The Hardware/Software Interface
CSE351 Spring 2015
Lecture 6

Instructor:
Katelin Bailey

Teaching Assistants:
Today

- Floating-point in C
- Basics of Machine Programming & Architecture
  - What is an ISA (Instruction Set Architecture)?
  - A brief history of Intel processors and architectures
  - C, assembly, machine code
  - x86 basics: registers
Today

- **Floating-point in C**
- **Basics of Machine Programming & Architecture**
  - What is an ISA (Instruction Set Architecture)?
  - A brief history of Intel processors and architectures
  - C, assembly, machine code
  - x86 basics: registers
Floating Point in C

- C offers two levels of precision
  
  ```
  float single precision (32-bit)
  double double precision (64-bit)
  ```

- `#include <math.h>` to get `INFINITY` and `NAN` constants

- Equality (==) comparisons between floating point numbers are tricky, and often return unexpected results
  
  ```
  Just avoid them!
  ```
Floating Point in C

- Conversions between data types:
  - Casting between int, float, double changes the bit representation.
  - int → float
    - May be rounded; overflow not possible
  - int → double or float → double
    - Exact conversion (32-bit ints; 52-bit frac + 1-bit sign)
  - long int → double
    - Rounded or exact, depending on word size
  - double or float → int
    - Truncates fractional part (rounded toward zero)
    - Not defined when out of range or NaN: generally sets to Tmin
Number Representation Really Matters

- **1991: Patriot missile targeting error**
  - clock skew due to conversion from integer to floating point
- **1996: Ariane 5 rocket exploded ($1 billion)**
  - overflow converting 64-bit floating point to 16-bit integer
- **2000: Y2K problem**
  - limited (decimal) representation: overflow, wrap-around
- **2038: Unix epoch rollover**
  - Unix epoch = seconds since 12am, January 1, 1970
  - signed 32-bit integer representation rolls over to Tmin in 2038
- **other related bugs**
  - 1994: Intel Pentium FDIV (floating point division) HW bug ($475 million)
  - 1997: USS Yorktown “smart” warship stranded: divide by zero
  - 1998: Mars Climate Orbiter crashed: unit mismatch ($193 million)
Floating Point and the Programmer

```c
#include <stdio.h>

int main(int argc, char* argv[]) {

    float f1 = 1.0;
    float f2 = 0.0;
    int i;
    for ( i=0; i<10; i++ ) {
        f2 += 1.0/10.0;
    }

    printf("0x%08x  0x%08x
", *(int*)&f1, *(int*)&f2);
    printf("f1 = %10.8f
", f1);
    printf("f2 = %10.8f
\n", f2);

    f1 = 1E30;
    f2 = 1E-30;
    float f3 = f1 + f2;
    printf("f1 == f3? %s\n", f1 == f3 ? "yes" : "no" );

    return 0;
}
```

```
$ ./a.out
0x3f800000  0x3f800001
f1 = 1.000000000
f2 = 1.000000119
f1 == f3? yes
```
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>Location accessed by fun(i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>d7 ... d4</td>
<td>4</td>
</tr>
<tr>
<td>d3 ... d0</td>
<td>3</td>
</tr>
<tr>
<td>a[1]</td>
<td>2</td>
</tr>
<tr>
<td>a[0]</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>
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

Saved State

| d7 ... d4 | 0100 0000 0000 1001 0001 1110 1011 1000 |
| d3 ... d0 | 0101 0000 ...
| a[1]       |
| a[0]       | 0 |

Location accessed by fun(i)
Memory Referencing Bug (Revisited)

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

Saved State

<table>
<thead>
<tr>
<th>Location accessed by fun(i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>d7  ...  d4</td>
</tr>
<tr>
<td>0100 0000 0000 1001 0001 1110 1011 1000</td>
</tr>
<tr>
<td>d3  ...  d0</td>
</tr>
<tr>
<td>0100 0000 0000 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>a[1]</td>
</tr>
<tr>
<td>0100 0000 0000 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>a[0]</td>
</tr>
<tr>
<td>0100 0000 0000 0000 0000 0000 0000 0000</td>
</tr>
</tbody>
</table>
Summary

- As with integers, floats suffer from the fixed number of bits available to represent them
  - Can get overflow/underflow, just like ints
  - Some “simple fractions” have no exact representation (e.g., 0.2)
  - Can also lose precision, unlike ints
    - “Every operation gets a slightly wrong result”

- Mathematically equivalent ways of writing an expression may compute different results
  - Violates associativity/distributivity

- Never test floating point values for equality!
- Careful when converting between ints and floats!
Many more details for the curious...

- Exponent bias
- Denormalized values – to get finer precision near zero
- Distribution of representable values
- Floating point multiplication & addition algorithms
- Rounding strategies

- We won’t be using or testing you on any of these extras in 351.
Chat with your neighbors for a couple minutes:

• What have you learned in the last 6 lectures?
  • Why do you think it was important to learn these things?

• What big questions do you have left with hex, binary, signed/unsigned integers, floats, or pointers?
  • If we don’t get to you: ask your questions on the discussion board so we can answer them for the class
Roadmap

C:

car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);

Java:

Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
c.getMPG();

get_mpg:
    pushq  %rbp
    movq   %rsp, %rbp
    ...
    popq   %rbp
    ret

Machine code:
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111

Computer system:

Memory, data, & addressing
Integers & floats
Machine code & C
x86 assembly
Procedures & stacks
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C
Today

- Floating-point in C
- **Basics of Machine Programming & Architecture**
  - What is an ISA (Instruction Set Architecture)?
  - A brief history of Intel processors and architectures
  - C, assembly, machine code
  - x86 basics: registers
HW/SW Interface: Code/Compile/Run Times

Code Time

User program in C

C compiler

Compile Time

Assembler

Run Time

Hardware

.c file

.exe file
The time required to execute a program depends on:

- **The program** (as written in C, for instance)
- **The compiler**: what set of assembler instructions it translates the C program into
- **The instruction set architecture (ISA)**: what set of instructions it makes available to the compiler
- **The hardware implementation**: how much time it takes to execute an instruction

What should the HW/SW interface contain?
Instruction Set Architectures

• The ISA defines:
  • The system’s state (e.g. registers, memory, program counter)
  • The instructions the CPU can execute
  • The effect that each of these instructions will have on the system state
General ISA Design Decisions

- **Instructions**
  - What instructions are available? What do they do?
  - How are they encoded?

- **Registers**
  - How many registers are there?
  - How wide are they?

- **Memory**
  - How do you specify a memory location?
X86 ISA

• Processors that implement the x86 ISA completely dominate the server, desktop and laptop markets

• Evolutionary design
  • Backwards compatible up until 8086, introduced in 1978
  • Added more features as time goes on

• Complex instruction set computer (CISC)
  • Many different instructions with many different formats
    • But, only small subset encountered with Linux programs
  • (as opposed to Reduced Instruction Set Computers (RISC), which use simpler instructions)
## Intel x86 Evolution: Milestones

<table>
<thead>
<tr>
<th>Name</th>
<th>Date</th>
<th>Transistors</th>
<th>MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>8086</td>
<td>1978</td>
<td>29K</td>
<td>5-10</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>386</td>
<td>1985</td>
<td>275K</td>
<td>16-33</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pentium 4F</td>
<td>2005</td>
<td>230M</td>
<td>2800-3800</td>
</tr>
</tbody>
</table>

- First 16-bit processor: Basis for IBM PC & DOS
- 1MB address space
- First 32 bit processor, referred to as IA32
- Added “flat addressing”
- Capable of running Unix
- 32-bit Linux/gcc targets i386 by default
- First 64-bit Intel x86 processor, referred to as x86-64
## Intel x86 Processors

### Machine Evolution
- **486** 1989 1.9M
- **Pentium** 1993 3.1M
- **Pentium/MMX** 1997 4.5M
- **PentiumPro** 1995 6.5M
- **Pentium III** 1999 8.2M
- **Pentium 4** 2001 42M
- **Core 2 Duo** 2006 291M
- **Core i7** 2008 731M

### Added Features
- Instructions to support multimedia operations
- Parallel operations on 1, 2, and 4-byte data
- Instructions to enable more efficient conditional operations
- More cores!
More information

- References for Intel processor specifications:
  - Intel’s “automated relational knowledgebase”:
    - http://ark.intel.com/
  - Wikipedia:
x86 Clones: Advanced Micro Devices (AMD)

- Same ISA, different implementation
- Historically
  - AMD has followed just behind Intel
  - A little bit slower, a lot cheaper
- Then
  - Recruited top circuit designers from Digital Equipment and other downward trending companies
  - Built Opteron: tough competitor to Pentium 4
  - Developed x86-64, their own extension of x86 to 64 bits
Intel’s Transition to 64-Bit

• Intel attempted radical shift from IA32 to IA64 (2001)
  • Totally different architecture (Itanium) and ISA than x86
  • Executes IA32 code only as legacy (and about 10x as slow!)
  • Performance disappointing

• AMD stepped in with evolutionary solution (2003)
  • x86-64 (also called “AMD64”)

• Intel felt obligated to focus on IA64
  • Hard to admit mistake or that AMD is better

• Intel announces “EM64T” extension to IA32 (2004)
  • Extended Memory 64-bit Technology
  • Almost identical to AMD64!

• Today: all but low-end x86 processors support x86-64
  • But, lots of code out there is still just IA32
Our Coverage in 351

- IA32
  - The traditional 32-bit x86 ISA

- x86-64
  - The new 64-bit x86 ISA - all lab assignments use x86-64!
Definitions

• **Architecture**: (also instruction set architecture or ISA)
  The parts of a processor design that one needs to understand
to write assembly code
  • “What is directly visible to software”
• **Microarchitecture**: Implementation of the architecture
  • CSE 352

• Is cache size “architecture”?
• How about CPU frequency?
• And number of registers?
Assembly Programmer’s View

- **Programmer-Visible State**
  - PC: Program counter
    - Address of next instruction
    - Called “EIP” (IA32) or “RIP” (x86-64)
  - Register file
    - Heavily used program data
  - Condition codes
    - Store status information about most recent arithmetic operation
    - Used for conditional branching

- **Memory**
  - Byte addressable array
  - Code, user data, (some) OS data
  - Includes stack used to support procedures (we’ll come back to that)
Turning C into Object Code

- Code in files p1.c p2.c
- Compile with command: `gcc -O1 p1.c p2.c -o p`
  - Use basic optimizations (-O1)
  - Put resulting machine code in file p

- C program (p1.c p2.c) → Compiler (gcc -S)
- Asm program (p1.s p2.s) → Assembler (gcc or as)
- Object program (p1.o p2.o) → Linker (gcc or ld)
- Executable program (p) → Static libraries (.a)
Compiling Into Assembly

```c
int sum(int x, int y) {
    int t = x+y;
    return t;
}
```

Generated IA32 Assembly

```assembly
sum:
    pushl %ebp
    movl %esp,%ebp
    movl 12(%ebp),%eax
    addl 8(%ebp),%eax
    movl %ebp,%esp
    popl %ebp
    ret
```

Obtain with command

```
gcc -O1 -S code.c
```

Produces file code.s
Compiling Into Assembly

```c
int sum(int x, int y)
{
    int t = x+y;
    return t;
}
```

Generated IA32 Assembly

```
sum:
pushl %ebp
movl %esp,%ebp
movl 12(%ebp),%eax
addl 8(%ebp),%eax
movl %ebp,%esp
popl %ebp
ret
```

Obtain with command

```
gcc -O1 -S code.c
```

Produces file `code.s`
Machine Instruction Example

- **C Code**: add two signed integers

```c
int t = x+y;
```

- **Assembly**
  - Add two 4-byte integers
  - “Long” words in GCC speak
  - Same instruction whether signed or unsigned

```assembly
addl 8(%ebp),%eax
```

-Similar to expression:
  - `x += y`

-More precisely:
  - `int eax;
    int *ebp;
    eax += ebp[2]`

- **Object Code**
  - 3-byte instruction
  - Stored at address 0x401046

```
0x401046: 03 45 08
```
Object Code

Code for sum

0x401040 <sum>:
  0x55
  0x89
  0xe5
  0x8b
  0x45
  0x0c
  0x03
  0x45
  0x08
  0x89
  0xec
  0x5d
  0xc3

  • Total of 13 bytes
  • Each instruction 1, 2, or 3 bytes
  • Starts at address 0x401040
  • Not at all obvious where each instruction starts and ends

• Assembler
  • Translates .s into .o
  • Binary encoding of each instruction
  • Nearly-complete image of executable code
  • Missing links between code in different files

• Linker
  • Resolves references between object files and (re)locates their data
  • Combines with static run-time libraries
    • E.g., code for malloc, printf
  • Some libraries are dynamically linked
  • Linking occurs when program begins execution
Disassembling Object Code

Disassembled

```
00401040 <_sum>:
  0:      55  push  %ebp
  1:     89 e5  mov  %esp,%ebp
  3:     8b 45 0c  mov  0xc(%ebp),%eax
  6:     03 45 08  add  0x8(%ebp),%eax
  9:     89 ec  mov  %ebp,%esp
 b:      5d  pop  %ebp
c:      c3  ret
```

- **Disassembler**
  - `objdump -d p`
  - Useful tool for examining object code (man 1 objdump)
  - Analyzes bit pattern of series of instructions (delineates instructions)
  - Produces near-exact rendition of assembly code
  - Can be run on either `p` (complete executable) or `p1.o` / `p2.o` file
Alternate Disassembly

<table>
<thead>
<tr>
<th>Object</th>
<th>Disassembled</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x401040:</td>
<td>0x401040 &lt;sum&gt;: push %ebp</td>
</tr>
<tr>
<td></td>
<td>0x401041 &lt;sum+1&gt;: mov %esp,%ebp</td>
</tr>
<tr>
<td></td>
<td>0x401043 &lt;sum+3&gt;: mov 0xc(%ebp),%eax</td>
</tr>
<tr>
<td></td>
<td>0x401046 &lt;sum+6&gt;: add 0x8(%ebp),%eax</td>
</tr>
<tr>
<td></td>
<td>0x401049 &lt;sum+9&gt;: mov %ebp,%esp</td>
</tr>
<tr>
<td></td>
<td>0x40104b &lt;sum+11&gt;: pop %ebp</td>
</tr>
<tr>
<td></td>
<td>0x40104c &lt;sum+12&gt;: ret</td>
</tr>
</tbody>
</table>

- Within gdb debugger
  - gdb p
  - disassemble sum
    - (disassemble function)
  - x/13b sum
    - (examine the 13 bytes starting at sum)
What Can be Disassembled?

```
% objdump -d WINWORD.EXE

WINWORD.EXE: file format pei-i386

No symbols in "WINWORD.EXE".
Disassembly of section .text:

30001000 <.text>:
30001000: 55 push %ebp
30001001: 8b ec mov %esp,%ebp
30001003: 6a ff push $0xffffffff
30001005: 68 90 10 00 30 push $0x30001090
3000100a: 68 91 dc 4c 30 push $0x304cdc91
```

- Anything that can be interpreted as executable code
- Disassembler examines bytes and reconstructs assembly source