Final Exam  Spring 2026

This exam consists of 8 substantive questions. You have 3 hours – budget your time wisely. Assume the armlab/gcc217 environment unless otherwise stated in a problem.

Do all of your work on these pages. You may use the provided blank spaces for scratch space, however this exam is preprocessed by computer, so for your final answers to be scored you must write them inside the designated spaces and fill in selected circles and boxes completely (⚫ and ⬛, not ✔ or ✘). Please make text answers dark and neat.

Name: _____ NetID: _____

Precept:

◯ P01 - MW 1:20 Xiaoyan Li	◯ P03 - TTh 12:15 Polly Ren	◯ P06 - TTh 1:20 Lana Glisic
◯ P02 - MW 3:30 Nicholas Yap	◯ P04 - TTh 12:15 David Shustin	◯ P08 - TTh 3:30 Viola Chen

This is a closed-book, closed-note exam, except you are allowed one two-sided study sheet. Please place items that you will not need out of view in your bag under your working space at this time. Electronic devices, including phones, laptops, smartwatches, smart glasses, etc. may not be used during this exam.

This examination is administered under the Princeton University Honor Code. Students should sit one seat apart from each other and refrain from talking to other students during the exam. All suspected violations of the Honor Code must be reported to honor@princeton.edu.

In the box below, copy **and** sign the Honor Code pledge before turning in your exam:
"I pledge my honor that I have not violated the Honor Code during this examination."

X_____

Question 0: First Things First. *That's it*. 0 points

Make sure you have filled out your name, NetID (i.e., armlab login – not PUID, not email alias), precept, and the complete Honor Code pledge text on the front page. Sign your name once you have finished the exam.

The substantive questions begin on Page 3. You should go there now. Ignore all the cryptic problem titles until you are completely finished with the exam; only then should you return to the puzzle at the bottom of this page, if there's time and it interests you.

For 0 points (so please don't even think about this if your time would be better spent reviewing your answer to one of the actual questions!), but significant puzzling respect:

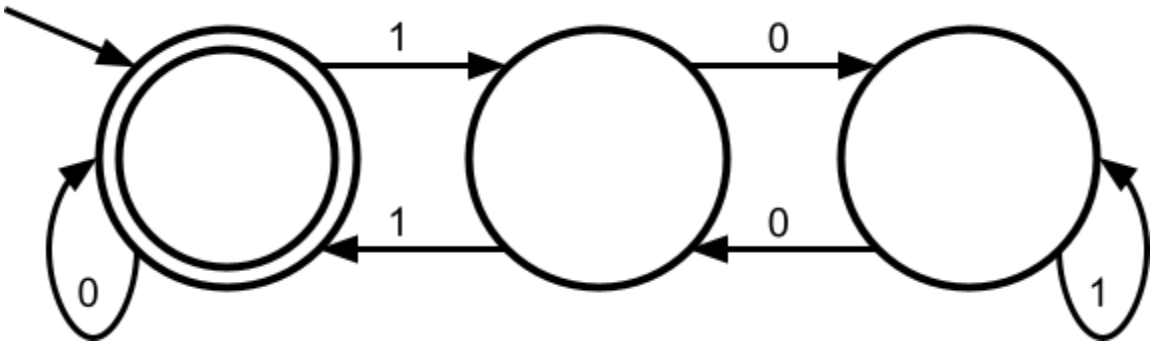
a. This exam has 9 questions, including Q0, with each cryptic title referencing one member of a not-quite-closed class (set) that you're intimately familiar with. What is the group, and which set element goes with each problem? Feel free to share your answers in the blank space above on this page.

b. Was that too easy? You're just getting started! Use the other information from the problem titles to **find** each set element (or, in one case, associated term) that has been hidden within its corresponding problem. Feel free, again, to use the space on this page to describe what you've found/solved.

Puzzle hints: The set is ordered; ?s are approximate interpretations; 'A'=65 except when it's 1; you might have to get a problem right to find where the set element is hiding!

Question 1: Princeton Inn the Nation's Service. How *dashing*! 6 points

Consider the following DFA for the first two parts of this question:



a. What are the first three unsigned integers whose binary representations are accepted by this DFA? Enter the three numbers in the boxes on the right.

b. In the box below, define the more general set of – or best English description of – binary unsigned integers this DFA accepts. Use no more than 10 words.

c. A student brings a LabTA the following line of code to compute the midpoint between lower bound `int a` and upper bound `int b`:

```
mid = (a + b) / 2;
```

The LabTA, remembering a concept from COS 217 and noticing a potential pitfall, suggests that the following computation would be better practice:

```
mid = a + (b - a) / 2;
```

In the box below, explain the problem with the original code that is fixed in the revised version. Use no more than 10 words.

Question 2: Paradigmatic Billionaire, for short, *vertically* en*code*d. 8 points

a. Consider the following string copy operation using the C standard library:

```
strcpy(pc1, (const char *) pc2);
```

Select all of the following that could be the type of `pc1` for this function call to compile cleanly (i.e., without warning or error):

`const char *`	`char * const`	`const char * const`	`void *`
☐	☐	☐	☐

Recall the signatures for these A3 functions:

```
int SymTable_put(SymTable_T oST, const char *pcKey, const void *pvValue);
int SymTable_contains(SymTable_T oST, const char *pcKey);
void *SymTable_get(SymTable_T oST, const char *pcKey);
```

Now consider the following **flawed** testing code for A3's `SymTable_put` function. Assume `N`, `oST`, `copyRet`, and `FALSE` were properly defined and initialized immediately before this snippet and that all necessary header files have been included.

```
char acNameString[N] = "ok";
SymTable_put(oST, acNameString, "ok");
copyRet = strcpy(acNameString, "bad");
if( strcmp("ok", SymTable_get(oST, "ok")) != 0 ||
            SymTable_contains(oST, copyRet) ) return FALSE;
```

b. In the box below, using no more than 10 words, explain the design error in `SymTable_put` that this test is intended to catch.

c. As noted, however, this testing code is incorrect. In the box below, in 1-2 sentences, explain why the testing code will crash in the event that the error it is intended to catch is present.

Question 3: Quantitatively Inclined? Look at *both sides*. 4 points

Most of the challenge of A4 was to read and understand a large code base, in as much as you could then debug or expand it. Write a COS 217 function comment for this code that details its parameters, behavior, and side effects. Be specific, but answer concisely.

```
static int DT_redacted(
     const char *pcPath,
     Node_T *poNResult) {

   Path_T oPPath = NULL;
   Node_T oNFound = NULL;
   int iStatus;

   assert(pcPath != NULL);
   assert(poNResult != NULL);

   if(!bIsInitialized) {
      *poNResult = NULL;
      return INITIALIZATION_ERROR;
   }

   iStatus = Path_new(pcPath, &oPPath);
   if(iStatus != SUCCESS) {
      *poNResult = NULL;
      return iStatus;
   }

   iStatus = DT_traversePath(
                oPPath, &oNFound);
   if(iStatus != SUCCESS) {
      Path_free(oPPath);
      *poNResult = NULL;
      return iStatus;
   }

   if(oNFound == NULL) {
      Path_free(oPPath);
      *poNResult = NULL;
      return NO_SUCH_PATH;
   }

   if(Path_comparePath(
                Node_getPath(oNFound),
                oPPath) != 0) {
      Path_free(oPPath);
      *poNResult = NULL;
      return NO_SUCH_PATH;
   }

   Path_free(oPPath);
   *poNResult = oNFound;
   return SUCCESS;
}
```

Unlike with function comments for your assignments, for this problem you do not have to discuss the various return values for this function in your comment.

Question 4: Lucan, Jeeves, Alfred, or Geoffrey's *Tactile* Mascot. 9 points

Consider this `Makefile`, which executes each of the four stages of the build process separately for a program consisting of three source files: a client (`client.c`) that calls functions from a library interface (`lib.h`) and an implementation of that library (`lib.c`).

```
client: client.o lib.o
     gcc217 client.o lib.o -o client

client.o: client.s
     gcc217 -c client.s

client.s: client.i
     gcc217 -S client.i

client.i: client.c lib.h
     gcc217 -E client.c >| client.i

lib.o: lib.s
     gcc217 -c lib.s

lib.s: lib.i
     gcc217 -S lib.i

lib.i: lib.c lib.h
     gcc217 -E lib.c >| lib.i

clean:
     rm -f client *.o *.s *.i
```

Recall these `bash` details: `touch` changes the date/time stamp of the given file(s) to the current date/time (as if they had just been edited); >| is just like > except it overwrites the destination file if it already exists; `rm -f` removes the given file(s); and the wildcard (*) expands to specify all files whose names match the expression it is used in, e.g., `make clean` removes the `client` executable and *all* the `.o`, `.s`, and `.i` files.

Assume that each part on the next page is **independent**, and that for each part the working directory initially contains the three source files, the `Makefile`, and already contains the seven file targets produced by the `Makefile`.

For each item below, mark whether the statement is true (**T**), false (**F**), or there is not enough information to say for sure (**NEI**) based on this `Makefile` and set of files.

		T	F	NEI
a.	`make` and `make client` are always equivalent	◯	◯	◯
b.	`rm -f client.o; make` will execute 2 `gcc` commands	◯	◯	◯
c.	`client.c` has only 1 `#include` directive: `#include "lib.h"`	◯	◯	◯
d.	`touch *.c; make` will execute all 7 `gcc` commands	◯	◯	◯
e.	`make clean; make; make` will execute all 7 `gcc` commands twice	◯	◯	◯
f.	The assembled object file `lib.o` will include relocation record(s)	◯	◯	◯
g.	`touch lib.h; make` will execute all 7 `gcc` commands	◯	◯	◯
h.	`touch lib.c; make` will rebuild `client.o`	◯	◯	◯
i.	Instructions from `client.o` will appear at lower memory addresses than instructions from `lib.o` in the final `client` executable	◯	◯	◯

You may refer to this abbreviated ARM assembly language reference for Q5-Q8.

Instruction(s)	Description
`{add,sub,mul, lsl} dst, src1, src2`	`dst = src1 {+, -, *, <<} src2`
`{beq,bne} label`	Go to `label` if comparison was {“equal”, “not equal”}
`{b,bl} label`	{Unconditionally go to , Call function at} `label`
`cmp first, second`	Compare `first` with `second`, setting bits in PSTATE
`ldr dst, [src]`	Load 4 or 8 bytes pointed to by `src` into `dst`
`str src, [dst]`	Store 4 or 8 bytes in `src` to memory pointed to by `dst`
`mov dst, src`	Copy contents of register `src` to register `dst`
`sxtw dst, src` (`src` is `w`, `dst` is `x`)	Copy sign-extended contents of `src` to `dst`
`ret`	Return to address pointed to by `x30`
`R0 – R7` and `R0` (`w` or `x`)	Used for arguments to and return value from functions

Question 5: Myself of Song of Myself, *think just outside the box*. 7 points

Fill in the box to the left of each description with the letter of the appropriate assembler directive from the options bank on the right.

A. `.equ`
B. `.quad`
C. `.req`
D. `.section`
E. `.size`
F. `.skip`
G. `.word`

☐ Working aliases can be assigned to registers with this directive

☐ Helps the linker and debugger track the length of a symbol

☐ `int`-sized allocation and initialization

☐ Tells the assembler where subsequent data/code is allocated

☐ Makes space for a variable in the BSS section

☐ Allocates and initializes a pointer or `long`

☐ Names an immediate value with a semantically meaningful alias

Question 6: Causal inquiry north of the border? Be *bold*! 12 points

The following assembly language code was generated by `gcc217` then very lightly cleaned up for clarity. The original source was a small C function that takes a pointer to a structure as a parameter and returns a `long`.

```
 1        .section .text
 2        .global  mystery
 3        .type mystery, %function
 4  mystery:
 5        sub   sp, sp, #32
 6        cmp   x0, 0
 7        bne   .setup
 8        mov   x0, 0
 9        b     .return
10  .setup:
11        str   x0, [sp, 8]
12        str   x0, [sp, 24]
13        mov   x1, 1
14        str   x1, [sp, 16]
15        b     .check
16
```

```
17  .loop:
18        ldr   w1, [x0, 0]
19        sxtw  x1, w1
20        ldr   x2, [sp, 16]
21        mul   x2, x2, x1
22        str   x2, [sp, 16]
23        ldr   x0, [sp, 24]
24        ldr   x1, [x0, 8]
25        str   x1, [sp, 24]
26  .check:
27        ldr   x0, [sp, 24]
28        cmp   x0, 0
29        bne   .loop
30        ldr   x0, [sp, 16]
31  .return:
32        add   sp, sp, #32
33        ret
```

a. As noted, the parameter is a pointer to a structure. Based on how the parameter is used in the function, fill out the C `struct` below, using reasonable field names.

```
struct mysteryArg {



}
```

b. Translate the assembly language function into idiomatic (not flattened) C code.

```
long mystery(struct mysteryArg *psArg) {
```

Question 7: Perelman unrealized *as key* to the last 3 4ths. 10 points

Recall from lecture the machine language instruction template for three-register ALU instructions, e.g., `add` :

Op. Group: Data processing – 3-register

- Instruction width in bit 31: 0 = 32-bit, 1 = 64-bit
- Whether to set condition flags (e.g. ADD vs ADDS) in bit 29
- Second source register in bits 16-20
- First source register in bits 5-9
- Destination register in bits 0-4
- Remaining bits encode additional information about instruction

a. A processor's instruction set architecture is like a software module's interface, in that it should stay fixed even in the face of underlying implementation changes. Thus, code written for early versions of the processor can still run on later versions.

   But not all underlying implementation changes can be architecture agnostic. For example, upgrading the processor design to have 61 general purpose registers instead of just 31 would force a change to the instruction set representation. In the box below, in no more than 1 sentence, explain why this particular upgrade would require a change to the instructions.

b. Consider the **buggy** `count` function below, which is intended to examine each element of an array of `N` ARM machine language 3-register add instructions and return the number that use register `r` for any of the three register operands.

```
size_t count(unsigned int adds[], size_t N, unsigned char r) {
   size_t count = 0;
   size_t index;
   unsigned char mask = 0xF;
   assert(adds != NULL);
   for(index = 0; index < N; index++) {
      if((adds[index] & mask) == r)
         count++;
      if(((adds[index] >> 5) & mask) == r)
         count++;
      if(((adds[index] >> 17) & mask) == r)
         count++;
   }
   return count;
}
```

Find and fix **four** errors in the code above in-place (add missing code, cross out superfluous code, or cross out and replace incorrect code). If the same error is made multiple times, each occurrence counts separately towards the total of four.

To help you debug, here are some example calls to `count` on an array `auiArr` with the following instruction machine code values (corresponding assembly language shown beside for ease of understanding); recall `xzr` is encoded as 31.

```
8b1f03e0   (add  x0, xzr, xzr)
8b1f001f   (add xzr,  x0, xzr)
8b00004e   (add x14,  x2,  x0)
8b1f0043   (add  x3,  x2, xzr)
8b010057   (add x23,  x2,  x1)
```

Sample call	Correct result	This version's result
`count(auiArr, 5, 0);`	3	4
`count(auiArr, 5, 1);`	1	0
`count(auiArr, 5, 2);`	3	3
`count(auiArr, 5, 31);`	3	0

Question 8: FitzChivalry Farseer? Soon, but construction is *left*! 9 points

Hopefully you recall the `struct BigInt` and the `BigInt_T` opaque pointer from A5:

```
struct BigInt {
   /* The number of used digits in the BigInt object */
   long lLength;

   /* The digits comprising the BigInt object.
      aulDigits[0] stores the least significant digit.
      Unused digits are set to 0. */
   unsigned long aulDigits[MAX_DIGITS];
};
typedef struct BigInt *BigInt_T;
```

Of course you also recall the code shown below, from the assembly language function `BigInt_swapDigits`, because you were asked to translate it to C in the *very* hard Q5.

☐	`sub sp, sp, 32`
☐	`str x0, [sp, A]`
☐	`str x1, [sp, B]`
☐	`str x2, [sp, C]`
☐	`ldr x0, [sp, A]`
☐	`ldr x1, [sp, B]`
☐	`add x0, x0, 8`
☐	`ldr x2, [x0, x1, lsl 3]`
☐	`str x2, [sp, D]`
☐	`ldr x0, [sp, A]`
☐	`ldr x1, [sp, C]`
☐	`add x0, x0, 8`
☐	`ldr x2, [x0, x1, lsl 3]`
☐	`ldr x0, [sp, A]`
☐	`ldr x1, [sp, B]`
☐	`add x0, x0, 8`
☐	`str x2, [x0, x1, lsl 3]`
☐	`ldr x0, [sp, A]`
☐	`ldr x1, [sp, C]`
☐	`ldr x2, [sp, D]`
☐	`add x0, x0, 8`
☐	`str x2, [x0, x1, lsl 3]`
☐	`add sp, sp, 32`
☐	`ret`

Some of the instructions in this implementation, however, could be safely removed. Optimize the code above by marking the boxes to remove each instruction for which nothing changes to remaining instructions, nor to the function's behavior, in its absence.