CS367 – Test 1 – Review Guide

This guide tries to revisit what topics we've covered, and also to briefly suggest/hint at types of questions that might show up on the test. Anything on slides, assigned reading, recitation work, or in our assignments are good places to look for practice.

"ints aren't integers, floats aren't reals." Be able to explain when differences arise between math and our representations.

**C language** - pointers and arrays
- pointer code: what happens?
- pointer parameters allow the function to update values (pass by reference).
- printf missing arguments: whatever else on the stack gets used.

**representing info as bits**
- bits often written as binary or hex.
- numbers: we considered unsigned int, two's complement int, floating point
- chars/strings: bytes according to ascii, null-terminating (\0)
- instructions: individual instructions of our program are stored in a binary encoding, like 'ascii for instructions', with #'s and register names embedded. (Just know that even instructions are stored as bits somehow – we actually will learn how in the next 3rd of the semester).

**boolean algebra**
- Know how to use these ops, and do bit-manipulation/calculations like our homework.
- bit ops: & | ^ ~ << >> (both >>'s: logical(0's) and arithmetic(MSB’s))
- logic ops: && || !

**big-endian vs. little-endian** ([wiki page is nice!](https://en.wikipedia.org/wiki/Endianness))
- *We always* use the lowest address used by a multi-byte value as the address. Big-endian stores the MSByte (most significant byte) at this lowest address, little-endian stores the LSByte (least significant byte) there.
- Humans are sort of big-endian: 1,234₁₀ has the MSD (1) earliest (the start), while the majority of systems you'll likely use (esp. x86) are little-endian (would store 0x12345678 as the bytes 78-56-34-12).
- When does this show up, and not show up, in our own data representations we've discussed so far this semester?
Encoding Numbers

When representing integers with some fixed number of bits, we covered unsigned and two's complement. In both cases, each column is worth a magnitude of successive powers of two. The only difference between the two is that the leftmost column is worth a positive amount with unsigned, and a negative amount with two's complement. All the differences arise from this single change.

- **unsigned.** Assume width \( w \) below. (Constants in hex are for \( w=32 \)).
  - \( \text{UMax} = (2^w) - 1 = 0xFFFFFFFF \)
    - -1 b/c zero is one of the \( (2^w) \) represented numbers.
  - \( \text{UMin} = 0 = 0x00000000 \)
    - leading bit is nothing special.

- **signed** (two's complement).
  - leading bit has same magnitude as unsigned \( (2^{w-1}) \), but is worth negative the amount. All other columns have the same positive value as in unsigned.
  - \( \text{TMax}: 2^{(w-1)} - 1 = 2^{(8-1)} - 1 = 128 - 1 = 127. \)
  - \( \text{TMin}: -2^{(w-1)} = -2^{(8-1)} = -128. \)
  - \( |\text{TMin}| = |\text{TMax}| + 1. \) (Asymmetric range of values)
  - Example: 8-bit numbers. 256 values, from \(-128\) to \(+127\).
    - \( \text{TMax}: 2^{(w-1)} - 1 = 2^{(8-1)} - 1 = 128 - 1 = 127. \)
    - \( \text{TMin}: -2^{(w-1)} = -2^{(8-1)} = -128. \)

- casting between signed and unsigned integer representations: doesn't change the bits, only the interpretation (e.g., less-than check).
  - C language: comparing unsigned and signed? the signed is implicitly casted to unsigned. So \(-1<0U\) becomes \(2^w - 1 < 0\) (all unsigned), and would be false (0).

- **Addition**
  - unsigned add and signed add have identical behavior at the bit-level.
  - true addition requires \( w+1 \) bits; extra (top) bit is discarded (overflow may occur!).
  - Two's complement overflow: consider \( s = u+v. \)
    - overflow occurs iff: \((u,v<0, s >=0) \) or \((u,v >=0, s<0)\).
    - negative overflow and positive overflow are possible.
    - in C: \( \text{ovf} = (u<0 == v<0) \&\& (u<0 != s<0) \)

- **Multiplication**
  - true multiplication requires \( w*2 \) bits. discard top \( w \) bits (overflow). We get the \((\text{mod } 2^w)\) answer.
  - \( u<<k \) equates to \( u*2^k \). (signed or unsigned). Easily calculate doubling as shifting by a bit.
  - can combine for non ~ "2\(^n\)" constant multipliers: \( u<<5 - u<<3 == u*24 \). \textbf{shift and add} are often much faster than \textbf{multiply} instructions (though the gap is closing...)

- **Division**
  - divide by power of two:
    - unsigned: \( u>>k \) gives floor\( u/(2^k) \). Logical shift (fills in zeroes)
    - signed: can't use \( u>>k \) directly when \( u \) is negative; floor \((x/(2^k))\) rounds down always, which is wrong for negative results (should round towards zero).
    - "Biasing." for negative numbers only, compute as: \( \text{floor}( (x+(2^k)-1) / (2^k) ) \)
Floating Point

- IEEE standard 754
- can represent non-whole numbers (and whole numbers, too)
- only an approximation of the real numbers. Floating-point numbers are more dense near origin. Sparser and sparser as magnitude increases (same # of #'s per order of magnitude: \(2^5, 2^6, 2^7\), etc each have same # of #'s – this means the gap between numbers increases at each jump in order of magnitude).

form: \((-1)^s \times M \times (2^E)\)
- \(s\) is sign bit
- \(M\) is the significand
- \(E\) is the exponent.
- \((s \mid \text{exp} \mid \text{frac})\) is the layout of bits.
- \(\text{exp}\) encodes \(E\), using a bias to offset the value.
  - bias = \(2^{(\text{expWidth}-1)}-1\). (e.g., with four bits for exponent, bias is \(2^{(4-1)}-1 = 7\).
- \(\text{frac}\) encodes \(M\) (the significant bits of scientific notation).
  - encodes bits to right of bit-point.
  - denormalized (exp=0's): implicit leading 0 to left of bit-point.
  - normalized (exp !=0's, exp!=1's): implicit leading 1 left of bit-point.

Three groups of values:
- **normalized**: exp is mix of 0's and 1's.
  - \(E = \text{exp} - \text{bias}\).
  - \(M = (1.)\text{xxx}, \text{where frac}=\text{xxx}\).
- **denormalized**: exp==0's.
  - \(E = 1 - \text{bias}\).
  - \(M = (0.)\text{xxx}, \text{where frac}=\text{xxx}\).
- **special**: exp = 1's.
  - frac = 0's: infinity (pos or neg)
  - frac != 0's: NaN.

Some example representations:
- 8 bits: 1 sign bit 4 exp bits 3 frac bits.
- 32 bits: 1 sign bit 8 exp bits 23 frac bits.
- 64 bits: 1 sign bit 11 exp bits 52 frac bits.

**Concept**: a floating point number really stores a sign bit and two unsigned ints to encode \(V=(-1)^s \times M \times 2^E\). Consider 12.0 == 1.100\_2 \times 2^3= 0 1010 100 (8-bit minifloat, with 1/4/3 s/exp/frac bits). \(M\) is encoded as frac, equaling "how many eighths?", (here, 100\_2=4 eighths). \(E\) is encoded as exp, as \(E+\text{bias}=\text{exp}, 3+7=1010 = 1010_2\) since it's normalized. If you can identify these two unsigned values and read them, floating point becomes a lot easier. (more frac bits? different power-of-two fraction than 8ths; more exp bits? larger bias).

**Rounding modes**: calculate value with more precision than storable (use more bits while calculating). Then either round-to-even (default), round up, round down, or truncate.
- round-to-even: when there’s a closest value, round to that no matter what!; but when our temporary calculation's full-precision value is exactly halfway between two representable values, always choose the even value. See slides on rounding for more information.

**Converting between integral/f.p. representations**
- bit-pattern IS changed.
- int → double: exact (all ints representable with 52 frac bits and E≤31).
- int → float: rounding for larger-magnitude numbers (>2^{24}).
- fp → int: truncates fractional part (rounds to zero). May overflow (*usually* defined as TMax or TMin)
Compilation Process

- from source to assembly code file (.s)
- from assembly to binary (.o). Basically non-human-friendly version of assembly.
- linker commits to address locations, combines multiple files, and substitutes labels for addresses, creating an executable program.

Disassembly

- turning object code (.o or other executable) to assembly (.s) for human-readable version.
- command: `objdump -d executable_file`

Registers.

Registers are physical locations in the CPU (usually 64-bit in our discussions) that go by a 3-letter name, instead of a numeric address like memory. They allow faster calculations and moving of values, mostly due to being part of the processor itself (when compared to caches and memory/RAM, which are accessible at a further distance).

- Here are some special purpose registers:
  - `%rip`: program counter (address of the next assembly instruction to be run)
  - `%rsp`: stack pointer (boundary of used/unused portion of stack)
  - `%rbp`: frame pointer ("base pointer" – beginning of current frame, close to top of stack)
  - `%rax`: used for return value in function call’s return. (integer values)

- The general purpose registers, which are used for calculations of the program, are:
  - `caller-save`: `%rax, %rdi, %rsi, %rcx, %rdx, %r8-%r11`
  - `callee-save`: `%rbx, %rbp, %r12 - %r15`

Assembly Code

- We are using AT&T format (not Intel) for our class → `op src, dest`
- x86-64: CISC style (Complex Instruction-Set Computer). It has many different instructions with different formats.
- RISC style (Reduced Instruction Set Computer) tends to have more registers, fewer instructions, three operand instructions, and tends to be quite fast.
- x86-64: new for 64-bit machines. Added more registers (RISC-like), avoids stack usage (by using more registers). As a result we need 64-bit pointers.

Data Types

In assembly, we only have these data types:

- integral types of 1,2,4, 8 bytes. (labeled `byte`, `word`, `long` word, or `quadword`). These names originated long ago enough that "word" meant 16 bits at the time.
- float types of 4,8,10 bytes. (labeled `single`, `long`, and `extended`).
- no complex types like arrays or structs--must be simulated byte by byte.
Addressing Modes

Addressing modes are the different ways we can represent a value, or indicate a location. For registers and memory locations, we can read or write from them. Here are some examples and their meaning; the general case is last. For Test One: be able to interpret the address of these addressing styles.

- $0x0123456789ABCDEF$: immediate value (literal value). A dollar sign followed by a C style integer value. Must fit in our word length.
- %reg: register's contents. (note: %reg is not a real register name!)
- (%reg): treat register's contents as memory address, and access that memory address's contents. \( \rightarrow \) %rsp and %rbp are usually used in this "de-referencing" way.

**General Case:** imm(%reg\_base, %reg\_index, scale): calculate \( \text{imm} + \text{%reg}\_base + \text{%reg}\_index*\text{scale} \). This means that we add the immediate value \( \text{imm} \) to the contents of \( \text{%reg}\_base \), and further add the result of multiplying the contents of \( \text{%reg}\_index \) by \( \text{scale} \). This result is treated as a memory address; the instruction using it determines whether we read/write the address or value at that address.
  - think of \( \text{%reg}\_base \) as the base address of an array.
  - think of \( \text{%reg}\_index \) as the index of one spot in the array.
  - think of \( \text{scale} \) as the byte-width of each item in the array.
    - **NOTE:** scale can only be the number 1,2,4, or 8. (It is implemented as a bit-shift, which is very fast and simple to implement).
  - variants:
    - (basereg)
    - (basereg,indexreg)
    - D(basereg)
    - D(basereg,indexreg)
    - (basereg,indexreg,scale)
    - (, indexreg,scale)
    - D(basereg,indexreg,scale)
    - D(, indexreg,scale)

- movq src, dest:
  - either the source or destination can use the general case. When the source has the general case format, the value at the calculated address will be copied into the destination. When it's the destination, the source value will be copied into the address of the destination.
    - possible moves:
      - immediate to register
      - immediate to memory
      - register to register
      - register to memory
      - memory to register
    - disallowed:
      - memory to memory (too slow!)
      - anything to immediate (e.g., doesn't make sense to store a value into 5).

- leaq instruction: same addressing modes as with the general case that movq used, except it's used to calculate memory address *without* performing memory reference. The address itself is stored to the destination (result is the address instead of its contents, skipping the "lastly" sentence of the general case description above).
  \( \rightarrow \) leaq is very useful for performing mathematical calculations that have nothing to do with memory addresses. We get two adds and a limited version of muldiv in one fast instruction!