University of Washington – Computer Science & Engineering

Autumn 2025 Instructors: Justin Hsia, Amber Hu 2025-12-10

CSE351 FINAL

Last Name:		
First Name:		
Student ID Number:		
Name of person to your Left | Right		
All work is my own. I had no prior knowledge of the exam contents, nor will I share the contents with others in CSE351 who haven't taken it yet. Violation of these terms could result in a failing grade. **(please sign)**		

Do not turn the page until 12:30.

Instructions

- This exam contains 14 pages, including this cover page. Show scratch work for partial credit, but put your final answers in the boxes and blanks provided.
- The last page is a reference sheet. *Please* detach it from the rest of the exam.
- The exam is closed book (no laptops, tablets, wearable devices). You are allowed two pages (US letter, double-sided) of *handwritten* notes. Scientific calculators are allowed.
- Please silence and put away all cell phones and other mobile or noise-making devices. Remove all hats, headphones, and watches.
- You have 110 minutes to complete this exam.

Advice

- Read questions carefully before starting. Skip questions that are taking a long time.
- Read *all* questions first and start where you feel the most confident.
- Relax. You are here to learn.

Question	M1	M2	M3	M4	M5	F6	F7	F8	F9	F10	Total
Possible Points	8	4	17	24	18	12	18	14	20	15	**150**

Question M1: Number Representation [8 pts]

Suppose we're writing a program that records **monetary transactions**. Each transaction is recorded in the following 32-bit representation scheme (U = unsigned, S = signed/two's complement):

```
          +----------+----------+----------+----------+
          |   Year   |  Month   |    Day   |  Amount  |
          +----------+----------+----------+----------+
bits:       31-23(U)   22-19(U)   18-14(U)    13-0(S)
```

(A) What is the **maximum** (*i.e.*, most positive) **transaction amount**? You may leave your answer unsimplified. [2 pt]

(B) Our year field will encode "years since 2000" (*e.g.*, 0x0 → year 2000, 0x1 → year 2001). **What is the *first* year after 2000 that we *cannot* represent?** [2 pt]

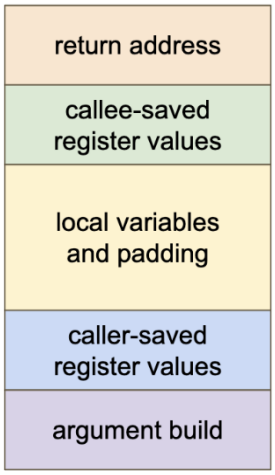
(C) Fill in the C expression that completes the function to **extract the month field value from txn:** [4 pt]

```
// The returned unsigned int should have the least 4 significant
// bits set to the month and the rest set to 0.
unsigned int get_month(unsigned int txn) {

    return _____;
}
```

Question M2: Design Question [4 pts]

We introduced the different portions of a stack frame. *Briefly* explain why the **caller-saved registers** must be found in the position/order given.



Why after the *local variables and padding*?

Why before the *argument build*?

Student ID Number: _____

Question M3: Pointers & Memory [17 pts]

For this problem, we are using a 64-bit x86-64 machine (**little endian**). A partial view of the current state of memory (values in hex) is shown below, along with a snippet of the ASCII table.

```
// "uw" is shown in memory
char* cp = "uw";
long  lg = 0x351;
short ar[5];
```

Word Addr	+0	+1	+2	+3	+4	+5	+6	+7
0xA0	33	00	75	77	00	74	6F	70
0xA8	00	CE	BE	9F	50	24	9D	D6
0xB0	5A	96	8E	5E	EF	5C	DF	9E
0xB8	D9	81	3E	43	34	DE	D7	CB
0xC0	B1	6B	AA	FB	BD	0B	7E	67

Partial ASCII table:

'n'	'o'	'p'	'q'	'r'	's'	't'	'u'	'v'	'w'	'x'	'y'	'z'
0x6E	0x6F	0x70	0x71	0x72	0x73	0x74	0x75	0x76	0x77	0x78	0x79	0x7A

(A) **How many bytes are allocated** by the declarations and initializations in the C code given above? [4 pt]

bytes

(B) **What is the output of `printf("%s", cp+3)`** – printing `cp+3` as a C string? [3 pt]

(C) Assume `ar` is stored at address `0xF0` (not shown in memory), and consider the following C expressions. **Provide the C type and value** (no leading zeros). [10 pt]

C Expression	C Type	Hex Value
`&ar[-8]`		`0x`
`*cp + 5`		`0x`

Question M4: Procedures & The Stack [24 pts]

The recursive function `run_back_sum()` and its x86-64 disassembly are shown below. This function recursively computes the sum of an array of `long`s and replaces the array values with the running partial sum, in reverse order. For example, `run_back_sum({1, 2, 3}, 3)` returns 6 and modifies the array to be `{6, 5, 3}`.

```
long run_back_sum (long* addr, int len) {
  if (len == 0)
    return 0;
  long elem = *addr;
  *addr = elem + run_back_sum(addr+1, len-1);
  return *addr;
}
```

```
#:  0000000000401136 <run_back_sum>:
1     401136:   b8 00 00 00 00     mov    $0x0,%eax
2     40113b:   85 f6              test   %esi,%esi
3     40113d:   75 01              jne    401140 <run_back_sum+0xa>
4     40113f:   c3                 ret
5     401140:   55                 push   %rbp
6     401141:   53                 push   %rbx
7     401142:   48 83 ec 08        sub    $0x8,%rsp
8     401146:   48 89 fb           mov    %rdi,%rbx
9     401149:   48 8b 2f           mov    (%rdi),%rbp
10    40114c:   83 ee 01           sub    $0x1,%esi
11    40114f:   48 8d 7f 08        lea    0x8(%rdi),%rdi
12    401153:   e8 de ff ff ff     call   401136 <run_back_sum>
13    401158:   48 01 e8           add    %rbp,%rax
14    40115b:   48 89 03           mov    %rax,(%rbx)
15    40115e:   48 83 c4 08        add    $0x8,%rsp
16    401162:   5b                 pop    %rbx
17    401163:   5d                 pop    %rbp
18    401164:   c3                 ret
```

(A) What is the **return address to `run_back_sum`** that gets stored on the stack? Answer in hex. [2 pt]

`0x` _____

(B) The 18 assembly instructions are numbered on the left. Below, **list *only* the instruction numbers that access memory**, separated by commas. [4 pt]

READ from memory: _____

WRITE to memory: _____

(C) **What values are saved across each recursive call?** Answer in terms of the C code. [4 pt]

(D) **How big is a `run_back_sum` stack frame** on a recursive (*i.e.,* not base case) call? [2 pt]

bytes

(E) Which stack frame portions exist in a recursive call to `run_back_sum`? **Write "Y" or "N"**. [2 pt]

Return address _____

Callee-saved register values _____

Local variables and padding _____

Caller-saved register values _____

Argument build _____

(F) Assume `main` calls `run_back_sum({4, 2}, 2)`. **How many `run_back_sum` stack frames are created**? [2 pt]

frames

(G) Assume `main` calls `run_back_sum({4, 2}, 2)`, with the array at address `0x100`. **Fill in the memory snapshot below** as this call to `run_back_sum` returns to `main`. The number of boxes is arbitrary; for unknown words, write "`0x unknown`". [8 pt]

0x7fffffffe008	<ret addr to main>
0x7fffffffe000	0x
0x7fffffffdff8	0x
0x7fffffffdff0	0x
0x7fffffffdfe8	0x
0x7fffffffdfe0	0x
0x7fffffffdfd8	0x
0x7fffffffdfd0	0x
0x7fffffffdfc8	0x

Question M5: C & Assembly [18 pts]

Answer the questions below about the following x86-64 assembly function:

```
mystery:
        cmpl    $1, %esi                # Line 1
        jg      .L7                     # Line 2
        ret                             # Line 3
.L7:    movslq  %esi, %rax              # Line 4
        leaq    -1(%rdi,%rax), %rax     # Line 5
        movzbl  (%rax), %edx            # Line 6
        movzbl  (%rdi), %ecx            # Line 7
        movb    %cl, (%rax)             # Line 8
        movb    %dl, (%rdi)             # Line 9
        subl    $2, %esi                # Line 10
        addq    $1, %rdi                # Line 11
        call    mystery                 # Line 12
        ret                             # Line 13
```

`mystery` takes two arguments and contains an `if`-statement. **Fill in the C function skeleton below**, using the provided variable names `arr`, `len`, and `c`. Hint: Recall that `arr[i]` ↔ `*(arr+i)`.

```
void mystery (_____________ arr, _____________ len) {

    if (_____________) {                        // Lines 1-2

        char c = _____________;                 // Lines 4-6

        _____________ = _____________;          // Lines 7-8

        _____________ = _____________;          // Line  9

        mystery(_____________, _____________);  // Lines 10-12
    }
}
```

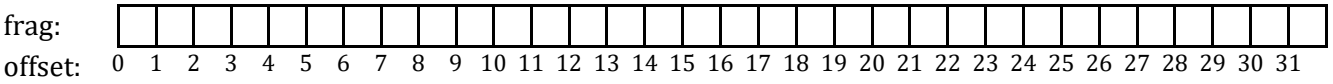
Question F6: Structs [12 pts]

For this question, assume a 64-bit machine and the following C struct definition.

```
struct TA {
  char* name;          // e.g., "Rose"
  char section[6];     // e.g., "AE/CE"
  struct TA* partner;
  int employeeID;
};
```

(A) The boxes below represent the bytes in an instance of `struct TA`, starting with offset 0 on the left. **Label each field above the boxes with a bracket and name**. **Shade in the boxes representing fragmentation**. Draw a thick vertical line at the end of the struct instance; if there are extra unused boxes at the end, leave them blank. [6 pt]

fields:



(B) **Can we reorder the fields in `struct TA` to reduce the size of an instance**? If yes, provide a field ordering; if not, briefly explain why not. [4 pt]

(C) *Briefly* give one reason why adding external fragmentation in struct instances is good/beneficial. [2 pt]

Question F7: Caching [18 pts]

We have an L1 data cache with the following settings:

- 128-byte cache size
- direct-mapped
- 32-byte blocks
- LRU replacement
- write-back and write allocate policies

The width of a physical address is ≥ 8 bits, but its exact length is not important. This code snippet swaps the elements of **two *adjacent* arrays** of `double`s of size `LEN`.

```
#define LEN 6
double temp;
double A[LEN];  // &A[0] = 0x60
double B[LEN];  // &B[0] = 0x90
for (int i = 0; i < LEN; i++) {
  temp = A[i];
  A[i] = B[i];
  B[i] = temp;
}
```

Assume `i` and `temp` are stored in registers and no elements of either array are in the cache at the start. Answer the following questions to analyze the code:

(A) **How many array elements per cache block**? [1 pt]

(B) **How wide** (in bits) **are the block offset and index fields**? [2 pt]

index: bits	block offset: bits

(C) **Which set and offset** (in binary) do `A[0]` and `B[0]` map into? [4 pt]

A[0] set: `0b`	A[0] offset: `0b`
B[0] set: `0b`	B[0] offset: `0b`

(D) **What is the memory access pattern** of *one* iteration of the `for`-loop (*e.g.*, RA for "read from A", WB for "write to B")? Separate accesses with commas (*e.g.*, "RA, WB"). [2 pt]

(E) **How many cache misses** occur in the execution of this *entire* code snippet? [3 pt]

(F) For each of the proposed (independent) changes, draw ↑ for "increased", — for "no change", or ↓ for "decreased" to indicate the **effect on the number of misses from Part E**: [6 pt]

Make 2-way set associative _____

Double the block size (64 B) _____

No-write allocate _____

Student ID Number: _____

Question F8: Memory Allocation [14 pts]

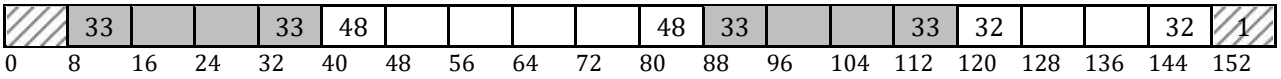
(A) Consider the C code shown here. Assume that the `malloc` call succeeds and that all variables are stored in memory (not registers). In the following groups of expressions, **circle the one** whose returned *value* (assume just before `return 0`) is **smallest/lowest**. [6 pt]

Group 1:	`ip`	`&str`	`&TWO`
Group 2:	`head`	`*str`	`TWO`
Group 3:	`&head`	`&ip`	`str`

```
#include <stdlib.h>
long TWO = 2;
struct node* head = NULL;

int main() {
  int* ip = malloc(88);
  char* str = "351";
  free(ip);
  return 0;
}
```

(B) We are using a dynamic memory allocator on a 64-bit machine with an explicit free list, 8-byte boundary tags, and 16-byte alignment. There is no prec-used? tag, so both allocated and free heap blocks have a footer. Answer the following questions, given the current state of the heap shown below.



i. What is the **minimum block size** of this allocator? [2 pt]

bytes

ii. If the shown allocated blocks were the result of calls to `malloc(16)` and `malloc(12)`, how much **internal fragmentation** is there in the heap? [2 pt]

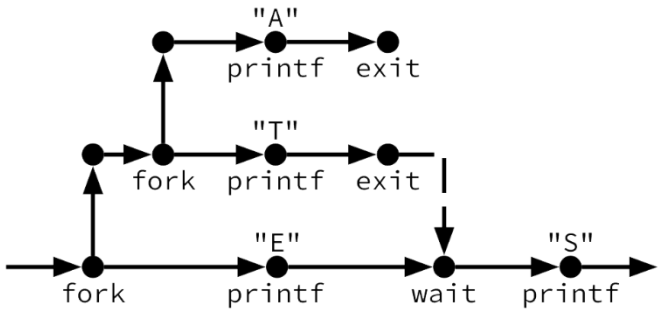
bytes

iii. Assuming a *best-fit allocation strategy*, what would be the **returned value from a call to `malloc(24)`**? [2 pt]

iv. Assuming the `malloc` from (iii) hasn't taken place, which argument to `free()` **yields the largest free block afterward**? [2 pt]

Question F9: Processes [20 pts]

(A) For the following process diagram, **determine whether or not each output is possible** (circle one). For impossible outputs, **circle the first invalid letter in the sequence**. *Spaces have been added for readability, but should not be considered when determining the validity of outputs.* [7 pt]



i. `"A T E S"` Possible / Impossible

ii. `"E A S T"` Possible / Impossible

iii. `"S E A T"` Possible / Impossible

iv. `"T E S A"` Possible / Impossible

(B) Following Part A's formatting, **draw a process diagram for the following code snippet**. Assume that the original process has a PID of 3 and that PIDs are assigned sequentially (*i.e.*, the next processes will have PIDs 4, 5, 6, etc.). [9 pt]

```
int x;
if (fork()) {
  printf("%d", getpid());
} else {
  x = fork();
  if (x)
    printf("%d", x);
  else {
    x++;
    printf("%d", x);
  }
  exit(0);
}
```

Draw here:

(C) *Briefly* explain how the following enable each process in Part B to maintain a distinct x value. [4 pt]

`fork`:
Page tables:

Question F10: Virtual Memory [15 pts]

Our system has the following setup:

- 16-bit virtual addresses and 4 KiB of RAM with 256-byte pages
- A PTE contains bits for valid (V), dirty (D), read (R), write (W), and execute (X)
- A 4-entry, 2-way set associative TLB with LRU replacement in the state shown (permission bits: 1 = allowed, 0 = disallowed)

TLBT	Valid	D	R	W	X	PPN
0x05	1	1	1	1	0	0xA
0x7F	1	1	1	1	0	0xB
0x7F	0	0	1	1	0	0xE
0x02	1	0	1	0	1	0xD

(A) Compute the following system values: [6 pt]

page offset width _____ # of physical address bits _____

# of PTEs in a page table _____ TLBT field width _____

(B) Given the current TLB state, what will happen if we try to **write to the address `0x05ED`**? Circle one item in each row. [3 pt]

TLB:	TLB Hit / TLB Miss / Neither
Page Table:	Page Table Hit / Page Fault / Neither
Data Access:	Cache Access / Segmentation Fault

(C) Given the current TLB state, give the **smallest address that results in accessing physical page 10**. [3 pt]

`0x`

(D) Circle one of the options below for each row: [3 pt]

Smallest storage capacity:	Caches	Disk	RAM	Registers
Slowest access time:	Caches	Disk	RAM	Registers
Updated on a context switch:	Caches	Disk	RAM	Registers

End of Exam

Did you write your Student ID Number on the top-right corner of every odd page?

THIS PAGE PURPOSELY
LEFT BLANK

CSE 351 Final Reference Sheet

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

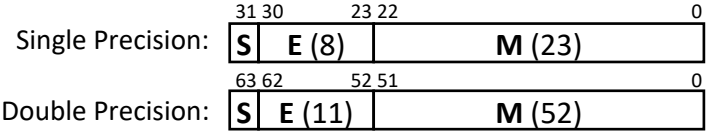
Bit fields: $(-1)^{S} \times 1.M \times 2^{(E-\text{bias})}$

single precision bias = 127

double precision bias = 1023

IEEE 754 Encodings

E	M	Meaning
all zeros	all zeros	± 0
all zeros	non-zero	± denorm num
1 to MAX-1	anything	± norm num
all ones	all zeros	± ∞
all ones	non-zero	NaN



Single Precision:

Double Precision:

Assembly Instructions

mov a, b	Copy from a to b.
movs a, b	Copy from a to b with sign extension. Needs *two* width specifiers.
movz a, b	Copy from a to b with zero extension. Needs *two* width specifiers.
lea a, b	Compute effective address and store in b. *Note:* the scaling parameter of memory operands can only be 1, 2, 4, or 8.
push src	Push `src` onto the stack and decrement stack pointer.
pop dst	Pop from the stack into `dst` and increment stack pointer.
call <func>	Push return address onto stack and jump to a procedure.
ret	Pop return address and jump there.
add a, b	Add from a to b and store in b (and sets flags).
sub a, b	Subtract a from b (compute b-a) and store in b (and sets flags).
imul a, b	Multiply a and b and store in b (and sets flags).
and a, b	Bitwise AND of a and b, store in b (and sets flags).
sar a, b	Shift value of b *right* (*arithmetic*) by a bits, store in b (and sets flags).
shr a, b	Shift value of b *right* (*logical*) by a bits, store in b (and sets flags).
shl a, b	Shift value of b *left* by a bits, store in b (and sets flags). Same as **sal**.
cmp a, b	Compare b with a (compute b-a and set condition codes based on result).
test a, b	Bitwise AND of a and b and set condition codes based on result.
jmp <label>	Unconditional jump to address.
j* <label>	Conditional jump based on condition codes (*more on next page*).
set* a	Set byte a to 0 or 1 based on condition codes.

Conditionals

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

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