CS 33 Final Review Guide

Note: This review guide may not contain all topics needed on the exam. It is intended to simply help guide you as you study for your exam.

I. Bits and Bytes

- **a.** Representation of Numbers
  - **i.** Can you convert numbers between binary, decimal, and hexadecimal forms?
- **b.** How computers represent data types of C (particularly for the x86-64 architecture)
- **c.** Data Operations
  - **i.** Boolean Operations
    1. And (&)
    2. Or (|)
    3. Xor (^)
    4. Not (~)
    5. Arithmetic and Logical Shifts (<<, >>)
       - **a.** For which types of data would you do arithmetic shifts and for which would you do logical shifts?
  - **ii.** Logical Operations
    1. And (&&)
    2. Or (||)
    3. Not (!)
  - **iii.** What is the difference in the output of boolean operations and logical operations?
  - **iv.** Memory and Strings
    1. How is memory addressed (particularly for x86-64)?

2. Endianness
    a. Little Endian vs. Big Endian
    b. What Endianness is used for x86-64?
3. Strings
    a. How are strings are denoted and represented in C?

II. Integers

a. Unsigned
    i. Minimum and Maximum represented numbers for an n bit integer.
b. Signed
    i. 2’s complement representation of an integer.
        1. How to calculate the negative of a 2’s complement number
        2. Minimum and maximum represented numbers for an n bit 2’s complement integer.
c. Casting between unsigned and signed integers (implicit and explicit)
    i. Results when comparing unsigned and signed integers (any combination of the 2)
d. Integer addition
    i. How is addition performed?
    ii. Overflow
        1. Unsigned overflow vs. Signed Overflow
        2. How to detect overflow?
e. Integer shifting
    i. Shifting left by n bits is equivalent to what arithmetic operation?
    ii. Shifting right?

III. Assembly Basics (x86-64)

- **a.** Concepts of registers, PC, and memory.
    - **i.** What exactly do we store in registers?
- **b.** Assembly instructions
    - **i.** Basic format
        - **1.** Where src and dest values are located.
        - **2.** Instruction suffixes:
            - **a.** What does the b, w, l, q suffixes for instructions mean?
            - **b.** Can you identify which portion of a register is being used by its name?
                - **i.** (i.e. %rax vs. %eax)
    - **ii.** lea vs. mov
    - **iii.** Be familiar with other basic operations like add, sub, etc.
    - **iv.** Calculating memory addresses
        - **1.** When in assembly would you dereference an address?
        - **2.** Format and formula for calculating memory addresses.
            - **a.** Ex. What is the address calculated by 20(%rax, %rcx, 2)?

IV. Control

- **a.** Condition Codes
    - **i.** CF, SF, ZF, OF
        - **1.** What are these control codes and when are they set?
        - **2.** Explicit and Implicit code setting
- **b.** Control Instructions
    - **i.** Instructions that explicitly set condition codes
        - **1.** test, cmp

            a. What operations do these instructions perform, and how do they set the condition codes?
            b. Do these instructions have any side effects besides setting condition codes?
    ii. Instructions which use condition codes.
        1. Jump instructions (jX)
            a. e.g. jg, jle, jne, je, etc.
        2. Other conditional instructions
            a. SetX
            b. Conditional Move
                i. Ex. Cmovle
        3. How can you use these instructions to implement branches, loops and switch statements?

V. Procedures

- **a.** Stack
    - **i.** Stack organization
        - **1.** Direction stack grows and location of the top of the stack.
        - **2.** %rsp register
            - **a.** What does it contain?
    - **ii.** Stack commands
        - **1.** pushq
        - **2.** popq
        - **3.** What does using sub on %rsp do?
- **b.** Procedure control
    - **i.** Instructions
        - **1.** callq
        - **2.** retq

3. What do these instructions do? How do they affect the PC (%rip), %rsp, and the stack?

ii. Data conventions

1. In which registers are procedure arguments stored in?
2. In which register is the return value stored in?

iii. Be familiar with the overall process of passing control when a procedure is called and when it is returned.

iv. Caller-saved registers vs. Callee-saved registers.

v. Concept of stack frames

1. What type of data is typically stored in a stack frame?

vi. Recursion

1. How recursion looks as assembly code and how it is represented as stack frames?

VI. Data

a. Arrays

i. Is data contiguous?

ii. How do you measure size of array?

iii. How is array stored in memory?

iv. C syntax

1. Suppose you have array: int val[5]
    a. Indexing: what does val[3] return?
    b. What does val return?
    c. What does val + 2 return?
    d. What does *(val + 1) return?
    e. What does &(val[0]) return?

v. Multidimensional arrays:

1. How are multidimensional arrays stored in memory?

            a. What flattening method is used?
        2. If given an index for a value or address in an array, can you find it in memory?
- **b.** Structs
    - **i.** How is data stored in a struct?
        - **1.** Is data contiguous?
        - **2.** Memory representation
            - **a.** Alignment (Which addresses would you store the data?)
                - **i.** What are the padding rules for individual elements?
                - **ii.** What are the padding rules for the overall struct?
            - **b.** How would you access an element if you only had a pointer to the struct?
            - **c.** Reorder elements to save space.
- **c.** Unions
    - **i.** How is data stored in unions?
        - **1.** Which address(s) would you store the data?
    - **ii.** How to measure size of unions?

VII. Advanced Machine Programming Topics

- **a.** Memory space.
    - **i.** How is memory organized?
        - **1.** Stack
        - **2.** Heap
        - **3.** Data
        - **4.** Text/Shared Libraries
    - **ii.** Which types of data goes into which location?
        - **1.** I.e., where would local procedure data be stored?
        - **2.** Where would static variables be stored?

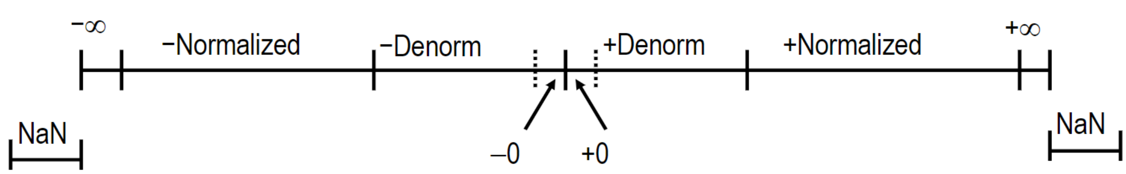
3. Where would procedures and instructions be stored?

b. Buffer Overflow
   i. How does buffer overflow manifest?
   ii. What are some issues with buffer overflow?
      1. i.e. Stack smashing
   iii. What are some defenses for buffer overflow?

VIII. **Floating Point**

a. IEEE Floating Point Formula
   i. Normalized Floating Point
   ii. Denormalized Floating Point
   iii. Special Cases (Inf, -Inf, and NaN)
   iv. In each case, what is the value of the exponent?
   v. For Normalized and Denormalized Floating Point, What is the leading value of M?
   vi. Given an exponent field size, how do you calculate bias and E in the floating point formula?
   vii. How do you determine the sign of a floating point value?

b. Useful Visualization



c. Properties of Floating Point operations
   i. Rounding
      1. Rules

            a. Towards 0
            b. Round down
            c. Round up
            d. Nearest Even
        2. Rounding in binary
    ii. Addition
        1. How to add up the M fields in conjunction with the E fields, and round M to remove excess bits.
    iii. Multiplication
        1. Basic understanding of what is done.
    iv. How are Inf and NaN values generated from addition and/or multiplication ops?
    v. Casting
        1. Cast from float to double and vice versa?
        2. Cast from float/double to int and vice versa?
        3. In which cases will the value be preserved?
        4. Rounding and truncation rules when casting in general.

d. Comparisons and Equations
    i. What is the result of a comparison with Inf?
    ii. What is the result of a comparison with NaN?
    iii. Can you determine if 2 expressions are equivalent?
        1. Study examples from lecture and discussion.

IX. Optimization

a. General Optimizations
    i. Code/Motion
    ii. Strength reduction
    iii. Sharing common subexpressions
    iv. Removing unnecessary procedure calls

b. Optimization Blockers

i. Procedure Calls
   1. Why are procedure calls expensive?
   2. Why can't compilers optimize these easily?
ii. Memory Aliasing
   1. When multiple references refer to same or overlapping memory?
   2. Why can't compiler optimize away certain memory accesses if the programmer knows they don't refer to the same memory?
   3. Look at examples.

c. Instruction Level Parallelism
   i. In general, why can we do this?
   ii. Cycles Per Element/ Cycles per op
      1. In general: T (Total execution time) = CPE*n(number of iterations) + overhead
      2. Would adding m functional units improve T by a factor of m?
   iii. Latency and Throughput
      1. Latency is thought of as how long a function/operation takes overall.
      2. Throughput is how many ops can be completed in a time unit (i.e. cycle).
      3. If there was no parallelism, Throughput = 1/Latency.
      4. How about if there was parallelism?
   iv. ILP optimizations
      1. Loop-unrolling
         a. What is it?
         b. Why does this improve performance?

            - **i.** Does it remove sequential dependencies?
        - **c.** Does this improve latency and/or throughput?
    - **2.** Reassociation
        - **a.** What is it?
        - **b.** Why does this improve performance?
            - **i.** Does it remove sequential dependencies?
        - **c.** Does this improve latency and/or throughput?
        - **d.** Consider cases of simple reassociation of operations vs. using multiple accumulators.
    - **3.** Estimate improvement of performance using latency and throughput bounds (given).
        - **a.** Understand how/why optimizations may improve/reduce performance.
- **v.** Additional Topics
    - **1.** SIMD
        - **a.** Vector operations
        - **b.** How does this improve performance?
    - **2.** Branch
        - **a.** Why are these operations typically so expensive?
        - **b.** Branch Prediction: How does a CPU use this to improve branch performance?

X. Memory Hierarchy (Caching in Particular)

- **a.** Locality
    - **i.** Temporal
        - **1.** What does this mean?

        - **ii.** Spatial
            - **1.** What does this mean?
        - **iii.** Consider examples for both data and instruction memory.
    - **b.** Hierarchy
        - **i.** Understand at a high level how the cache hierarchy works.
            - **1.** How is the hierarchy organized?
            - **2.** Why does the hierarchy work?
        - **ii.** Memory hits vs. misses
            - **1.** Understand at a high-level different policies for hits and misses exists, which includes how blocks are evicted from the cache and replaced.
    - **c.** Cache Organization
        - **i.** Cache Structure
            - **1.** What is a block/line?
            - **2.** What is a set?
        - **ii.** Cache addressing and accessing
            - **1.** Which bits of a memory address determine the block offset?
                - **a.** Given a block size B, how many offset bits do you need at least?
            - **2.** Which bits of a memory address determine which set you are accessing in the cache?
                - **a.** How is this done?
                - **b.** If you have S sets, how many set bits do you need at least?
            - **3.** Once you know which set the address is referring to, how do you determine which block the set is referring to?

a. What happens if the block is not found in the set?

4. What is the formula relating cache size C to block size B, set size E, and number of sets S?
    a. Given certain values about the cache structure and memory address structure, calculate the remaining ungiven values.
5. Familiarize yourself with cache organization on multi-core system
    a. What is False Sharing?

d. Practical Optimization using Cache Structure
- **i.** Given code example, modify to take advantage of cache locality.
- **ii.** Particular examples for multi-dimensional arrays.
- **iii.** Estimate misses of certain implementation given block size.

XI. Multithreading

a. High-level overview
- **i.** Understand how to decompose program into multiple threads
    1. Domain decomposition
    2. Task decomposition
    3. Pipelining
- **ii.** Fork-Join Multithreading Model
    1. What does this mean? How does it work?
- **iii.** What is a dependency graph? How is it used for decomposition?

b. OpenMP General
- **i.** Know the various pragma's, macros and clauses used by OpenMP for multithreading

            1. Highlights (this is not an exhaustive list of all macros required):
                a. parallel for
                b. parallel
                c. for
                d. private, firstprivate, lastprivate
                e. single
                f. nowait
    c. Race Conditions and Locking
        i. What is a race condition?
            1. Why does it occur?
        ii. Locking
            1. Could we use a simple flag variable to prevent race conditions?
                a. Give a simple logical reason or counterexample.
            2. If a thread is in a locked section, can another thread enter that locked section?
            3. Implementing locking with OpenMP
                a. Macros
                    i. critical
                        1. How does locking work with this macro?
                        2. What is a benefit and downside of this approach?
                    ii. omp_lock_t (lock objects)
                        1. How is this different from the critical section?
                        2. What is the benefit of this approach?

3. Downside?

iii. Reductions: How is this similar to the above approaches? Different?

d. Deadlock
    i. What is deadlock?
    ii. Why does it occur?
        1. Four conditions
            a. Mutual Exclusion
            b. Partial Resource Allocation
            c. No Pre-empting locks
            d. Cycle in Resource Allocation/Locking
        2. Given a code sample, could you identify the deadlock?
            a. Prove it using the four deadlock conditions
            b. Remove the deadlock
        3. If your code only uses one of the 3 OpenMP locking macros described above, for which of the macros is deadlock possible? Why?
        4. Examples from class/discussion
            a. Circular Dining Table/ Circular Spongebob cooking

XII. Linking

a. Why use Linkers? What is benefit?

b. Static Linking vs. Dynamic Linking?
    i. List benefits and costs of both
        1. Think of storage, performance, etc.
    ii. Types of library files used for each
    iii. At what point of the program does linking occur in each case?

                    1. For dynamic linking, there are multiple possible points at which linking can occur. What are they?
        c. Overview of linking steps
            i. Symbol Resolution
                1. Types of Symbols
                    a. Global
                    b. External
                    c. Local
                2. What does resolution actually mean?
            ii. Relocation
                1. What is happening in this step?
                2. What does the program look like before and after the linking?

XIII. Exceptions

- **a.** What is Exceptional Control Flow?
    - **i.** How is it different from branching and jumping?
- **b.** Compare and contrast types of exceptional control flow
    - **i.** Exceptions
    - **ii.** Process Context Switch
    - **iii.** Signals
    - **iv.** Nonlocal jumps
- **c.** Exceptions
    - **i.** What are they?
    - **ii.** How are they run (handled)?
    - **iii.** Types:
        - **1.** Interrupts
        - **2.** Traps
        - **3.** Faults
        - **4.** Aborts
    - **iv.** Compare types of exceptions

            1. Asynchronous vs. Synchronous
            2. How they manifest in general?
            3. Return behavior in general
            4. Examples
    d. Understand general examples of exceptions
        i. System Calls
        ii. Page Faults

XIV. **Virtual Memory**
    a. Why do we use it? What do we get from virtual memory?
    b. Benefits and Costs of Virtual Memory
    c. How VM works?
        i. Page Table
            1. Comparison to caching
                a. Page offset
                b. Page Numbers
            2. Page Translation from virtual address to physical one
            3. Page Hits vs. Page Faults
                a. What happens in each
                b. Compare to cache hits and misses
            4. How are pages allocated during a page fault?
            5. Permission bits: read, write and execute
            6. TLB and multi-level page table
                a. What are the purposes of each?
                b. What benefits do they provide?

XV. **RISC Architectures and MIPS**
    a. Compare and contrast RISC/CISC architectures?
        i. Example: # of available instructions
        ii. Size of typical programs.
        iii. Etc.

b. Is x86-64 CISC or RISC?
c. Is MIPS CISC or RISC?
d. MIPS
    i. Know the different types of instructions and their formats?
        1. Memory accesses
        2. Arithmetic
        3. Logical
        4. Comparison
        5. Control (branching, jumping)
    ii. Can you convert x86-64 code to MIPS and vice versa?
    iii. Can you read a MIPS program, understand what it does, and find a solution given some input?
        1. Similar to our x86-64 assembly problems, but with MIPS.

XVI. Practical Experience

a. Linux and gdb
    i. Basic gdb commands
    ii. Can read output of linux and gdb commands, particularly those shown in class.
        1. Can read and interpret register and memory dumps from gdb.
b. Data Lab
    i. Familiarize yourself with all the functions. You should know the basic algorithm for implementing each of these functions.
c. Bomb Lab
    i. Be able to understand and interpret functions written in assembly.

    ii. Given input register values, be able to calculate output register values, and vice versa.
    iii. Particular examples: recursions, switch jump tables, loops, …
    iv. Be able to read memory and stack dumps.

d. Attack Lab

- **i.** Buffer Overflow tests
    - **1.** Be able to read an assembly algorithm and stack trace, and understand if and how buffer overflow occurs.
    - **2.** Be able to recognize a code injection attack: usually if you are jumping into the region of the stack you are overflowing.
    - **3.** Be able to recognize a return-oriented programming attack. Follow along the gadget instructions and returned addresses to see what the function is doing.

e. Parallel Lab

- **i.** Why was it so difficult to parallelize the base algorithm given to you?
- **ii.** Understand the modification tip given to you to parallelize the code.
    - **1.** Why did we do it this way?
- **iii.** Understand tradeoffs between parallelization versus cache locality.