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

Professor Patrick McDaniel
Fall 2013

Base-X Systems

PENNSTATE 1855

- 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

PENNSTATE 1855

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

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

$$\begin{array}{rcr} 10^0 * 4 & = & 4 \\ 10^1 * 3 & = & 30 \\ 10^2 * 2 & = & 200 \\ 10^3 * 1 & = & 1000 \\ \hline value & = & 1000 + 200 + 30 + 4 \end{array}$$

A Simple Example

PENNSTATE 1855

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

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

$$\begin{array}{lcr}
2^0 * 1 & = & 1 \\
2^1 * 1 & = & 2 \\
2^2 * 0 & = & 0 \\
2^3 * 0 & = & 0 \\
2^4 * 1 & = & 16 \\
2^5 * 1 & = & 32 \\
2^6 * 0 & = & 0 \\
2^7 * 1 & = & 128 \\
\hline
value & = & 1 + 2 + 16 + 32 + 128 (= 179)
\end{array}$$

$$\begin{array}{lcr}
2^7 & = & 128 \\
2^6 & = & 64 \\
2^5 & = & 32 \\
2^4 & = & 16 \\
2^3 & = & 8 \\
2^2 & = & 4 \\
2^1 & = & 2 \\
2^0 & = & 1
\end{array}$$

Converting Decimal to Binary

- Converting decimal to hex is just the reverse!

```
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
```

$$
\begin{aligned}
2^7 &= 128 \\
2^6 &= 64 \\
2^5 &= 32 \\
2^4 &= 16 \\
2^3 &= 8 \\
2^2 &= 4 \\
2^1 &= 2 \\
2^0 &= 1
\end{aligned}
$$

Converting Decimal to Binary

```
          235
-         128
------------
          107

Val : 1
```

```
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
```

$$
\begin{aligned}
2^7 &= 128 \\
2^6 &= 64 \\
2^5 &= 32 \\
2^4 &= 16 \\
2^3 &= 8 \\
2^2 &= 4 \\
2^1 &= 2 \\
2^0 &= 1
\end{aligned}
$$

Converting Decimal to Binary

```
          107
-          64
------------
           43

Val: 11
```

```
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
```

$$
\begin{aligned}
2^7 &= 128 \\
2^6 &= 64 \\
2^5 &= 32 \\
2^4 &= 16 \\
2^3 &= 8 \\
2^2 &= 4 \\
2^1 &= 2 \\
2^0 &= 1
\end{aligned}
$$

Converting Decimal to Binary

```
            43
-           32
------------

            11


Val: 111
```

```
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
```

$$
\begin{aligned}
2^7 &= 128 \\
2^6 &= 64 \\
2^5 &= 32 \\
2^4 &= 16 \\
2^3 &= 8 \\
2^2 &= 4 \\
2^1 &= 2 \\
2^0 &= 1
\end{aligned}
$$

Converting Decimal to Binary

```
              11
-              0
------------
              11

Val: 1110
```

```
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
```

2^7	=	128
2^6	=	64
2^5	=	32
2^4	=	16
2^3	=	8
2^2	=	4
2^1	=	2
2^0	=	1

Converting Decimal to Binary

```
              11
-              8
-------------
               3

Val: 1110 1
```

```
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
```

2^7	=	128
2^6	=	64
2^5	=	32
2^4	=	16
2^3	=	8
2^2	=	4
2^1	=	2
2^0	=	1

Converting Decimal to Binary

```
                3
-               0
-------------
                3

Val: 1110 10
```

```
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
```

2^7	=	128
2^6	=	64
2^5	=	32
2^4	=	16
2^3	=	8
2^2	=	4
2^1	=	2
2^0	=	1

Converting Decimal to Binary

```
                3
-               2
-------------
                1

Val: 1110 101
```

```
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
```

2^7	=	128
2^6	=	64
2^5	=	32
2^4	=	16
2^3	=	8
2^2	=	4
2^1	=	2
2^0	=	1

Converting Decimal to Binary

```
                1
-               1
-------------
                0

Val: 1110 1011
```

```
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
```

$$
\begin{aligned}
2^7 &= 128 \\
2^6 &= 64 \\
2^5 &= 32 \\
2^4 &= 16 \\
2^3 &= 8 \\
2^2 &= 4 \\
2^1 &= 2 \\
2^0 &= 1
\end{aligned}
$$

In class (decimal to hex)

PENNSTATE 1855

a) 25	c) 3,274	e) 2,864,434,397
b) 437	d) 7,108	f) 287,454,020

```
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
```

$$
\begin{array}{rcrrcr}
2^{31} & = & 2147483648 & 2^{30} & = & 1073741824 \\
2^{29} & = & 536870912 & 2^{28} & = & 268435456 \\
2^{27} & = & 134217728 & 2^{26} & = & 67108864 \\
2^{25} & = & 33554432 & 2^{24} & = & 16777216 \\
2^{23} & = & 8388608 & 2^{22} & = & 4194304 \\
2^{21} & = & 2097152 & 2^{20} & = & 1048576 \\
2^{19} & = & 524288 & 2^{18} & = & 262144 \\
2^{17} & = & 131072 & 2^{16} & = & 65536 \\
2^{15} & = & 32768 & 2^{14} & = & 16384 \\
2^{13} & = & 8192 & 2^{12} & = & 4096 \\
2^{11} & = & 2048 & 2^{10} & = & 1024 \\
2^{9} & = & 512 & 2^{8} & = & 256 \\
2^{7} & = & 128 & 2^{6} & = & 64 \\
2^{5} & = & 32 & 2^{4} & = & 16 \\
2^{3} & = & 8 & 2^{2} & = & 4 \\
2^{1} & = & 2 & 2^{0} & = & 1
\end{array}
$$

In class (decimal to hex)

a) 25 = 19	c) 3,274 = 0xCCA	e) 2,864,434,397 = 0xAABBCCDD
b) 437 = 0x1B5	d) 7,108 = 0x1BC4	f) 287454020 = 0x11223344

```
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
```

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}	=	2097152	2^{20}	=	1048576
2^{19}	=	524288	2^{18}	=	262144
2^{17}	=	131072	2^{16}	=	65536
2^{15}	=	32768	2^{14}	=	16384
2^{13}	=	8192	2^{12}	=	4096
2^{11}	=	2048	2^{10}	=	1024
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

OK, I lied

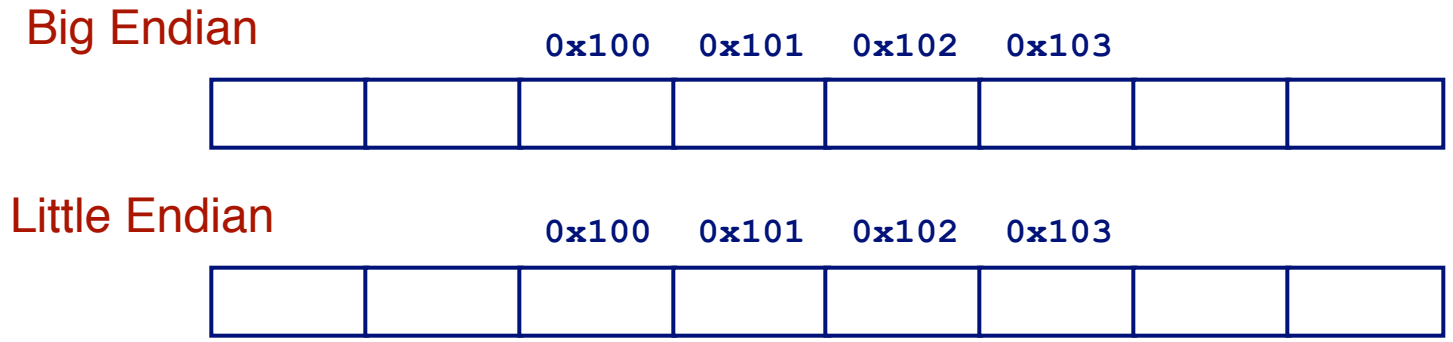
Byte Ordering

PENNSTATE 1855

- 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

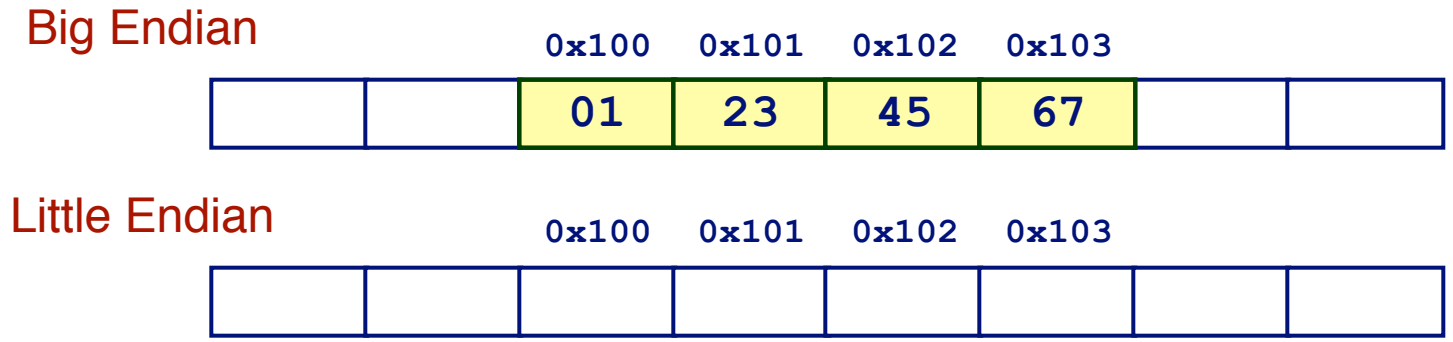
Byte Ordering Example

- 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



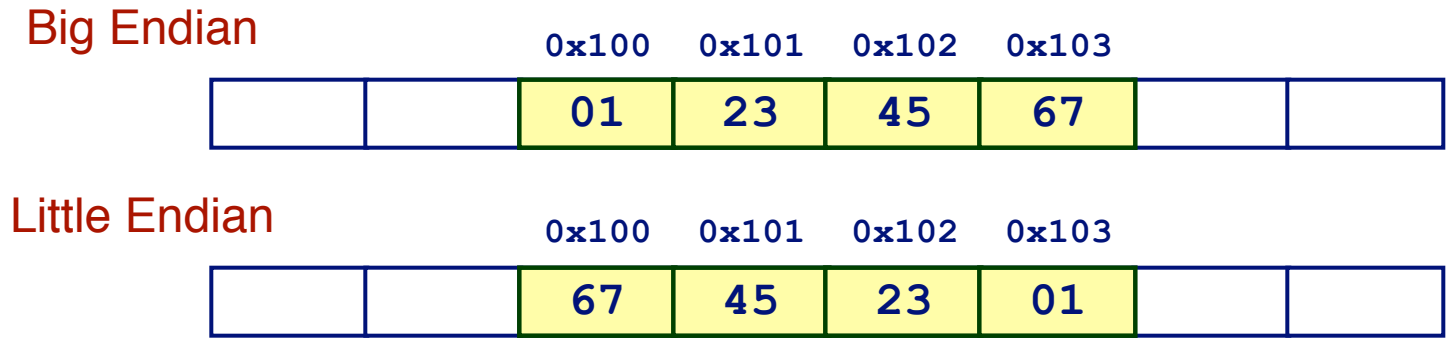
Byte Ordering Example

- 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



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



Reading Byte-Reversed Listings

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

Address	Instruction Code	Assembly Rendition
8048365:	5b	pop %ebx
8048366:	81 c3 ab 12 00 00	add $0x12ab,%ebx
804836c:	83 bb 28 00 00 00 00	cmpl $0x0,0x28(%ebx)

- 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

```
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));
```

Result (Linux):

```
int a = 15213;
0x11ffffcb8   0x6d
0x11ffffcb9   0x3b
0x11ffffcba   0x00
0x11ffffcbb   0x00
```

Printf directives:

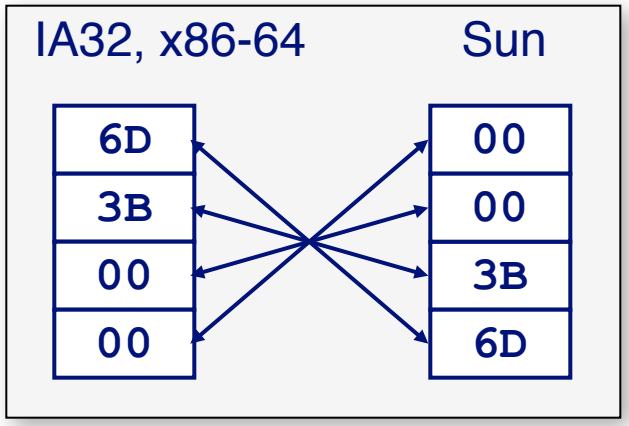
%p: Print pointer

%x: Print Hexadecimal

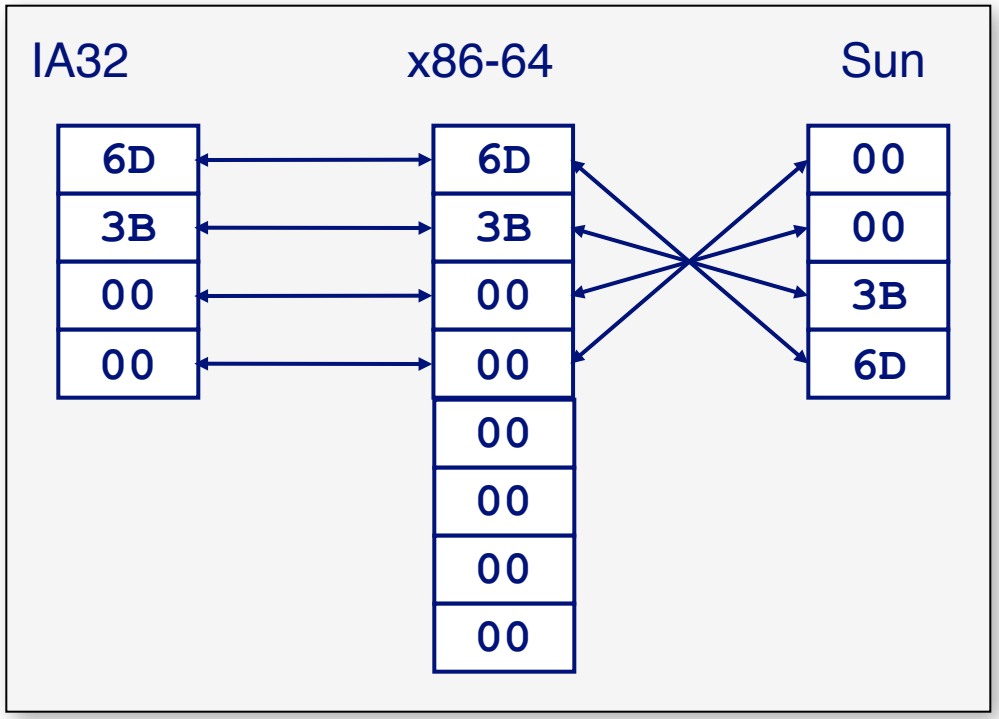
Representing Integers

Decimal:	15213
Binary:	0011 1011 0110 1101
Hex:	3 B 6 D

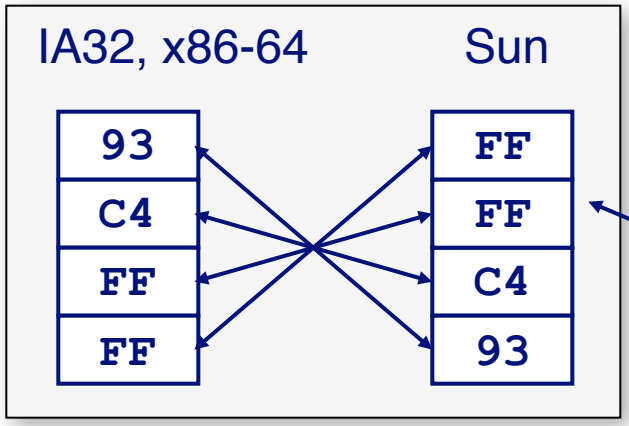
```
int A = 15213;
```




```
long int C = 15213;
```



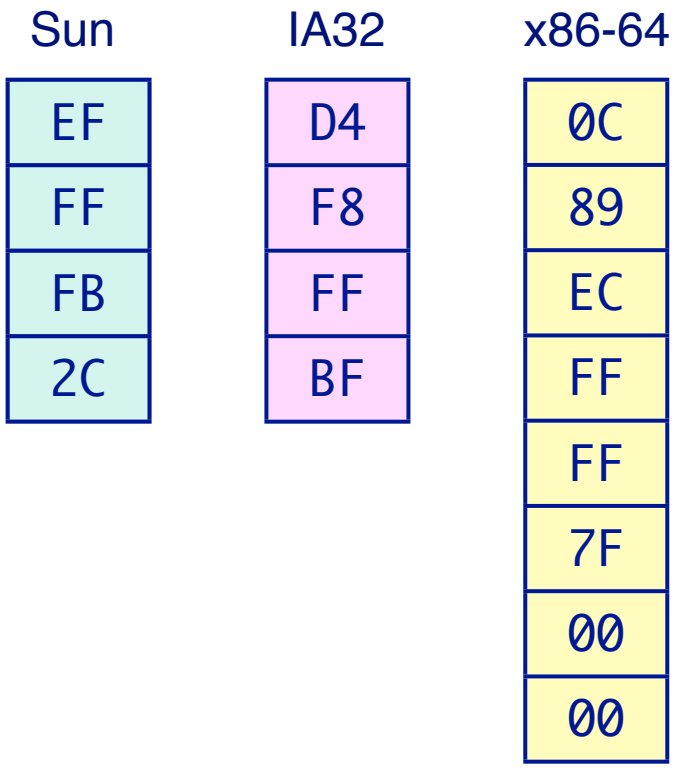

```
int B = -15213;
```



Two's complement representation
(Covered later)

Representing Pointers

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



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

&	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

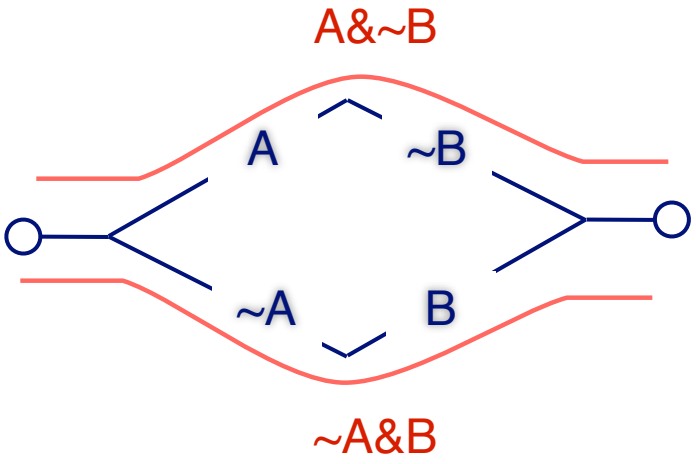
Exclusive-Or (Xor)

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

^	0	1
0	0	1
1	1	0

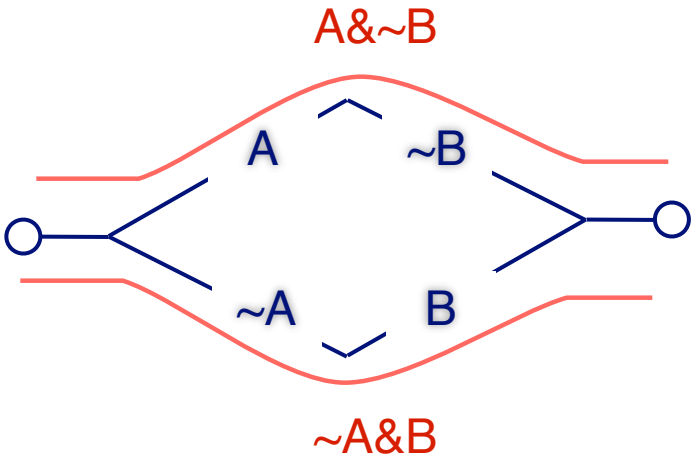
Application of Boolean Algebra

- 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



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

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

- All of the Properties of Boolean Algebra Apply

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

Representing & Manipulating Sets

- Representation
  - Width w bit vector represents subsets of {0, …, w–1}
  - aj = 1 if j ∈ A

```
            01010101   { 0, 2, 4, 6 }
            76543210

            01101001   { 0, 3, 5, 6 }
            76543210
```

```
Operations On Sets:

& 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

PENNSTATE 1855

- 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`
    - $\sim 01000001_2 \rightarrow 10111110_2$
  - `~0x00 → 0xFF`
    - $\sim 00000000_2 \rightarrow 11111111_2$
  - `0x69 & 0x55 → 0x41`
    - $01101001_2 \;\&\; 01010101_2 \rightarrow 01000001_2$
  - `0x69 | 0x55 → 0x7D`
    - $01101001_2 \mid 01010101_2 \rightarrow 01111101_2$

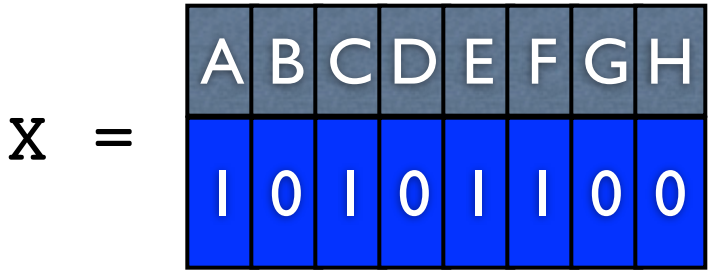
Contrast: Logic Operations in C

PENNSTATE 1855

- 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)

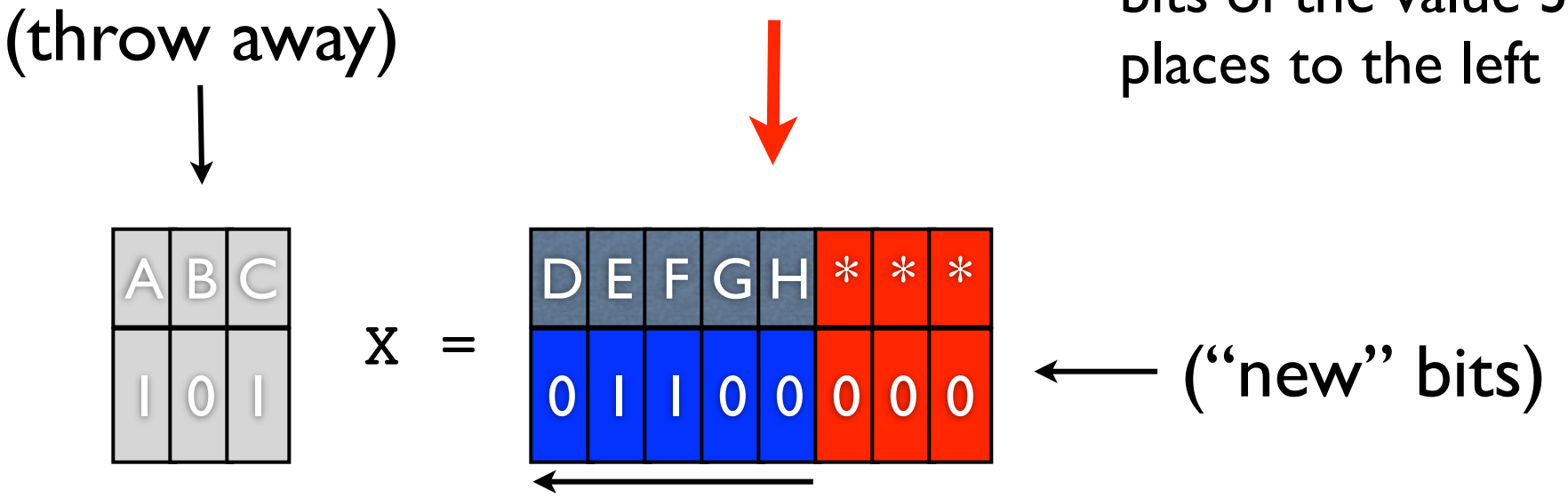
Shift Operations

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




```
X = X << 3;
```

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



Types of Shift

- 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 right
- Undefined Behavior
  - Shift amount < 0 or ≥ word size

Argument **x**	`01100010`
`<< 3`	
Log. `>> 2`	
Arith. `>> 2`	

Argument **x**	`10100010`
`<< 3`	
Log. `>> 2`	
Arith. `>> 2`	

Types of Shift

- 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 right
- Undefined Behavior
  - Shift amount < 0 or ≥ word size

Argument **x**	01100010
<< 3	00010
Log. >> 2	
Arith. >> 2	

Argument **x**	10100010
<< 3	
Log. >> 2	
Arith. >> 2	

Types of Shift

- 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 right
- Undefined Behavior
  - Shift amount < 0 or ≥ word size

Argument x	01100010
<< 3	00010000
Log. >> 2	
Arith. >> 2	

Argument x	10100010
<< 3	
Log. >> 2	
Arith. >> 2	

Types of Shift

- 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 right
- Undefined Behavior
  - Shift amount < 0 or ≥ word size

Argument x	01100010
<< 3	00010000
Log. >> 2	011000
Arith. >> 2	

Argument x	10100010
<< 3	
Log. >> 2	
Arith. >> 2	

Types of Shift

- 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 right
- Undefined Behavior
  - Shift amount < 0 or ≥ word size

Argument x	01100010
<< 3	00010000
Log. >> 2	00011000
Arith. >> 2	

Argument x	10100010
<< 3	
Log. >> 2	
Arith. >> 2	

Types of Shift

- 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 right
- Undefined Behavior
  - Shift amount < 0 or ≥ word size

Argument x	01100010
<< 3	00010000
Log. >> 2	00011000
Arith. >> 2	00011000

Argument x	10100010
<< 3	
Log. >> 2	
Arith. >> 2	

Types of Shift

- 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 right
- Undefined Behavior
  - Shift amount < 0 or ≥ word size

Argument x	01100010
<< 3	00010000
Log. >> 2	00011000
Arith. >> 2	00011000

Argument x	10100010
<< 3	
Log. >> 2	
Arith. >> 2	

Types of Shift

- 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 right
- Undefined Behavior
  - Shift amount < 0 or ≥ word size

Argument x	01100010
<< 3	00010000
Log. >> 2	00011000
Arith. >> 2	00011000

Argument x	10100010
<< 3	00010
Log. >> 2	
Arith. >> 2	

Types of Shift

- 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 right
- 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	
Arith. >> 2	

Types of Shift

- 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 right
- 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	101000
Arith. >> 2	

Types of Shift

- 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 right
- 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	

Types of Shift

- 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 right
- 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	101000

Types of Shift

- 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 right
- 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

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

Bits	31-24	23-16	8-15	0-7
Values	d	c	b	a

```
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) );
```

```
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) );
```

```
A: 0x00000011
B: 0x00002200
C: 0x00330000
D: 0x44000000
Packed bytes : 0x44332211
```