Final Exam Spring 2023

COS 217

This exam consists of 8 questions. You have 180 minutes – budget your time wisely. Assume the ArmLab/Linux/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 1:30 Donna Gabai	◯	P04 TTh 1:30 Wei Luo	◯	P06 TTh 3:30 Ashwini Raina
◯	P02 - MW 3:30 Donna Gabai	◯	P04A TTh 1:30 Samuel Ginzburg	◯	P07 TTh 7:30 Wei Tang
◯	P03 - TTh 12:30 Guðni Nathan Gunnarsson	◯	P05 TTh 2:30 Jianan Lu		

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 or under your working space at this time. Electronic devices such as cell phones, laptops, music players, smartwatches except to check the time, 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."

Exam statistics:
Mean: 56.8 / 80 (71%)
StdDev: 11.7 / 80
{25th/50th (median)/75th} %ile: { 49 , 58 , 66 } / 80
Max: 78 / 80

X________________________________

(The exam questions begin on page 3. This page may be used for scratch work, however any answers given on this page will not be graded.)

Question 1: *Snap Twice*, Bubble Once   10 points

Identify whether each statement is **True** or **False**. Fill in exactly one circle per line.

		True	**False**
a.	ARM registers have shorter (i.e., faster) latency than RAM	●	○
b.	An ARM ALU instruction may store its result directly to RAM	○	●
	ARM is a load/store architecture: `ldr` first, then manipulate, then `str` separately		
c.	Fixed-length instructions are more typical of RISC than CISC	●	○
	RISC tendencies: {fewer, simpler, fixed-length} instructions; ldr/str paradigm		
d.	The assembler generates a relocation record for every branch	○	●
	Assembler can build ML itself for branches within the same file's text section		
e.	`adds w0, w1, wzr` is completely equivalent to `mov w0, w1`	○	●
	`adds` sets the `PSTATE` flags, but `mov` doesn't		
f.	Both spatial and temporal locality concern near-future data access	●	○
	"Access x → access x's neighbor *soon*" and "Access x → access x again *soon*"		
g.	`gprof` can identify what function is using most of a program's time	●	○
h.	`main`'s initial `SP` value may be variable to hamper stack smashes	●	○
	Attacks relying on absolute stack addresses are harder if `SP` differs in each run		
i.	Using big-endian byte order allows for larger `signed long` values	○	●
	The range depends on the number of bits and representation, not byte order		
j.	Machine language is more portable than assembly language	○	●
	Neither is portable, but assembly language is at least not **less** portable		

Question 2: *Hyde* trailing bits that are always 0   4 points

The machine language instructions for `str w1, [x2, imm]` and `strb w1, [x2, imm]` both use 12 bits to represent the unsigned immediate offset value (`imm`). The maximum possible values of these offsets **differ by a factor of 4**, however, due to assumptions afforded by alignment guarantees. In the boxes below, give the maximum possible unsigned offset value, in base 10. You may choose to represent your answer in terms of a power of 2, e.g., $2^8 - 39$:

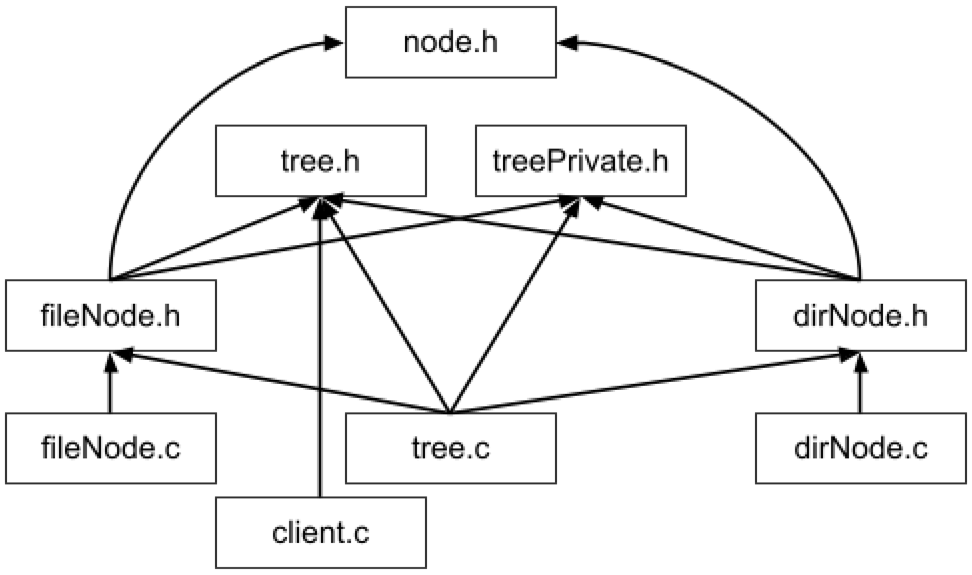
`str`	`strb`
16380 (2^{14} - 4) (14: offset is shifted left by 2 bits; -4: must be divisible by 4)	4095 (2^{12} - 1)

Question 3: Redundant Rebuilds *Nevermore* 12 points

Imagine that A4 has been changed for next semester in the following (dubious) ways:

- Eliminate the `Path` and `DynArray` modules
- Require separate modules for file nodes and directory nodes
- Allow the two node modules to see the definitions of the file tree state variables

Consider a proposed set of source code relationships, where arrows indicate `#include`, e.g., `client.c` `#includes` `tree.h`:



As with your A4, one requirement is a `Makefile` to produce the executable `ft`! Here is the skeleton of a correct `Makefile` for the diagram, but it has placeholders for targets `client.o`, `dirNode.o`, `fileNode.o`, `ft`, and `tree.o`, and their dependency lists:

```
TARGET1: DEPENDENCIES1
     gcc217 client.o tree.o dirNode.o fileNode.o -o ft

TARGET2: DEPENDENCIES2
     gcc217 -c client.c

TARGET3: DEPENDENCIES3
     gcc217 -c tree.c

TARGET4: DEPENDENCIES4
     gcc217 -c dirNode.c

TARGET5: DEPENDENCIES5
     gcc217 -c fileNode.c
```

For each of the following items concerning the `Makefile`'s **correctly completed** version – one where each **`TARGET`** is replaced by an actual filename and each **`DEPENDENCIES`** is replaced by a list of zero or more filenames – answer in the box to the right of the question:

a. What file is `TARGET1`?	`ft`
	`(based on what comes after -o)`

b. How many `.c` files are listed in `DEPENDENCIES1`?	`0`
	`(executables depend only on .o files)`

c. How many `.h` files are listed in `DEPENDENCIES1`?	`0`
	`(executables depend only on .o files)`

d. How many `.o` files are listed in `DEPENDENCIES1`?	`4`
	`(the ones listed in the gcc217 command)`

e. What file is `TARGET2`?	`client.o`
	`(gcc217 -c does P/C/A, producing .o)`

f. How many `.c` files are listed in `DEPENDENCIES2`?	`1`
	`(a .o file is made from only 1 .c file)`

g. How many `.h` files are listed in `DEPENDENCIES2`?	`1`
	`(tree.h is the only #included file)`

h. How many `.o` files are listed in `DEPENDENCIES2`?	`0`
	`(.o files depend only on .c/.h files)`

i. What file is `TARGET3`?	`tree.o`

j. How many `.h` files are listed in `DEPENDENCIES3`?	`5`
	`(4 tree.c #included, node.h indirectly)`

k. What file is `TARGET4`?	`dirNode.o`
l. What file is `TARGET5`?	`fileNode.o`

(The exam questions continue on page 7. This page may be used for scratch work, however any answers given on this page will not be graded.)

Question 4: `add`*ams Family*

5 points

Consider the following pattern for 3-register ALU instructions in ARM:

```
msb: bit 31                                     lsb: bit 0
↓                                                        ↓
wxsx 101x xxxr rrrr xxxx xxrr rrrr rrrr
```

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

Here are the full sets of red opgroup+opcode bits (bits 30, 28-24) that span both sides of the aqua s bit (bit 29) for some specific instructions that follow this pattern:

```
adc/adcs 0s11010
add/adds 0s01011
```

In the box beside each machine language instruction encoding below, write the number of the corresponding assembly language instruction from the list on the right (only half of the numbers will be used):

a. `0xba010040`	10
b. `0x9a020020`	7
c. `0x1a020020`	6
d. `0x8b010040`	3
e. `0xba020020`	9

1. `add w0, w1, w2`
2. `add x0, x1, x2`
3. `add x0, x2, x1`
4. `adds w0, w1, w2`
5. `adds w0, w2, w1`
6. `adc w0, w1, w2`
7. `adc x0, x1, x2`
8. `adcs w0, w1, w2`
9. `adcs x0, x1, x2`
10. `adcs x0, x2, x1`

Question 5: *Were* is *enid*? 10 points

Consider the following combined `enum` declaration and `typedef`:

```
typedef enum e { E=5, F=6, I=9, L=12, N=14, T=20, X=24 } e;
```

a. In the box below, indicate in what section this line allocates memory, assuming it appears outside of any function. If it does not allocate memory, write "NONE".

```
NONE (e is a type,not a
variable)
```

Further consider this program shell, where locations within the code are numbered:

```
#include <stdlib.h>
typedef struct Node *Node_T;
/* 1 */ ;
static /* 2 */ ;
/* 3 */ = NULL;

void fun( /* 4 */ ) {
   /* 5 */ ;
   static /* 6 */ = 21.7;
   /* other code follows */
}
```

b. For each row of the table, in each of the STACK, BSS, and DATA columns indicate **all** the location numbers from the program above where the expression in the first column could be placed that would result in memory being allocated in that section of memory **and** would not result in a compiler warning or error. If no location number would accomplish both, indicate this with "NONE".

	STACK	BSS	DATA
Node_T n	4, 5	1, 2	3 (6: compiler error)
int i	4, 5	1, 2	6 (unlike in Java) (3: compiler warning)
double d	4, 5	1, 2	6 (3: compiler error)

Question 6: *Cello*, world! 10 points

Consider a program made up of the following two files:

`cello.c`

```
   #include <stdio.h>
   #define H 'H'

L✧ extern void brood();
   int data = H;

   int main() {
      brood();
      printf("%cello, world!\n", data);
      return 0;
   }
```

`brood.s`

```
.section .text

        .global brood
brood:
        adr x0, data
        mov w1, 'C'
        str w1, [x0]
        ret
```

For each statement, identify to which stage of the build process (**P**reprocessor, **C**ompiler, **A**ssembler, **L**inker) the statement applies, or **N**one if it applies to none of the four stages.

	P	C	A	L	N
a. Emits a warning if the C line L✧ is omitted entirely Compiler reports “implicit declaration” of `brood` when seeing the function call	○	●	○	○	○
b. Emits a warning if L✧ is `void` **`brood`**`();` (no **`extern`**) `extern` is the default linkage for a function declaration in C, so these are equivalent	○	○	○	○	●
c. Emits an error if the `.global` directive is omitted Compiler generates a `bl`; assembler makes a relocation record; linker reports error	○	○	○	●	○
d. Emits an error if `.global` is misspelled without the “`.`” Without “`.`” the assembler treats this as an instruction it doesn’t know, not a directive	○	○	●	○	○
e. Finds the `stdio` library declarations Preprocessor imports them from `/usr/include/stdio.h` when handling the `#include`	●	○	○	○	○
f. Finds the `stdio` library definition for `printf` Linker collects object code from `/usr/lib64/libc.a`	○	○	○	●	○
g. Replaces all instances of `H` with `'H'` in source code Preprocessor does this when handling the `#define`	●	○	○	○	○
h. Generates a `beq` assembly instruction from `cello.c` There is no conditional in the file, so no need to generate a `cmp` + conditional branch	○	○	○	○	●
i. Generates a `bl` assembly instruction from `cello.c` Compiler translates `brood()` and `printf()` function calls into `bl` instructions	○	●	○	○	○
j. Calculates the relative address from `data` to the `adr` Assembler makes a relocation record; Linker computes address + patches `adr` ML	○	○	○	●	○

Question 7: A String *Thing* or Two 13 points

This problem will be working with two different implementations of a basic String ADT with the following interface, defined in `stringthing.h`:

```
#ifndef STRINGTHING_H
#define STRINGTHING_H
#include <stddef.h>
typedef enum {FALSE, TRUE} boolean;

/* An S_T is a "string": a series of characters indexed from 0 */
typedef struct S* S_T;

/* Create a new empty String of length 0 return the String,
   or NULL if insufficient memory */
S_T S_new(void);

/* Deallocate all memory associated with s */
void S_free(S_T s);

/* Return the length of s's String contents */
size_t S_length(S_T s);

/* Return the character at index i of String s
   Behavior is undefined if i >= S_length(s) */
char S_charAt(S_T s, size_t i);

/* Append pc's contents (up to but not including the '\0') to the back of s
   Return TRUE for success, or FALSE if insufficient memory */
boolean S_append(S_T s, char *pc);

/* Set pc's contents (up to but not including the '\0') into s beginning
   at index i, overwriting existing contents and expanding s if
   needed. Return TRUE for success, or FALSE leaving s unchanged if
   i > S_length(s) or insufficient memory for necessary expansion */
boolean S_set(S_T s, size_t i, char *pc);

#endif
```

Two implementations, a client, and questions about them follow on subsequent pages.

(The exam questions about this module continue on page 12. This page may be used for scratch work, however any answers given on this page will not be graded.)

s1.c – a partial implementation; assume necessary standard libraries are included:

```
#include "stringthing.h"
struct S {
   size_t len; /* number of characters in chars */
   char * chars; /* characters in the string */
};
S_T S_new(void) {
   S_T s = calloc(1, sizeof(struct S));
   return s;
}
void S_free(S_T s) { /* to be completed in part a. */ }
size_t S_length(S_T s) {
   assert(s);
   return s->len;
}
char S_charAt(S_T s, size_t i) {
   assert(s);
   assert(i < s->len);
   return s->chars[i];
}
boolean S_append(S_T s, char *pc) {
   assert(s);
   assert(pc);
   return /* to be completed in part b. */ ;
}
boolean S_set(S_T s, size_t i, char *pc) {
   size_t pc_len;
   char *chars;
   assert(s);
   assert(pc);
   if(i > s->len)
      return FALSE;
   pc_len = strlen(pc);
   if(i + pc_len > s->len) {
      /* if s->chars is NULL, realloc is identical to malloc(i + pc_len); */
      chars = realloc(s->chars, i + pc_len);
      if(!chars)
         return FALSE;
      s->chars = chars;
      s->len = i + pc_len;
   }
   /* strncpy(dest, src, n) copies up to n characters from src to dest */
   strncpy(&s->chars[i], pc, pc_len);
   return TRUE;
}
```

a. In the box below, implement the `S_free` function from the first implementation (`s1.c`) such that the module will not leak any memory:

```
void S_free(S_T s) {
  assert(s != NULL); /* ok to do if(s != NULL) instead*/
  free(s->chars);
  free(s);

}
```

b. In the box below, complete the return statement of the `S_append` from the first implementation (`s1.c`) to make the function accord to `StringThing.h`.

```
return  S_set(s, s->len /* or S_length(s) */, pc)           ;
```

Now consider this client program; assume necessary standard libraries are included and that all memory allocations in `S_new`, `S_append`, and `S_set` always succeed. Recall that `argv[0]` is the program's name and `argc` is the number of elements in `argv`.

```
#include "stringthing.h"
int main(int argc, char** argv) {
   S_T s = S_new();
   size_t i;
   if(!s) return EXIT_FAILURE;
   S_append(s, argv[0]);
   for(i = 1; i < (unsigned int) argc; i++) {
      S_set(s, i-1, argv[i]);
   }
   for(i = 0; i < S_length(s); i++)
      putchar(S_charAt(s, i));
   S_free(s);
   return EXIT_SUCCESS;
}
```

c. In the box below, write the output of the client (built into the program `kooky`) when invoked in `bash` as the following command:
`./kooky cold ocean swimming: the "water's" only seventeen`

```
costwoseventeen
```

s2.c – a 2nd partial implementation; assume necessary standard libraries are included:

```
#include "stringthing.h"
struct S {
   /* first sizeof(size_t) bytes are string's length as a size_t,
      followed by that many characters of contents */
   char * chars;
};
S_T S_new(void) {
   S_T s = calloc(1, sizeof(struct S));
   if(!s) return s;
   s->chars = calloc(1,sizeof(size_t));
   if(!s->chars) {
      free(s);
      return NULL;
   }
   return s;
}
void S_free(S_T s) { /* redacted so as not to spoil part a*/ }
size_t S_length(S_T s) {
   assert(s);
   return *(size_t*)s->chars;
}
char S_charAt(S_T s, size_t i) {
   assert(s);
   assert(i < S_length(s));
   return /* return value to be completed in part e. */ ;
}
boolean S_append(S_T s, char *pc) { /* redacted so as not to spoil part b*/ }
boolean S_set(S_T s, size_t i, char *pc) {
   size_t pc_len;
   char *chars;
   assert(s);
   assert(pc);
   if(i > S_length(s)) return FALSE;
   pc_len = strlen(pc);
   if(i + pc_len > S_length(s)) {
      chars = realloc(s->chars, sizeof(size_t) + i + pc_len);
      if(!chars) return FALSE;
      s->chars = chars;
      /* length update to be completed in part f. */
   }
   /* strncpy(dest, src, n) copies up to n characters from src to dest */
   strncpy(&/* operand to be completed in part e. */, pc, pc_len);
   return TRUE;
}
```

You may find it useful to think about some armlab examples of this data representation:

- The empty string – 8 bytes total pointed to by the `s->chars` field:
  - 8 bytes representing `(size_t) 0`, then no additional contents
- The string with contents "abcd" – 12 bytes total:
  - 8 bytes representing `(size_t) 4`, then the `chars` `'a'`, `'b'`, `'c'`, and `'d'`
- The string with 64 newlines – 72 bytes total :
  - 8 bytes representing `(size_t) 64`, then 64 `'\n' chars`
- The string with 1073741824 (2^{30}) `'a'`s: 1073741832 bytes total:
  - 8 bytes representing `(size_t) 1073741824`, then 1073741824 `'a' chars`
- Notice: there are no trailing `'\0'`s in these strings!

d. In the box below, describe in under 10 words what changes would have to be made to the interface `stringthing.h` in order to make the `kooky` client work using this implementation instead of the first one:

None
(The whole point of the interface is that clients know how to use the module without having to care about the underlying representation or implementation!)

e. In the second implementation (`s2.c`), two redacted expressions are the same:
   i. In `S_charAt`, the code required to complete the `return` statement
   ii. In `S_set`, the address-of operator's operand in `strncpy`'s first argument

   In the box below, write the expression that would correctly complete both lines:

`s->chars[sizeof(size_t) + i]`
(or, like we did for `BigInt` in A5: **`(s->chars + sizeof(size_t))[i]`**)

f. In the second implementation (`s2.c`), the line that updates the length of the string after expansion in `S_set` has also been redacted. In the box below, write the one-line statement that would complete the appropriate update:

`* (size_t*) s->chars = i + pc_len;` (dereference casted `s->chars`)
(or, equivalently: **`((size_t*) s->chars)[0] = i + pc_len;`**)

Question 8: *Goody* Two chars 16 points

Consider the following C function, which reads in up to two chars from standard input, stores any chars read into its array argument, and returns the number of chars read:

```
size_t readTwo(char ac[])
{
   size_t j = 0;
   int c;
   assert(ac != NULL);
   do {
      c = getchar();
      if(c == EOF)
         return j;
      ac[j] = c;
      j++;
   } while(j < 2);
   return j;
}
```

In the box on the next page, translate this function into ARM assembly language faithfully without optimization (i.e., maintain function state on the stack at offsets defined by .equ constants, not in callee-saved registers). Where possible use offset memory addressing modes instead of composing a target address through separate instructions.

You can refer to this abbreviated ARM assembly language reference guide:

Instruction(s)	Description
`{add,sub,lsl} dst, src1, src2`	`dst = src1 {+, -, <<} src2`
`{blo,beq} label`	Go to `label` if comparison was {“lower than”, “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`
`strb src, [dst]`	Store lowest 1 byte in `src` to memory pointed to by `dst`
`mov dst, src`	Copy `src` to `dst`
`ret`	Return to address pointed to by `x30`
`{x,w}0 – {x,w}7` and `{x,w}0`	Used for arguments to and return value from functions

If you run out of space in the box on the next page, you may use page 18 for the remainder of your response. Clearly indicate within the box that you have done so.

```
      .section .text
      .equ EOF, -1
      // fill in your stack size/offsets here and use these symbols in readTwo
      .equ STACK_NUMBYTES, __32__
      .equ AC, __8__
      .equ J,  __16__
      .equ C,  __24__ // other orders are possible
      .global readTwo
readTwo:
      sub sp, sp, STACK_NUMBYTES
      str x30, [sp]    // save return address to restore after calling getchar
      str x0, [sp, AC] // save parameter (base address of array ac)
      str xzr, [sp, J] // size_t j = 0;
      // c is not initialized in the C code
      // the assert macro is not translated to assembly

      // do {
loop:
      // c = getchar()
      bl getchar
      str w0, [sp, C]

      // if(c == EOF) return j
      cmp w0, EOF          // optional to reload C before doing this
      beq epilog

      // ac[j] = c;
      ldr x1, [sp, AC]
      ldr x2, [sp, J]
      strb w0, [x1, x2]    // optional to reload C before doing this

      // j++;
      add x2, x2, 1        // optional to reload J before doing this
      str x2, [sp, J]

      // } while(j < 2)
      cmp x2, 2
      blo loop

      // return j;
epilog:
      ldr x0, [sp, J]
      ldr x30, [sp]
      add sp, sp, STACK_NUMBYTES
      ret
```

(Question 8 was the last question. The space below is intentionally left blank. You may use it for scratch work – which will not be graded – or to complete Question 8 as previously instructed.)

The theme of this exam was the fall 2022 Netflix hit release *Wednesday* … alas, the vast majority of students this term reported not having seen it. References in problem titles are italicized. Other references include the file/function name `brood` (which Wednesday does while playing the cello), the executable name `kooky` (a throwback to the old *Addams Family* theme song's lyrics), and the first line at the bottom of this page (a play on the traditional English nursery rhyme *Monday's Child*, which was used as the title for the first episode and is likely the source of the series' title character's name).

I hope your Wednesday's *exam* was **not** full of woe!
Thanks for a great semester!