

University of Washington — Computer Science & Engineering  
**CSE 351, Summer 2019 — Final Exam**

Friday, August 23rd, 2019

Name: \_\_\_\_\_

Student ID: \_\_\_\_\_

Name of student to your left: \_\_\_\_\_

Name of student to your right: \_\_\_\_\_

I certify that all work is my own. I had no prior knowledge of exam contents nor will I share the contents with any student in CSE 351 who has not yet taken the exam. Violation of these terms may result in a failing grade.  
*(Please sign below.)*

Signature: \_\_\_\_\_

### Instructions

- You may fill out this page, but **do not turn the page until 10:50am**.
- This exam is closed-book, except for one *handwritten* double-sided 8.5×11” note sheet. Calculators, cell phones, smart watches, notes written underneath your sleeves, Google Glasses, Hololens, neural links, and any other futuristic devices are not allowed.
- You have 60 minutes to complete the exam. Please stop promptly at 11:50am.
- The last page of the exam is a reference sheet. Please detach it before turning in your exam.
- Write your Student ID number (*not* your UW NetID) on the top-right corner of each page.
- We will scan your exams to grade them. Please write *clearly* and *legibly*.
- There are 7 questions, totaling 75 points, across 10 pages (including this one).

### Advice

- Read the questions thoroughly before answering.
- Write down your thoughts and intermediate steps so that you can get partial credit. But be sure to clearly indicate your final answer.
- Questions are not necessarily in order of difficulty. Skip around or read ahead. Make sure you have a chance to attempt all the questions.
- *Relax*. You are here to learn 😊.

Question:	1	2	3	4	5	6	7	Total
Points:	14	18	8	14	11	9	1	75

## 1. (14 points) A Pretty structured Question

Your instructor decides that his company, WolfBytes™, needs a new website to market their Intel clone CPUs. He's decided to write the website in C and wants to use a linked list of structs, the definition of which is shown below, to store their product information.

```
struct cpu {
    float    clock_speed;
    struct   cpu *next;
    short    num_cores;
    int      cache_size;
    short    cache_assoc;
};
```

- (a) (5 points) We've come out with a new CPU and want to add to our catalog. Our new CPU has a clock speed of 2.4 GHz. Given a pointer `start` to the beginning of our catalog list, create a new struct `cpu` on the *heap*, set its clock speed, and add it to the *end* of the list. You don't need to set the rest of the parameters or `#include` any libraries.

```
void add_new_cpu(struct cpu *start) {
    struct cpu *cpu = (struct cpu *) malloc(sizeof(struct cpu)) ;
```

**Solution:**

```
    cpu->clock_speed = 2.4;

    while (start->next != NULL) {
        start = start->next;
    }

    start->next = cpu;
```

```
}
```

For the following parts, assume x86-64.

- (b) (6 points) How much space does an instance of struct `cpu` use? How many bytes are lost to internal and external fragmentation?

Total Space	Internal Fragmentation	External Fragmentation
<u>32</u> bytes	<u>6</u> bytes	<u>6</u> bytes

- (c) (3 points) By rearranging the elements, we may be able to reduce the size of a struct `cpu`. Give the minimized size in bytes.

**Solution:** 24 bytes

2. (18 points) Cache Ya Later!

WolfBytes's new entry-level CPU, GREYWOLF, uses the following cache parameters:

- A 2-way set-associative cache with 8 total sets and 4 byte cache blocks.
- Write-back, write-allocate, and least-recently-used policies.
- 10-bit wide physical addresses.

(a) (1 point) What is the total size of the cache in bytes?

(a) 64 bytes

(b) (3 points) How many bits do we use for the Tag, Index, and Offset?

Tag	Index	Offset
<u>5</u>	<u>3</u>	<u>2</u>

(c) (10 points) Now simulate, *in order*, each of the following 1-byte memory accesses. Fill out the table with the binary representation of the memory address, whether the access resulted in a cache hit or miss, and whether the access resulted in *any* data being written to memory. Assume that the cache starts cold, but that each access updates the cache. The first entry is completed for you.

Access	Binary Address	Cache hit?	Data written to memory?
Read 0x023	0b00 0010 0011	No	No
Read 0x028	<u>0b00 0010 1000</u>	<u>No</u>	<u>No</u>
Write 0x14B	<u>0b01 0100 1011</u>	<u>No</u>	<u>No</u>
Read 0x02A	<u>0b00 0010 1010</u>	<u>Yes</u>	<u>No</u>
Read 0x369	<u>0b11 0110 1001</u>	<u>No</u>	<u>Yes</u>

(d) (4 points) Give one benefit and one drawback of using a *fully-associative* cache as opposed to a *set-associative* cache.

**Solution:** Benefit: impossible for conflict misses to occur  
 Drawback: more complicated to implement in hardware (and therefore, generally slower)

## 3. (8 points) If You See a fork In The Road—Take It!

You are given the following snippet of code:

```
void pillow() {
    int x = 8;

    if (fork()) {
        x++;
        printf("%d ", x);
        if (!fork()) {
            printf("%d ", x - 3);
            return;
        }
    } else {
        x--;
    }

    printf("%d ", x);
}
```

- (a) (3 points) Give three possible outputs from the above code.

**Solution:** Many solutions possible. At least one “9” must be printed before “6”.

- (b) (2 points) How many total processes are created?

(b) 3

- (c) (3 points) What happens to each of the child processes after the parent process terminates? How can the parent prevent this?

**Solution:** They will become zombies and will eventually be reaped by `init` or `systemd`. The parent should explicitly reap its children using `wait`.

4. (14 points) Let's Get Virtual (Memory)

WolfBytes's midrange CPU, TIMBERWOLF, uses the following parameters for virtual memory:

- 24-bit virtual addresses, 256 KiB of RAM with 512-byte pages.
- A fully associative 8-entry TLB with LRU replacement.

(a) (4 points) Calculate the following values:

Page offset width	# of physical pages	# of virtual pages	TLBT width
<u>9 bits</u>	<u>512</u>	<u><math>2^{15}</math></u>	<u>15 bits</u>

One process running on the CPU uses a page-aligned array `arr` containing 2048 shorts in the code shown below. Assume that the entirety of `arr[]` starts on disk.

```
void pandas(short arr[], short s) {
    for (int i = 0; i < 2048; i += 4) {
        arr[i] += s;
    }
}
```

(b) (2 points) What is the stride (in bytes) of memory accesses to `arr` *between each iteration of the loop*?

(b) 8

(c) (3 points) What is the TLB hit rate?

(c) 127/128

(d) (3 points) What is the page table hit rate?

(d) 0

(e) (2 points) We've said in lecture that pages are analogous to cache blocks (i.e., they are the unit of data transfer between memory and disk). *Why are pages significantly larger than cache blocks?*

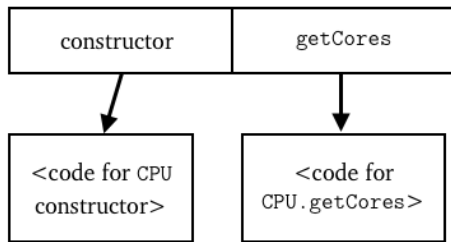
**Solution:** Since disk is so much slower than memory, we want to pull as much data as possible in one go to avoid disk accesses as much as possible.

5. (11 points) A Nice Hot Cup of Java

WolfBytes has gotten wind of this fancy new language called “Java” and has decide to re-write their website using it. They’ve written two classes to store information about their CPUs:

<pre>class CPU {     float clockSpeed;     int cacheSize;     int cacheAssoc;      int getCores() {         return 1;     } }</pre>	<pre>class MultiCoreCPU extends CPU {     int numberOfCores;     float[] coreSpeeds = new float[16];      int getCores() {         return numberOfCores;     }      float[] getCoreSpeeds() {         return coreSpeeds;     } }</pre>
---	--

(a) (4 points) The vtable for CPU is shown below. Annotate the diagram with the *changes* that we would need to make for the vtable of MultiCoreCPU.



**Solution:** getCores should point to code for MultiCoreCPU.getCores and there should be a new entry at the end of the table for MultiCoreCPU.getCoreSpeeds

You may assume that the alignment for this JVM implementation is the same as C on x86-64, and that fields are stored in memory in the order that they are declared.

(b) (2 points) How much space does an instance of CPU take up?  
 (b) \_\_\_\_\_ **32B**

(c) (3 points) How much space does an instance of MultiCoreCPU take up?  
 (c) \_\_\_\_\_ **40B**

(d) (2 points) Give an example of something that is allowed in C, but *not* in Java, because it would prevent the garbage collector from working properly.

**Solution:** pointers to middle of structs/objects, casting pointers to other types, etc.

## 6. (9 points) Don't You Forget About Me(mory)

Consider the following C code:

```

1 int *foo() {
2     int **bar = (int **) malloc(sizeof(int) * 4);
3     bar++;
4
5     int *quux = bar[0];
6     free(bar);
7     return 0;
8 }
```

- (a) (3 points) Assume that, in the code shown above, the call to `malloc` succeeds and that `bar` and `quux` are stored in memory (i.e., not in registers). Fill in the following blanks with “>”, “<”, or “unknown” to compare the values returned by the following expression right before the return statement.

<code>&amp;bar</code>	<u>        &gt;        </u>	<code>&amp;foo</code>
<code>bar</code>	<u>        &lt;        </u>	<code>&amp;quux</code>
<code>quux</code>	<u>    <b>unknown</b>    </u>	<code>&amp;quux</code>

- (b) (4 points) The code shown above contains two memory-related errors. Describe the two errors, including the line numbers on which they occur.

**Solution:** On line 2: `malloc` should be given `sizeof(int *) * 4`, since we want a list of `int` pointers.  
On line 6: freeing `bar` after changing the pointer.

- (c) (2 points) Give a *specific* advantage of using an explicit free list as opposed to an implicit list when writing a memory allocator.

**Solution:** You don't need to search the entire list to find a free block, just the known free blocks.

Student ID: \_\_\_\_\_

7. (1 point) I Want Free Points!!

If you still have time left over, draw something cool on this page?

**Solution:** All answers accepted.

Thank you for a great quarter! Have an awesome fall! 😊



# CSE 351 Reference Sheet (Final)

Binary	Decimal	Hex
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

$2^0$	$2^1$	$2^2$	$2^3$	$2^4$	$2^5$	$2^6$	$2^7$	$2^8$	$2^9$	$2^{10}$
1	2	4	8	16	32	64	128	256	512	1024

SI Size	Prefix	Symbol	IEC Size	Prefix	Symbol
$10^3$	Kilo-	K	$2^{10}$	Kibi-	Ki
$10^6$	Mega-	M	$2^{20}$	Mebi-	Mi
$10^9$	Giga-	G	$2^{30}$	Gibi-	Gi
$10^{12}$	Tera-	T	$2^{40}$	Tebi-	Ti
$10^{15}$	Peta-	P	$2^{50}$	Pebi-	Pi
$10^{18}$	Exa-	E	$2^{60}$	Exbi-	Ei
$10^{21}$	Zetta-	Z	$2^{70}$	Zebi-	Zi
$10^{24}$	Yotta-	Y	$2^{80}$	Yobi-	Yi

## Sizes

C type	Suffix	Size
char	b	1
short	w	2
int	l	4
long	q	8

## IEEE 754 FLOATING-POINT STANDARD

Value:  $\pm 1 \times \text{Mantissa} \times 2^{\text{Exponent}}$

Bit fields:  $(-1)^S \times 1.M \times 2^{(E-\text{bias})}$

where Single Precision Bias = 127,

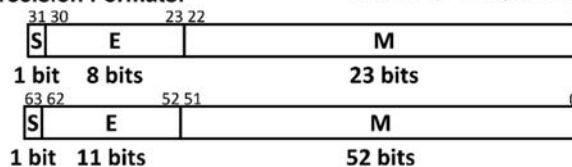
Double Precision Bias = 1023.

## IEEE 754 Symbols

Exponent	Fraction	Object
0	0	$\pm 0$
0	$\neq 0$	$\pm$ Denorm
1 to MAX - 1	anything	$\pm$ Fl. Pt. Num.
MAX	0	$\pm\infty$
MAX	$\neq 0$	NaN

S.P. MAX = 255, D.P. MAX = 2047

## IEEE Single Precision and Double Precision Formats:



## Assembly Instructions

<b>mov a, b</b>	Copy from a to b.
<b>movs a, b</b>	Copy from a to b with sign extension. Needs <i>two</i> width specifiers.
<b>movz a, b</b>	Copy from a to b with zero extension. Needs <i>two</i> width specifiers.
<b>lea a, b</b>	Compute address and store in b. <i>Note:</i> the scaling parameter of memory operands can only be 1, 2, 4, or 8.
<b>push src</b>	Push <i>src</i> onto the stack and decrement stack pointer.
<b>pop dst</b>	Pop from the stack into <i>dst</i> and increment stack pointer.
<b>call &lt;func&gt;</b>	Push return address onto stack and jump to a procedure.
<b>ret</b>	Pop return address and jump there.
<b>add a, b</b>	Add from a to b and store in b (and sets flags).
<b>sub a, b</b>	Subtract a from b (compute $b-a$ ) and store in b (and sets flags).
<b>imul a, b</b>	Multiply a and b and store in b (and sets flags).
<b>and a, b</b>	Bitwise AND of a and b, store in b (and sets flags).
<b>sar a, b</b>	Shift value of b <i>right (arithmetic)</i> by a bits, store in b (and sets flags).
<b>shr a, b</b>	Shift value of b <i>right (logical)</i> by a bits, store in b (and sets flags).
<b>shl a, b</b>	Shift value of b <i>left</i> by a bits, store in b (and sets flags).
<b>cmp a, b</b>	Compare b with a (compute $b-a$ and set condition codes based on result).
<b>test a, b</b>	Bitwise AND of a and b and set condition codes based on result.
<b>jmp &lt;label&gt;</b>	Unconditional jump to address.
<b>j* &lt;label&gt;</b>	Conditional jump based on condition codes ( <i>more on next page</i> ).
<b>set* a</b>	Set byte based on condition codes.

## Conditionals

Instruction	(op) s, d	test a, b	cmp a, b
<b>je</b> "Equal"	d (op) s == 0	b & a == 0	b == a
<b>jne</b> "Not equal"	d (op) s != 0	b & a != 0	b != a
<b>js</b> "Sign" (negative)	d (op) s < 0	b & a < 0	b-a < 0
<b>jns</b> (non-negative)	d (op) s >= 0	b & a >= 0	b-a >= 0
<b>jg</b> "Greater"	d (op) s > 0	b & a > 0	b > a
<b>jge</b> "Greater or equal"	d (op) s >= 0	b & a >= 0	b >= a
<b>jl</b> "Less"	d (op) s < 0	b & a < 0	b < a
<b>jle</b> "Less or equal"	d (op) s <= 0	b & a <= 0	b <= a
<b>ja</b> "Above" (unsigned >)	d (op) s > 0U	b & a < 0U	b > a
<b>jb</b> "Below" (unsigned >)	d (op) s < 0U	b & a > 0U	b < a

## Registers

Name	Convention	Name of "virtual" register		
		Lowest 4 bytes	Lowest 2 bytes	Lowest byte
%rax	Return value – Caller saved	%eax	%ax	%al
%rbx	Callee saved	%ebx	%bx	%bl
%rcx	Argument #4 – Caller saved	%ecx	%cx	%cl
%rdx	Argument #3 – Caller saved	%edx	%dx	%dl
%rsi	Argument #2 – Caller saved	%esi	%si	%sil
%rdi	Argument #1 – Caller saved	%edi	%di	%dil
%rsp	Stack Pointer	%esp	%sp	%spl
%rbp	Callee saved	%ebp	%bp	%bpl
%r8	Argument #5 – Caller saved	%r8d	%r8w	%r8b
%r9	Argument #6 – Caller saved	%r9d	%r9w	%r9b
%r10	Caller saved	%r10d	%r10w	%r10b
%r11	Caller saved	%r11d	%r11w	%r11b
%r12	Callee saved	%r12d	%r12w	%r12b
%r13	Callee saved	%r13d	%r13w	%r13b
%r14	Callee saved	%r14d	%r14w	%r14b
%r15	Callee saved	%r15d	%r15w	%r15b

## C Functions

**void\*** malloc(**size\_t** size):

Allocate size bytes from the heap.

**void\*** calloc(**size\_t** n, **size\_t** size):

Allocate n\*size bytes and initialize to 0.

**void** free(**void\*** ptr):

Free the memory space pointed to by ptr.

**size\_t** sizeof(**type**):

Returns the size of a given type (in bytes).

**char\*** gets(**char\*** s):

Reads a line from stdin into the buffer.

**pid\_t** fork():

Create a new child process (duplicates parent).

**pid\_t** wait(**int\*** status):

Blocks calling process until any child process exits.

**int** execv(**char\*** path, **char\*** argv[]):

Replace current process image with new image.

## Virtual Memory Acronyms

<b>MMU</b>	Memory Management Unit	<b>VPO</b>	Virtual Page Offset	<b>TLBT</b>	TLB Tag
<b>VA</b>	Virtual Address	<b>PPO</b>	Physical Page Offset	<b>TLBI</b>	TLB Index
<b>PA</b>	Physical Address	<b>PT</b>	Page Table	<b>CT</b>	Cache Tag
<b>VPN</b>	Virtual Page Number	<b>PTE</b>	Page Table Entry	<b>CI</b>	Cache Index
<b>PPN</b>	Physical Page Number	<b>PTBR</b>	Page Table Base Register	<b>CO</b>	Cache Offset