Integer C Puzzles

Argue that it is always true or provide a counter example.

Assume 32-bit architecture

Initialization

```c
int x = foo();
int y = bar();
unsigned ux = x;
unsigned uy = y;
```

- $x < 0$  \implies  \((x*2) < 0\)
- $ux >= 0$  \implies  \((x<<30) < 0\)
- $x & 7 == 7$  \implies  \(-x < -y\)
- $ux > -1$  \implies  \(x + y > 0\)
- $x >= 0$  \implies  \(-x <= 0\)
- $x <= 0$  \implies  \(-x >= 0\)
- $(x|-x)>>31 == -1$
- $ux >> 3 == ux/8$
- $x >> 3 == x/8$
- $x & (x-1) != 0$
• $x < 0 \Rightarrow x \times 2 < 0$
  • False: Underflow
  • Result not defined for signed integers, but wraps around on x86
• $ux \geq 0$
  • True: No negative unsigned integers.
• $x \& 7 == 7 \Rightarrow x \ll 30 < 0$
  • True: $x \& 7 == 7$ means that all bits that are 1 in 7 (111) are set in $x$
  • 2s bit set $\Rightarrow x \ll 30$ has high bit set, so < 0
• $ux > -1$
  • False: Signed integer compared with unsigned integer, promoted to unsigned
  • $(\text{unsigned int})-1 == \text{INT}_\text{MAX}$
• $x > y \Rightarrow -x < -y$
  • False: $x = 0$, $y = \text{T\_MIN}$
  • $-x == 0$, $-y$ is undefined; in 2’s complement, $-y == \text{T\_MIN}$
• $x \times x >= 0$
  • False: Overflow
• $x > 0$ && $y > 0 \Rightarrow x + y > 0$
  • False: Overflow
• $x >= 0 \Rightarrow -x <= 0$
  • True: all positive numbers have distinct inverses
• $x <= 0 \Rightarrow -x >= 0$
  • False: $-\text{T\_MIN} == \text{T\_MIN}$
• \((x|-x)\lll 31 == -1\)
  • False: \(x = 0\)
  • (true for all other values)
• \(ux \lll 3 == ux / 8\)
  • True: Integer division is truncated, not rounded
  • So is right shift
• \(x \lll 3 == x / 8\)
  • False: With negative numbers, division truncates toward 0
  • Right shift truncates toward \(-\infty\)
• \(x & (x - 1) != 0\)
  • False: \(x = 0\) or \(x = 1\)
  • \(0 & y == 0\) for all \(y\)
Floating Point Puzzles

• For each of the following C expressions, either:
  • Argue that it is true for all argument values
  • Explain why not true

- \( x == (\text{int})(\text{float}) x \)
- \( x == (\text{int})(\text{double}) x \)
- \( f == (\text{float})(\text{double}) f \)
- \( d == (\text{float}) d \)
- \( f == -(-f); \)
- \( 2/3 == 2/3.0 \)
- \( d < 0.0 \Rightarrow ((d*2) < 0.0) \)
- \( d > f \Rightarrow -f > -d \)
- \( d * d >= 0.0 \)
- \( (d+f)-d == f \)

Assume neither \( d \) nor \( f \) is NaN

\( int x = \ldots; \)
\( float f = \ldots; \)
\( double d = \ldots; \)
• \( x == (\text{int})(\text{float})x \)
  • False: floating-point conversion can lose precision
• \( x == (\text{int})(\text{double})x \)
  • True: 64-bit double has enough fraction bits to hold an int with enough precision
• \( f == (\text{float})(\text{double})f \)
  • True: No precision lost converting to a larger data type
• \( d == (\text{float})d \)
  • False: Implicit conversion back to double, but some precision is lost in initial conversion
• \( f = -(-f) \)
  • True: negation only affects the sign bit
• \( \frac{2}{3} == \frac{2}{3.0} \)
  • False: First is integer division, truncates to 0
  • Second is floating point division
• \( d < 0.0 \Rightarrow ((d*2) < 0.0) \)
  • True: No overflow in floating point
  • If it’s too small, returns \(-\infty\)
• \( d > f \Rightarrow -f > -d \)
  • True: No possibility of overflow in negation
• \( d * d >= 0.0 \)
  • True: No overflow. \( d * d \) might be \( \infty \), but \( \infty > 0.0 \)
• \( (d + f) – d == f \)
  • False: \( d + f \) could become infinite
Assembly and gdb

- Bomb lab out now!
- Do this lab on the Shark machines
  - shark.ics.cmu.edu
  - Log in with your Andrew credentials
- Tools
  - objdump -d
  - gdb
  - input file
Assembly Language

- Low level, directly correlated with hardware operations
- Not the final binary code (machine code)
  - Each assembly mnemonic corresponds to a single machine code instruction
  - Transformed to machine code by an assembler
- Machine specific – IA32 vs. x86-64 vs. SPARC, ARM, AVR, etc.
  - For this lab, we’ll use IA32 assembly
- Most operations manipulate registers – 32-bit memory locations in the processor itself
  - Referred to by name (%eax, %esp, etc)
Memory Addressing

• Terminology used in assembly language to denote memory locations
  • Syntax differs between assemblers, but semantics are constant
• Memory locations of the form \( D(R_b, R_i, S) \)
  • \( D \) is a constant offset (“displacement”)
  • \( R_b \) is a register containing the base of the address
    • use \((R_b)\) to simply access the memory at the address in a register
  • \( R_i \) is an index register, used to index into arrays
  • \( S \) is the size of the objects in an array
• Address = \( D + R_b + R_i \times S \)
Arithmetic Operations

- Store results in the second operand
  - `addl [src] [dest]` => `dest = src + dest`
  - `subl [src] [dest]` => `dest = dest – src`

- Set flags indicating properties of the result
  - Global CPU flags indicating “the result of the last arithmetic operation”
  - `cmpl [src] [dest]` sets flags the same as `subl [src] [dest]`, but does not store the result in `dest`. 
Control Flow

- Jump instructions – set the program counter to a specified address
- Allows execution of code from most parts of memory – including self-modification
  - We’ll go into this more during Buffer Lab
- Conditional jumps
  - jump if certain conditions of the arithmetic flags apply – for example, if the last computation returned 0
  - `cmpl %eax, %ebx`
  - `je 0x00000e80`
Debugging

- **gdb** – the GNU debugger
  - invoked with “gdb [program]” or “gdb --args [program] [arg1] [arg2] ...”
  - To allow gdb access to information about your C source, compile with “gcc -g”
- Allows you to step through your code, examine program state
- **Breakpoints** – tell gdb to “Run until you get to this point”
  - In gdb, type “break [argument]”
  - Function names
  - Line numbers (with -g)
  - Addresses – e.g. “break *0x00000e80”
Debugging – Running your program

- **start**
  - Loads your program, but pauses before running any code

- **run**
  - Loads your program and starts it executing
  - Runs until it terminates or hits a breakpoint
Debugging – Moving through code

• **step/stepl**
  - **step**: Proceed to the next line of code, passing into any function call
  - **stepl**: Proceed to the next instruction, passing into any function call
  - Without -g, step doesn’t know where the next line is
  - proceeds till program exits

• **next**
  - Only works with -g
  - Proceeds to the next line of code, running through any functions required to get there

• **continue**
  - Runs the program until it hits a breakpoint or terminates
Debugging – Examining your Program

• print
  • Takes an expression as its argument
  • With -g, can use variable names
  • Access registers with e.g. $eax
  • Can control formatting:
    • print/x for hex
    • print/t for binary
    • print/s for a null-terminated string
    • others – see “help print” for more information

• examine
  • Shows a memory location
  • Approximately, “examine x” = “print *x”
Resources

- Intel Software Developer’s Manual
  - Very large
  - Full documentation of the IA32 and x86-64 architectures, including all assembly instructions

- GDB quick reference
  - [http://users.ece.utexas.edu/~adnan/gdb-refcard.pdf](http://users.ece.utexas.edu/~adnan/gdb-refcard.pdf)
  - Lists commands for gdb
  - Use internal help feature for more details