CS61C Summer 2025 — Yokota Midterm

Solutions last updated: Tuesday, July 22, 2025

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

Solution: N/A

Q1.2 (1 point) −128

0b

Solution: 0b10000000

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
```

$\rightarrow$ 0x

Solution: 0x00D7F413

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.

2^{34}

Solution: `0x6100 = 0b0 1100001 00000000`.

Sign bit = 0 $\longrightarrow$ positive.

Exponent bits = $\texttt{1100001}_2$ = 97 $\longrightarrow$ unbiased exponent = 97 − 63 = 34.

Mantissa bits = `0`: Mantissa = 1.0

Value = 1.0×2^{34}.

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

255.5

Solution: 2^8 is when the step size becomes 1. Now subtract 0.5 or 2^{-1} to get the first non-representable non-integer.

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

1030

Solution: 2^9 is when the step size becomes 2. However we realize that every multiple of 10 is representable between 2^9 and 2^{10} as only odd numbers are not representable if the step size is 2. Then we go to 2^{10}, which is when the step size becomes 4. We see 1024 is representable, add 4 so 1028 is, and the first non representable multiple of 10 is 1030.

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?

```
uint32_t a[] = {
    0x72657665,
    0x616c0079,
    0x72736500,
    0x00747365
};
printf("%s\n", (char*) a);
```

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

Solution: every
Order the bytes according to the endianness of the system, then read until the first null terminator is reached.

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

Solution: reveal
Same as above.

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:

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

Useful C `stdlib` function prototypes:

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

```
char *mega_string(node_t *head) {
    size_t total_len = 0;
    node_t *curr = head;
    while (curr) {

        total_len += strlen(curr->buffer);
                               Q2.3

        curr = curr->next;
                 Q2.4
    }

    char *result = calloc(total_len + 1, 1);
                               Q2.5
    char *p = result;
    curr = head;
    while (curr) {
        size_t len = strlen(curr->buffer);

        strncpy(p, curr->buffer, len+1);
                        Q2.6

        p += len ;
              Q2.7

        node_t *temp = curr;
                  Q2.8
        curr = curr->next;

        free(temp);
             Q2.9
    }
    return result;
}
```

Solution: This solution concatenates all the null-terminated strings from a linked list of fixed-size buffers into a single dynamically allocated string while freeing the list nodes. It first traverses the list to compute the total length of all strings, then allocates a buffer of size total_len + 1 to hold the concatenated result. In the provided code, calloc is used to allocate and zero-initialize the buffer, so the final null terminator at the end of the string is already guaranteed. It then iterates through the list again, copying each string into the result using **`strncpy`** with **`len + 1`** to include the intermediate null terminators from each node's buffer, though advancing only by **`len`** to overwrite those intermediate nulls in subsequent iterations, since these strings are 'concatenated' (and thus we do not want null terminators between them). Note that because of this, one can actually use **`len`** with **`strncpy`**, due to calloc's zeroing out of memory. If malloc were used instead of calloc, the buffer would not be zero-initialized - so strncpy would need to copy **`len + 1`** bytes on each call to ensure a null terminator is copied into the end of the megastring.

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?

```
1190
```

Solution: 119*10 + 1 120 - 1 possible chars per string (-1 for null terminator), 10 strings.

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.

```c
typedef struct {
  char name[8];                    // Byte Offset 0
  uint8_t acorns;                  // Byte Offset 8
  uint8_t seeds;                   // Byte Offset 9

  // points to the address where the struct is stored
  struct chipmunk *home;           // Byte Offset 16
} chipmunk;

int main() {
  chipmunk *chippy = malloc(sizeof(chipmunk));
  strcpy(chippy->name, "CHIPPY"); // Sets name to CHIPPY
  chippy->acorns = 8;              // Sets number of acorns
  chippy->seeds = 16;              // Sets number of seeds
  chippy->home = chippy;           // Sets chippy's home to their address

  // Set seeds_ptr = &chippy->seeds
  uint8_t *seeds_ptr = chippy + 9;

  return 0;
};
```

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.

- A `p &chippy`
- [✓ `p chippy`]
- [✓ `p *chippy`]
- [✓ `p chippy->home`]
- E `p chippy.home`
- F 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) p &chippy->seeds
             Q3.2

# 2. execute line 17:

(gdb)  n
      Q3.3

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

(gdb) p seeds_ptr
           Q3.4
```

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

```
uint8_t *seeds_ptr = (uint8_t*) chippy + 9
```

Solution: The pointer `chippy` must be cast to a `uint8_t*` before doing pointer arithmetic. This cast ensures we increment by the size of a `uint8_t*` instead of a `chipmunk` struct.

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

```
typedef struct {
  char *name;
  // rest of the struct is the same as previously defined
} chipmunk;

int main() {
  chipmunk *chippy =  malloc(sizeof(chipmunk));
  chippy->name = malloc(sizeof(char) * 22);
  strncpy(chippy->name, "Chippy the 61Chipmunk", 22);

  // Code omitted

  free(chippy);
  return 0;
}
```

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.

`did not free chippy->name`

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((x >> 52) & 2047) {
                 Q4.1

       case  0 :
            Q4.2

           return (x << 1) | (x & (1 << 63));
                             Q4.3

       case 2047:
            Q4.4

           return x;
             Q4.5

       case 2046:
            Q4.6

           return INTINF | (x & (1 << 63));
                            Q4.7
       default:

           return x + (1 << 52);
                       Q4.8
    }
}
```

Solution: For this question, it is easiest to think about a floating point number as a value of the form $2^{\text{exp}} * y$ for some value y. For most normal numbers, double this is $2^{\text{exp}+1} * y$, so we can compute this by increasing the exponent by 1; this can be done by adding `1<<52` to our number, which corresponds to the bottom bit of the exponent in a double. There's a few cases, though, which cannot be computed in this manner:

- For infinity and NaN, doubling does nothing. Thus, if the exponent is all 1s, we can simply return x directly. Note that both infinity AND -infinity are accounted for in this.
- For denormalized numbers, we can't simply add 1 to the exponent. In this case, we use the fact that denormalized numbers (and exponent 1) use the same linear scale, so we can left-shift the mantissa by 1 to get the correct double (just as how we can double integers by left-shifting). We do need to take care to keep the sign bit from the original number, but the exponent can also be left-shifted, since it's already 0 in this case.
- For numbers greater than half the maximum double, we need to set the result to the appropriate infinity, since just incrementing the exponent by 1 might yield a NaN. This occurs precisely when we have an exponent one less than the maximum exponent. We again need to be careful to maintain the sign of the original number.

Notably, all of the above cases are contingent only on the exponent bits of x, and exactly one exponent is used for each case. As such, we can isolate the exponent bits and switch on them to yield the correct doubled number.

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).

```
power_of_two:
    addi t0 a0 -1

    and t0 t0 a0
        Q5.1

    slti a0 t0 1
        Q5.2
    jr ra
```

Solution: Making use of the fact that a power-of-two in binary is a `1` followed by some number of `0`s, the solution to this problem detects this pattern using bitwise operations. The first instruction is given and subtracts 1 from the input - this value would be represented in binary as a series of `1`s. So, if the input is a power-of-two, the preset registers would contain:

a0 | `0b10000..00`

t0 | `0b01111..11`

To then detect a power-of-two, we can perform an **`and`** operation between these two values, which returns `0` only for powers-of-two. Then, the last step is to flip this value (`0` to `1`, non-zero to `0`) using **`slti`**.

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

```
typedef struct TreeNode {
    uint32_t value;
    struct TreeNode* left;  // NULL if there is no left child
    struct TreeNode* right; // NULL if there is no right child
} 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.

```
sum_powers_of_two:
    # Prologue (Q5.12)
    # ...

    beq a0 x0 null_case
                 Q5.3
    mv s0 a0
    li s1 0

    # Left node

    lw a0 4(s0)
        Q5.4

    jal sum_powers_of_two
             Q5.5

    addi s1 s1 a0
         Q5.6

    # Right node

    lw a0 8(s0)
        Q5.7

    jal sum_powers_of_two
             Q5.8

    addi s1 s1 a0
         Q5.9

    # Check value

    lw a0 0(s0)
        Q5.10

    jal power_of_two
           Q5.11

    beq a0 x0 exit
    lw t0 0(s0)
    add s1 s1 t0
    j exit

null_case:
    li s1 0

exit:
    mv a0 s1
    # Epilogue (Q5.12)
    # ...
    jr ra
```

Solution: This implementation recursively calls `sum_powers_of_two` to check every node in the tree.

Since the input is a pointer, a load needs to be performed to get each of the struct members from memory. Each member is 4 bytes large, so the input pointer can be offset by 0 to get the value, 4 to get the `left` pointer, and 8 to get the `right` pointer.

Then, we must accumulate the results of calling `sum_powers_of_two` on each child node, along with the current node if its a power-of-two. Since the latter is implemented for us, we must only concern ourselves with the former.

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?.

A `a0` [✓ `s0`] [✓ `s1`] D `t0`

[✓ `ra`] F Other (specify below) G None of the above

Solution: By calling convention, any s-type registers that are used must be saved and restored before returning. `s0` and `s1` are used in the given instructions and must therefore be saved. `ra` is also changed when `sum_powers_of_two` and `power_of_two` are called - so it must also be saved and restored to allow `sum_powers_of_two` to return to the correct address.

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.

[✓]

```
validate_tree_value:
    li t0 0x61C
    slli t0 t0 12
    sltu a0 a0 t0
    jr ra
```

B

```
validate_tree_value:
    li t0 0x61C000
    sub t1 a0 t0
    slti a0 t1 0
    jr ra
```

C

```
validate_tree_value:
    auipc t0 0x61C
    sltu a0 a0 t0
    jr ra
```

[✓]

```
validate_tree_value:
    lui t0 0x61C
    sltu a0 a0 t0
    jr ra
```

(E) None of the above

Solution: Options 1 and 4 correctly implement the function by correctly loading `0x61C000` into t0, followed by an unsigned **`sltu`** operation. Option B is incorrect because it uses **`slti`**, which will treat the input as a signed number. Sufficiently large inputs would therefore be treated as negative values and result in erroneous outputs. Option C is incorrect because it uses **`auipc`**. While **`auipc`** is similar to **`lui`**, **`auipc`** also adds the value of the program counter to the destination register, potentially also resulting in erroneous outputs.

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

Solution: $4 + 0.02 * 11 = 4.22\text{ns}$

Rubric:
- Partial: Correct answer, wrong or un-simplified format [/1]
- Fully Correct [/2]

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: 5 bit(s)	I: 5 bit(s)	O: 6 bit(s)

Solution: 64 KiB address space => $\log_2(64 * 1024 \text{ bytes}) = 16$ bits for the address.

TIO breakdown:

T: addr size - I - O = $16 - 5 - 6 = 5$

I: $\log_2(\# \text{ lines}) = \log_2\left(\frac{\frac{\$ \text{ size}}{\text{line size}}}{\# \text{ ways}}\right)$ = $\log_2\left(\frac{\frac{2^{12}}{2^6}}{2}\right) = \log_2(32) = 5$

O: $\log_2(\# \text{ bytes in line}) = 6$

Rubric:
- T: +0.5 pt (all or nothing)
- I: +0.5 pt (all or nothing)
- O: +0.5 pt (all or nothing)

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: 480

Compulsory Misses: 32

Non-compulsory Misses: 0

Solution: Hits:

Correct: 480 Hits

Total Array Accesses: 512

$512 * 4 = 2048$ Bytes accessed

Hit: 512 accesses - (16 sets will be used $*$ 2 compulsory misses / set) = $512 - 32 = 480$ hits

(Why 16 sets?):

2048 bytes accessed / 64 bytes per line = 32 blocks.

We have a 2 way-set associative cache => 32 / 2 = 16 sets used.

Compulsory Misses:

Correct: 32 Compulsory Misses

Total Accesses: 512

Compulsory Misses: 512 - 480 hits = 32 misses

Non-compulsory Misses:

Correct: 0 Non-Compulsory Misses

Total Accesses: 512

Hits: 480

Compulsory Misses: 32

Non-Compulsory Misses: $512 - 480 - 32 = 0$

Rubric: Cache Hits [+1]:
1. All or nothing

Compulsory Misses [+1]:
1. All or nothing

Non-compulsory Misses [+1]:
1. All or nothing

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: 0

Compulsory Misses: 512

Non-compulsory Misses: 0

Solution: Hits: 0 Hits

In this cache, accessing data in the cache `STRIDE` = 2048 = 0x800 bytes apart causes the index to be always 0 while the tag changes, causing misses on every access.

For example:

Access 0x0000: T = 00000 I = 00000 O = 000000

Access 0x0800: T = 00001 I = 00000 O = 000000

Access: 0x1000: T = 00010 I = 00000 O = 000000

Access: 0x1800: T = 00011 I = 00000 O = 000000

Compulsory Misses:

Correct: 512 Compulsory Misses

We bring in a new block during every memory access. This causes a compulsory miss. Since we are only accessing forwards (ie: `i` is strictly incrementing), we never re-access any blocks we newly brought in. Meaning we do not have any Non-compulsory misses.

Non-Compulsory Misses: Correct: 0 Non-Compulsory Misses

512 accesses - 0 hits - 512 compulsory misses = 0 non-compulsory misses

Another way to think about this: We only access elements in incremental order, so the data we brought into the cache is never accessed again.

Rubric: Cache Hits [/1]:
1. All or nothing

Compulsory Misses [/1]:
1. All or nothing

Non-compulsory Misses; [/1]:
1. All or nothing

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.

Solution: **`a[i] Hit Rate`:**

Correct: 480/512 = 15/16 = 0.9375 = 93.75%

This can be seen from the fact that a 2-way set-associative cache is basically a 2-banked direct-mapped cache. Each bank is 2 KiB here. Therefore, the total size of this banked cache is 4 KiB, which is the same size as the 2-way set-associative cache above.

Similar to a 2-way set-associative cache, we place in the left bank (or think of this as way 1) first, then the right (think: way 2). The LRU replacement policy remains the same as the 2-way set-associative cache above.

Therefore, we essentially have a 2-way set-associative cache here. The workload statistics from above carry over to this question and should remain the same.

Rubric:

- +1 - Fully correct
- +1 - Double Jeopardy Prevention: Incorrect solution; But work in the box below shows a hit rate calculation in agreement with your solution/numbers in Q6.2.2, even if any/all of your answers for Q6.2.2 were incorrect. If you showed no work, yet your calculation reflected the correct hit rate calculation using your answer from Q6.2.2, submit a regrade request.
- +0.9 - Partial Credit: Answer correct but not simplified
- +0.5 - Partial Credit: Minor error or arithmetic error
- +0 - Incorrect. Work shown but not worthy of partial credit. Student work makes no advance towards calculating hit rate of `a[i]` **OR** Student work has an error in approach or shows a misunderstanding of the core concept tested in the question.

`b[i * STRIDE_INTS] Hit Rate`: Correct: 0 Hits / 512 Accesses = 0% Hit rate

This can be seen from the fact that a 2-way set-associative cache is basically a 2-banked direct-mapped cache. Each bank is 2 KiB here. Therefore, the total size of this banked cache is 4 KiB, which is the same size as the 2-way set-associative cache above.

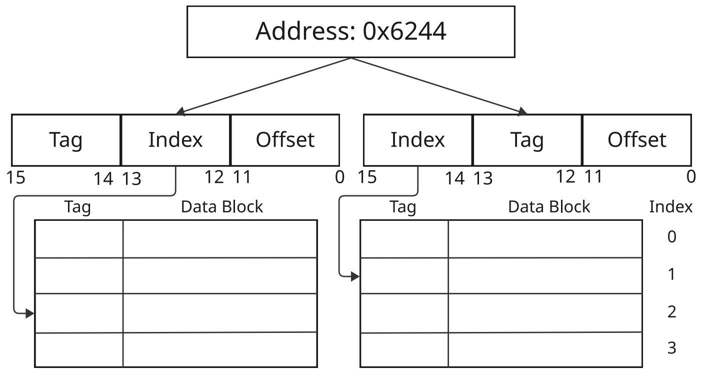
Similar to a 2-way set-associative cache, we place in the left bank (or think of this as way 1) first, then the right (think: way 2). The LRU replacement policy remains the same as the 2-way set-associative cache above.

Therefore, we essentially have a 2-way set-associative cache here. The workload statistics from above carry over to this question and should remain the same.

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	6.	0xC953
2.	0xF543	7.	0x8206
3.	0x3524	8.	0xB361
4.	0x6554	9.	0xF636
5.	0x2204	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.

Solution: Fully Correct:

States: TIO direct-mapped design is better.

With reasoning: Middle bits vary more typically, so using a TIO-organized cache reduces conflicts and replacements. Upper bits often remain constant within small address regions, leading to poor index distribution and higher misses.

TIO design is good for **spatial locality** - Nearby addresses (ex, from arrays or loops) are likely to differ in middle bits than upper bits, which means indexing with upper bits (ITO) will result in many addresses mapping to the same set.

Rubric:

- Too long to include here, see Gradescope.

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.

Solution: Fully Correct:

States: It is better to have a TIO-ITO design.

With reasoning:

TIO-ITO has better performance across a larger selection of workloads. The design might result in more hits if you have a workload that consistently thrashes the left TIO cache because you can rely on the right ITO cache to potentially hit.

Example workload: C code above with both ordinary incremental accesses such as `a[i]` **and** strided accesses such as `b[i  * STRIDE_INTS]`.

Rubric:
- Too long to state here, check Gradescope.

[1][2]

[1]Fun fact: This banked cache question was inspired by the 2-way skewed-associative cache proposed by André Seznec at the 20th annual ACM International Symposium on Computer Architecture (ISCA) in 1993. You can read more about it here: https://dl.acm.org/doi/pdf/10.1145/165123.165152

[2]Fun fact #2: There was another more difficult version of this question (oops…) that allowed you to evict any piece of data from the cache, regardless of whether they were in the set that you were accessing – achieving true "LRU" replacement across your entire cache. That version of the question (which, in hindsight, was good that we cut out of the exam) was modeled after the ZCache proposed by Daniel Sanchez and Christos Kozyrakis at the 43rd Annual IEEE/ACM Symposium on Microarchitecture (MICRO-43) in 2010. You can read more about it here: https://people.csail.mit.edu/sanchez/papers/2010.zcache.micro.pdf

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]))`

Solution: The $ECRET will not be revealed until someone determines the $ECRET. Hints and discussion will be on Ed.

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!