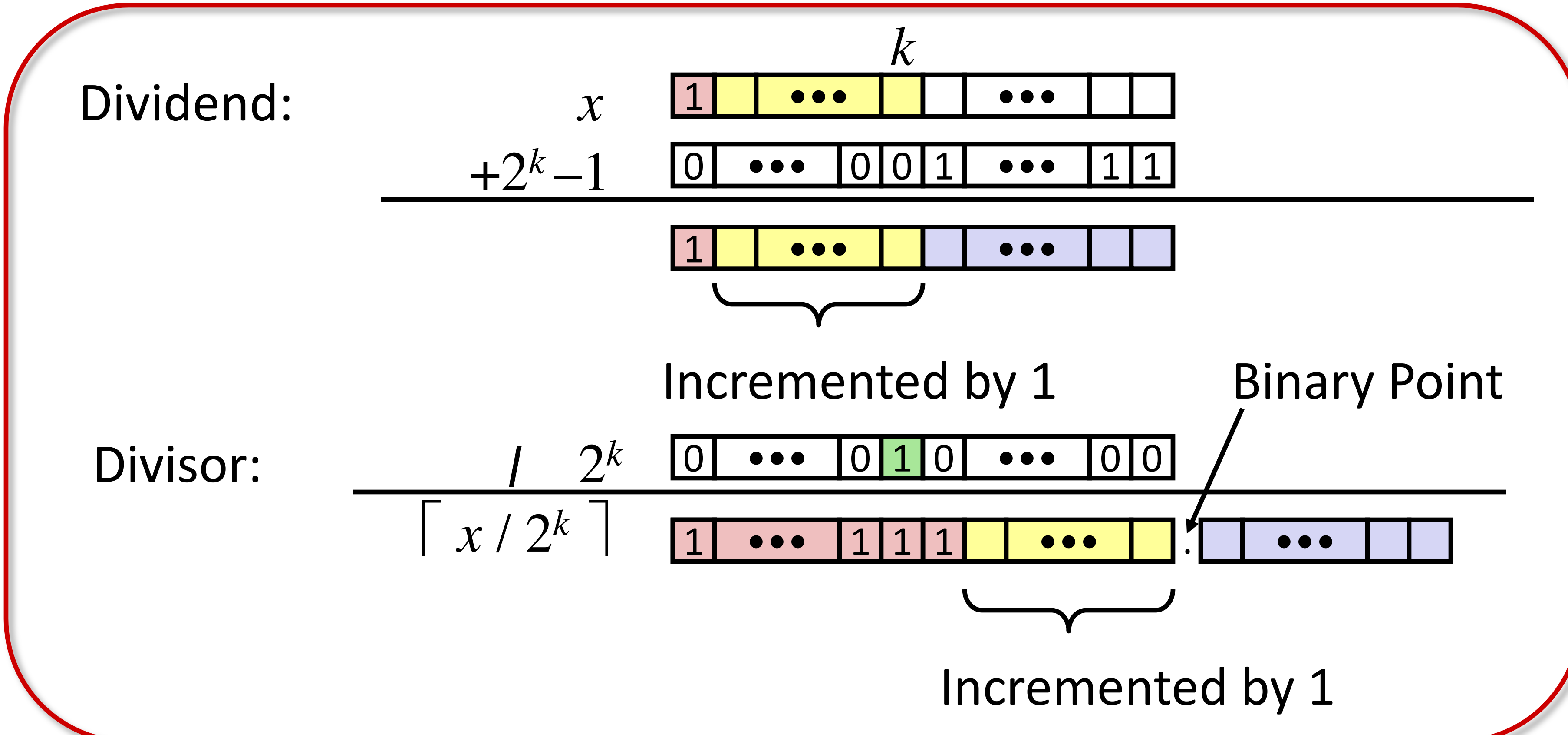
Case 1: No rounding is required (`x >= 0`)



Biasing has no effect

Correct power-of-2 divide with shift

Case 2: Rounding is required ($x < 0$)



Biasing adds 1 to final result

Compiled signed division code

C function

```
long int idiv8(long 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;
```

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
- **CPUs can use the same machine instruction to perform either unsigned or signed addition**

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)
- **On most machines, integer multiplication and division are fairly slow**
  - Replace them by constant factors with combinations of shift and addition/subtraction operations

Arithmetic: Basic rules

- **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