CS61C Summer 2025 — Yokota Midterm

PRINT Your Name: _____

PRINT Your Student ID: _____

PRINT the Name and Student ID of the person to your left: _____

PRINT the Name and Student ID of the person to your right: _____

PRINT the Name and Student ID of the person in front of you: _____

PRINT the Name and Student ID of the person behind you: _____

You have 110 minutes. There are 7 questions of varying credit. (100 points total)

Question:	1	2	3	4	5	6	7	Total
Points:	20	14	11	12	21	22	0	100

For questions with **circular bubbles**, you may select only one choice.

- ○ Unselected option (Completely unfilled)
- ◌ Don't do this (it will be graded as incorrect)
- ● Only one selected option (completely filled)

For questions with **square checkboxes**, you may select one or more choices.

- ☐ You can select
- ☐ multiple squares
- ☑ (Don't do this)

Anything you write outside the answer boxes or you ~~cross out~~ will not be graded. If you write multiple answers, your answer is ambiguous, or the bubble/checkbox is not entirely filled in, we will grade the worst interpretation. For coding questions with blanks, you may write at most one statement per blank and you may not use more blanks than provided.

If an answer requires hex input, you must only use capitalized letters (`0xB0BACAFE` instead of `0xb0bacafe`). For hex and binary, please include prefixes in your answers unless otherwise specified, and do not truncate any leading 0's. For all other bases, do not add any prefixes or suffixes.

As a member of the UC Berkeley community, I act with honesty, integrity, and respect for others. I will follow the rules of this exam.

Acknowledge that you have read and agree to the honor code above and sign your name below:

_____

This page left intentionally (mostly) blank

The exam begins on the next page.

Q1 *Potpourri* 🍲 **(20 points)**

Convert the following numbers from decimal to two's complement 8-bit binary. If the number is not representable in this format, write N/A.

Q1.1 (1 point) 128

0b

Q1.2 (1 point) −128

0b

For Q1.3-Q1.4, select the number representation that would best represent the described variable, given that all representations use 8 bits.

Q1.3 (1.5 points) The current elevation of a person jumping at the top of Mt. Everest (in centimeters above sea level).

◯ Unsigned ◯ Sign Magnitude ◯ 2's Complement ◯ Biased

Q1.4 (1.5 points) `size_t`, a `typedef` in C used to denote the size of an object in bytes.

◯ Unsigned ◯ Sign Magnitude ◯ 2's Complement ◯ Biased

Q1.5 (4 points) Convert the below RISC-V instruction into 32-bit hexadecimal machine code.

```
andi s0 a5 13
```

➡

0x

For Q1.6-Q1.7, select the step of CALL that performs the operation.

Q1.6 (1.5 points) Combines multiple object files into a single executable.

◯ Compiler ◯ Assembler ◯ Linker ◯ Loader

Q1.7 (1.5 points) Performs syntax analysis, optimization, and code generation.

◯ Compiler ◯ Assembler ◯ Linker ◯ Loader

For Q1.8-Q1.10, find the decimal representation of the floating point number described, given that we are using a 16-bit IEEE-754 standard floating-point format with 1 sign bit, 7 exponent bits (with a standard bias of −63), and 8 mantissa bits.

Q1.8 (2 points) `0x6100`. Express your answer as a power of two.

Q1.9 (3 points) The **largest positive non-integer** that can be represented. (Clarification during the exam: largest positive **finite** non-integer)

Q1.10 (3 points) The **smallest positive multiple of 10** that can't be represented.

Q2 *Stringing Along*  **(14 points)**

Q2.1 *Fun With Endianness*

What gets printed if this C code is run on a 32-bit Little-Endian and 64-bit Big-Endian system?

```
1 uint32_t a[] = {
2     0x72657665,
3     0x616c0079,
4     0x72736500,
5     0x00747365
6 };
7 printf("%s\n", (char*) a);
```

Q2.1.1 (2 points) 32-bit Little-Endian system:

Q2.1.2 (2 points) 64-bit Big-Endian system:

Q2.2 *Megastring*

For the remainder of this question, you are given a singly-linked list where each node contains a fixed-size memory buffer of `MAX_LEN` bytes, interpreted as a null-terminated string of variable length. The list continues until the `next` pointer is `NULL`.

You have the following structure:

```
1 #define MAX_LEN 120
2 typedef struct node {
3     char buffer[MAX_LEN];   // guaranteed to store well-formed strings
4     struct node *next;      // next node in the list, or NULL if no next exists
5 } node_t;
```

Useful C `stdlib` function prototypes:

```
1 // Return the length of string s, not including null terminator
2 size_t strlen(const char *s);
3 // Copy at most n bytes of the string in src to dest, and returns dest.
4 // src and dest may not overlap, and dest must be at least n bytes long.
5 // If there is no null byte among the first n bytes of src,
6 // a null terminator will not be copied.
7 // If the length of src is less than n, strncpy writes additional null bytes
8 // to dest to ensure that a total of n bytes are written.
9 char *strncpy(char *dest, const char *src, size_t n);
```

Q2.3-2.9 (8 points) Write a function that concatenates all the strings in a linked list into a single, heap-allocated string. The function must allocate the minimal amount of space to store the combined string, and the linked list should be freed by the end. You may assume that all allocations succeed.

`mega_string`: Concatenates all strings stored in the given linked list into one long string, in the order they are found in the list.		
Arguments	`node_t *head`	The head of the linked list.
Return value	`char *result`	A pointer to the single, heap-allocated string.

```
1  char *mega_string(node_t *head) {
2      size_t total_len = 0;
3      node_t *curr = head;
4      while (curr) {
5          total_len += ____________________;
                        Q2.3
6          ____________________;
               Q2.4
7      }
8      char *result = ____________________;
                      Q2.5
9      char *p = result;
10     curr = head;
11     while (curr) {
12         size_t len = strlen(curr->buffer);
13         ____________________;
               Q2.6
14         p += ____________________;
                  Q2.7
15         ____________________;
               Q2.8
16         curr = curr->next;
17         ____________________;
               Q2.9
18     }
19     return result;
20 }
```

Q2.10 (2 points) What is the maximum length of a string output by `mega_string`, given that `MAX_LEN` = 120 and you are given a linked list of length 10?

Q3 *Chippy's 61Crash(out)* 🐹 **(11 points)**

Chippy the chipmunk is trying to keep track of their stuff, but they're having a bit of trouble tracking memory addresses. Let's help Chippy debug with `gdb`! For your reference, common `gdb` commands are listed below:

Command	Abbreviation	Description
`start`	`start`	Begin running the program and stop at line 1 in `main`.
`step`	`s`	Execute the current line of code, stepping into function.
`next`	`n`	Execute the current line of code, stepping over functions.
`finish`	`fin`	Executes the remainder of the function and returns to the caller.
`print [arg]`	`p`	Prints the value of the argument.
`quit`	`q`	Exits `gdb`.

Note: Chippy has commented their code with *what they intended to do*, **not** what the code actually does.

```
1  typedef struct {
2    char name[8];                  // Byte Offset 0
3    uint8_t acorns;                // Byte Offset 8
4    uint8_t seeds;                 // Byte Offset 9
5
6    // points to the address where the struct is stored
7    struct chipmunk *home;         // Byte Offset 16
8  } chipmunk;
9
10 int main() {
11   chipmunk *chippy = malloc(sizeof(chipmunk));
12   strcpy(chippy->name, "CHIPPY"); // Sets name to CHIPPY
13   chippy->acorns = 8;            // Sets number of acorns
14   chippy->seeds = 16;            // Sets number of seeds
15   chippy->home = chippy;         // Sets chippy's home to their address
16
17   // Set seeds_ptr = &chippy->seeds
18   uint8_t *seeds_ptr = chippy + 9;
19
20   return 0;
21 };
```

Q3.1 (2 points) Chippy wants to confirm that `chippy->home` and `chippy` point to the same address in memory. Which `gdb` commands will show us the address? Assume that the program has stopped just before the return statement. Select all that apply.

- ☐ `p &chippy`
- ☐ `p chippy`
- ☐ `p *chippy`
- ☐ `p chippy->home`
- ☐ `p chippy.home`
- ◯ None of the above

Q3.2-3.4 (3 points) Chippy runs their code, and they track their error down to line 17. Fill in the **gdb** commands that will show the addresses of **chippy->seeds** and **seeds_ptr** below. No commands have been run besides the ones shown. You may fill in at most one command on each line.

```
(gdb) b chippy.c:17
Breakpoint 1 at 0x11ea: file chippy.c, line 17.
(gdb) run
...output omitted...
Breakpoint 1, main () at chippy.c:17
17        uint8_t *seeds_ptr = chippy + 9;
(gdb) p chippy
(chipmunk *) 0x5555555592a0

# 1. print the address of chippy->seeds
# this will output (uint8_t *) 0x5555555592a9:

(gdb) _____
                                        Q3.2

# 2. execute line 17:

(gdb) _____
                                        Q3.3

# 3. print chip_ptr:
# this will output (uint8_t *) 0x5555555592b0

(gdb) _____
                                        Q3.4
```

Q3.5 (2 points) Edit line 17 to fix the pointer arithmetic:

```
uint8_t *seeds_ptr =
```

Suppose we want to let a `struct chipmunk` have a `name` of any length:

```
1  typedef struct {
2    char *name;
3    // rest of the struct is the same as previously defined
4  } chipmunk;
5
6  int main() {
7    chipmunk *chippy =  malloc(sizeof(chipmunk));
8    chippy->name = malloc(sizeof(char) * 22);
9    strncpy(chippy->name, "Chippy the 61Chipmunk", 22);
10
11   // Code omitted
12
13   free(chippy);
14   return 0;
15 }
```

However, even though we get the correct behavior, `valgrind` outputs this:

```
$ valgrind ./chippy
==2068826== Memcheck, a memory error detector
==2068826== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==2068826== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==2068826== Command: ./chippy
==2068826==
==2068826== HEAP SUMMARY:
==2068826==     in use at exit: 22 bytes in 1 blocks
==2068826==   total heap usage: 2 allocs, 1 frees, 46 bytes allocated
==2068826==
==2068826== LEAK SUMMARY:
==2068826==    definitely lost: 22 bytes in 1 blocks
==2068826==    indirectly lost: 0 bytes in 0 blocks
==2068826==      possibly lost: 0 bytes in 0 blocks
==2068826==    still reachable: 0 bytes in 0 blocks
==2068826==         suppressed: 0 bytes in 0 blocks
==2068826== Rerun with --leak-check=full to see details of leaked memory
==2068826==
==2068826== For lists of detected and suppressed errors, rerun with: -s
==2068826== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Q3.6 (4 points) In at most 15 words, explain the memory error `valgrind` is detecting, and why it occurred.

Q4 *Double Double* 🍔 **(12 points)**

(12 points) Write the program `double_double`, defined below:

`double_double`: Doubles a double using integer operations.		
Arguments	`uint64_t x`	The bitwise representation of a `double y`
Return value	`uint64_t`	A `uint64_t` that stores the bitwise representation of the double `2*y`

For example:

```
#include <math.h> // defines constants INFINITY and NAN, equal to inf and nan
double y[] = {123.0, 1e308, INFINITY, NAN}; //Max double = approx. 1.8e308
uint64_t *x = (uint64_t*) y;
for(int i = 0; i < 4; i++)
    x[i] = double_double(x[i]);
printf("%f %f %f %f\n", y[0],y[1],y[2],y[3]); //Prints out "246.0 inf inf nan"
```

You may assume:

- Doubles are defined using IEEE-754 standard double-precision floating point,
- Any constants defined will be interpreted as 64-bit integers (even without the appropriate suffix).
- **You may only use bitwise, basic arithmetic, and comparison operators. In particular, you may not typecast, define float-type variables, use pointers, or use any functions defined in math.h.**
- You may use integer constants, but may not use any float-type constants (e.g. `INFINITY` and `NAN`).
- You may not need to use all the lines/cases.

```
#define INTINF (0x7FF << 52)  // bitwise representation of INFINITY
uint64_t double_double(uint64_t x) {

  switch(_____Q4.1_____) {

      case _____Q4.2_____:

          _____Q4.3_____;

      case _____Q4.4_____:

          _____Q4.5_____;

      case _____Q4.6_____:

          _____Q4.7_____;
      default:

          _____Q4.8_____;
    }
}
```

Q5 *I Speak For The Trees* 🌳 **(21 points)**

Congratulations! You have been hired as the Lorax's assistant, and you've been tasked with ensuring the Truffula trees are healthy using your coding skills.

(4 points) As your first task, implement `power_of_two` as defined below.

Hint: Bitwise operations may be helpful for this question.

`power_of_two`: Determines whether a number is a power of two (i.e. can be represented as 2^n where n is some non-negative integer).		
Arguments	`a0`	A nonzero, unsigned integer.
Return value	`a0`	A boolean value (1 if the input if a power of two, 0 otherwise).

```
1  power_of_two:
2      addi t0 a0 -1
3      _____________________________
                   Q5.1
4      _____________________________
                   Q5.2
5      jr ra
```

(12 points) The Lorax uses the following C `struct` to implement a binary tree:

```
1  typedef struct TreeNode {
2      uint32_t value;
3      struct TreeNode* left;  // NULL if there is no left child
4      struct TreeNode* right; // NULL if there is no right child
5  } TreeNode;
```

Implement the following recursive RISC-V function, `sum_powers_of_two`.

`sum_powers_of_two`: Sums all powers of two contained within a tree.		
Arguments	`a0`	A pointer to the root of a tree, represented by a `TreeNode`.
Return value	`a0`	The sum of all `value`s within a tree that are powers of two. If `a0` is `NULL` or contains no powers of two, return `0`.

You may assume that `power_of_two` is implemented correctly, though it may not match the implementation above.

```
1  sum_powers_of_two:
2      # Prologue (Q5.12)
3      # ...
4      beq a0 x0 ____________________
                       Q5.3
5      mv s0 a0
6      li s1 0
7
8      # Left node
9      ____________________________________
                    Q5.4
10     ____________________________________
                    Q5.5
11     ____________________________________
                    Q5.6
12
13     # Right node
14     ____________________________________
                    Q5.7
15     ____________________________________
                    Q5.8
16     ____________________________________
                    Q5.9
17
18     # Check value
19     ____________________________________
                    Q5.10
20     ____________________________________
                    Q5.11
21
22     beq a0 x0 exit
23     lw t0 0(s0)
24     add s1 s1 t0
25     j exit
26
27 null_case:
28     li s1 0
29
30 exit:
31     mv a0 s1
32     # Epilogue (Q5.12)
33     # ...
34     jr ra
```

Q5.12 (2 points) Which registers need to be saved in the prologue and restored in the epilogue for `sum_powers_of_two` to satisfy calling convention?.

☐ `a0` ☐ `s0` ☐ `s1` ☐ `t0`

☐ `ra` ☐ Other (specify below) ◯ None of the above

Q5.13 (3 points) The Once-ler, in an attempt to sabotage your tree management system, manages to change the values of some trees to unreasonable values. The Lorax has determined that all of the bad values are greater than or equal to `0x61C000`.

Given the below definition of `validate_tree_value`, select all of the options that correctly implement `validate_tree_value`.

`validate_tree_value`: Detects bad `TreeNode` values.		
Arguments	`a0`	`value` contained in a `TreeNode`.
Return value	`a0`	`1` if the input in `a0` is valid (less than `0x61C000`), and `0` otherwise.

☐
```
1 validate_tree_value:
2     li t0 0x61C
3     slli t0 t0 12
4     sltu a0 a0 t0
5     jr ra
```

☐
```
1 validate_tree_value:
2     li t0 0x61C000
3     sub t1 a0 t0
4     slti a0 t1 0
5     jr ra
```

☐
```
1 validate_tree_value:
2     auipc t0 0x61C
3     sltu a0 a0 t0
4     jr ra
```

☐
```
1 validate_tree_value:
2     lui t0 0x61C
3     sltu a0 a0 t0
4     jr ra
```

◯ None of the above

Q6 *2-Way Skewed Direct-Mapped Cache* 💵 **(22 points)**

Q6.1 *AMAT Warmup*

The Annapurna Labs Graviton RISC CPU used in Amazon's Web Services EC2 Cloud Computing servers has the following cache performance.

L1 Instruction Cache	
Miss Rate	2%
L1$ Miss Penalty	11 cycles
Hit Time	4 cycles

Q6.1.1 (2 points) What is the AMAT for the L1 Instruction Cache in cycles?

cycles

Q6.2 *Optimizing a Cache*

We know that associativity reduces conflict misses, but depending on the workload, there can still be many conflict misses due to the temporal and spatial locality of the data being cached. Jim proposes a few new cache designs that attempt to solve this problem.

We have a 4 KiB 2-way set-associative cache with a block size of 64 bytes on a system with a 64 KiB address space.

Q6.2.1 (1.5 points) What is the tag-index-offset breakdown for the 2-way set-associative cache defined above?

T: bit(s) | I: bit(s) | O: bit(s)

Below is some sample C code that computes a dot product. Assume that the cache is cold immediately before running the **for** loop on line 18. The cache implements write-back using an LRU replacement policy.

```
1  #define N 512                    // Vector size
2  #define LINE_SIZE 64             // 64 bytes per cache line
3  #define CACHE_SIZE (4 * 1024)    // 4 KiB total cache
4  #define ASSOC 2                  // 2 ways
5  #define NUM_SETS                 // Number of sets in the cache, value omitted
6
7  #define STRIDE (LINE_SIZE * NUM_SETS)  // 2048 bytes stride to hit same set
8  #define STRIDE_INTS (STRIDE / sizeof(int))
9
10 int main() {
11     int *a = malloc(N * sizeof(int));  // assume that a is aligned
12     int *b = malloc(N * STRIDE);       // assume that b is aligned
13
14     // Code filling *a and *b have been omitted.
15
16     // Compute dot product
17     register int sum = 0; // sum stored in register
18     for (register int i = 0; i < N; i++) {
19         sum += a[i] * b[i * STRIDE_INTS];
20     }
21
22     free(a);
23     free(b);
24     return 0;
25 }
```

Q6.2.2 (3 points) For memory accesses to **only** `a[i]` throughout the lifetime of the program above, how many cache accesses are hits? How many accesses are compulsory misses? How many accesses are non-compulsory misses?

Cache Hits:

Compulsory Misses:

Non-compulsory Misses:

Q6.2.3 (3 points) For memory accesses to **only** `b[i * STRIDE_INTS]` throughout the lifetime of the program above, how many cache accesses are hits? How many accesses are compulsory misses? How many accesses are non-compulsory misses?

Cache Hits:
Compulsory Misses:
Non-compulsory Misses:

To improve the performance of our cache, Jim tries a new design.

Jim sets up two distinct direct-mapped L1 caches (a left bank and a right bank). Each bank is 2 KiB in size. When a new cache block is added, we attempt to place it in the left bank. If the slot in the left bank is full, we attempt to place data in the correct spot in the right bank. If both slots are full, then we replace the least recently used block among the two slots we checked (LRU replacement policy).

Q6.2.4 (2 points) What is the hit rate for memory accesses to `a[i]` and `b[i * STRIDE_INTS]` for the newly proposed cache design above? Evaluate using the same C code as the one provided in the previous parts.

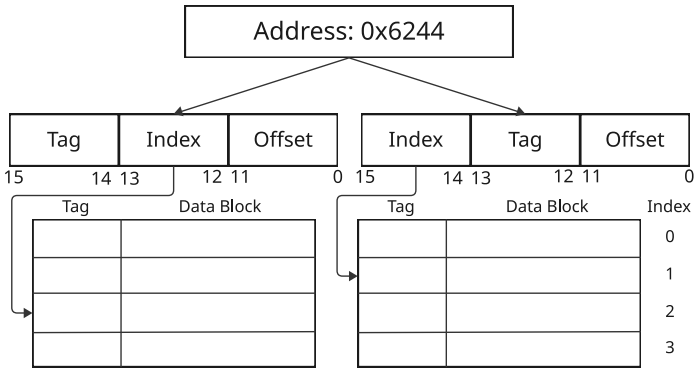
`a[i]` Hit Rate:
`b[i * STRIDE_INTS]` Hit Rate:

Show your reasoning in the box below. This is optional, but we may use your answers in the box to award partial credit.

Q6.2.5 (2.5 points) Jim realizes that he can make the cache more efficient if the left and right banks use different schemes to decide their index. Jim changes the cache setup so that the right bank computes its index using the top bits of the tag instead of the bottom bits (in other words, we split addresses in ITO order instead of TIO order, as shown below).

You may assume:

- Both caches use the same number of index bits. **For this subpart only, assume that we split into a TIO breakdown of 2:2:12.**
- The left bank still uses TIO order as before.
- The placement and replacement policies from the previous subpart still apply (Insert in the left bank before the right bank and replace in LRU order).



Memory Accesses:

1. `0x6244`
2. `0xF543`
3. `0x3524`
4. `0x6554`
5. `0x2204`
6. `0xC953`
7. `0x8206`
8. `0xB361`
9. `0xF636`
10. `0x5865`

For example, the above diagram shows the cache slots associated with address `0x6244`.

Given the ten memory accesses above, show the **final state** of the cache after all addresses have been accessed. Assume the cache starts cold. For each memory access:

- If the memory access is a cache hit, do nothing.
- Otherwise look at the newly inserted block.
  - If the block is inserted into an empty slot, write the accessed memory address in that slot.
  - If a cache block insertion replaces another block, cross out the address of the replaced block, and write the accessed memory address in the same slot.

The first five accesses have been done for you.

2-Banked Cache with Alternative Indexing		
Left Bank	**Right Bank**	**Index**
0xC953	~~0x3524~~ 0x2204	0
0x5865		1
0x6244	0x8206	2
~~0xF543~~ 0xB361	0xF636	3

Q6.2.6 (4 points) Is it better to use a single ITO direct-mapped cache or a single TIO direct-mapped cache? Explain. Only the first three sentences of your answer will be graded.

Your reasoning should be workload-independent and representative of common use cases or memory access patterns in software.

Q6.2.7 (4 points) Is it better to have a TIO-TIO cache (design used in Q6.2.4) or a TIO-ITO cache (design used in Q6.2.5)? Explain. Only the first three sentences of your answer will be graded.

Your reasoning should be workload-independent and representative of common use cases or memory access patterns in software.

Q7 *61C-cret* ? **(0 points)**

These questions will not be assigned credit. Feel free to leave them blank.

Q7.1 What is the $ECRET? Hint: `2.1.2(2.2(6.2.5[0]))`

Q7.2 If there's anything else you want us to know, or you feel like there was an ambiguity in the exam, please put it in the box below.

For ambiguities, you must qualify your answer and provide an answer for both interpretations. For example, "if the question is asking about A, then my answer is X, but if the question is asking about B, then my answer is Y". You will only receive credit if it is a genuine ambiguity and both of your answers are correct. We will only look at ambiguities if you request a regrade.

Alternatively, draw a chipmunk with a snake headband eating a double double jumping on a trampoline freaking out, with truffula trees and a gravitron in the background!