CMPSC 311 - Introduction to Systems Programming
Module: Bits/Byte Operations

Professor Patrick McDaniel
Fall 2013
There was a mistake on the original assignment slides.

You have to install `libgcrypt11-dev` via `apt-get`.

```
sudo apt-get install libgcrypt11-dev
```
Base-X Systems

- All base-X systems have the following characteristic:

Assume a base $b$ and digits $P = \{p_k, p_{k-1}, p_{k-2}, \ldots, p_1, p_0\}$

$$value = \sum_{i=0}^{k} b^i * p_i$$

where $\forall p_i \in P, p_i = [0, b - 1]$

Example: decimal 1,234 $\rightarrow$ $P = \{1, 2, 3, 4\}$
A Simple Example

Consider $b = 10$ and $P = \{1, 2, 3, 4\}$

\[
value = \sum_{i=0}^{k} b^i \times p_i
\]

\[
\begin{align*}
10^0 \times 4 &= 4 \\
10^1 \times 3 &= 30 \\
10^2 \times 2 &= 200 \\
10^3 \times 1 &= 1000
\end{align*}
\]

\[
value = 1000 + 200 + 30 + 4
\]
A Simple Example

Consider \( b = 2 \) and \( P = \{1, 0, 1, 1, 0, 0, 1, 1\} \)

\[
\text{value} = \sum_{i=0}^{k} b^i \cdot p_i
\]

\[
\begin{align*}
2^0 \cdot 1 &= 1 \\
2^1 \cdot 1 &= 2 \\
2^2 \cdot 0 &= 0 \\
2^3 \cdot 0 &= 0 \\
2^4 \cdot 1 &= 16 \\
2^5 \cdot 1 &= 32 \\
2^6 \cdot 0 &= 0 \\
2^7 \cdot 1 &= 128
\end{align*}
\]

\[
\text{value} = 1 + 2 + 16 + 32 + 128(= 179)
\]

\[
\begin{align*}
2^7 &= 128 \\
2^6 &= 64 \\
2^5 &= 32 \\
2^4 &= 16 \\
2^3 &= 8 \\
2^2 &= 4 \\
2^1 &= 2 \\
2^0 &= 1
\end{align*}
\]
Converting Decimal to Binary

• Converting decimal to hex is just the reverse!

Input: decimal number x
1: Find largest power i of 2 <= x
2: while (i>=0)
   2a: if (x>2^i) then
      2a1: next digit is a 1
      2a2: x = x-2^i
   2b: else
      2b1: next digit is a 0
   2c: i=i-1
3: done

\[
\begin{align*}
2^7 &= 128 \\
2^6 &= 64 \\
2^5 &= 32 \\
2^4 &= 16 \\
2^3 &= 8 \\
2^2 &= 4 \\
2^1 &= 2 \\
2^0 &= 1
\end{align*}
\]
Converting Decimal to Binary

235      Input: decimal number x
-       1: Find largest power i of 2 <= x
---------  2: while (i>=0)
  107    2a: if (x>2^i) then
            2a1: next digit is a 1
            2a2: x = x-2^i
       2b: else
            2b1: next digit is a 0
       2c: i=i-1
   3: done
Converting Decimal to Binary

Input: decimal number x

1: Find largest power i of 2 <= x

2: while (i>=0)
   2a: if (x>2^i) then
      2a1: next digit is a 1
      2a2: x = x-2^i
   2b: else
      2b1: next digit is a 0
   2c: i=i-1

3: done
Converting Decimal to Binary

Input: decimal number x

1: Find largest power i of 2 <= x

2: while (i>=0)
   2a: if (x>2^i) then
       2a1: next digit is a 1
       2a2: x = x-2^i
   2b: else
       2b1: next digit is a 0
   2c: i=i-1

3: done
Converting Decimal to Binary

11 Input: decimal number $x$
- 0 1: Find largest power $i$ of 2 $\leq x$
- --------- 2: while ($i >= 0$)
  11 2a: if ($x > 2^i$) then
        2a1: next digit is a 1
        2a2: $x = x - 2^i$
  2b: else
        2b1: next digit is a 0
  2c: $i = i - 1$
3: done

Val: 1110

<table>
<thead>
<tr>
<th>Power</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^7$</td>
<td>128</td>
</tr>
<tr>
<td>$2^6$</td>
<td>64</td>
</tr>
<tr>
<td>$2^5$</td>
<td>32</td>
</tr>
<tr>
<td>$2^4$</td>
<td>16</td>
</tr>
<tr>
<td>$2^3$</td>
<td>8</td>
</tr>
<tr>
<td>$2^2$</td>
<td>4</td>
</tr>
<tr>
<td>$2^1$</td>
<td>2</td>
</tr>
<tr>
<td>$2^0$</td>
<td>1</td>
</tr>
</tbody>
</table>
Converting Decimal to Binary

11  Input: decimal number x
-  8  1: Find largest power i of 2 <= x
  --------  2: while (i>=0)
      3  2a: if (x>2^i) then
       2a1: next digit is a 1
            2a2: x = x-2^i
       2b: else
       2b1: next digit is a 0
       2c: i=i-1
3: done
Converting Decimal to Binary

3  Input: decimal number x

- 0  1: Find largest power i of 2 <= x

-------------  2: while (i>=0)

3  2a: if (x>2^i) then
    2a1: next digit is a 1
    2a2: x = x-2^i

    2b: else
    2b1: next digit is a 0

    2c: i=i-1

3: done

Val: 1110 10

<table>
<thead>
<tr>
<th>n</th>
<th>2^n</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>64</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
Converting Decimal to Binary

3  Input: decimal number x
-  2  1: Find largest power i of 2 <= x
    --------------  2: while (i>=0)
    1          2a: if (x>2^i) then
                2a1: next digit is a 1
                2a2: x = x-2^i
          2b: else
                2b1: next digit is a 0
                2c: i=i-1
3: done
Converting Decimal to Binary

1 Input: decimal number \( x \)
- 1 1: Find largest power \( i \) of 2 <= \( x \)
-- 2: while (\( i >= 0 \))
  0 2a: if (\( x > 2^i \)) then
    2a1: next digit is a 1
    2a2: \( x = x - 2^i \)
  2b: else
    2b1: next digit is a 0
    2c: \( i = i - 1 \)
3: done

Val: 1110 1011

\[
\begin{array}{c|c}
2^7 & 128 \\
2^6 & 64 \\
2^5 & 32 \\
2^4 & 16 \\
2^3 & 8 \\
2^2 & 4 \\
2^1 & 2 \\
2^0 & 1 \\
\end{array}
\]
<table>
<thead>
<tr>
<th></th>
<th>a) 25</th>
<th>c) 3,274</th>
<th>e) 2,864,434,397</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>b) 437</td>
<td>d) 7,108</td>
<td>f) 287,454,020</td>
</tr>
</tbody>
</table>

Input: decimal number \( x \)
1: Find largest power \( i \) of 2 less than \( x \)
   2: while \((x>0)\)
      2a: if \((x>2^i)\) then
         2a1: next digit is a 1
         2a2: \( x = x-2^i \)
      2b: else
         2b1: next digit is a 0
      2c: \( i=i-1 \)
3: done

\[
\begin{align*}
2^{31} & = 2147483648 & 2^{30} & = 1073741824 \\
2^{29} & = 536870912 & 2^{28} & = 268435456 \\
2^{27} & = 134217728 & 2^{26} & = 67108864 \\
2^{25} & = 33554432 & 2^{24} & = 16777216 \\
2^{23} & = 8388608 & 2^{22} & = 4194304 \\
2^{21} & = 2,097,152 & 2^{20} & = 1,048,576 \\
2^{19} & = 524,288 & 2^{18} & = 262,144 \\
2^{17} & = 131,072 & 2^{16} & = 65,536 \\
2^{15} & = 32,768 & 2^{14} & = 16,384 \\
2^{13} & = 8,192 & 2^{12} & = 4,096 \\
2^{11} & = 2,048 & 2^{10} & = 1,024 \\
2^9 & = 512 & 2^8 & = 256 \\
2^7 & = 128 & 2^6 & = 64 \\
2^5 & = 32 & 2^4 & = 16 \\
2^3 & = 8 & 2^2 & = 4 \\
2^1 & = 2 & 2^0 & = 1
\end{align*}
\]
In class (decimal to hex)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a) 25 = 19</td>
<td>c) 3,274 = 0xCCA</td>
<td>e) 2,864,434,397 = 0xAABBCCDD</td>
</tr>
<tr>
<td>b) 437 = 0x1B5</td>
<td>d) 7,108 = 0x1BC4</td>
<td>f) 287454020 = 0x11223344</td>
</tr>
</tbody>
</table>

Input: decimal number x
1: Find largest power i of 2 less than x
2: while (x>0)
   2a: if (x>2^-i) then
      2a1: next digit is a 1
      2a2: x = x-2^-i
   2b: else
      2b1: next digit is a 0
   2c: i=i-1
3: done
OK, I lied
Byte Ordering

• How should bytes within a multi-byte word be ordered in memory?

• Conventions
  ‣ Big Endian: Sun, PPC Mac, Internet
    • Least significant byte has highest address
  ‣ Little Endian: x86
    • Least significant byte has lowest address
Byte Ordering Example

- **Big Endian**
  - Least significant byte has highest address

- **Little Endian**
  - Least significant byte has lowest address

- **Example**
  - Variable `x` has 4-byte representation `0x01234567`
  - Address given by `&x` is `0x100`

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

<table>
<thead>
<tr>
<th>Little Endian</th>
<th>0x100</th>
<th>0x101</th>
<th>0x102</th>
<th>0x103</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>67</td>
<td>45</td>
<td>23</td>
<td>01</td>
</tr>
</tbody>
</table>
Reading Byte-Reversed Listings

- Disassembly
  - Text representation of binary machine code
  - Generated by program that reads the machine code
- Example Fragment

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction Code</th>
<th>Assembly Rendition</th>
</tr>
</thead>
<tbody>
<tr>
<td>8048365:</td>
<td>5b</td>
<td>pop %ebx</td>
</tr>
<tr>
<td>8048366:</td>
<td>81 c3 ab 12 00 00</td>
<td>add $0x12ab,%ebx</td>
</tr>
<tr>
<td>804836c:</td>
<td>83 bb 28 00 00 00 00</td>
<td>cmpl $0x0,0x28(%ebx)</td>
</tr>
</tbody>
</table>

- Deciphering Numbers
  - Value: 0x12ab
  - Pad to 32 bits: 0x000012ab
  - Split into bytes: 00 00 12 ab
  - Reverse: ab 12 00 00
Examining Data Representations

- Code to Print Byte Representation of Data
  - Casting pointer to unsigned char * creates byte array

```c
typedef unsigned char *pointer;

void show_bytes(pointer start, int len){
    int i;
    for (i = 0; i < len; i++)
        printf("%p\t0x%.2x\n", start+i, start[i]);
    printf("\n");
}
...
int a = 15213;
printf("int a = 15213;\n");
show_bytes((pointer) &a, sizeof(int));
```

Printf directives:
- %p: Print pointer
- %x: Print Hexadecimal

Result (Linux):
```
int a = 15213;
0x11fffcbb 0x00
0x11fffcba 0x00
0x11fffcb9 0x3b
0x11ffffcb8 0x6d
```
Representing Integers

int A = 15213;

IA32, x86-64       Sun

6D  3B  00  00

long int C = 15213;

IA32   x86-64       Sun

6D  3B  00  00

int B = -15213;

IA32, x86-64       Sun

93  C4  FF  FF

Two’s complement representation (Covered later)
Representing signed numbers

- The top bit is always “sign bit”, where the value is interpreted to be positive if it is zero and negative if the value is one.

For positive integers, just interpret as same as unsigned.

\[ \begin{array}{cccccccc}
0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\
\hline
3 & 5 \\
\end{array} \]

\[ = 0x35 = 53 \]

Positive
Representing signed numbers

- Interpret a negative number from the twos complement
  - To compute the twos complement of a k-bit value, subtract the bottom k-1 from $2^k$

\[
\begin{array}{cccccccc}
1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\
\hline
3 & 5
\end{array}
\]
\[= -(128 - 53) = -75\]

Negative

\[
\begin{array}{cccccccc}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]
\[= 2^k = 128\]

\[
\begin{array}{cccccccc}
0 & 1 & 1 & 0 & 1 & 0 & 1 & 1
\end{array}
\]
\[= 0x35 = 53\]
**Exercise**

What are the signed interpretations of these values ...

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

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>8-0 =-8</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Representing Pointers

```c
int B = -15213;
int *P = &B;
```

<table>
<thead>
<tr>
<th></th>
<th>Sun</th>
<th>IA32</th>
<th>x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>EF</td>
<td></td>
<td>D4</td>
<td>0C</td>
</tr>
<tr>
<td>FF</td>
<td></td>
<td>F8</td>
<td>89</td>
</tr>
<tr>
<td>FB</td>
<td></td>
<td>FF</td>
<td>EC</td>
</tr>
<tr>
<td>2C</td>
<td></td>
<td>BF</td>
<td>FF</td>
</tr>
</tbody>
</table>

**Note:** Different compilers & machines assign different locations to objects
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>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</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>
Application of Boolean Algebra

• Applied to Digital Systems by Claude Shannon
  ‣ 1937 MIT Master’s Thesis
  ‣ Reason about networks of relay switches
  ‣ Encode closed switch as 1, open switch as 0

Connection when
A&~B | ~A&B = A^B
General Boolean Algebras

• Operate on Bit Vectors
  ▪ Operations applied bitwise

  \[
  \begin{array}{c}
  01101001 \\
  \& 01010101
  \end{array}
  \begin{array}{c}
  01101001 \\
  \| 01010101
  \end{array}
  \begin{array}{c}
  01101001 \\
  ^ 01010101
  \end{array}
  \begin{array}{c}
  01000001 \\
  \sim 01010101
  \end{array}
  \begin{array}{c}
  01111101 \\
  | 01111101
  \end{array}
  \begin{array}{c}
  01101001 \\
  ^ 01010101
  \end{array}
  \begin{array}{c}
  00111100 \\
  \sim 01010101
  \end{array}
  \begin{array}{c}
  10101010 \\
  ~ 01010101
  \end{array}
  \]

• All of the Properties of Boolean Algebra Apply
General Boolean Algebras

• Operate on Bit Vectors
  ‣ Operations applied bitwise

\[
\begin{array}{c}
01101001 \\
\& 01010101 \\
\hline
01000001
\end{array}
\quad
\begin{array}{c}
01101001 \\
| 01010101 \\
\hline
01111101
\end{array}
\quad
\begin{array}{c}
01101001 \\
^ 01010101 \\
\hline
00111100
\end{array}
\quad
\begin{array}{c}
\sim 01010101 \\
\hline
10101010
\end{array}
\]

• All of the Properties of Boolean Algebra Apply
Representing & Manipulating Sets

• Representation
  ‣ Width \( w \) bit vector represents subsets of \( \{0, \ldots, w-1\} \)
  ‣ \( a_j = 1 \) if \( j \in A \)

\[
\begin{align*}
01010101 &= \{0, 2, 4, 6\} \\
76543210
\end{align*}
\]

\[
\begin{align*}
01101001 &= \{0, 3, 5, 6\} \\
76543210
\end{align*}
\]

Operations On Sets:

\& Intersection \quad 01000001 \quad \{0, 6\}

| Union \quad 01111101 \quad \{0, 2, 3, 4, 5, 6\}

^ Symmetric difference \quad 00111100 \quad \{2, 3, 4, 5\}

~ Complement \quad 10101010 \quad \{1, 3, 5, 7\}
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

- Examples (Char data type)
  - ~0x41 ➔ 0xBE
    - ~01000001₂ ➔ 1011110₂
  - ~0x00 ➔ 0xFF
    - ~00000000₂ ➔ 1111111₂
  - 0x69 & 0x55 ➔ 0x41
    - 01101001₂ & 01010101₂ ➔ 01000001₂
  - 0x69 | 0x55 ➔ 0x7D
    - 01101001₂ | 01010101₂ ➔ 01111101₂
Contrast: Logic Operations in C

- Contrast to Logical Operators (&&, ||, !)
  - View 0 as “False”
  - Anything nonzero as “True”
  - Always return 0 or 1
  - Early termination

- Examples (char data type)
  - !0x41 ➔ 0x00
  - !0x00 ➔ 0x01
  - !!0x41 ➔ 0x01
  - 0x69 && 0x55 ➔ 0x01
  - 0x69 || 0x55 ➔ 0x01
  - p && *p (avoids null pointer access)
Shift Operations

- A shift operator (<< or >>) moves bits to the right or left, throwing away bits and adding bits as necessary.

\[
\begin{array}{cccccccc}
A & B & C & D & E & F & G & H \\
1 & 0 & 1 & 0 & 1 & 1 & 0 & 0
\end{array}
\]

\[
X = \quad \begin{array}{cccccccc}
A & B & C & D & E & F & G & H \\
1 & 0 & 1 & 0 & 1 & 1 & 0 & 0
\end{array}
\]

\[
X = X \ll 3;
\]

(throw away)

\[
\begin{array}{cccccccc}
A & B & C & D & E & F & G & H \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}
\]

Read as this: shift the bits of the value 3 places to the left.

("new" bits)
Types of Shift

• 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 right

• 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>Log. ( \gg 2 )</td>
<td>00011000</td>
</tr>
<tr>
<td>Arith. ( \gg 2 )</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>Log. ( \gg 2 )</td>
<td>00101000</td>
</tr>
<tr>
<td>Arith. ( \gg 2 )</td>
<td>11101000</td>
</tr>
</tbody>
</table>
Putting it all together

• Suppose you want to create a function to place multiple values in the same 32-bit
  ‣ Value a in least significant byte
  ‣ Value b in 2nd byte
  ‣ Value c in 3rd byte
  ‣ Value d in 4th byte
Using bit operations ...

```c
uint32_t pack_bytes( uint32_t a, uint32_t b, uint32_t c, uint32_t d ) {

    // Setup some local values
    uint32_t retval = 0x0, tempa, tempb, tempc, tempd;

    tempa = a&0xff; // Make sure you are only getting the bottom 8 bits
    tempb = (b&0xff) << 8; // Shift value to the second byte
    tempc = (c&0xff) << 16; // Shift value to the third byte
    tempd = (d&0xff) << 24; // Shift value to the top byte
    retval = tempa|tempb|tempc|tempd; // Now combine all of the values

    // Print out all of the values
    printf( "A: 0x%08x\n", tempa );
    printf( "B: 0x%08x\n", tempb );
    printf( "C: 0x%08x\n", tempc );
    printf( "D: 0x%08x\n", tempd );

    // Return the computed value
    return( retval );
}

...  

printf( "Packed bytes : 0x%08x\n", pack_bytes(0x111, 0x222, 0x333, 0x444) );
```