CS 33

Data Representation

Number Representation

- Hindu-Arabic numerals
  - developed by Hindus starting in 5th century
    - » positional notation
    - » symbol for 0
  - adopted and modified somewhat later by Arabs
    - » known by them as “Rakam Al-Hind” (Hindu numeral system)
  - 1999 rather than MCMXCIX
    - » (try doing long division with Roman numerals!)

Which Base?

- 1999
  - base 10
    - » $9 \cdot 10^0 + 9 \cdot 10^1 + 9 \cdot 10^2 + 1 \cdot 10^3$
  - base 2
    - » 11111001111
      - $1 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3 + 0 \cdot 2^4 + 0 + 2^5 + 1 \cdot 2^6 + 1 \cdot 2^7 + 1 \cdot 2^8 + 1 \cdot 2^9 + 1 \cdot 2^{10}$
  - base 8
    - » 3717
      - $7 \cdot 8^0 + 1 \cdot 8^1 + 7 \cdot 8^2 + 3 \cdot 8^3$
    - » why are we interested?
  - base 16
    - » 7CF
      - $15 \cdot 16^0 + 12 \cdot 16^1 + 7 \cdot 16^2$
    - » why are we interested?

Words …

$$\underbrace{0\,1\,1}_{3}\;\underbrace{1\,1\,1}_{7}\;\underbrace{0\,0\,1}_{1}\;\underbrace{1\,1\,1}_{7}$$

12-bit computer word

$$\underbrace{0\,0\,0\,0}_{0}\;\underbrace{0\,1\,1\,1}_{7}\;\underbrace{1\,1\,0\,0}_{C}\;\underbrace{1\,1\,1\,1}_{F}$$

16-bit computer word

Algorithm …

```
void baseX(unsigned int num, unsigned int base) {
   char digits[] = {'0', '1', '2', '3', '4', '5', '6', … };
   char buf[8*sizeof(unsigned int)+1];
   int i;

   for (i = sizeof(buf) - 2; i >= 0; i--) {
      buf[i] = digits[num%base];
      num /= base;
      if (num == 0)
         break;
   }

   buf[sizeof(buf) - 1] = '\0';
   printf("%s\n", &buf[i]);
}
```

Or …

```
$ bc
obase=16
1999
7CF
$
```


Quiz 1

- What's the decimal (base 10) equivalent of 23_{16}?
  a) 19
  b) 33
  c) 35
  d) 37

Encoding Byte Values

- **Byte = 8 bits**
  - **binary 00000000_2 to 11111111_2**
  - **decimal: 0_{10} to 255_{10}**
  - **hexadecimal 00_{16} to FF_{16}**
    - **base 16 number representation**
    - **use characters ‘0’ to ‘9’ and ‘A’ to ‘F’**
    - **write $FA1D37B_{16}$ in C as**
      - **0xFA1D37B**
      - **0xfa1d37b**

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

Boolean Algebra

- **Developed by George Boole in 19th Century**
  - **algebraic representation of logic**
    - » **encode “true” as 1 and “false” as 0**

And

- A&B = 1 when both A=1 and B=1

&	0	1
0	0	0
1	0	1

Or

- A|B = 1 when either A=1 or B=1

|	0	1
0	0	1
1	1	1

Not

- ~A = 1 when A=0

~	
0	1
1	0

Exclusive-Or (Xor)

- A^B = 1 when either A=1 or B=1, but not both

^	0	1
0	0	1
1	1	0

General Boolean Algebras

- **Operate on bit vectors**
  - **operations applied bitwise**

```
  01101001     01101001     01101001
& 01010101   | 01010101   ^ 01010101   ~ 01010101
  01000001     01111101     00111100     10101010
```

- **All of the properties of boolean algebra apply**

Example: Representing & Manipulating Sets

- Representation
  - width-w bit vector represents subsets of {0, …, w–1}
  - $a_j = 1$ iff $j \in A$

01101001	{ 0, 3, 5, 6 }
76543210	
01010101	{ 0, 2, 4, 6 }
76543210	

- Operations

&	intersection	01000001	{ 0, 6 }
|	union	01111101	{ 0, 2, 3, 4, 5, 6 }
^	symmetric difference	00111100	{ 2, 3, 4, 5 }
~	complement	10101010	{ 1, 3, 5, 7 }

Bit-Level Operations in C

- Operations &, `|`, `~`, `^` available in C
  - apply to any "integral" data type
    - » `long`, `int`, `short`, `char`
  - view arguments as bit vectors
  - arguments applied bit-wise
- Examples (char datatype)

`~0x41` → `0xBE`

  $\sim 01000001_2 \rightarrow 10111110_2$

`~0x00` → `0xFF`

  $\sim 00000000_2 \rightarrow 11111111_2$

`0x69 & 0x55` → `0x41`

  $01101001_2 \; \& \; 01010101_2 \rightarrow 01000001_2$

`0x69 | 0x55` → `0x7D`

  $01101001_2 \mid 01010101_2 \rightarrow 01111101_2$

Contrast: Logic Operations in C

- **Contrast to Logical Operators**
  - `&&, ||, !`
    - » view 0 as "false"
    - » anything nonzero as "true"
    - » always return 0 or 1
    - » early termination/short-circuited execution
- **Examples (char datatype)**

```
!0x41 → 0x00
!0x00 → 0x01
!!0x41 → 0x01

0x69 && 0x55 → 0x01
0x69 || 0x55 → 0x01
p && *p     (avoids null pointer access)
```

Contrast: Logic Operations in C

- **Contrast to Logical Operators**
  - `&&, ||, !`
    - » **vie**... **“false”**

Watch out for && vs. & (and || vs. |)… One of the more common oopsies in C programming

- **E**...

```
!0x41 → 0x00
!0x00 → 0x01
!!0x41 → 0x01

0x69 && 0x55 → 0x01
0x69 || 0x55 → 0x01
p && *p
```
(avoids null pointer access)

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

Signed Integers

- **Sign-magnitude**

$$\underbrace{\boxed{b_{w-1}}}_{\text{sign}}\underbrace{\boxed{b_{w-2}}\boxed{b_{w-3}}\boxed{\quad\ldots\quad}\boxed{b_2}\boxed{b_1}\boxed{b_0}}_{\text{magnitude}}$$

value = $$(-1)^{b_{w-1}} \cdot \sum_{i=0}^{w-2} b_i \cdot 2^i$$

- two representations of zero!

Signed Integers

- **Ones' complement**
  - negate a number by forming its bitwise complement
    - » e.g., (-1)·01101011 = 10010100

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

$$= \sum_{i=0}^{w-2} b_i \cdot 2^i \quad \text{if } b_{w\text{-}1}=0$$

$$= \sum_{i=0}^{w-2} (b_i\text{-}1) \cdot 2^i \quad \text{if } b_{w\text{-}1}=1$$

two zeroes!

Signed Integers

- **Two's complement**

  b_{w-1} = 0 ⇒ non-negative number

$$\text{value} = \sum_{i=0}^{w-2} b_i \cdot 2^i$$

  b_{w-1} = 1 ⇒ negative number

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

Signed Integers

- **Negating two's complement**

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

  - **how to compute –*value*?**
    (~value)+1

Signed Integers

- **Negating two's complement (continued)**

$$\mathbf{\mathit{value} + (\sim\mathit{value} + 1)}$$

$$\mathbf{= \; (\mathit{value} + \sim\mathit{value}) + 1}$$

$$\mathbf{= \; (2^{w}-1) + 1}$$

$$\mathbf{= \; 2^{w}}$$

$$= \; \boxed{1}\,\overbrace{\boxed{0}\,\boxed{0}\,\boxed{0}\,\boxed{\quad\ldots\quad}\,\boxed{0}\,\boxed{0}\,\boxed{0}}^{w}$$

Quiz 2

- We have a computer with 4-bit words that uses two's complement to represent negative numbers. What is the result of subtracting 0010 (2) from 0001 (1)?
  a) 0111
  b) 1001
  c) 1110
  d) 1111

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**

  $|TMin| \quad = \quad TMax + 1$

  » Asymmetric range

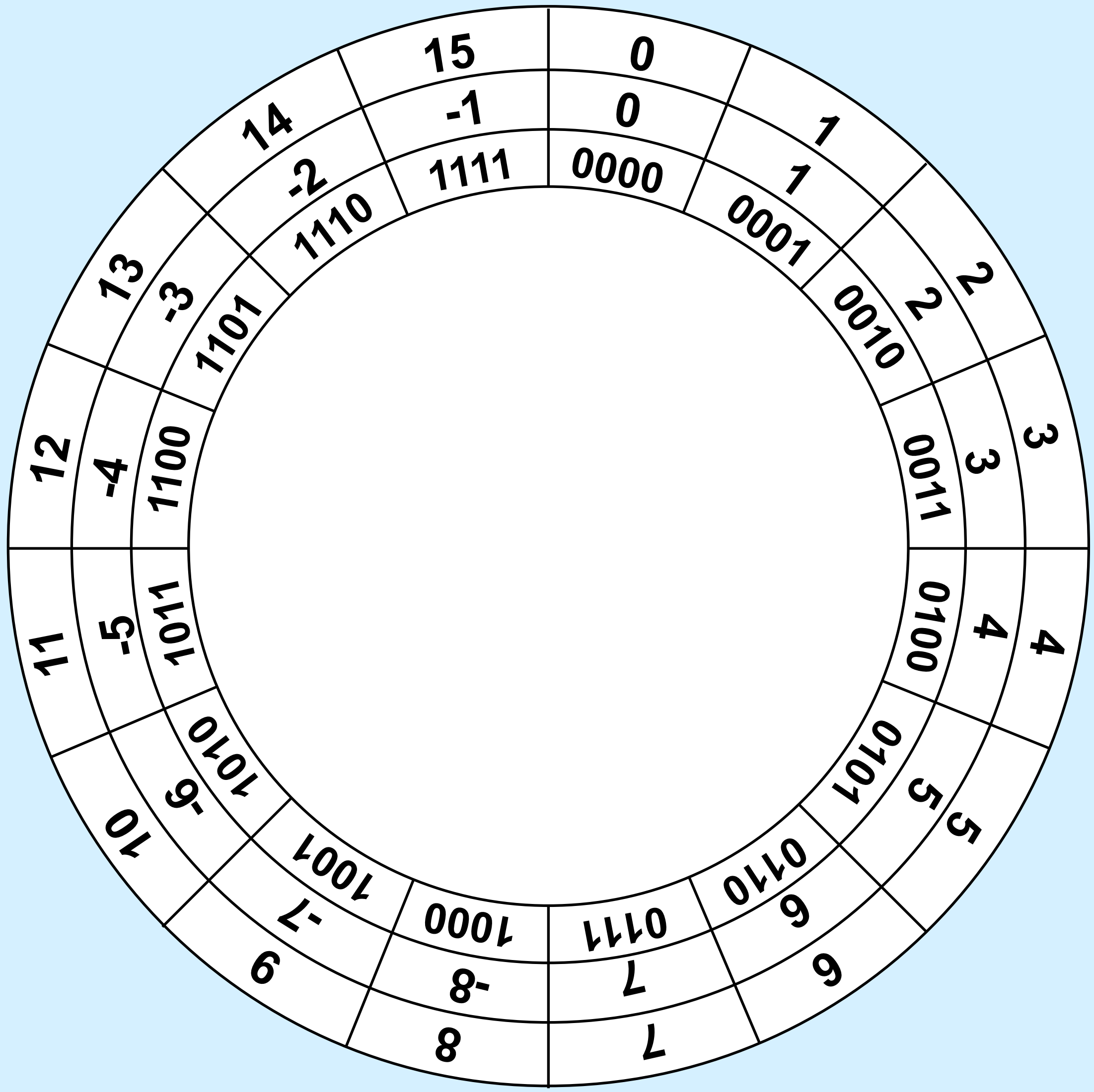
  $UMax \quad = \quad 2 * TMax + 1$

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

Quiz 3

- **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 int 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;
    ```

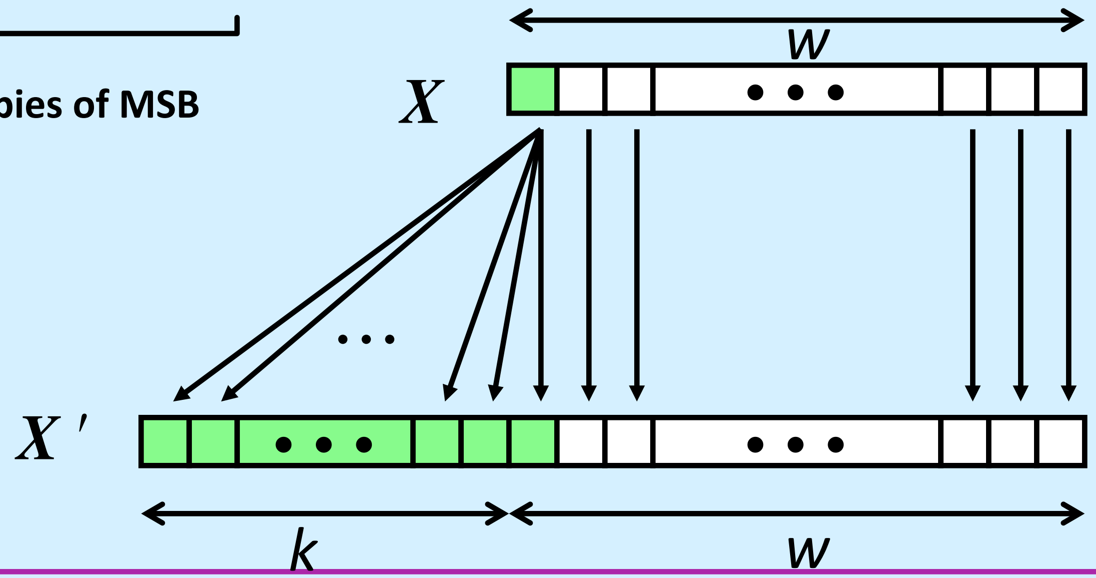
Casting Surprises

- **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?

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

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

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

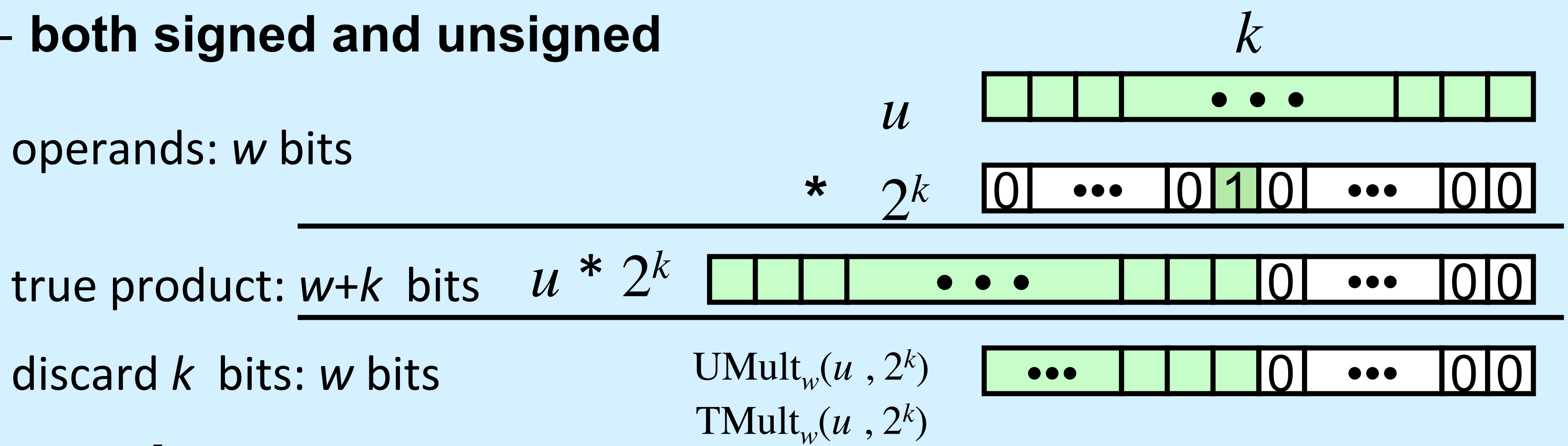
$$val_{w+2} = -2^{w+1} + 2^w + 2^{w-1} + \sum_{i=0}^{w-2} b_i \cdot 2^i$$

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

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

Power-of-2 Multiply with Shift

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



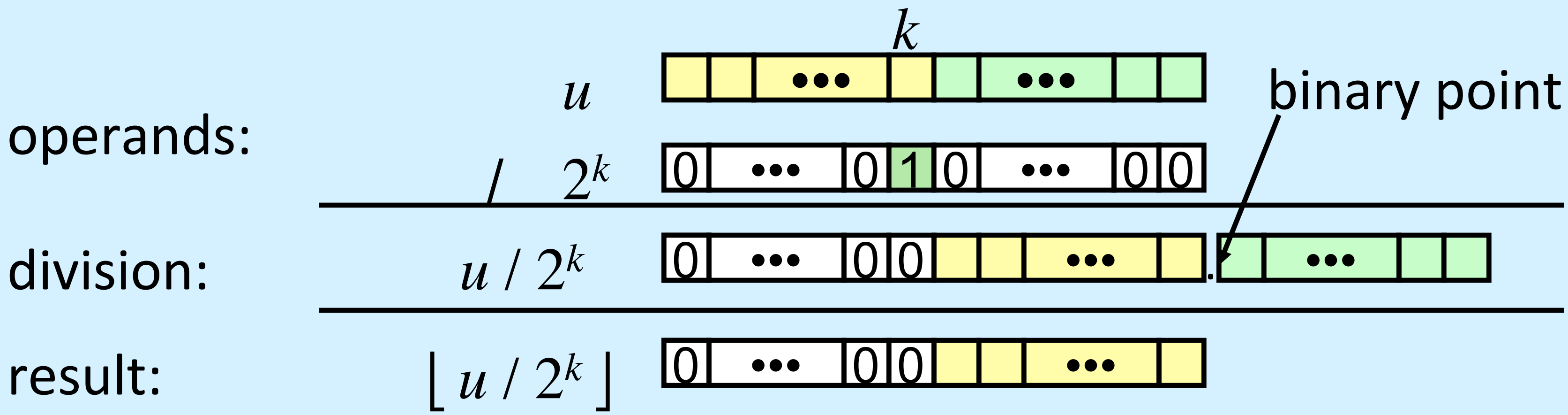
- **Examples**

```
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

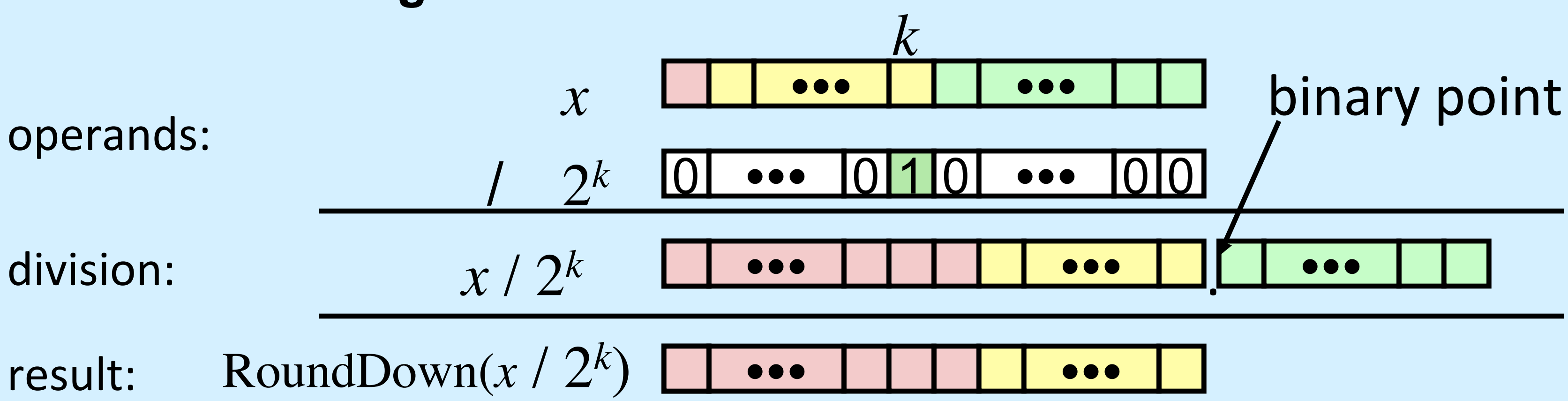
- **Quotient of unsigned by power of 2**
  - `u >> k` gives $\lfloor$ `u / ` $2^k \rfloor$
  - uses logical shift



	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`

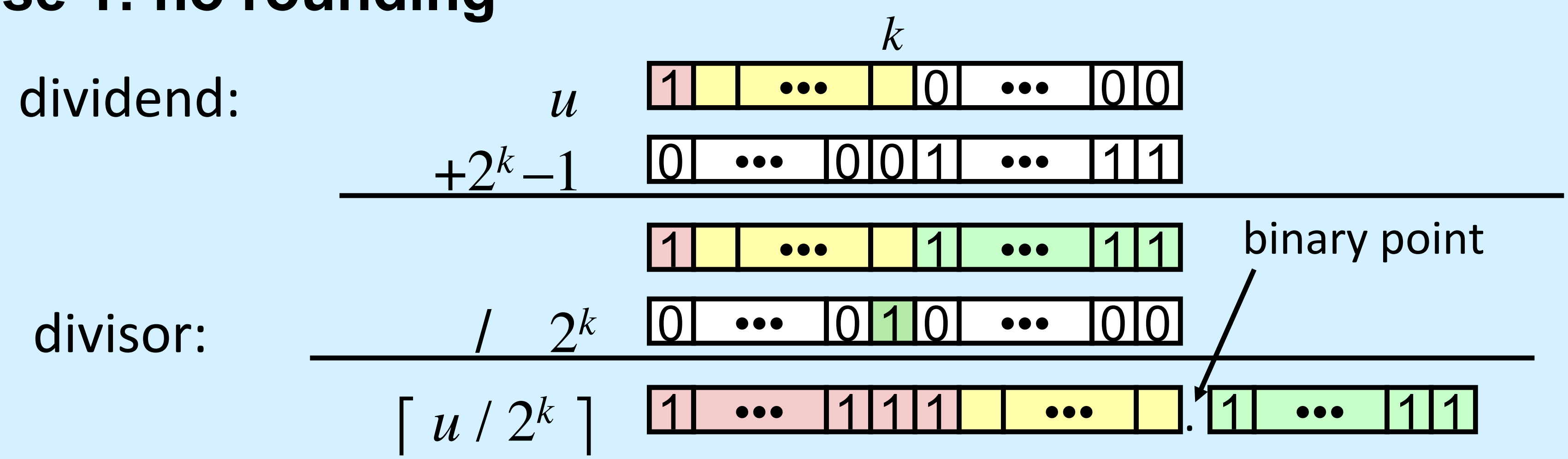


	Division	Computed	Hex	Binary
`y`	-15213	-15213	`C4 93`	`11000100 10010011`
`y >> 1`	-7606.5	-7607	`E2 49`	`11100010 01001001`
`y >> 4`	-950.8125	-951	`FC 49`	`11111100 01001001`
`y >> 8`	-59.4257813	-60	`FF C4`	`11111111 11000100`

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**

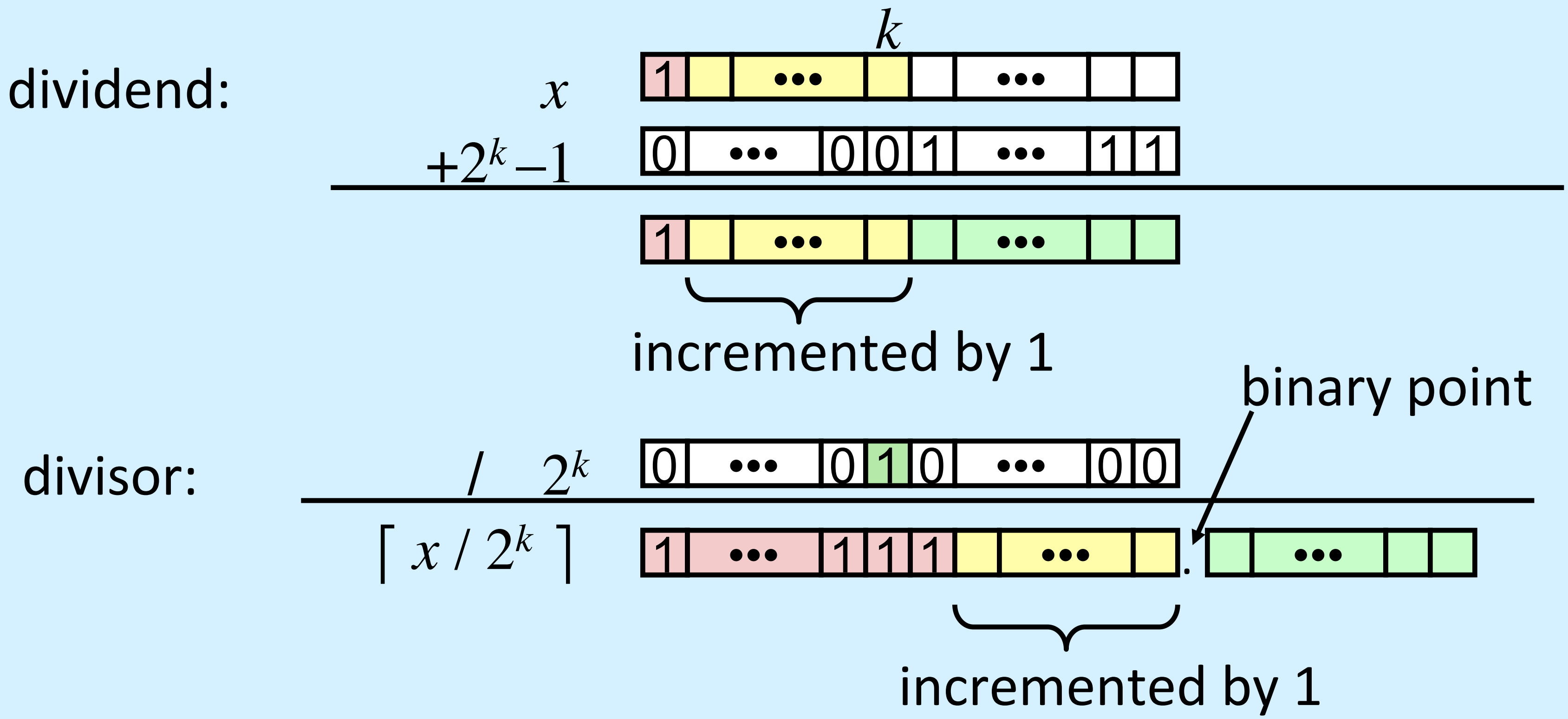
Case 1: no rounding



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

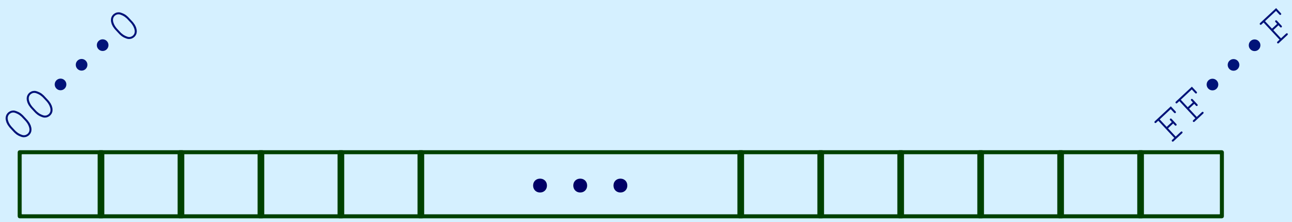
```
unsigned i;
for (i = cnt-2; i >= 0; i--)
  a[i] += a[i+1];
```

  - 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

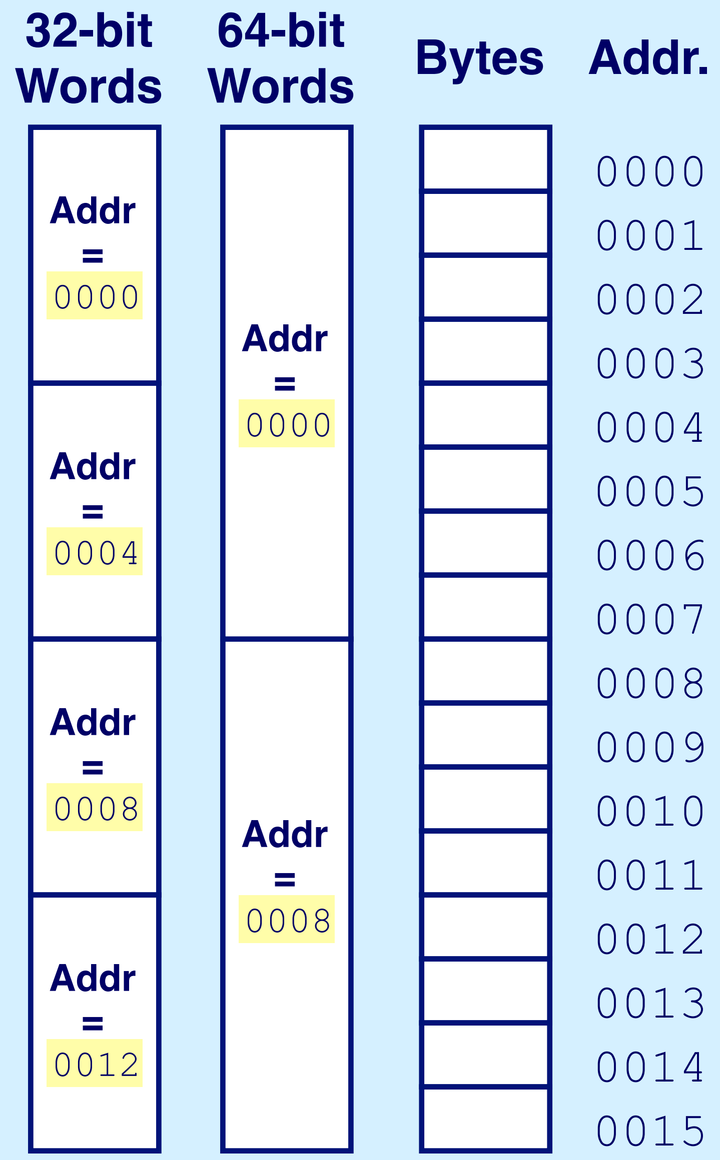
- Note: system provides private address spaces to each "process"
  - think of a process as a program being executed
  - so, a program can clobber its own data, but not that of others

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)
    - » become 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)



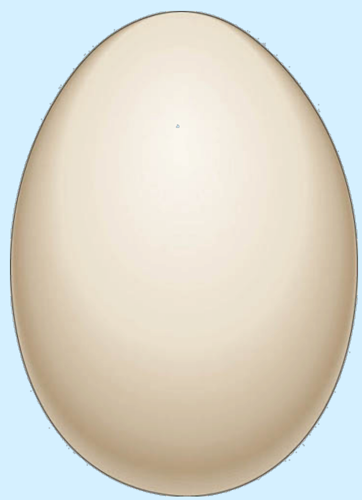
Byte Ordering

- Four-byte integer
  - 0x7654321
- Stored at location 0x100
  - which byte is at 0x100?
  - which byte is at 0x103?

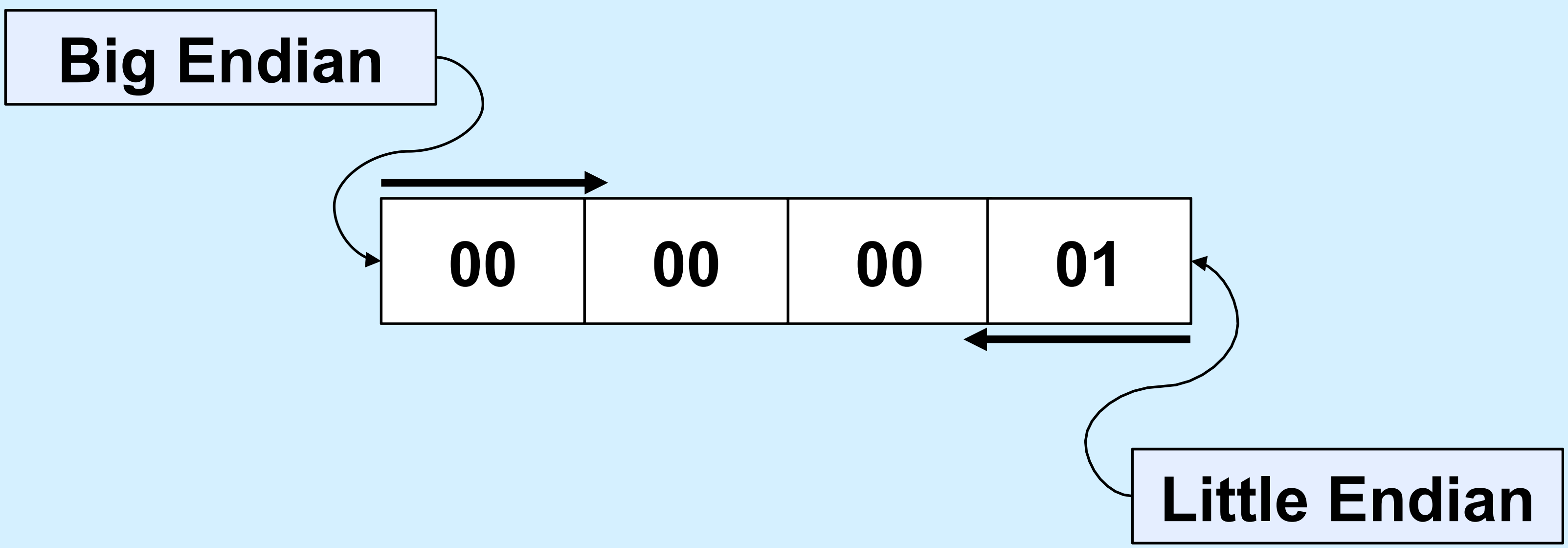
?

01	23	45	67	Little-endian
0x100	0x101	0x102	0x103	

67	45	23	01	Big-endian
0x100	0x101	0x102	0x103	



Byte Ordering (2)



Quiz 4

```
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) $2^{32}-1$