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 ...

12-bit computer word

0 1 1 1 1 1 1 0 0 1 1 1 1
3 7 1 7

16-bit computer word

0 0 0 0 0 1 1 1 1 1 1 0 0 1 1 1 1
0 7 C F
Algorithm ...

```c
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 ...

```bash
$ 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 00000000₂ to 11111111₂
  - decimal: 0₁₀ to 255₁₀
  - hexadecimal 00₁₆ to FF₁₆
    » base 16 number representation
    » use characters ‘0’ to ‘9’ and ‘A’ to ‘F’
    » write FA1D37B₁₆ in C as
      • 0xFA1D37B
      • 0xfa1d37b
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

<table>
<thead>
<tr>
<th>&amp;</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Not
- ~A = 1 when A=0

<table>
<thead>
<tr>
<th>~</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Exclusive-Or (Xor)
- A^B = 1 when either A=1 or B=1, but not both

<table>
<thead>
<tr>
<th>^</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
General Boolean Algebras

• Operate on bit vectors
  – operations applied bitwise

\[
\begin{align*}
01101001 & \quad 01101001 & \quad 01101001 \\
\& 01010101 & | 01010101 & ^ 01010101 & ~ 01010101 \\
01000001 & 01111101 & 00111100 & 10101010
\end{align*}
\]

• All of the properties of boolean algebra apply
Example: Representing & Manipulating Sets

• Representation
  – width-\(w\) bit vector represents subsets of \{0, \ldots, w-1\}
  – \(a_j = 1\) iff \(j \in A\)

\[
\begin{align*}
01101001 & \quad \{0, 3, 5, 6\} \\
76543210 & \\
01010101 & \quad \{0, 2, 4, 6\} \\
76543210 &
\end{align*}
\]

• Operations
  & intersection \quad \begin{align*}01000001 & \quad \{0, 6\} \\
| & \quad \begin{align*}01111101 & \quad \{0, 2, 3, 4, 5, 6\} \\
^ & \quad \begin{align*}00111100 & \quad \{2, 3, 4, 5\} \\
\sim & \quad \begin{align*}10101010 & \quad \{1, 3, 5, 7\}
\end{align*}
\end{align*}
\end{align*}
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)
  \(~0x41 \rightarrow 0xBE\)
  \(~01000001_2 \rightarrow 10111110_2\)
  \(~0x00 \rightarrow 0xFF\)
  \(~00000000_2 \rightarrow 11111111_2\)

- Bitwise operations:
  \(0x69 \& 0x55 \rightarrow 0x41\)
  \(01101001_2 \& 01010101_2 \rightarrow 01000001_2\)
  \(0x69 \mid 0x55 \rightarrow 0x7D\)
  \(01101001_2 \mid 01010101_2 \rightarrow 01111101_2\)
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
  - &&, ||, !
    » 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)

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

• **Left Shift:** \( x \ll y \)
  - shift bit-vector \( x \) left \( y \) positions
    - throw away extra bits on left
      » fill with 0’s on right

• **Right Shift:** \( x \gg y \)
  - 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 \(\geq\) word size

<table>
<thead>
<tr>
<th>Argument ( x )</th>
<th>01100010</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \ll 3 )</td>
<td>00010000</td>
</tr>
<tr>
<td><strong>Log. ( \gg 2 )</strong></td>
<td>00011000</td>
</tr>
<tr>
<td><strong>Arith. ( \gg 2 )</strong></td>
<td>00011000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Argument ( x )</th>
<th>10100010</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \ll 3 )</td>
<td>00010000</td>
</tr>
<tr>
<td><strong>Log. ( \gg 2 )</strong></td>
<td>00101000</td>
</tr>
<tr>
<td><strong>Arith. ( \gg 2 )</strong></td>
<td>11101000</td>
</tr>
</tbody>
</table>
Signed Integers

- Sign-magnitude

\[
\begin{array}{cccccccc}
\text{sign} & b_{w-1} & b_{w-2} & b_{w-3} & \ldots & b_2 & b_1 & b_0 \\
\text{magnitude} & & & & & & & \\
\end{array}
\]

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

- two representations of zero!
Signed Integers

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

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

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

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

two zeroes!
Signed Integers

- Two’s complement
  
  \[ b_{w-1} = 0 \implies \text{non-negative number} \]

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

  \[ b_{w-1} = 1 \implies \text{negative number} \]

  \[
  \text{value} = (-1) \cdot 2^{w-1} + \sum_{i=0}^{w-2} b_i \cdot 2^i
  \]
Signed Integers

• Negating two’s complement

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

– how to compute –value?

(\sim\text{value})+1
Signed Integers

• Negating two’s complement (continued)

\[ value + (\sim value + 1) \]

\[ = (value + \sim value) + 1 \]

\[ = (2^w-1) + 1 \]

\[ = 2^w \]

\[ \begin{array}{cccccc}
1 & 0 & 0 & 0 & \ldots & 0 & 0 & 0 \\
\end{array} \]
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**
  - $U_{\text{Min}} = 0$
    - 000...0
  - $U_{\text{Max}} = 2^w - 1$
    - 111...1

- **Two’s Complement Values**
  - $T_{\text{Min}} = -2^{w-1}$
    - 100...0
  - $T_{\text{Max}} = 2^{w-1} - 1$
    - 011...1

- **Other Values**
  - Minus 1
    - 111...1

Values for $W = 16$

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>$U_{\text{Max}}$</td>
<td>65535</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>$T_{\text{Max}}$</td>
<td>32767</td>
<td>7F FF</td>
<td>01111111 11111111</td>
</tr>
<tr>
<td>$T_{\text{Min}}$</td>
<td>-32768</td>
<td>80 00</td>
<td>10000000 00000000</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 00</td>
<td>00000000 00000000</td>
</tr>
</tbody>
</table>
Values for Different Word Sizes

<table>
<thead>
<tr>
<th>W</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>UMax</td>
<td>255</td>
<td>65,535</td>
<td>4,294,967,295</td>
<td>18,446,744,073,709,551,615</td>
</tr>
<tr>
<td>Tmax</td>
<td>127</td>
<td>32,767</td>
<td>2,147,483,647</td>
<td>9,223,372,036,854,775,807</td>
</tr>
<tr>
<td>Tmin</td>
<td>-128</td>
<td>-32,768</td>
<td>-2,147,483,648</td>
<td>-9,223,372,036,854,775,808</td>
</tr>
</tbody>
</table>

• Observations

\[ |T_{Min}| = T_{Max} + 1 \]

» Asymmetric range

\[ U_{Max} = 2 \times T_{Max} + 1 \]

• C Programming

• `#include <limits.h>`

• declares constants, e.g.,

  • ULONG_MAX
  • LONG_MAX
  • LONG_MIN

• values platform-specific
Quiz 3

- What is $-\text{TMin}$ (assuming two’s complement signed integers)?
  a) $\text{TMin}$
  b) $\text{TMax}$
  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 \( W = 32 \):
    
    \[
    T_{\text{MIN}} = -2,147,483,648 , \quad T_{\text{MAX}} = 2,147,483,647
    \]

<table>
<thead>
<tr>
<th>Constant_1</th>
<th>Constant_2</th>
<th>Relation</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0U</td>
<td>&lt;=</td>
<td>unsigned</td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
<td>&lt;</td>
<td>signed</td>
</tr>
<tr>
<td>-1</td>
<td>0U</td>
<td>&gt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>-2147483647-1</td>
<td>&gt;</td>
<td>signed</td>
</tr>
<tr>
<td>2147483647U</td>
<td>-2147483647-1</td>
<td>&lt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>-1</td>
<td>-2</td>
<td>&gt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>(unsigned)-1</td>
<td>-2</td>
<td>&gt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>2147483648U</td>
<td>&lt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>(int) 2147483648U</td>
<td>&gt;</td>
<td>signed</td>
</tr>
</tbody>
</table>
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:
  - $X' = x_{w-1}, \ldots, x_{w-1}, x_{w-2}, \ldots, x_0$
Sign Extension Example

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

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>15213</td>
<td>3B 6D 00111011 01101101</td>
</tr>
<tr>
<td>ix</td>
<td>15213</td>
<td>00 00 3B 6D 00000000 00000000 00111011 01101101</td>
</tr>
<tr>
<td>y</td>
<td>-15213</td>
<td>C4 93 11000100 10010011</td>
</tr>
<tr>
<td>iy</td>
<td>-15213</td>
<td>FF FF C4 93 11111111 11111111 11000100 10010011</td>
</tr>
</tbody>
</table>

- Converting from smaller to larger integer data type
  - C automatically performs sign extension
Does it Work?

\[
\begin{align*}
\text{val}_w &= -2^{w-1} + \sum_{i=0}^{w-2} b_i \cdot 2^i \\
\text{val}_{w+1} &= -2^w + 2^{w-1} + \sum_{i=0}^{w-2} b_i \cdot 2^i \\
&= -2^{w-1} + \sum_{i=0}^{w-2} b_i \cdot 2^i \\
\text{val}_{w+2} &= -2^{w+1} + 2^w + 2^{w-1} + \sum_{i=0}^{w-2} b_i \cdot 2^i \\
&= -2^w + 2^{w-1} + \sum_{i=0}^{w-2} b_i \cdot 2^i \\
&= -2^{w-1} + \sum_{i=0}^{w-2} b_i \cdot 2^i
\end{align*}
\]
Power-of-2 Multiply with Shift

- **Operation**
  - \( u << k \) gives \( u \times 2^k \)
  - both signed and unsigned

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

- 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
  - \( u \gg k \) gives \( \lfloor u / 2^k \rfloor \)
  - uses logical shift

<table>
<thead>
<tr>
<th>Division</th>
<th>Computed</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>x 15213</td>
<td>15213</td>
<td>3B 6D</td>
<td>00111011 01101101</td>
</tr>
<tr>
<td>x &gt;&gt; 1 7606.5</td>
<td>7606</td>
<td>1D B6</td>
<td>00011101 10110110</td>
</tr>
<tr>
<td>x &gt;&gt; 4 950.8125</td>
<td>950</td>
<td>03 B6</td>
<td>00000011 10110110</td>
</tr>
<tr>
<td>x &gt;&gt; 8 59.4257813</td>
<td>59</td>
<td>00 3B</td>
<td>00000000 00111011</td>
</tr>
</tbody>
</table>
Signed Power-of-2 Divide with Shift

- Quotient of signed by power of 2
  - $x \gg k$ gives $\lfloor x / 2^k \rfloor$
  - uses arithmetic shift
  - rounds wrong direction when $x < 0$

<table>
<thead>
<tr>
<th>Division</th>
<th>Computed</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y$</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
<tr>
<td>$y \gg 1$</td>
<td>-7606.5</td>
<td>E2 49</td>
<td>11100010 01001001</td>
</tr>
<tr>
<td>$y \gg 4$</td>
<td>-950.8125</td>
<td>FC 49</td>
<td>111111100 01001001</td>
</tr>
<tr>
<td>$y \gg 8$</td>
<td>-59.4257813</td>
<td>FF C4</td>
<td>111111111 11000100</td>
</tr>
</tbody>
</table>
Correct Power-of-2 Divide

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

Case 1: no rounding

<table>
<thead>
<tr>
<th>dividend:</th>
<th>( u )</th>
<th>+</th>
<th>( +2^k - 1 )</th>
<th>( \lfloor \frac{u}{2^k} \rfloor )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 1 \ldots 0 \ldots 0 \ldots )</td>
<td>( 1 \ldots 0 \ldots 0 \ldots )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( 0 \ldots 0 \ldots 0 \ldots )</td>
<td>( 0 \ldots 0 \ldots 0 \ldots )</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>divisor:</th>
<th>( / )</th>
<th>( 2^k )</th>
<th>( \lfloor \frac{u}{2^k} \rfloor )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 0 \ldots 0 \ldots 0 \ldots )</td>
<td>( 0 \ldots 0 \ldots 0 \ldots )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Biasing has no effect
Correct Power-of-2 Divide (Cont.)

Case 2: rounding

\[
\begin{align*}
\text{dividend:} & & x & = & 1.\cdots & \cdots & \cdots & \cdots \\
 & & \quad +2^k - 1 & = & 0.\cdots & 001 & \cdots & 111 \\
\end{align*}
\]

\[
\begin{align*}
\text{divisor:} & & 1 & = & \cdots & \cdots & \cdots & \cdots \\
 & & \quad / & = & 0.\cdots & 010 & \cdots & 00 \\
\quad \left[ x / 2^k \right] & & 1 & = & \cdots & 111 & 1 & \cdots \\
\end{align*}
\]

- incremented by 1
- binary point
- incremented by 1

Biasing adds 1 to final result
Why Should I Use Unsigned?

• *Don’t* use just because number nonnegative
  – easy to make mistakes
    
    ```c
    unsigned i;
    for (i = cnt-2; i >= 0; i--)
        a[i] += a[i+1];
    
    – can be very subtle
    ```

    ```c
    #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?

Little-endian:

<table>
<thead>
<tr>
<th>01</th>
<th>23</th>
<th>45</th>
<th>67</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>0x101</td>
<td>0x102</td>
<td>0x103</td>
</tr>
</tbody>
</table>

Big-endian:

<table>
<thead>
<tr>
<th>67</th>
<th>45</th>
<th>23</th>
<th>01</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>0x101</td>
<td>0x102</td>
<td>0x103</td>
</tr>
</tbody>
</table>
Byte Ordering (2)

<table>
<thead>
<tr>
<th>Big Endian</th>
<th>Little Endian</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 00 00 01</td>
<td>00 00 00 01</td>
</tr>
</tbody>
</table>
Quiz 4

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