CS 33

Data Representation, Part 2

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
- **Other Values**
  - Minus 1
    111…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

	W			
	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

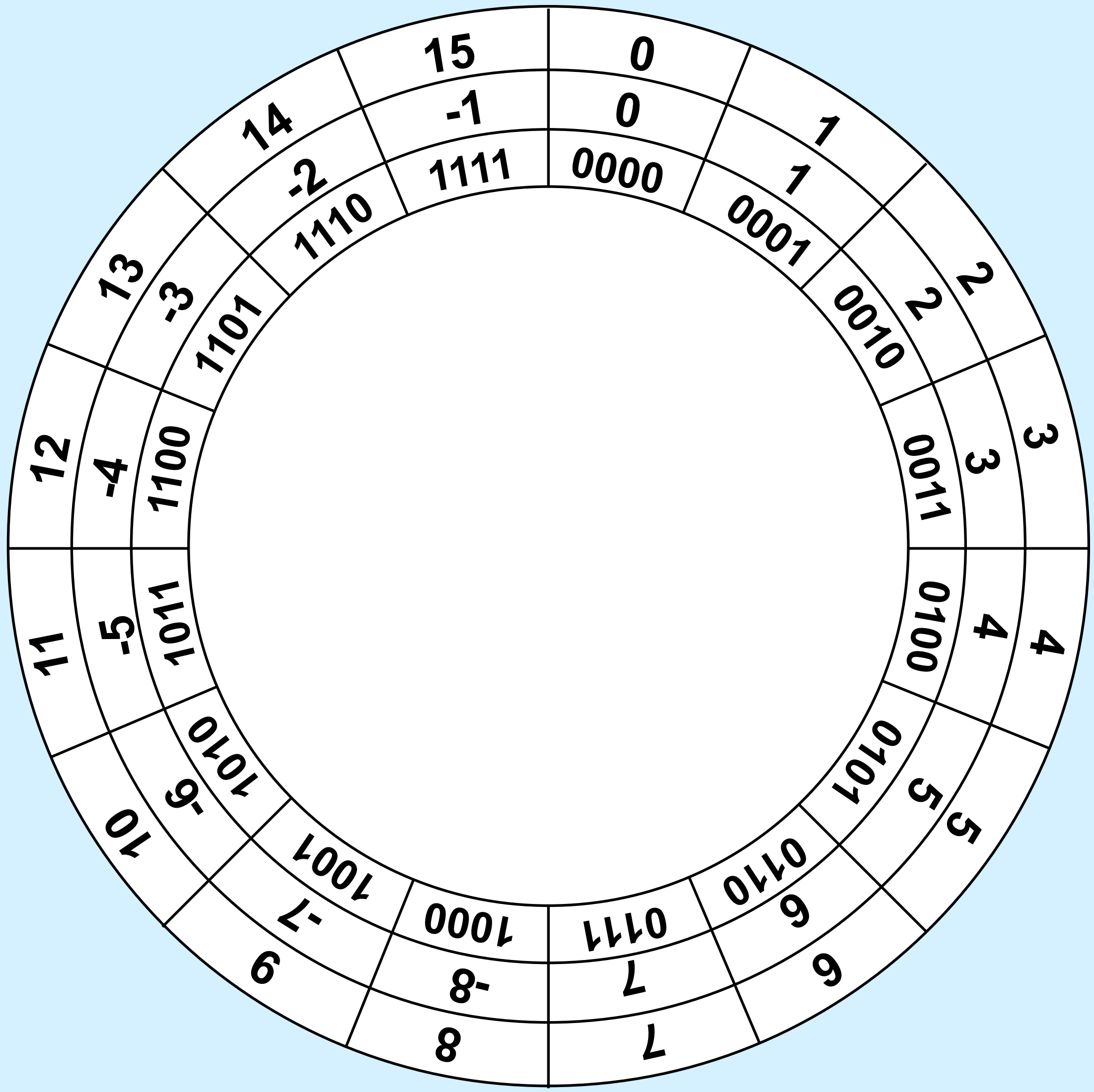
  $UMax = 2 * TMax + 1$

- **C Programming**
  - **#include** <limits.h>
  - declares constants, e.g.,
    - ULONG_MAX
    - LONG_MAX
    - LONG_MIN
  - values platform-specific

Quiz 1

- **What is –TMin (assuming two's complement signed integers)?**
  a) TMin
  b) TMax
  c) 0
  d) 1

4-Bit Computer Arithmetic



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

    ```
    int tx, ty;
    unsigned ux, uy; // “unsigned” means “unsigned int”
    tx = (int) ux;
    uy = (unsigned int) ty;
    ```

  - implicit casting also occurs via assignments and procedure calls

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

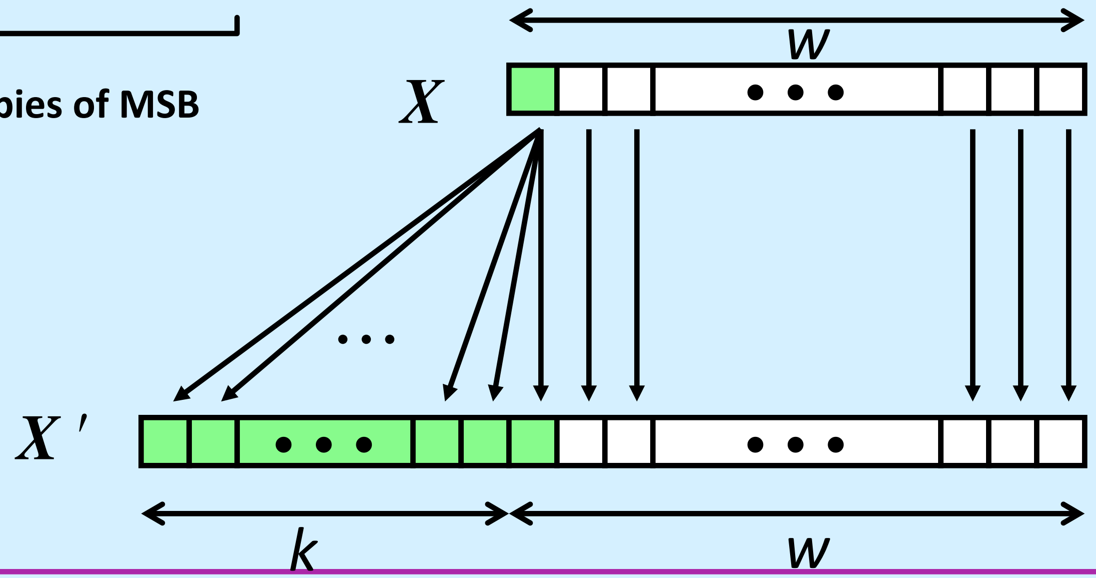
Casting Surprises

- **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	-2147483647-1	>	signed
2147483647U	-2147483647-1	<	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 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

Does it Work?

$$val_w = -2^{w-1} + \sum_{i=0}^{w-2} b_i \cdot 2^i$$

$$val_{w+1} = -2^w + 2^{w-1} + \sum_{i=0}^{w-2} b_i \cdot 2^i$$

$$= -2^{w-1} + \sum_{i=0}^{w-2} b_i \cdot 2^i$$

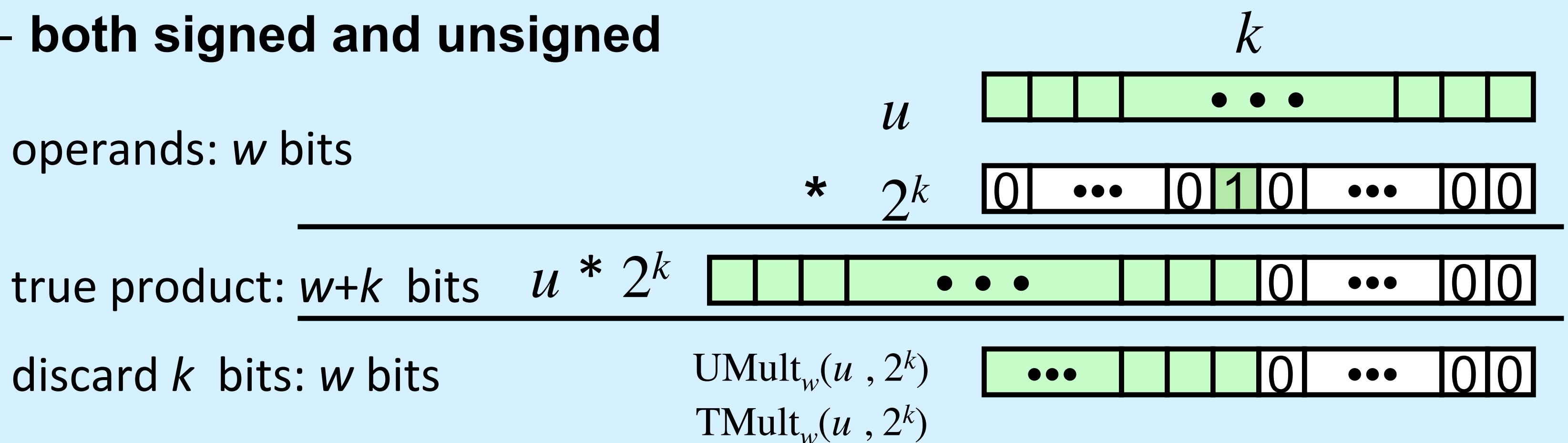
$$val_{w+2} = -2^{w+1} + 2^w + 2^{w-1} + \sum_{i=0}^{w-2} b_i \cdot 2^i$$

$$= -2^w + 2^{w-1} + \sum_{i=0}^{w-2} b_i \cdot 2^i$$

$$= -2^{w-1} + \sum_{i=0}^{w-2} b_i \cdot 2^i$$

Power-of-2 Multiply with Shift

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



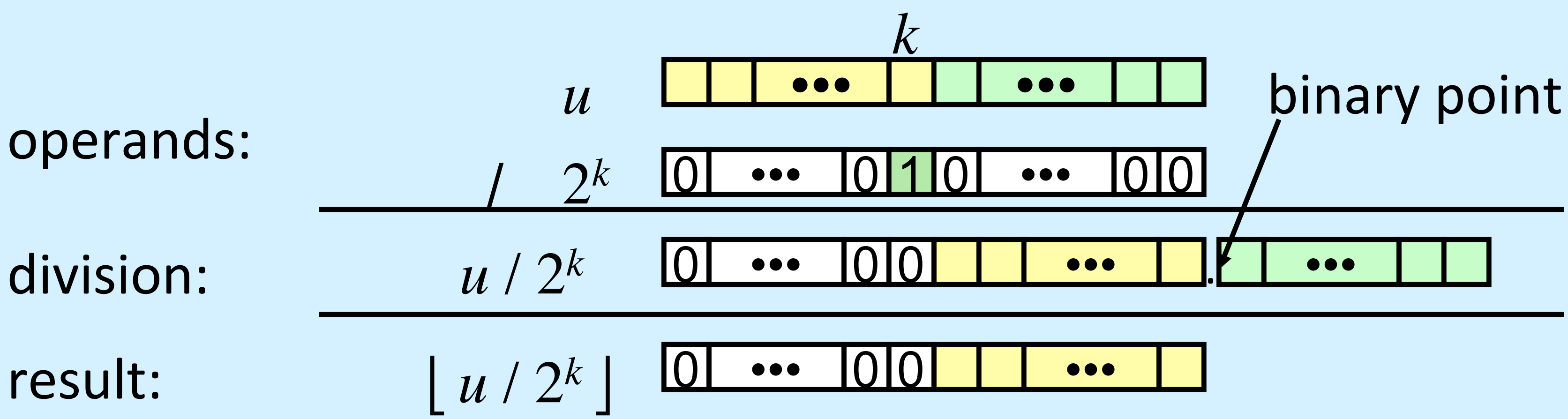
- **Examples**

```
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

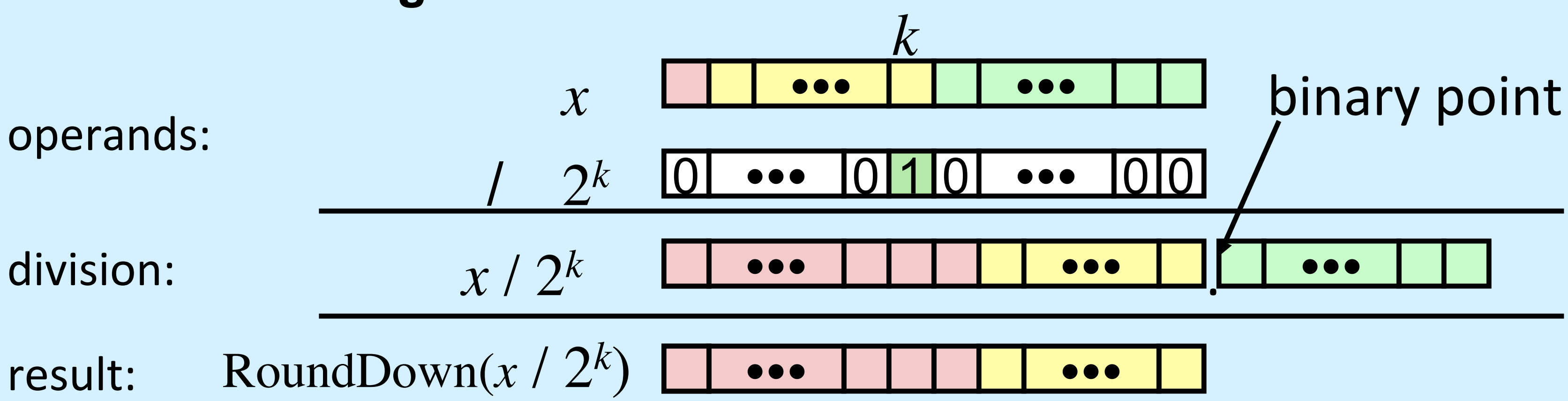
- **Quotient of unsigned by power of 2**
  - `u >> k` **gives** $\lfloor$ `u /` 2^k $\rfloor$
  - **uses logical shift**



	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`

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
  - rounds wrong direction when `x < 0`

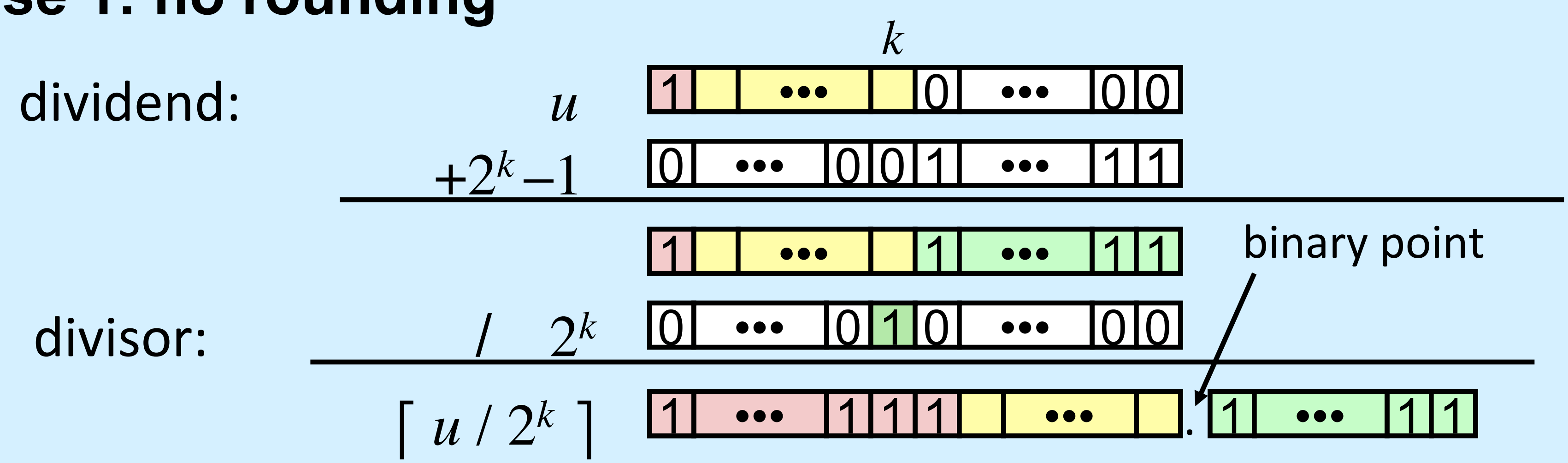


	Division	Computed	Hex	Binary
`y`	-15213	-15213	`C4 93`	`11000100 10010011`
`y >> 1`	-7606.5	-7607	`E2 49`	`11100010 01001001`
`y >> 4`	-950.8125	-951	`FC 49`	`11111100 01001001`
`y >> 8`	-59.4257813	-60	`FF C4`	`11111111 11000100`

Correct Power-of-2 Divide

- **Quotient of negative number by power of 2**
  - want $\lceil$ `x / 2`$^k\rceil$ (round toward 0)
  - compute as $\lfloor$ `(x+2`^{k}`-1) / 2`$^k\rfloor$
    - » in C: `(x + (1<<k)-1) >> k`
    - » biases dividend toward 0

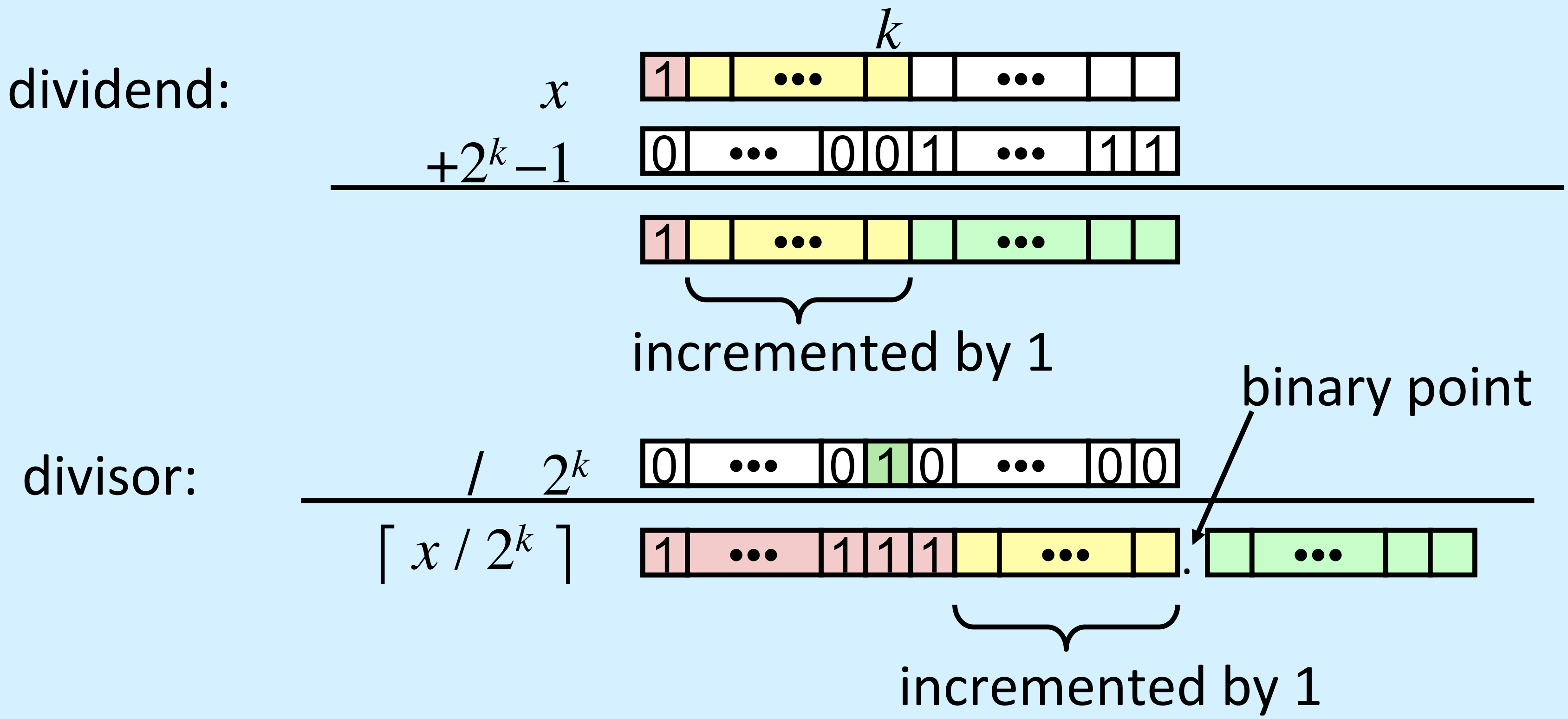
Case 1: no rounding



Biasing has no effect

Correct Power-of-2 Divide (Cont.)

Case 2: rounding



Biasing adds 1 to final result

Why Should I Use Unsigned?

- *Don't* use just because number nonnegative
  - easy to make mistakes

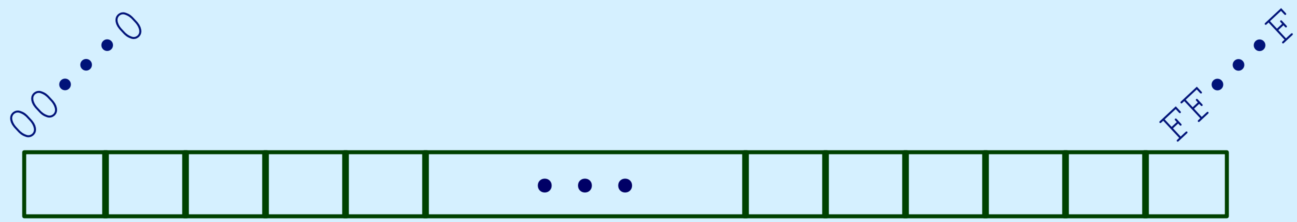
```
unsigned i;
for (i = cnt-2; i >= 0; i--)
  a[i] += a[i+1];
```

  - can be very subtle

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

- *Do* use when performing modular arithmetic
  - multiprecision arithmetic
- *Do* use when using bits to represent sets
  - logical right shift, no sign extension

Byte-Oriented Memory Organization



- Programs refer to data by address
  - conceptually, envision it as a very large array of bytes
    - » in reality, it's not, but can think of it that way
  - an address is like an index into that array
    - » and, a pointer variable stores an address

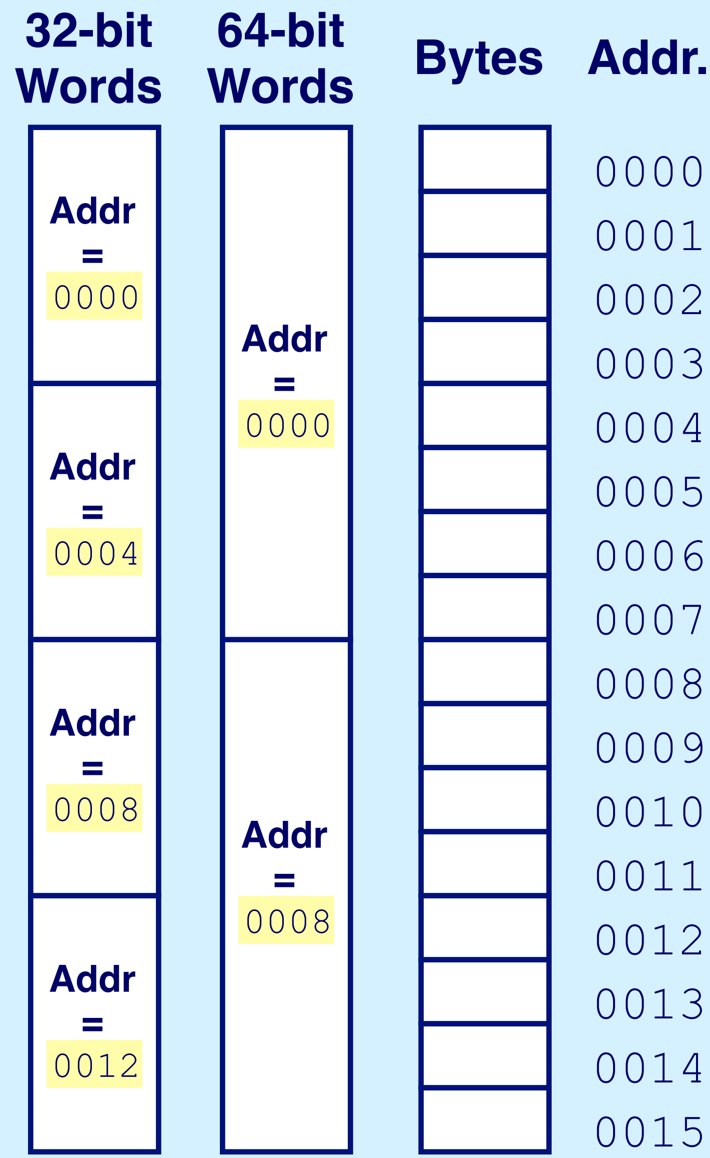
- Note: system provides private address spaces to each "process"
  - think of a process as a program being executed
  - so, a program can clobber its own data, but not that of others

Machine Words

- Any given computer has a "word size"
  - nominal size of integer-valued data
    - » and of addresses
  - until recently, most machines used 32 bits (4 bytes) as word size
    - » limits addresses to 4GB (2^{32} bytes)
    - » becomes too small for memory-intensive applications
      - leading to emergence of computers with 64-bit word size
  - machines still support multiple data formats
    - » fractions or multiples of word size
    - » always integral number of bytes

Word-Oriented Memory Organization

- **Addresses specify byte locations**
  - address of first byte in word
  - addresses of successive words differ by 4 (32-bit) or 8 (64-bit)



Byte Ordering

- Four-byte integer
  - 0x76543210
- Stored at location 0x100
  - which byte is at 0x100?
  - which byte is at 0x103?

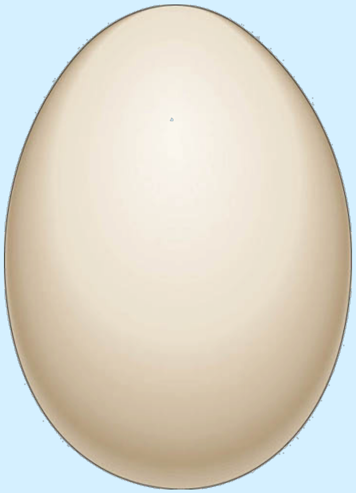
?

10	32	54	76
0x100	0x101	0x102	0x103

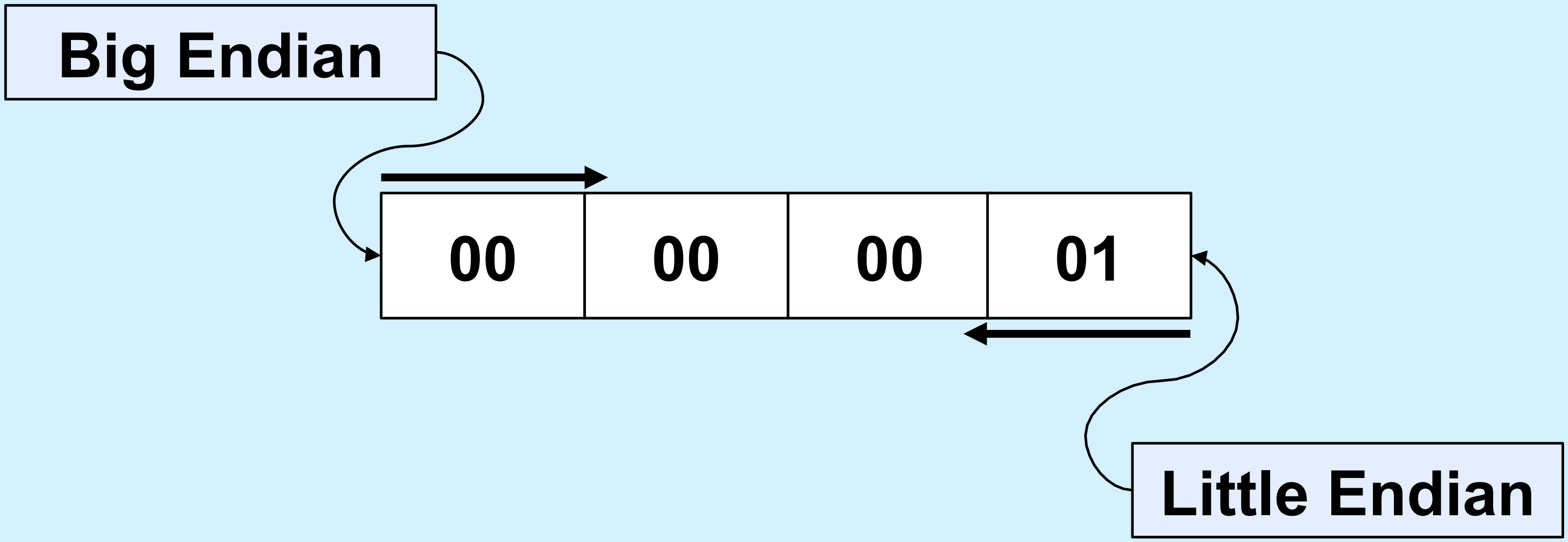
Little-endian

76	54	32	10
0x100	0x101	0x102	0x103

Big-endian



Byte Ordering (2)



Quiz 2

```
int main() {
  long x=1;
  proc(x);
  return 0;
}

void proc(int arg) {
  printf("%d\n", arg);
}
```

What value is printed on a big-endian 64-bit computer?

a) 0
b) 1
c) 2^{32}
d) 2^{32}-1