Computer Architecture and Systems Programming
(252-0061-00)

Timothy Roscoe
Herbstsemester 2012
You will have seen...

- **Courses:**
  - Programming & software engineering
  - Parallel programming
  - Data structures and algorithms
- **Languages:** Eiffel, Java, C#, ...
- **Constructs and techniques:**
  - Object-orientation
  - Design-by-contract
  - Locks, atomicity, OpenMP, etc.
This course covers in depth...

• Programming in C
  – *Still* the systems programming language of choice
• Programming in Assembly Language
  – What the machine understands
• What *really* happens at the bit-level
  – Machine instructions
  – Memory systems
  – I/O devices
• Basic elements of processor design
• What makes things go fast (and slow)
Course Components

• Lectures
  – Higher level concepts and ideas

• Recitations
  – Applied concepts, important tools and skills for labs, clarification of lectures, exam coverage, C tutorial

• Lab exercises
  – The heart of the course
  – 1 week each (sometimes 2 weeks)
  – Provide in-depth understanding of aspects of systems
  – Programming and measurement

• Exam (100% of grade)
  – Test your understanding of concepts
Language

• I’ll teach in English (and C...)
  – If I speak too fast, or say something unclear, raise your hand!
  – Please ask questions!

• Assistants’ groups are 5 x German, 2 x English
  – So far...

• Examination:
  – Paper will be in English
  – Answers may be in German or English
Logistics

• Lectures here in CAB G.61
  – Tuesday, Wednesday 10:00 – 12:00

• Recitations – very important!
  – Thursday 13:00 – 15:00, IFW various rooms
  – Learn C, simulator, tools
  – Briefings for Lab exercises
  – Knowledge needed for exams, but not in the lectures!

• There **will** be a session this Thursday
  – Sign up sheets at the front of the room today
  – Check the course web page on Wednesday
More logistics

• Web site:
  http://www.systems.ethz.ch/courses/fall2012/SPCA
  – Lecture notes should appear in advance on web site
  – The notes are not intended to be understood without lectures...

• Procedure for answering additional questions:
  1. Ask your friends
  2. Check the web
  3. Ask your teaching assistant
  4. Ask another teaching assistant
  5. Email me (troscoe@inf.ethz.ch)
Acknowledgements

• Course based on CS 15-213 at Carnegie Mellon University
  – Lots of material gratefully borrowed from CMU

• New material: multicore, devices, etc.
  – All my fault 😊
Textbooks

• Randal E. Bryant and David R. O’Hallaron,
  – http://csapp.cs.cmu.edu

• Brian Kernighan and Dennis Ritchie,
Motivation

• Most CS courses emphasize abstraction
  – Abstract data types (objects, contracts, etc.)
  – Asymptotic analysis (worst-case, complexity)
• These abstractions have limitations
  – Often don’t survive contact with reality
  – Especially in the presence of bugs
  – Need to understand details of underlying implementations
Goals

• Become more effective programmers
  – Find and eliminate bugs efficiently
  – Understand and tune for program performance

• Prepare for later *systems* classes at ETHZ
  – Compilers, Operating Systems, Networks, Computer Architecture, Embedded Systems
“Systems” as a field

• Encompasses:
  – Operating systems
  – Database systems
  – Networking protocols and routing
  – Compiler design and implementation
  – Distributed systems
  – Cloud computing & online services

• On and above hardware/software boundary
You are here:

Computational Science
Visual Computing
Computer Security
Etc.

*Systems topics*

---

Software

---

Hardware

---

Processor design
Digital design, electrical engineering

---

Systems Programming and Computer Architecture
Lecture 1:
Introduction, Bits and Bytes
Computer Architecture and Systems Programming
(252-0061-00)

Timothy Roscoe
Herbstsemester 2012
Motivation: 5 realities
Reality #1:

int’s are not integers.
float’s are not real numbers.
ints are not integers, floats are not reals

• Is $x^2 \geq 0$?
  – floats: Yes!
  – ints:
    • $40000 \times 40000 \rightarrow 16000000000$
    • $50000 \times 50000 \rightarrow ??$

• Is $(x + y) + z = x + (y + z)$?
  – unsigned & signed ints: Yes!
  – floats:
    • $(1e20 + -1e20) + 3.14 \rightarrow 3.14$
    • $1e20 + (-1e20 + 3.14) \rightarrow ??$

http://xkcd.com/571
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}

• Similar to code found in a version of FreeBSD’s implementation of getpeername
• There are legions of smart people trying to find vulnerabilities in programs
Typical usage

/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}

#define MSIZE 528

void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, MSIZE);
    printf("%s\n", mybuf);
}
Malicious usage

```c
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}

#define MSIZE 528

void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, -MSIZE);
    
    . . .
}
```
Computer arithmetic

- Does not generate random values
  - Arithmetic operations have important mathematical properties

- Cannot assume all “usual” mathematical properties
  - Due to finiteness of representations
  - Integer operations satisfy “ring” properties
    - Commutativity, associativity, distributivity
  - Floating point operations satisfy “ordering” properties
    - Monotonicity, values of signs

- Observation
  - Need to understand which abstractions apply in which contexts
  - Important issues for compiler writers and serious application programmers
Reality #2:

You’ve Got to Know Assembly
You’ve got to know assembly

• Chances are, you’ll never write program in assembly
  – Compilers are much better & more patient than you are
• But: understanding assembly is key to machine-level execution model
  – Behavior of programs in presence of bugs
    • High-level language model breaks down
  – Tuning program performance
    • Understand optimizations done/not done by the compiler
    • Understanding sources of program inefficiency
  – Implementing system software
    • Compiler has machine code as target
    • Operating systems must manage process state
• Creating / fighting malware
  • x86 assembly is the language of choice!
Assembly code example

• Time Stamp Counter
  – Special 64-bit register in Intel-compatible machines
  – Incremented every clock cycle
  – Read with \texttt{rdtsc} instruction

• Application
  – Measure time (in clock cycles) required by procedure

```c
double t;
start_counter();
P();
t = get_counter();
printf("P required %f clock cycles\n", t);
```
Code to read counter

- Write small amount of assembly code using GCC’s asm facility
- Inserts assembly code into machine code generated by compiler

```c
static unsigned cyc_hi = 0;
static unsigned cyc_lo = 0;

/* Set *hi and *lo to the high and low order bits
   of the cycle counter. */
void access_counter(unsigned *hi, unsigned *lo)
{
    asm("rdtsc; movl %%edx,%0; movl %%eax,%1"
         : "=r" (*hi), "=r" (*lo)
         : "%edx", "%eax");
}
```
Reality #3:

Memory matters.
RAM is an unrealistic abstraction.
Memory matters

• Memory is **not unbounded**
  – It must be allocated and managed
  – Many applications are memory-dominated

• Memory referencing bugs especially pernicious
  – Effects are distant in both time and space

• Memory performance is **not uniform**
  – Cache and virtual memory effects can greatly affect program performance
  – Adapting program to characteristics of memory system can lead to major speed improvements
Memory referencing bug

double fun(int i)
{
    volatile double d[1] = {3.14};
    volatile long int a[2];
    a[i] = 1073741824; /* Possibly out of bounds */
    return d[0];
}

fun(0)  ->  3.14
fun(1)  ->  3.14
fun(2)  ->  3.1399998664856
fun(3)  ->  2.00000061035156
fun(4)  ->  3.14, then segmentation fault

Explanation:

<table>
<thead>
<tr>
<th>Saved State</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>d7 … d4</td>
<td>3</td>
</tr>
<tr>
<td>d3 … d0</td>
<td>2</td>
</tr>
<tr>
<td>a[1]</td>
<td>1</td>
</tr>
<tr>
<td>a[0]</td>
<td>0</td>
</tr>
</tbody>
</table>

Location accessed by fun(i)
Memory referencing errors

• C and C++ do not provide any memory protection
  – Out of bounds array references
  – Invalid pointer values
  – Abuses of malloc/free

• Can lead to nasty bugs
  – Whether or not bug has any effect depends on system and compiler
  – Action at a distance
    • Corrupted object logically unrelated to one being accessed
    • Effect of bug may be first observed long after it is generated

• How can I deal with this?
  – Program in C#, or ML, or Scala, or Haskell, or...
  – Understand what possible interactions may occur
  – Use or develop tools to detect referencing errors
Memory system performance

- Hierarchical memory organization
- Performance depends on access patterns
  - Including how step through multi-dimensional array

```c
void copyji(int src[2048][2048],
            int dst[2048][2048])
{
    int i,j;
    for (i = 0; i < 2048; i++)
        for (j = 0; j < 2048; j++)
            dst[i][j] = src[i][j];
}

void copyij(int src[2048][2048],
            int dst[2048][2048])
{
    int i,j;
    for (j = 0; j < 2048; j++)
        for (i = 0; i < 2048; i++)
            dst[i][j] = src[i][j];
}
```

21 times slower
(Pentium 4)
The Memory Mountain

Read throughput (MB/s)

Stride (words) Working set size (bytes)

Pentium III Xeon
550 MHz
16 KB on-chip L1 d-cache
16 KB on-chip L1 i-cache
512 KB off-chip unified L2 cache

This processor is a little old, but it’s the same on new ones!
Reality #4:

There’s much more to performance than asymptotic complexity
There’s much more to performance than asymptotic complexity

- **Constant factors** matter too!
- Even exact op count does not predict performance
  - Easily see 10:1 performance range depending on how code written
  - Must optimize at multiple levels: algorithm, data representations, procedures, and loops
- Must understand **system** to optimize performance
  - How programs compiled and executed
  - How to measure program performance and identify bottlenecks
  - How to improve performance without destroying code modularity and generality
Example: matrix multiplication

Matrix-Matrix Multiplication (MMM) on 2 x Core 2 Duo 3 GHz (double precision)

Gflop/s

- Standard desktop computer, vendor compiler, using optimization flags
- Both implementations have **exactly** the same operations count ($2n^3$)
- What is going on?

Best known code (K. Goto)

Triple loop

160x
Matrix-Matrix Multiplication (MMM) on 2 x Core 2 Duo 3 GHz

Gflop/s

- Memory hierarchy and other optimizations: 20x
- Vector instructions: 4x
- Multiple threads: 4x

- Reason for 20x: Blocking or tiling, loop unrolling, array scalarization, instruction scheduling, search to find best choice
- Effect: less register spills, less L1/L2 cache misses, less TLB misses
Reality #5:

Computers don’t just execute programs
Computers don’t just run programs

• They need to get data in and out
  – I/O system critical to program reliability and performance

• They communicate with each other over networks
  – Many system-level issues arise in presence of network
    • Concurrent operations by autonomous processes
    • Coping with unreliable media
    • Cross platform compatibility
    • Complex performance issues
To start:
Bits, Bytes and Integers
Bits, Bytes, and Integers

• Topics
  – Representing information as bits
  – Bit-level manipulations
    • Boolean algebra
    • Expressing in C
  – Representations of Integers
    • Basic properties and operations
    • Implications for C
Binary representations

• Base 2 number representation
  – Represent $15213_{10}$ as $11101101101101_2$
  – Represent $1.20_{10}$ as $1.0011001100110011[0011]..._2$
  – Represent $1.5213 \times 10^4$ as $1.1101101101101_2 \times 2^{13}$

• Electronic implementation
  – Easy to store with bistable elements
  – Reliably transmitted on noisy and inaccurate wires
Encoding byte values

• **Byte = 8 bits**
  
  – Binary: $00000000_2$ to $11111111_2$
  
  – Decimal: $0_{10}$ to $255_{10}$
    
    • First digit must not be 0 in C
  
  – 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 $0xFA1D37B$
      
      – Or $0xfa1d37b$

<table>
<thead>
<tr>
<th>Hex</th>
<th>Dec</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0010</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0011</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>0100</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>0110</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>0111</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>1000</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>1001</td>
</tr>
<tr>
<td>A</td>
<td>10</td>
<td>1010</td>
</tr>
<tr>
<td>B</td>
<td>11</td>
<td>1011</td>
</tr>
<tr>
<td>C</td>
<td>12</td>
<td>1100</td>
</tr>
<tr>
<td>D</td>
<td>13</td>
<td>1101</td>
</tr>
<tr>
<td>E</td>
<td>14</td>
<td>1110</td>
</tr>
<tr>
<td>F</td>
<td>15</td>
<td>1111</td>
</tr>
</tbody>
</table>
Byte-oriented memory organization

• Programs refer to *virtual addresses*
  – Conceptually, a very large array of bytes
  – Secretly, a hierarchy of different memory types
  – System provides *address space* private to particular “process”
    • Program being executed
    • Program can clobber its own data, but not that of others

• Compiler + runtime system control allocation
  – Where different program objects should be stored
  – All allocation within single virtual address space
Machine words

• Machines have a “word size”
  – Nominal size of integer-valued data
    • Including addresses
  – Many current machines use 32-bit (4 byte) words
    • Limits addresses to 4GB
    • Becoming too small for memory-intensive applications
  – Modern machines often use 64-bit (8 byte) words
    • Potential address space ~ $1.8 \times 10^{19}$ bytes
    • x86-64 machines support 48-bit addresses: 256 Terabytes

• Machines support multiple data formats
  • Fractions or multiples of word size
  • Always integral number of bytes
Word-oriented memory

- Addresses specify byte locations
  - Address of first byte in word
  - Addresses of successive words differ by 4 (32-bit) or 8 (64-bit)
## Data representations

- Sizes of C objects (in bytes)

<table>
<thead>
<tr>
<th>C data type</th>
<th>Typical 32-bit</th>
<th>ia32</th>
<th>Intel x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>long long</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>long double</td>
<td>8</td>
<td>10/12</td>
<td>10/16</td>
</tr>
<tr>
<td>char *</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>
Byte ordering

- How should bytes within multi-byte word be ordered in memory?
  - Big Endian: Sun, PPC, Internet
    • Least significant byte has highest address
  - Little Endian: x86
    • Least significant byte has lowest address

- Origin: “Gullivers Reisen” (Gulliver’s Travels)

- Which end to crack a soft-boiled egg?

Egg in “little endian” configuration (Wikipedia)
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>Little Endian</td>
<td>0x100</td>
<td>0x101</td>
<td>0x102</td>
<td>0x103</td>
</tr>
</tbody>
</table>
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 \( 0x100 \)

---

**Big Endian**

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

**Little Endian**

<table>
<thead>
<tr>
<th>( 0x100 )</th>
<th>( 0x101 )</th>
<th>( 0x102 )</th>
<th>( 0x103 )</th>
</tr>
</thead>
</table>
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 4 bytes: 0x000012ab
  Split into bytes: 00 00 12 ab
  Reverse (endian): 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("0x%p\t0x%.2x\n", start+i, start[i]);
    }
}
```

`printf` directives:
- `%p`: print pointer
- `%x`: print hexadecimal
show_bytes example

```c
int a = 15213;
printf("int a = 15213;\n");
show_bytes((pointer) &a, sizeof(int));
```

Result using Linux on Intel x86:

```c
int a = 15213;
0x11fffffcb8  0x6d
0x11fffffcb9  0x3b
0x11fffffcca  0x00
0x11fffffccc  0x00
```
Representing integers

\[
\begin{align*}
\text{int } A &= 15213; \\
\text{int } B &= -15213; \\
\text{long int } C &= 15213;
\end{align*}
\]

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

Two’s complement representation (covered later)
Representing pointers

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

Different compilers & machines assign different locations to objects
Representing strings

• Strings in C
  – Represented by array of characters
  – Each character encoded in ASCII format
    • Standard 7-bit encoding of character set
      • Character “0” has code 0x30
        – Digit i has code 0x30+i
    – String should be null-terminated
      • Final character = 0
  
• Compatibility
  – Byte ordering not an issue

char *S[6] = "15213";
**Boolean Algebra**

- **Developed by George Boole in 19th Century**
  - Algebraic representation of logic
    - Encode “True” as 1 and “False” as 0

**AND:**
\[ A \& B = 1 \text{ when } A=1 \text{ and } B=1 \]

<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:**
\[ A \mid B = 1 \text{ when } A=1 \text{ or } B=1 \]

<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:**
\[ \sim A = 1 \text{ when } A=0 \]

<table>
<thead>
<tr>
<th>~</th>
<th></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):**
\[ A ^ B = 1 \text{ when } A=1 \text{ or } B=1 \text{ but not both} \]

<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

A ∧ ¬B
¬A ∧ B

Connection when
A ∧ ¬B ∨ ¬A ∧ B = A^B
General boolean algebras

• Operate on bit vectors
  – Operations applied bitwise

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

• All of the properties of boolean algebra apply
Representing & manipulating sets

• Width \( w \) bit vector represents subsets of \( \{0, \ldots, w-1\} \)

• \( a_j = 1 \) if \( j \in A \):

  01101001  \( \{0, 3, 5, 6\} \)
  76543210

  01010101  \( \{0, 2, 4, 6\} \)
  76543210

• Operations:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Result</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>Intersection</td>
<td>01000001</td>
<td>( {0, 6} )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Union</td>
<td>01111101</td>
</tr>
<tr>
<td>^</td>
<td>Symmetric difference</td>
<td>00111100</td>
<td>( {2, 3, 4, 5} )</td>
</tr>
<tr>
<td>~</td>
<td>Complement</td>
<td>10101010</td>
<td>( {1, 3, 5, 7} )</td>
</tr>
</tbody>
</table>
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):
  \[ \sim0x41 \rightarrow 0xBE \]
  \[ \sim01000001_2 \rightarrow 1011110_2 \]
  \[ \sim0x00 \rightarrow 0xFF \]
  \[ \sim00000000_2 \rightarrow 11111111_2 \]
  \[ 0x69 \& 0x55 \rightarrow 0x41 \]
  \[ 01101001_2 \& 01010101_2 \rightarrow 01000001_2 \]
  \[ 0x69 \mid 0x55 \rightarrow 0x7D \]
  \[ 01101001_2 \mid 01010101_2 \rightarrow 01111111_2 \]
Contrast: Logic operations in C

• `&&`, `||`, `!`
  – 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

• **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 \( \geq \) word size

<table>
<thead>
<tr>
<th>Argument x</th>
<th>01100010</th>
</tr>
</thead>
<tbody>
<tr>
<td>( &lt;&lt; 3 )</td>
<td>00010000</td>
</tr>
<tr>
<td>Log. &gt;&gt; 2</td>
<td>00011000</td>
</tr>
<tr>
<td>Arith. &gt;&gt; 2</td>
<td>00011000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Argument x</th>
<th>10100010</th>
</tr>
</thead>
<tbody>
<tr>
<td>( &lt;&lt; 3 )</td>
<td>00010000</td>
</tr>
<tr>
<td>Log. &gt;&gt; 2</td>
<td>00101000</td>
</tr>
<tr>
<td>Arith. &gt;&gt; 2</td>
<td>11101000</td>
</tr>
</tbody>
</table>

Java writes this “\( \ggg \)”. 
Integer C puzzles

- Assume 32-bit word size, two’s complement integers
- For each of the following C expressions, either:
  - Argue that is true for all argument values
  - Give example where not true
- \( x < 0 \Rightarrow (x \times 2) < 0 \)
- \( ux \geq 0 \)
- \( x \& 7 == 7 \Rightarrow (x \ll 30) < 0 \)
- \( ux > -1 \)
- \( x > y \Rightarrow -x < -y \)
- \( x \times x \geq 0 \)
- \( x > 0 \&\& y > 0 \Rightarrow x + y > 0 \)
- \( x \geq 0 \Rightarrow -x \leq 0 \)
- \( x \leq 0 \Rightarrow -x \geq 0 \)
- \( (x | -x) \gg 31 == -1 \)
- \( ux \gg 3 == ux/8 \)
- \( x \gg 3 == x/8 \)
- \( x \& (x-1) \neq 0 \)

Initialization

```c
int x = foo();
int y = bar();
unsigned ux = x;
unsigned uy = y;
```
Next time: Integers

- Representation: unsigned and signed
- Conversion, casting
- Expanding, truncating
- Addition, negation, multiplication, shifting