CS 61C: Great Ideas in Computer Architecture
Lecture 2: Numbers & C Language
Krste Asanović & Randy Katz
http://inst.eecs.berkeley.edu/~cs61c

Agenda
• Numbers wrap-up
• This is not on the exam!
• Break
• C Primer
• Administrivia, Break
• C Type declarations
• And in Conclusion, ...

Recap: Binary Number Conversion

<table>
<thead>
<tr>
<th>Binary</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001010</td>
<td>7₂₆₁₀</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Binary</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>7₄₆₁₀</td>
<td>1001010</td>
</tr>
</tbody>
</table>

Signed Integer Representation

How to represent -5?
Sign & magnitude (8-bit example):

<table>
<thead>
<tr>
<th>Sign</th>
<th>7-bit magnitude (0...127)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Rules for addition, \( a + b \):
- If \( \text{sign}(a) = \text{sign}(b) \), add \( a \) to \( b \)
- If \( \text{sign}(a) \neq \text{sign}(b) \), subtract \( b \) from \( a \)
- If \( a = 0 \) or \( b = 0 \), are they equal? (If not equal ???)

Cumbersome
- "Complicated" hardware: reduced speed / increased power
- Is there a better way?

Going backwards by going forwards

- How to add -1 to a four-bit number?
  - \( 0111₂₆₁₀ + 0001₂₆₁₀ = 1000₂₆₁₀ \)
  - \( 0111₂₆₁₀ - 0001₂₆₁₀ = 0110₂₆₁₀ \)

Going backwards by going forwards

- Add 10000₂₆₁₀ and throw away fifth bit, result unchanged
  - Result wraps around to same value
  - \( 0111₂₆₁₀ + 10000₂₆₁₀ = 10001₂₆₁₀ \)
  - \( 0111₂₆₁₀ - 10000₂₆₁₀ = 0111₂₆₁₀ \)
Going backwards by going forwards

• Add $10000_{10} + 1_{10} = 01111_{10}$ and throw away fifth bit
  
  $0111_{10} = 7_{10}$  
  $01111_{10} + 16 = 16 - 1_{10}$

Two’s Complement (8-bit example)

<table>
<thead>
<tr>
<th>As Signed/Decimal</th>
<th>As Unsigned/Decimal</th>
<th>Binary Number (8-bit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>-128</td>
<td>-128</td>
<td>10000000</td>
</tr>
<tr>
<td>-127</td>
<td>-127</td>
<td>11111111</td>
</tr>
<tr>
<td>-126</td>
<td>-126</td>
<td>10000011</td>
</tr>
<tr>
<td>-125</td>
<td>-125</td>
<td>11111101</td>
</tr>
<tr>
<td>-124</td>
<td>-124</td>
<td>10000000</td>
</tr>
<tr>
<td>-123</td>
<td>-123</td>
<td>11111100</td>
</tr>
<tr>
<td>-122</td>
<td>-122</td>
<td>10000011</td>
</tr>
<tr>
<td>-121</td>
<td>-121</td>
<td>11111100</td>
</tr>
</tbody>
</table>

Most-significant bit (MSB) equals sign when interpreted as a signed number

4-bit Example

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0111</td>
</tr>
<tr>
<td>4</td>
<td>0110</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
</tr>
</tbody>
</table>

- Map negative $\rightarrow$ positive numbers
  
  - Example for 4-bit: $-3 \rightarrow 2^2 - 3 = 13$
  - "Two’s complement"
  - No special rules for adding positive and negative numbers

Signed versus Unsigned Binary Numbers

- Both are stored as a collection of bits
- The same set of bits can be used to represent a signed number or an unsigned number (or a string, a picture, or a...)
- It’s up to the operation interpreting the bits
  
  $1101_{10} = +13_{10}$ (as an unsigned number)
  $1101_{10} = -3_{10}$ (as a signed number)
  $1101_{10} = \text{orange}$ (as a color)
  $1101_{10} = \text{cat}$ (as a type of animal)
  $1101_{10} = \text{whatever you want it to mean}...$

Unary Negation (Two’s Complement)

4-bit Example (-8_{10} \rightarrow +7_{10})

\[
\begin{align*}
16_{\text{10}} & \rightarrow \ 00000000_{\text{2}} & 15_{\text{10}} & \rightarrow \ 00111111_{\text{2}} \\
15_{\text{10}} & \rightarrow \ 00001111_{\text{2}} & 15_{\text{10}} & \rightarrow \ 00111111_{\text{2}} \quad \text{(invert)} \\
14_{\text{10}} & \rightarrow \ 00001110_{\text{2}} & 14_{\text{10}} & \rightarrow \ 00111110_{\text{2}} \\
13_{\text{10}} & \rightarrow \ 00001101_{\text{2}} & 13_{\text{10}} & \rightarrow \ 00111101_{\text{2}} \\
\end{align*}
\]

Signed versus Unsigned Binary Numbers

- Both are stored as a collection of bits
- The same set of bits can be used to represent a signed number or an unsigned number (or a string, a picture, or a...)
- It’s up to the operation interpreting the bits

Signed versus Unsigned Binary Numbers

- Both are stored as a collection of bits
- The same set of bits can be used to represent a signed number or an unsigned number (or a string, a picture, or a...)
- It’s up to the operation interpreting the bits

Signed versus Unsigned Binary Numbers

- Both are stored as a collection of bits
- The same set of bits can be used to represent a signed number or an unsigned number (or a string, a picture, or a...)
- It’s up to the operation interpreting the bits

Your Turn

- What is the decimal value of the following binary 8-bit 2’s complement number?
  
  $1110\ 0001_{10}$

\[
\begin{array}{|c|c|}
\hline
\text{Answer} & \text{Value} \\
\hline
\text{RED} & 33_{10} \\
\text{ORANGE} & -31_{10} \\
\text{GREEN} & 225_{10} \\
\text{YELLOW} & -33_{10} \\
\hline
\end{array}
\]
Addition
4-bit Example

<table>
<thead>
<tr>
<th>Unsigned</th>
<th>Signed (Two’s Complement)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3\text{\textsubscript{16}} + 4\text{\textsubscript{16}} + 0100\text{\textsubscript{16}}</td>
<td>3\text{\textsubscript{16}} + 4\text{\textsubscript{16}} + 0100\text{\textsubscript{16}}</td>
</tr>
<tr>
<td>7\text{\textsubscript{16}}</td>
<td>7\text{\textsubscript{16}}</td>
</tr>
<tr>
<td>0111\text{\textsubscript{16}}</td>
<td>0111\text{\textsubscript{16}}</td>
</tr>
<tr>
<td>3\text{\textsubscript{16}} + 1011\text{\textsubscript{16}} + 1110\text{\textsubscript{16}}</td>
<td>3\text{\textsubscript{16}} + 1011\text{\textsubscript{16}} + 1110\text{\textsubscript{16}}</td>
</tr>
<tr>
<td>14\text{\textsubscript{16}}</td>
<td>1110\text{\textsubscript{16}}</td>
</tr>
</tbody>
</table>

No special rules for two’s complement signed addition

Overflow
4-bit Example

<table>
<thead>
<tr>
<th>Unsigned addtion</th>
<th>Signed addition (Two’s Complement)</th>
</tr>
</thead>
<tbody>
<tr>
<td>13\text{\textsubscript{16}} + 14\text{\textsubscript{16}} + 1110\text{\textsubscript{16}}</td>
<td>3\text{\textsubscript{16}} + 1110\text{\textsubscript{16}}</td>
</tr>
<tr>
<td>27\text{\textsubscript{16}}</td>
<td>0111\text{\textsubscript{16}}</td>
</tr>
<tr>
<td>1011\text{\textsubscript{16}}</td>
<td>1111\text{\textsubscript{16}}</td>
</tr>
<tr>
<td>7\text{\textsubscript{16}}</td>
<td>7\text{\textsubscript{16}}</td>
</tr>
<tr>
<td>0111\text{\textsubscript{16}}</td>
<td>0111\text{\textsubscript{16}}</td>
</tr>
<tr>
<td>27\text{\textsubscript{16}}</td>
<td>0111\text{\textsubscript{16}}</td>
</tr>
<tr>
<td>1011\text{\textsubscript{16}}</td>
<td>1111\text{\textsubscript{16}}</td>
</tr>
</tbody>
</table>

Addition Overflow Detection
4-bit Example

Unsigned addition
- Carry-out indicates overflow

Signed addition (Two’s Complement)
- Overflow if
  - Signs of operands are equal
  - AND
  - Sign of result differs from sign of operands
- No overflow is possible when signs of operands differ

Overflow rules depend on the operation (signed vs unsigned addition)

Adding Numbers of Different Bit Widths

Unsigned addition

Signed extension

<table>
<thead>
<tr>
<th>Decimal</th>
<th>4-bit</th>
<th>8-bit</th>
<th>32-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1101\text{\textsubscript{16}}</td>
<td>0011\text{\textsubscript{16}}</td>
<td>0000 0011\text{\textsubscript{16}}</td>
<td>0000 0000 0000 0011\text{\textsubscript{16}}</td>
</tr>
<tr>
<td>101000101\text{\textsubscript{16}}</td>
<td>100010101\text{\textsubscript{16}}</td>
<td>101000101\text{\textsubscript{16}}</td>
<td>101000101\text{\textsubscript{16}}</td>
</tr>
<tr>
<td>77770010</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
</tbody>
</table>

- What should we assume for upper four bits of first operand?
- Treat as zeros!
- “Zero-extension”

Sign Extension

<table>
<thead>
<tr>
<th>4-bit</th>
<th>8-bit</th>
<th>32-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1101\text{\textsubscript{16}}</td>
<td>101000101\text{\textsubscript{16}}</td>
<td>11111111111111111111111111111111</td>
</tr>
<tr>
<td>0011\text{\textsubscript{16}}</td>
<td>100010101\text{\textsubscript{16}}</td>
<td></td>
</tr>
<tr>
<td>1101\text{\textsubscript{16}}</td>
<td>101000101\text{\textsubscript{16}}</td>
<td></td>
</tr>
</tbody>
</table>

- When is this relevant?
- Any time you need a wider version of a narrower operand
  e.g., adding two integers of different widths

- “Sign-extension”
Your Turn

Which range of decimals can be expressed with a 6-bit two’s complement number?

<table>
<thead>
<tr>
<th>Answer</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>RED</td>
<td>-32 ... -32</td>
</tr>
<tr>
<td>GREEN</td>
<td>-64 ... -63</td>
</tr>
<tr>
<td>ORANGE</td>
<td>-31 ... -32</td>
</tr>
<tr>
<td>YELLOW</td>
<td>-32 ... -31</td>
</tr>
</tbody>
</table>

Agenda

- Numbers wrap-up
- This is not on the exam!
- Break
- C Primer
- Administrivia, Break
- C Type declarations
- And in Conclusion, ...

HotChips 2017: Brains & Dinosaurs

- Top conference introducing new processor chips

Z14, IBM Mainframe, 50+ years on

- Micro-architecture
  - 12 cores per CP-chip (5.2GHz)
  - Cache improvements
    - DRAM + 128MB L3
    - 12 larger 32-GB L2s
    - 12 smaller 128-GB L2s
  - New instruction set
    - Pipeline Optimizations
    - Improved instruction delivery
    - Improved instruction decode
    - Enhanced phase decoding
    - Enhanced ILP
    - Enhanced instruction type
  - New compiler
  - New architecture & TBB
- Accelerators
  - Reduced cay intro-processor
  - Improved performance
  - New functions (GEM, T9K, J9K)
  - Enhanced memory performance
  - Improved memory bandwidth
  - Improved memory latency
  - Enhanced memory compression
- Z14 On-Draw and System Topology
  - IBM
  - 24 CP modules (6M transistors)
  - 24 CP modules (6M transistors)
  - 24 CP modules (6M transistors)
  - 24 CP modules (6M transistors)
  - 24 CP modules (6M transistors)
Levels of Representation

<table>
<thead>
<tr>
<th>High Level Language</th>
<th>Medium Level Language</th>
<th>Low Level Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program (e.g., C)</td>
<td>Assembly Program (e.g., RISC-V)</td>
<td>Machine Language Program (RISC-V)</td>
</tr>
<tr>
<td>Instruction</td>
<td>Hardware Architecture Description (e.g., block diagrams)</td>
<td>Logic Circuit Description (Circuit Schematic Diagrams)</td>
</tr>
</tbody>
</table>

Current Focus

Anything can be represented as a number, i.e., data or instructions:

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

Introduction to C

“The Universal Assembly Language”

- Languages used in 61C:
  - C
  - Assembly
  - Python used in two labs
- Prior programming experience helpful, e.g.
  - Java
  - C++

Survey

Who has never written a program in Java

or C, C++, Objective-C?

Hello World

**C**

```c
#include <stdio.h>

int main(void) {
    printf("Hello World!\n");
    return 0;
}
```

**Java**

```java
public class L02_HelloWorld {
    public static void main(String args[]) {
        System.out.println("Hello World!");
    }
}
```
Compilation & Running

```
$ gcc HelloWorld.c
$ ./a.out
Hello World!
```

C Compilation Simplified Overview

```
foo.c  bar.c
    |    |
    v    v
Compiler Compiler
    |
foo.o  bar.o
    |    |
    v    v
Linker Linker
    |
lib.o
    |
a.out
    |
Machine code object files
    |
Pre-built object file libraries
    |
Machine code executable file
```

Compilation versus Interpretation

<table>
<thead>
<tr>
<th>C (compiled)</th>
<th>Python (interpreted)</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Compiler (&amp; linker) translates source into machine language</td>
<td>• Interpreter is written in some high-level language (e.g., C) and translated into machine language</td>
</tr>
<tr>
<td>• Machine language program is loaded by OS and directly executed by the hardware</td>
<td>• Loader by OS and directly executed by processor</td>
</tr>
<tr>
<td></td>
<td>• Interpreter reads source code (e.g., Python) and &quot;interprets&quot; it</td>
</tr>
</tbody>
</table>

Java “Byte Code”

- Java compiler (javac) translates source to “byte code”
- “Byte code” is a particular assembly language
  - Just like i86, RISC-V, ARM, ...
  - Can be directly executed by appropriate machine
    - implementations exist(ed), not commercially successful
  - More typically, “byte code” is
    - interpreted on target machine (e.g., i86) by java program
    - compiled to target machine code (e.g., by JIT)
- Program runs on any computer with a “byte code” interpreter (more or less)

Compilation

- Excellent run-time performance:
  - Much faster than interpreted code (e.g., Python)
  - Usually faster than Java (even with JIT)
- Note: Computers only run machine code
  - Compiled application program, or
  - Interpreter (that interprets source code)

Compilation: Disadvantages

- Compiled files, including the executable, are
  - architecture-specific, depending on processor type
    - e.g., RISC-V vs. ARM
  - and the operating system
    - e.g., Windows vs. Linux
- Executable must be rebuilt on each new system
  - I.e., “porting your code” to a new architecture
- “Change → Compile → Run [repeat]” iteration cycle can be slow during development
  - Recompile only parts of program that have changed
  - Tools (e.g. make) automate this
C Dialects

```c
$ gcc x.c
int main(void) {
    int a[S2]; // declare array
    for (int i=0; i<S2; i++) a[i] = 0;
    return 0;
}
```

```c
$ gcc -ansi -pedantic x.c
#define M_PI (3.14159) /* Define constant */
```

C Pre-Processor (CPP)

```text
foo.c → CPP → foo.i → Compiler

• C source files pass through macro processor, CPP, before compilation
• CPP replaces comments with a single space
• CPP commands begin with "#"
• #include "file.h" /* Inserts file.h */
• #include <stdio.h> /* Loads from standard loc */
• #define M_PI (3.14159) /* Define constant */
• #if/#endif /* Conditional inclusion of text */
• Use -save-temps option to gcc to see result of preprocessing
• Full documentation at: http://gcc.gnu.org/onlinedocs/cpp/
```

Agenda

• Numbers wrap-up
• This is not on the exam!
• Break
• C Primer
• Administrivia, Break
• C Type declarations
• And in Conclusion, ...

Administrativia

• Notify us of all known variances
  - DSP special accommodations: Email head TA (Steven Ho)
  - Exam conflicts: Fill out the form on Piazza
    (special accommodations can be made only in exceptional cases)
  - If you cannot get an instructional account, email TA
• Notify head TA by Tuesday, 9/5/2017
  - see http://www-inst.eecs.berkeley.edu/~cs61c/fa17/

Break!
Typed Variables in C

```c
int a = 4;
float f = 1.38e7;
char c = 'x';
```

- Declare before use
- Type cannot change
- Like Java

### Integers: Python vs. Java vs. C

<table>
<thead>
<tr>
<th>Language</th>
<th>sizeof(int)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python</td>
<td>&gt;=32 bits (plain ints), infinite (long ints)</td>
</tr>
<tr>
<td>Java</td>
<td>32 bits</td>
</tr>
<tr>
<td>C</td>
<td>Depends on computer; 16 or 32 or 64</td>
</tr>
</tbody>
</table>

- C: int — integer type that target processor works with most efficiently
- Only guaranteed:
  - sizeof(long long) ≥ sizeof(long) ≥ sizeof(int) ≥ sizeof(short)
  - Also, short >= 16 bits, long >= 32 bits
- All could be 64 bits
- Impacts portability between architectures

### Constants and Enumerations in C

- Constants
  - Assigned in typed declaration, cannot change
  - E.g.
    - `const float pi = 3.1415;`
    - `const unsigned long addr = 0xaf460;`
- Enumerations

```c
typedef enum {red, green, blue} colors;
switch(panels) {
    case red:
        break;
    case green:
        break;
    case blue:
        break;
}
```

### Integers:

- `int number_of_people() { return 3; }`
- `void news() { printf("no news\n"); }`
- `int sum(int x, int y) { return x + y; }`

- Like Java
- Declare return & argument types
- `void` for no value returned
- Functions MUST be declared before they are used
Uninitialized Variables

Code
```c
#include <stdio.h>
#include <stdlib.h>

void undefined_local1() {
  int x; /* undefined */
  printf("x = %d", x);
}

void some_calc(int a) {
  a = a % 7 + rand(); // a;
}

int main(void) {
  for (int i = 0; i < 5; i++) {
    some_calc(i + 1);
    undefined_local1();
  }
}
```

Output
```
$ gcc test.c
$ ./a.out
x = 0
x = 16807
x = -4
x = 282475249
x = -16
```

Struct’s in C

- Struct’s are structured groups of variables
- A bit like Java classes, but no methods
- E.g. `#include <stdio.h>`

```c
int main(void) {
    typedef struct { int x, y; } Point;
    Point p1;    // Point p1;
    p1.x = 0;    // p1.x = 0;
    p1.y = 123;  // p1.y = 123;
    Point p2 = { 77, -8 };  // Point p2 = { 77, -8 };
    printf("p1.x = %d, p1.y = %d, p2.x = %d, p2.y = %d\n",
            p1.x, p1.y, p2.x, p2.y);  // printf("p1.x = %d, p1.y = %d, p2.x = %d, p2.y = %d\n",
        );
}
```

More C ...

- Lecture does not cover C completely
  - You’ll still need your C reference for this course
  - K&R, The C Programming Language
  - & other references on the course website
- Next few lectures’ focus:
  - Pointers & Arrays
  - Memory management

And In Conclusion,…

- Signed integers represented in 2’s complement
- C Programming Language
  - Popular (still)
  - Similar to Java, but
    - no classes
  - explicit pointers (next lecture)
    - Beware
    - variables not initialized
    - variable size (# of bits) is machine & compiler dependent
- C is compiled to machine code
  - Unlike Python or Java, which are interpreted
  - Compilation is faster than interpretation