Exam Statistics:

Minimum: 36
Median: 82
Maximum: 100

Mean: 80.63
StdDev: 13.62

Question averages:
Q1: 100%
Q2: 92%
Q3: 73%
Q4: 76%
Q5: 83%
Q6: 60%
Q7: 81%
Q8: 78%
Q9: 97%

Q1 Instructions and Pledge

1 Point

This exam consists of 8 multi-part questions (plus the pledge), and you have 180 minutes -- budget your time wisely.

This is a closed-book, closed-note exam, and "cheat sheets" are not allowed. During the exam you must not refer to the textbook, course materials, notes, or any information on the Internet. You may not compile or run any code on armlab or any other machine.

You are not allowed to communicate with any other person, whether inside or outside the class. You may not send the exam problems to anyone, nor receive them from anyone, nor communicate any information about the problems or their topics. *If you have technical issues or need to ask a clarifying question about the wording of some problem, please post a **private** message on Ed.*

You may use blank paper as scratch space, but you must enter your answer in the online system in order to receive credit.

This examination is administered under the Princeton University Honor Code, and by signing the pledge below you promise that you have adhered to the instructions above.

Please type out the Honor Code pledge exactly as follows, including this exact spelling and punctuation:

> *I pledge my honor that I have not violated the Honor Code during this examination.*

I pledge my honor that I have not violated the Honor Code during this examination.

Now type your name as a signature confirming that you have adhered to the Honor Code:

Q2 Make me!

8 Points

Here are fragments of modules that will be built into one executable named `testtable`. All pertinent information is shown.

```c
/* testtable.c */
#include <stdio.h>
#include "table.h"
... rest of testtable.c

/* table.h */
#ifndef TABLE_INCLUDED
#define TABLE_INCLUDED
#include <stddef.h>
#include "mydefs.h"
... rest of table.h
#endif

/* table.c */
#include "table.h"
#include "node.h"
... rest of table.c

/* node.h */
#ifndef NODE_INCLUDED
#define NODE_INCLUDED
#include "mydefs.h"
... rest of node.h
#endif

/* node.c */
#include "node.h"
... rest of node.c

/* mydefs.h */
#ifndef MYDEFS_INCLUDED
#define MYDEFS_INCLUDED
... rest of mydefs.h
#endif
```

You must now write a `Makefile` for this project that compiles with COS 217 best practices. Its structure will be as follows:

```
TARGET1: DEPENDENCIES1
        gcc217 testtable.o table.o node.o -o testtable
TARGET2: DEPENDENCIES2
        gcc217 -c testtable.c
TARGET3: DEPENDENCIES3
        gcc217 -c table.c
TARGET4: DEPENDENCIES4
        gcc217 -c node.c
```

And here are some options for target/dependency rules:

```
(A)  node.c: node.h
(B)  node.o: node.c
(C)  node.o: node.c node.h mydefs.h
(D)  table.o: table.c table.h node.h
(E)  table.o: table.c table.h node.h mydefs.h
(F)  table.o: table.c table.h stddef.h mydefs.h node.h mydefs.h
(G)  testtable: testtable.o table.o node.o
(H)  testtable: testtable.c table.c node.c table.h node.h
(I)  testtable: testtable.o table.o node.o table.h node.h mydefs.h
(J)  testtable.o: testtable.c table.h mydefs.h
(K)  testtable.o: testtable.c table.h node.h mydefs.h
(L)  testtable.o: testtable.c table.h stdio.h stddef.h
(M)  None of the above
```

For each of the target/dependency lines to be included, ***write the letter corresponding to the best option from the list above***. You will not use all options.

Q2.1

2 Points

```
TARGET1: DEPENDENCIES1
```

G

EXPLANATION

This, being the first target in the `Makefile`, is the rule to build the executable. It must depend only on `.o` files.

Q2.2

2 Points

```
TARGET2: DEPENDENCIES2
```

J

EXPLANATION

`testtable.o` should be built from the corresponding `.c` file and any `.h` files included, directly or indirectly, by it. However, it should not depend on standard library `.h` files.

Q2.3
2 Points

```
TARGET3: DEPENDENCIES3
```

E

EXPLANATION

`table.o` should be built from the corresponding `.c` file and any `.h` files included, directly or indirectly, by it. However, it should not depend on standard library `.h` files. Also, `.h` dependencies should not be listed twice.

Q2.4
2 Points

```
TARGET4: DEPENDENCIES4
```

C

EXPLANATION

`node.o` should be built from the corresponding `.c` file and any `.h` files included, directly or indirectly, by it.

Q3 My memory is failing me
14 Points

For each code snippet below, indicate which of the listed memory management issues the code exhibits, if any. (If it exhibits multiple issues, select the *first* one encountered.) ***Assume that memory allocation always succeeds***, that all necessary `#include`s are present, and that there is no other relevant code outside of that shown. (Specifically, if the code shown fails to free some allocated memory, it has a memory leak -- assume that later code ***does not*** free anything.)

Q3.1
2 Points

```
int *pi = calloc(sizeof(int), 5);
int *pi2 = pi;
pi2[1] = 42;
```

```
free(pi2);
```

- ◯ This code leaks memory.
- ◯ This code writes to a memory location it shouldn't.
- ◯ This code reads from a memory location it shouldn't.
- ◯ This code calls `free` on a pointer it shouldn't.
- ◉ This code has none of the above issues.

EXPLANATION

`pi` and `pi2` point to the same place, so calling `free` on either one is equivalent.

Q3.2

2 Points

```
int *pi = calloc(sizeof(int), 5);
pi[pi[4]] = 42;
free(pi);
```

- ◯ This code leaks memory.
- ◯ This code writes to a memory location it shouldn't.
- ◯ This code reads from a memory location it shouldn't.
- ◯ This code calls `free` on a pointer it shouldn't.
- ◉ This code has none of the above issues.

EXPLANATION

`calloc` initializes allocated memory to zero, so the second line is equivalent to `pi[0] = 42`.

Q3.3

2 Points

```
char *vacation = "Summer of sun";
vacation[10] = 'f';
printf("%s\n", vacation);
```

- [ ] This code leaks memory.
- [x] This code writes to a memory location it shouldn't.
- [ ] This code reads from a memory location it shouldn't.
- [ ] This code calls `free` on a pointer it shouldn't.
- [ ] This code has none of the above issues.

EXPLANATION

String literals are stored in `rodata`, which means they can't be edited.

Q3.4

2 Points

```
char *palindrome = "racecar";
char reversed[7];
size_t i;
for (i = 0; i < strlen(palindrome); i++)
    reversed[i] = palindrome[6 - i];
printf("%s\n", reversed);
```

- [ ] This code leaks memory.
- [ ] This code writes to a memory location it shouldn't.
- [x] This code reads from a memory location it shouldn't.
- [ ] This code calls `free` on a pointer it shouldn't.
- [ ] This code has none of the above issues.

EXPLANATION

This code fails to allocate memory for, and copy, the terminating byte of the string. So, the `printf` call will read off the end of the array.

Q3.5

2 Points

```
char greeting[5];
strcpy(greeting, "Hiya");
printf("%s\n", greeting);
free(greeting);
```

- [ ] This code leaks memory.
- [ ] This code writes to a memory location it shouldn't.
- [ ] This code reads from a memory location it shouldn't.
- [x] This code calls `free` on a pointer it shouldn't.
- [ ] This code has none of the above issues.

EXPLANATION

`greeting` is a stack-allocated array, which means it should not be `free`'d.

Q3.6

2 Points

```
int **ppi = malloc(sizeof(int*));
*ppi = malloc(sizeof(int));
**ppi = 42;
free(ppi);
```

- [x] This code leaks memory.
- [ ] This code writes to a memory location it shouldn't.
- [ ] This code reads from a memory location it shouldn't.
- [ ] This code calls `free` on a pointer it shouldn't.
- [ ] This code has none of the above issues.

EXPLANATION

This code calls `malloc` twice, but only frees one of the resulting allocations.

Q3.7

2 Points

```
char *pc = malloc(sizeof(char));
int *pi = (int*)pc;
int i = *pi;
printf("%d", i);
free(pc);
```

- ○ This code leaks memory.
- ○ This code writes to a memory location it shouldn't.
- ◉ This code reads from a memory location it shouldn't.
- ○ This code calls `free` on a pointer it shouldn't.
- ○ This code has none of the above issues.

EXPLANATION

The call to `malloc` only allocated 1 byte of memory. Accessing it as an `int` reads 4 bytes.

Q4 Where did I put that variable again?

14 Points

Each of the following declarations, when encountered ***inside a function body***, will cause memory to be allocated. This may happen either at compile/link time or at run time, in one or more of the stack, heap, rodata, data, and/or bss sections. For each variable, how much memory is allocated and where? Assume that the code is compiled ***without optimization***.

Q4.1

1 Point

```
static int var1 = 42;
```

- ○ 1 byte
- ○ 2 bytes
- ○ 3 bytes
- ◉ 4 bytes
- ○ 8 bytes
- ○ 42 bytes

Q4.2

1 Point

(Same code as in 4.1)

- ○ stack
- ○ heap
- ○ rodata
- ◉ data
- ○ bss

EXPLANATION

An `int` on armlab is 32 bits long. This is a static variable, is not read-only, and is initialized, so it goes in data.

Q4.3

1 Point

```
static long var2;
```

- ○ 1 byte
- ○ 2 bytes
- ○ 3 bytes
- ○ 4 bytes
- ◉ 8 bytes
- ○ 42 bytes

Q4.4

1 Point

(Same code as in 4.3)

- ○ stack
- ○ heap
- ○ rodata
- ○ data
- ◉ bss

EXPLANATION

This is an uninitialized static variable, so it goes in the bss section. A `long` is 64 bits on armlab.

Q4.5

1 Point

```
unsigned short var3;
```

- ○ 1 byte
- ◉ 2 bytes
- ○ 3 bytes
- ○ 4 bytes
- ○ 8 bytes
- ○ 42 bytes

Q4.6

1 Point

(Same code as in 4.5)

- ◉ stack
- ○ heap
- ○ rodata
- ○ data
- ○ bss

EXPLANATION

This is a local variable, and is allocated on the stack. An `unsigned short` is 2 byes long.

Q4.7

1 Point

```
const char *var4 = "42";
```

Considering ***only*** `var4` (as opposed to `*var4`), how much memory is allocated and where?

- ○ 1 byte
- ○ 2 bytes
- ○ 3 bytes
- ○ 4 bytes
- ◉ 8 bytes
- ○ 42 bytes

Q4.8

1 Point

(Same code as in 4.7)

- ◉ stack
- ○ heap
- ○ rodata
- ○ data
- ○ bss

EXPLANATION

`var4` is a pointer, so is 8 bytes long. The pointer itself is a local variable, so it goes on the stack.

Q4.9

1 Point

(Same code as in 4.7)

Considering ***only*** `*var4` (as opposed to `var4`), how much memory is allocated and where?

- ○ 1 byte
- ○ 2 bytes
- ◉ 3 bytes
- ○ 4 bytes
- ○ 8 bytes
- ○ 42 bytes

Q4.10
1 Point

(Same code as in 4.7)

- ○ stack
- ○ heap
- ◉ rodata
- ○ data
- ○ bss

EXPLANATION

`var4` points to a string literal, which lives in rodata. The string is 3 bytes long, counting the `'\0'` at the end.

Q4.11
1 Point

```
char *var5 = malloc(42 * sizeof(char));
```

Considering ***only*** `var5` (as opposed to `*var5`), how much memory is allocated and where?

- ○ 1 byte
- ○ 2 bytes
- ○ 3 bytes
- ○ 4 bytes
- ◉ 8 bytes
- ○ 42 bytes

Q4.12
1 Point

(Same code as in 4.11)

- (selected) stack
- heap
- rodata
- data
- bss

EXPLANATION

`var5` is a pointer, so is 8 bytes long. The pointer itself is a local variable, so it goes on the stack.

Q4.13

1 Point

(Same code as in 4.11)

Considering ***only*** `*var5` (as opposed to `var5`), how much memory is allocated and where?

- 1 byte
- 2 bytes
- 3 bytes
- 4 bytes
- 8 bytes
- (selected) 42 bytes

Q4.14

1 Point

(Same code as in 4.11)

- stack
- (selected) heap
- rodata
- data
- bss

EXPLANATION

`var5` points to memory allocated on the heap by `malloc`. `sizeof(char)` is 1, so `malloc` allocated 42 bytes.

Q5 I can't find my bit whacker

10 Points

You are given the task of writing a function with the following signature:

```
int mask(int iSrc, int iNumBits);
```

The aim is to *mask off* the specified number of bits from a 32-bit `int`. That is, the function should set everything except the `iNumBits` least-significant (rightmost) bits of `iSrc` to zero, and return the result. For example, a call to `mask(27, 4)` should return 11, because 27 is 11011 in binary, and masking off the 4 least-significant bits yields 1011 in binary, or 11 in decimal.

Consider the following attempts at writing `mask`, not all of which are successful. For each function, ***indicate what it returns*** for `mask(27, 4)`. Assume that any needed header files have been included, and any needed libraries are linked.

The `pow(x, y)` function returns `x` raised to the power `y`.

The operation `~x` computes the bitwise complement of `x`.

The operation `x << y` computes `x` shifted left by `y` bits, filling in on the right with zeroes.

The operation `x >> y` computes `x` shifted right by `y` bits. You should assume that, when executed on signed numbers, it implements an ***arithmetic shift*** that fills in on the left with whatever is in `x`'s most-significant (leftmost) bit.

Hint: each of the possible answers occurs exactly once in the five code snippets below.

Q5.1

2 Points

```
int mask(int iSrc, int iNumBits) {
    return iSrc & iNumBits;
}
```

- ○ -5
- ◉ 0
- ○ 1
- ○ 11
- ○ Non-deterministic. No way to tell.

EXPLANATION

We'd need to bitwise-and `iSrc` with a variable containing `iNumBits` ones (and the rest zeroes), not with `iNumBits` itself.

Q5.2

2 Points

```
int mask(int iSrc, int iNumBits) {
    int result;
    int i;

    for (i = 0; i < iNumBits; i++)
      result = (result << 1) + 1;

    result = iSrc & result;
    return result;
}
```

- ○ -5
- ○ 0
- ○ 1
- ○ 11
- ◉ Non-deterministic. No way to tell.

EXPLANATION

`result` is uninitialized, so this may or may not succeed depending on the initial contents of `result`.

Q5.3

2 Points

```
int mask(int iSrc, int iNumBits) {
    int result;
    result = (int) pow(2, iNumBits) - 1;

    result = iSrc && result;
    return result;
}
```

- -5
- 0
- (selected) 1
- 11
- Non-deterministic. No way to tell.

EXPLANATION

The computation of `result` uses logical-and (`&&`) instead of bitwise-and (`&`). Recall that `&&` returns 0 or 1, instead of performing an operation for each bit.

Q5.4

2 Points

```
int mask(int iSrc, int iNumBits) {
    int result = 0;

    result = ~result;
    result = result >> iNumBits;
    result = result << iNumBits;
    result = ~result;

    result = iSrc & result;
    return result;
}
```

- -5
- 0
- 1
- (selected) 11
- Non-deterministic. No way to tell.

EXPLANATION

Despite being a bit convoluted, this code sequence always succeeds. Note that the first right-shift leaves `result` unchanged.

Q5.5

2 Points

```
int mask(int iSrc, int iNumBits) {
```

```
    int result = 0;

    result = iSrc << (32 - iNumBits);
    result = result >> (32 - iNumBits);

    return result;
}
```

- (selected) -5
- 0
- 1
- 11
- Non-deterministic. No way to tell.

EXPLANATION

This sometimes succeeds. However, `result` is signed, so if there's a 1 in the leftmost bit after the left shift, the right shift will fill in 1's on the left.

Q6 I'm casting about for answers

12 Points

Consider this translation from a portion of a C program to AARCH64 assembly language. A reference for the relevant AARCH64 instructions is included below.

```
// varI = (CAST_1) varA;
ldrb   w0, [sp, varA]
str    w0, [sp, varI]

// varJ = (CAST_2) varB;
ldrsb  w0, [sp, varB]
str    w0, [sp, varJ]

// if (varI < varJ + 1) goto label1;
ldr    w0, [sp, varI]
ldr    w1, [sp, varJ]
add    w1, w1, 1
cmp    w0, w1
blt    label1
```

Instruction	Description
`ldr dst, [src]`	Load word (32 bits) or quad (64 bits) to `dst`

Instruction	Description
`ldrb dst, [src]`	Load byte to `dst` with zero-extension
`ldrsb dst, [src]`	Load byte to `dst` with sign-exteinsion
`str src, [dst]`	Store word (32 bits) or quad (64 bits) at `dst`
`add dst, src1, src2`	Add `src1` and `src2`, storing result in `dst`
`cmp src1, src2`	Compare `src1` and `src2`, setting condition flags
`blt label`	Branch to `label` if (signed) less than

Q6.1

3 Points

What is the most likely type for `varA`?

- ○ `signed char`
- ◉ `char` / `unsigned char` (equivalent on armlab)
- ○ `int`
- ○ `unsigned int`
- ○ `long`
- ○ pointer

EXPLANATION

The `ldrb` implies a single-byte variable (a `char` of some kind), and the zero-extend implies that `varA` is unsigned.

Q6.2

3 Points

What is the most likely type for `CAST_1`?

- ○ `signed char`
- ○ `char` / `unsigned char` (equivalent on armlab)
- ◉ `int`
- ○ `unsigned int`
- ○ `long`
- ○ pointer

EXPLANATION

The `blt` instruction implies a signed comparison, while the `w` register implies 32 bits, which rules out a pointer.

Q6.3

3 Points

What is the most likely type for `varB`?

- ◉ `signed char`
- ○ `char` / `unsigned char` (equivalent on armlab)
- ○ `int`
- ○ `unsigned int`
- ○ `long`
- ○ pointer

EXPLANATION

The `ldrsb` implies a single-byte variable (a `char` of some kind), and the sign-extend implies that `varB` is signed.

Q6.4

3 Points

What is the most likely type for `CAST_2`?

- ○ `signed char`
- ○ `char` / `unsigned char` (equivalent on armlab)
- ◉ `int`
- ○ `unsigned int`
- ○ `long`
- ○ pointer

EXPLANATION

The `blt` instruction implies a signed comparison, while the `w` register implies 32 bits, which rules out a pointer. Note the fact that we used `blt` means that both `CAST_1` and `CAST_2` were to (signed) `int`. If either operand of the comparison were `unsigned`, the other operand would have been implicitly cast to `unsigned`, and we would have needed an unsigned branch instruction (such as `blo`).

Q7 I feel lucky

18 Points

Consider the following AARCH64 program:

```
    .section .rodata
scanfFormat:  .string "%d"
printfFormat: .string "%d\n"

    .section .text
f:
    sub sp, sp, 16
    str x30, [sp]
    bl rand
    and w0, w0, 1
    ldr x30, [sp]
    add sp, sp, 16
    ret

    .global main
main:
    sub sp, sp, 32
    str x30, [sp]
    str x19, [sp,8]
    str x20, [sp,16]

    adr x0, scanfFormat
    add x1, sp, 24
    bl scanf
    cmp w0, 1
    bne leave
```

```
    ldr w19, [sp,24]
    mov w20, 0
loop:
    cmp w19, 0
    ble postLoop
    bl f
    add w20, w20, w0
    sub w19, w19, 1
    b loop

postLoop:
    adr x0, printfFormat
    mov w1, w20
    bl printf

leave:
    ldr x30, [sp]
    ldr x19, [sp,8]
    ldr x20, [sp,16]
    add sp, sp, 32
    mov w0, 0
    ret
```

Quick AARCH64 reference:

Instructions / Registers	**Description**
`add/sub/and dst, src1, src2`	`dst = src1 +/-/& src2`
`mov dst, src`	Copy `src` to `dst`
`cmp src1, src2`	Compare registers, set condition flags
`adr dst, var`	Store address of `var` in `dst`
`ldr dst, [src]`	Load word or quad pointed to by `src` into `dst`
`str src, [dst]`	Store word or quad from `src` to memory pointed to by `dst`
`b label`	Unconditional branch to `label`
`bl label`	Branch to `label` and save return address in x30
`ret`	Return to address in x30

Instructions / Registers	Description
`bne/ble label`	Conditional branch if not equal / (signed) less than or equal
`x0..x7`	Hold parameters to function
`x0`	Holds return value from function
`x19..x28`	Callee-saved scratch registers

Q7.1
2 Points

Let's start by analyzing the function `f`.
How many parameters does it take as input?

- (•) 0
- ( ) 1
- ( ) 2
- ( ) A random number

EXPLANATION
The function does not use values that were provided in `x0..x7`. It manipulates `w0`, but only after getting the return value from `rand`.

Q7.2
2 Points

How many local variables does it use?

- (•) 0
- ( ) 1
- ( ) 2
- ( ) A random number

EXPLANATION
A giveaway is that it does not use the stack except for saving and restoring `x30`. It also does not use callee or caller saved registers.

Q7.3
2 Points

Recall that the `rand()` function returns a pseudorandom `int` in the range from 0 to some large number `RAND_MAX` (which happens to be 2147483647 on armlab). Given this, which game of chance is `f` most likely intended to simulate?

- (●) Flipping a coin -- odds 1 in 2
- ( ) Rolling a die -- odds 1 in 6
- ( ) Spinning a roulette wheel -- odds 1 in 38
- ( ) Playing the lottery -- odds 1 in 2147483648
- ( ) Guessing an answer on a COS 217 final -- odds unspecified, but probably not very good

EXPLANATION

It obtains a random integer, and then uses only its least-significant bit. So, it outputs 0 or 1 with equal probability.

Q7.4
2 Points

Now let's turn to `main`.
How many callee-saved registers does it use (not counting the return address)?

- ( ) 0
- ( ) 1
- (●) 2
- ( ) 3
- ( ) 4

EXPLANATION

The code saves, uses, and restores `x19` and `x20`.

Q7.5
2 Points

The first argument to `scanf` is `scanfFormat`. What is the *second* argument to `scanf`?

- ○ It doesn't have one
- ○ The value 24
- ◉ An address in `main`'s stack frame
- ○ An address in some other function's stack frame, possibly intended to cause a buffer overrun
- ○ The address of register `x1`

EXPLANATION

The stack frame is 32 bytes long (which we know because we subtracted 32 from `sp`), but the function prologue only stored 24 bytes' worth of stuff on it. We are free to use the last 8 bytes of the stack frame for a local variable, and the `add x1, sp, 24` instruction loads the address of that location in the stack into `x1`. This becomes the second argument to `scanf`. Note that this local variable is uninitialized at the point it's passed to `scanf` - it will get filled in with input from the command line.

Q7.6

2 Points

Recall that the return value of `scanf` is the number of format ("percent") directives that were successfully matched by user input. What does the program do if the user provides no valid input?

- ○ Behaves as if the user had typed in 0
- ○ Behaves as if the user had typed in 1
- ○ Uses an uninitialized value instead of user input
- ○ Crashes with a segmentation fault
- ◉ Exits cleanly without printing anything

EXPLANATION

If the return value from `scanf` is not equal to 1, the `bne` instruction jumps to `leave`.

Q7.7

2 Points

After `scanf` returns but before the loop, where does the value the user entered eventually wind up?

- ○ `w0`
- ○ `w1`
- ◉ `w19`
- ○ `w20`
- ○ Passed to `printf`

EXPLANATION

This is the purpose of the `ldr w19, [sp,24]` instruction - it loads the value that `scanf` placed at the location whose address was passed as its second argument.

Q7.8

2 Points

What does the program do if the user types in the number 42?

- ○ Prints `42`
- ◉ Prints the sum of 42 values returned by `f`
- ○ Enters an infinite loop
- ○ None of the above

EXPLANATION

The loop executes as long as `w19` is positive, and decrements that value at the end of each iteration.

Q7.9

2 Points

Suppose you change `ble postLoop` to `beq postLoop`. (The latter branches on "equal".) Now what does the program do if the user types in ***the number -42***?

- ○ Prints `-42`
- ○ Prints the negative of the sum of 42 values returned by `f`
- ○ Enters an infinite loop
- ◉ None of the above

EXPLANATION

The loop will execute until `x19` is decremented to the most negative 32-bit signed integer, wraps around to the most positive signed int, and then eventually is decremented back to 0. This isn't an infinite loop, though it will take a long time.

Q8 No clever title, just arithmetic

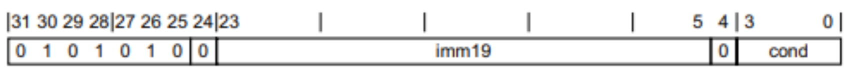
9 Points

Suppose that registers x0 and x1 contain variables corresponding to type `long`, and we have executed

```
cmp x0, x1
```

We now want to generate an AARCH64 machine language instruction that branches to `label1` if `x0 < x1`.

Here is the layout of the conditional branch instruction:



and here is what the `cond` bits mean:

cond	Mnemonic	Meaning (integer)	Meaning (floating-point)[a]	Condition flags
0000	EQ	Equal	Equal	Z == 1
0001	NE	Not equal	Not equal or unordered	Z == 0
0010	CS or HS	Carry set	Greater than, equal, or unordered	C == 1
0011	CC or LO	Carry clear	Less than	C == 0
0100	MI	Minus, negative	Less than	N == 1
0101	PL	Plus, positive or zero	Greater than, equal, or unordered	N == 0
0110	VS	Overflow	Unordered	V == 1
0111	VC	No overflow	Ordered	V == 0
1000	HI	Unsigned higher	Greater than, or unordered	C ==1 && Z == 0
1001	LS	Unsigned lower or same	Less than or equal	!(C ==1 && Z ==0)
1010	GE	Signed greater than or equal	Greater than or equal	N == V
1011	LT	Signed less than	Less than, or unordered	N! = V
1100	GT	Signed greater than	Greater than	Z == 0 && N == V
1101	LE	Signed less than or equal	Less than, equal, or unordered	!(Z == 0 && N == V)
1110	AL	Always	Always	Any
1111	NV[b]	Always	Always	Any

Q8.1

2 Points

The first thing we need to do is figure out the code for the condition we want. What bits should go in `cond` in the instruction format?

- ○ 0000
- ○ 0011
- ○ 1001
- ◉ 1011
- ○ 1101

EXPLANATION

We need a "signed less than", or `blt` instruction.

Q8.2

4 Points

Next, we need to figure out the offset to encode in the instruction. Suppose that the current instruction is at address 0x204 and that `label1` is at 0x1f4. What ***binary value*** should go in the 19-bit immediate (i.e., `imm19`) field of the instruction?

Hint 1: Remember that all AARCH64 instructions must be located at addresses that are a multiple of 4, so the conditional branch instruction saves space by not encoding the two least-significant (rightmost) bits of the offset, which must be 0.

Hint 2: Unless you're proficient in two's complement arithmetic, consider doing the subtraction and division *before* converting to binary.

- ○ 1111 1111 1111 1111 000
- ○ 1111 1111 1111 1111 011
- ◉ 1111 1111 1111 1111 100
- ○ 1111 1111 1111 1111 101
- ○ 1111 1111 1111 1111 110
- ○ 1111 1111 1111 1111 111

EXPLANATION

The offset is negative 0x10, or -16 decimal. Dropping two bits means dividing by 4, giving us -4. Finally, converting to two's complement gives us the 19-bit value above.

Q8.3

3 Points

What is the hex value of the ***byte*** at address 0x207? (Recall that the instruction starts at address 0x204.) ***Hint:*** Consider endianness.

- ○ 0x45
- ◉ 0x54
- ○ 0x63
- ○ 0x8b
- ○ 0xb8
- ○ None of the above

EXPLANATION

The full instruction is 0x54ffff8b. We are on a little-endian architecture, so the bytes are stored in the order 0x8b, 0xff, 0xff, 0x54 at locations 0x204, 0x205, 0x206, and 0x207, respectively.

Q9 Will this be on the test?

14 Points

Here are several possible strategies for testing:

A	Boundary Testing
B	Field Testing
C	Invariant Testing
D	Path Testing
E	Regression Testing
F	Statement Testing
G	Stress Testing

For each of the following descriptions, ***enter the letter*** corresponding to the type of testing being described. You will use each letter exactly once.

Q9.1

2 Points

Beta testing by clients

B

EXPLANATION

All of these common terms are described in the testing lecture.

Q9.2

2 Points

Checking known relationships among state variables

C

Q9.3

2 Points

Executing every line of code

F

Q9.4
2 Points

Executing every possible combination of lines of code

D

Q9.5
2 Points

Running all the tests again after making any change to the code

E

Q9.6
2 Points

Using a large quantity of randomly generated input

G

Q9.7
2 Points

Using inputs likely to trigger corner cases

A