CS 61C Summer 2020 Great Ideas in Computer Architecture

MIDTERM 1

INSTRUCTIONS

This is your exam. Complete it either at exam.cs61a.org or, if that doesn't work, by emailing course staff with your solutions before the exam deadline.

This exam is intended for the student with email address `cs61c@berkeley.edu`. If this is not your email address, notify course staff immediately, as each exam is different. Do not distribute this exam PDF even after the exam ends, as some students may be taking the exam in a different time zone.

For questions with **circular bubbles**, you should select exactly *one* choice.

- ◯ You must choose either this option
- ◯ Or this one, but not both!

For questions with **square checkboxes**, you may select *multiple* choices.

- ☐ You could select this choice.
- ☐ You could select this one too!

You may start your exam now. Your exam is due at <DEADLINE> Pacific Time. Go to the next page to begin.

Preliminaries

Please complete and submit these questions before the exam starts.

(a) What is your full name?

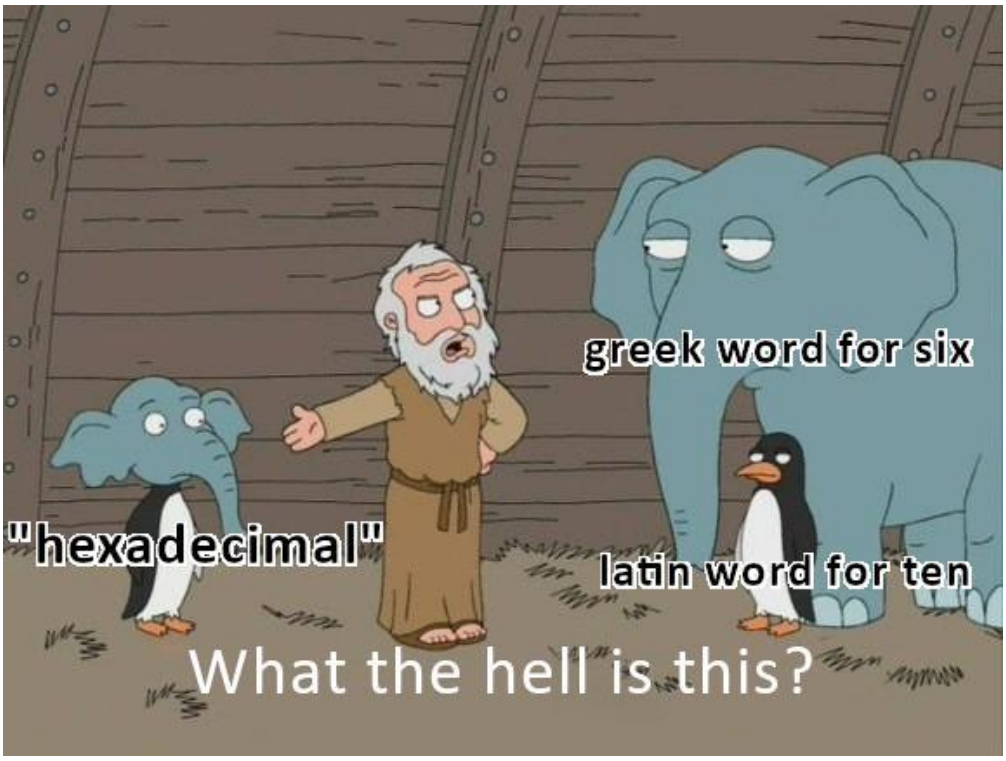
Solutions

(b) What is your student ID number?

dQw4w9WgXcQ (This is a YouTube video)

(c) If an answer requires hex input, make sure you only use capitalized letters! For example, 0xDEADBEEF instead of 0xdeadbeef. You will be graded incorrectly otherwise! Please always add the hex (0x) and binary (0b) prefix to your answers or you will receive 0 points. For all other bases, do not add the suffix or prefixes.

Some of the questions may use images to describe a problem. If the image is too small, you can click and drag the image to a new tab to see the full image. You can also right click the image and download it or copy its address to view it better. You can use the image below to try this. You can also click the star by the question if you would like to go back to it (it will show up on the side bar). In addition, you are able see a check mark for questions you have fully entered in the sidebar. Questions will auto submit about 5 seconds after you click off of them, though we still recommend you click the save button.



Good luck!

1. A Generic C Question

In object-oriented programming languages such as Java, the concept of a Generic data type exists. This means that, in a class definition of an object, we can leave the data types of chosen variables as an "unknown" type that is instead expected to be provided during instantiation of the object. In this problem, we will implement generics in C for a LinkedList. Remember, though, that we do not have objects to instantiate in C, so instead our GenericLinkedList should simply be versatile enough to accept any given data type without error or compiler warnings. **A user should not need to do any form of explicit or implicit casting when working with this new data type, except for when dealing with the void* pointer returned by the alloc functions.** For the following, assume we have included the correct `includes`.

(a) (3.0 pt) You may assume that our GenericLinkedList only has to account for 3 data type choices: `char`, `uint16_t`, `uint32_t`, where the # in `uint#_t` represents the number of bits the data type contains. It also supports structs and unions. In addition, we are working on a 32-bit addressable memory space, structs are word-aligned and padded appropriately, and all calls to `malloc()`, `calloc()`, and `realloc()` succeed. Fill in the skeleton for a `GenericLink`. **Your solution must use the minimum amount of space possible. A sub-optimal solution may not receive credit. You may not use void* in your approach.**

```
typedef struct {
  <YOUR CODE HERE>
} GenericLink;
```

(b) (1.0 pt) What does `sizeof(GenericLink)` evaluate to?

(c) I now want to store a String as a GenericLinkedList, i.e. each link should hold one char of the string, with the links ordered the same way as the chars in the string. You may assume that the length of the string is > 1. You do not need to worry about storing the null terminator. Please fill in the following function implementations:

i. (2.0 pt)

```
GenericLink* store_char(char c) {
    /* store_char takes in a char, and returns a
  pointer to a link containing this char */

    <YOUR CODE HERE>
}
```

ii. (6.0 pt)

```
GenericLink* store_string(char* str) {
    /* store_string takes in a string, and returns a
  pointer to the ‘‘head’’ of the GenericLinkedList
  holding the string, i.e. the link containing the first char*/

  <YOUR CODE HERE>
}
```

2. Doubly Linked Trouble!

For this problem, assume all pointers and integers are **four bytes** and all characters are **one byte**. Consider the following C code (all the necessary #include directives are omitted). C structs are properly aligned in memory and all calls to malloc succeed. **For all of these questions, assume we are analyzing them right before main returns.**

```
typedef struct node {
    void *data;
    struct node *nxt;
    struct node *prv;
} node;

void push_back(node *list, void *data) {
  node *n = (node *) malloc(sizeof(node));
  n->data = data; n->nxt = list; n->prv = list->prv;
  list->prv->nxt = n; list->prv = n;
}

int main() {
  char *r     = "CS 61C Rocks!";
  char s[]    = "CS 61C Sucks!";
  node sentinel; sentinel.nxt = &sentinel; sentinel.prv = &sentinel;
  push_back(&sentinel, r);
  push_back(&sentinel, s);
  push_back(&sentinel, &sentinel);
  push_back(&sentinel, calloc(sizeof(s) + 1, sizeof(char)));
}
```

(a) Each of the following evaluate to an address in memory. In other words, they "point" somewhere. Where in memory do they **point**?

i. (0.75 pt) `&sentinel`

- ◯ Heap
- ◯ Stack
- ◯ Static
- ◯ Code

ii. (0.75 pt) `sentinel.nxt->nxt->data`

- ◯ Heap
- ◯ Static
- ◯ Stack
- ◯ Code

iii. (0.75 pt) `&push_back`

- ◯ Stack
- ◯ Code
- ◯ Heap
- ◯ Static

iv. (0.75 pt) `sentinel.nxt->data`

- ◯ Stack
- ◯ Heap
- ◯ Static
- ◯ Code

v. (0.75 pt) `sentinel.prv->prv->data`

- ◯ Static
- ◯ Heap
- ◯ Stack
- ◯ Code

vi. (0.75 pt) `sentinel.prv->data`

- ◯ Static
- ◯ Heap
- ◯ Stack
- ◯ Code

vii. (0.75 pt) `sentinel.prv->prv`

- ◯ Code
- ◯ Heap
- ◯ Stack
- ◯ Static

(b) (3.0 pt) How many bytes of memory are allocated but not `free()`d by this program, if any? (assuming we have not called `free_list`) (Leave your answers as an integer. Do not include the units, we are telling you it's bytes after all!)

(c) (1.75 pt) Say we had this free function:

```
void free_list(node *n) {
  if (n == NULL) return;
  node *c = n->nxt;
  for (; c != n;){
    node *tmp = c; c = c->nxt;
    free(tmp);
  }
}
```

Given this free function, if we called `free_list(&sentinel)` **after all** the code in `main` is executed, this program would have well defined behavior.

◯ False

◯ True

3. RISC-V!

For each of the following, write a simple RISC-V function with one argument. Follow calling convention, use register mnemonic names (e.g., refer to t0 rather than x6), and add commas and a single space between registers/arguments (e.g. `addi a0, a1, 2`). If you do not follow this, you may be misgraded!

(a) Leave your answers fully simplified as integers. **Do not leave powers of 2 in your answer!** Feel free to use a calculator to simplify your answer.

You want to build a mini RISC-V instruction architecture that only supports 16 registers, which allows the length of the register fields to be shortened. Assuming that you use the extra bits to extend the immediate field, what is the range of half-word instructions that can be reached using a branch instruction in this new format? [`<lower bound>`, `<upper bound>`]

i. (0.75 pt) `<lower bound>`

ii. (0.75 pt) `<upper bound>`

(b) Find the length of a null-terminated string in bytes. The function should accept a pointer to a null-terminated string and return an integer. Your solution must be recursive!

```
strlen:
    __<CODE INPUT 1>__
    beq t0, zero, basecase
    __<CODE INPUT 2>__
    __<CODE INPUT 3>__
    __<CODE INPUT 4>__
    jal strlen
    __<CODE INPUT 5>__
    __<CODE INPUT 6>__
    __<CODE INPUT 7>__
    ret
  basecase:
    __<CODE INPUT 8>__
    ret
```

Fill in the following:

i. (0.75 pt) `<CODE INPUT 1>`

ii. (0.75 pt) `<CODE INPUT 2>`

iii. (0.75 pt) `<CODE INPUT 3>`

iv. (0.75 pt) `<CODE INPUT 4>`

v. (0.75 pt) `<CODE INPUT 5>`

vi. (0.75 pt) `<CODE INPUT 6>`

vii. (0.75 pt) `<CODE INPUT 7>`

viii. (0.75 pt) `<CODE INPUT 8>`

(c) Arithmetically negate a Two's Complement 32-bit integer without using the sub, mul or pseudo instructions.

```
negate:
  __<CODE INPUT 1>__
  __<CODE INPUT 2>__
  ret
```

Fill in the following:

i. (0.75 pt) `<CODE INPUT 1>`

ii. (0.75 pt) `<CODE INPUT 2>`

(d) i. (1.0 pt)

```
auipc t0, 0xABCDE # Assume this instruction is at 0x100
addi t0, t0, 0xABC
```

Write down the value of t0 in hex. Reminder: include the prefix in your answer!

ii. (2.0 pt)

```
li t0, 0xABCDEFAD
sw t0, 0(s0)
lb t0, 0(s0)
```

Write down the value of t0 in hex. Assume big-endianness. Reminder: include the prefix in your answer!

4. CALL!

Consider the following assembly code (Note these are the addresses the assembler give each of the instructions):

```
Address | Assembly
--------|---------------------------
0x0C    |         add  t0, x0, x0
0x10    |         addi t1, x0, 4
0x14    | loop:   beq  t0, t1, end
0x18    |         add  a0, a0, t0
0x1C    |         jal  ra, square
0x20    |         jal  ra, printf
0x24    | n:      addi t0, t0, 1
0x28    |         j    loop
0x2C    | end:    ecall
        |
0x30    | square: mul  a0, a0, a0
0x34    |         ret
```

(a) (1.0 pt) A poorly written but correct assembler can seriously slow down the speed of the compiled program.

◯ False

◯ True

Assuming an isolated assembler, create the symbol table after the first pass (top to down). If a line of the symbol table is not used, enter `N/A`.

(b) i. A. (0.25 pt) First label:

B. (0.25 pt) First address:

ii. A. (0.25 pt) Second label:

B. (0.25 pt) Second address:

iii. A. (0.25 pt) Third label:

B. (0.25 pt) Third address:

iv. A. (0.25 pt) Fourth label:

B. (0.25 pt) Fourth address:

(c) **(1.0 pt)** No address needs to be resolved at the linker stage.

◯ False

◯ True

(d) **(4.0 pt)** Translate the instruction at address 0x1C into machine code (in hex).

(e) **(0.5 pt)** This code is the input of...

◯ Loader

◯ Linker

◯ Compiler

◯ None of the other options

◯ Assembler

(f) **(1.5 pt)** Apple recently announced that it is switching from Intel processors to ARM ones, which have a different ISA (a RISC one!). To ensure that old programs can still run on these new devices, which stage(s) of the CALL stack do they need to re-run to create the executable binaries?

☐ Loader

☐ Linker

☐ Compiler

☐ Assembler

☐ None of the other options

(g) **(2.0 pt)** After the first pass of a top to bottom assembler, which of the following instructions do **NOT** have their addresses fully resolved?

☐ `jal ra, printf`

☐ None of the other options

☐ `beq t0, t1, end`

☐ `jal ra, square`

☐ `j loop`

5. Number Fun

(a) Does the resulting operation overflow given 6-bit, Two's Complement numbers?

i. (0.5 pt) `0b011111 + 0b000001`

- ◯ Correct
- ◯ Overflow

ii. (0.5 pt) `0b001111 + 0b001111`

- ◯ Overflow
- ◯ Correct

iii. (0.5 pt) `0b010001 + 0b001111`

- ◯ Overflow
- ◯ Correct

(b) Please answer the questions below, assume we are working with n bits.

i. A. (1.0 pt) It is possible to represent the same range of numbers with biased and 2's complement.

- ◯ True
- ◯ False

B. (1.0 pt) It is possible to represent the same range of numbers with 1's complement and bias.

- ◯ False
- ◯ True

ii. (1.0 pt)

$2 + 2$ can equal fish under the correct representation.

◯ False

◯ True

(c) Select all which is true for the following statements.

i. **(1.0 pt)** Which of the following interpretations allows for multiple different bit sequences to map to the same underlying value?

- ☐ Sign and Magnitude
- ☐ One's Complement
- ☐ Two's Complement
- ☐ Floating Point
- ☐ Biased (for at least 1 choice of bias)
- ☐ Unsigned
- ☐ None of the other options

ii. **(1.0 pt)** Which of the following interpretations allows us to deduce the sign just by looking at the most significant bit? (Ignore 0)

- ☐ Two's Complement
- ☐ One's Complement
- ☐ Floating Point
- ☐ Biased (for at least 1 choice of bias)
- ☐ None of the other options
- ☐ Sign and Magnitude

(d) (1.0 pt)

How many numbers are written the same way in 32-bit 2's complement and IEEE-754 single-precision floating point (32 bit)?

(e) Please fill out the following table. Write exactly "**N/A**" if the conversion is not possible. Some entries have already been filled out for you. You may assume all binary numbers are 8 bits. If you are writing your answer in hex or binary, make sure to include its prefix; you will not get credit if you forget! Also, do not include the suffix for any representation, i.e. for decimal, base 4, and base 8, just put in the raw number. (For example, if the answer for base 4 is 3210_4, just enter 3210). Please include all necessary leading zeros for any base other than decimal. You must fully simplify your answers.

Convert `0x3C` (Two's Complement) to...

i. (0.5 pt) Decimal

ii. (0.5 pt) Binary (Two's Complement)

iii. (0.5 pt) Base 4 (Two's Complement)

iv. (0.5 pt) Base 8 (Two's Complement)

v. (0.5 pt)

Binary (Biased w/ added bias of -127)

(f) **i.** **(1.0 pt)** For this question, assume that we are using 8-bit numbers! Make sure you fully simplify your answers. Note these problems are in numerical terms, not in terms of magnitude.

What is the distance between the **largest** number in 2's complement and the **largest** number in Sign and Magnitude?

ii. **(1.0 pt)** What is the distance between the **largest** number in 2's complement and the **largest** number in unsigned?

6. Don't Float Away!

Suppose we use an 8-bit floating point format similar to IEEE-754, with 1 sign bit, 3 exponent bits, and 4 significand bits. Assume the bias is -3 and we add the bias. For ALL parts of this question, express your answer a) in decimal, and b) in hex. Make sure you add the prefix to your hex value, fully simplify your answers, and do NOT leave them as fractions. Feel free to plug your fraction into Google to turn it into a decimal value. For all answers, write the exact decimal value, not a rounded one. All solutions have a finite number of decimal digits without rounding!

Quick reminder about intervals: `(` and `)` are exclusive while `[` and `]` are inclusive.

(a) i. (1.5 pt)

What's the gap (aka absolute value of the difference) between the **smallest positive** non-zero denorm and **smallest positive** non-zero norm? (Answer in decimal)

ii. (1.5 pt)

How many Floating Point numbers are in the interval of $(2^1, 2^3)$? (Answer in decimal)

iii. (1.0 pt)

How many positive non-zero denormalized Floating Point numbers can we represent? (Answer in decimal)

(b) Find the smallest positive non-zero denormalized number represented in this new format.

i. (1.0 pt) Decimal:

ii. (1.0 pt) Hex:

(c) Give the nearest representation of π ($\approx 3.14159\ldots$).

i. (3.0 pt) Decimal:

ii. (3.0 pt) Hex (using our floating point representation):

No more questions.