CS33: Intro Computer Organization
Fall 2021 Midterm

Name: _____

UID: _____

(Please write as legibly as possible!)

This is an open book, open notes exam, but you cannot share books/notes. Please follow the university guidelines in reporting academic misconduct.

Note: If the architecture of the machine is not specified, assume that the question is being asked in the context of a 64-bit little endian x86 machine, running Linux.

Please wait until everyone has their exam to begin. We will let you know when to start.

Good luck!

	Points Possible
1. Multiple Choice	14
2. Baba is Struct	8
3. Notable Floats	5
4. Array Interpretation	8
5. Imaginary Stack Allocator	10
6. Tricky Switch	7
7. Floating Point Mystery	*4*
8. Two Birds, One Bomb	*6*
Total	50 +12 bonus points

Question 1. Multiple Choice (14 points)

For the following multiple choice questions, **select all answers that apply.** If none are correct, leave the question blank. Put your answers in the table on the next page. (2pts each, no partial)

1. What is a difference between unsigned and signed integer representations?
    a. Unsigned integers can store a wider range for the same number of bits.
    b. Right-shifting an unsigned integer uses "logical" shift, while right-shifting a signed integer uses "arithmetic" shift.
    c. In a C expression that operates on two different datatypes, an unsigned datatype will take precedence over a floating-point datatype, but a signed datatype will not.
    d. It is meaningful to ask if a signed number is greater-than or equal to zero, while the same is meaningless for unsigned numbers.

2. Suppose you use the objdump command to disassemble a function, and you see this:

```
0000000000400595 <func>:
  400595:  53              push   %rbx
```

    What does the "53" in the above line represent?
    a. The function has a length (in terms of instructions) of 53 bytes.
    b. The push instruction is 53-bytes offset from the beginning of the function.
    c. "53" contains an encoding of the register %rbx.
    d. "53" contains an encoding of the push operation.
    e. Address 53 is the location of where a callee-saved register is pushed.

3. X86_64 contains an instruction which performs a conditional move -- **cmov** --, which moves one register to another based on the condition codes (aka. flags). This instruction can sometimes be used to perform if-statement control flow. When is performing if statements using conditional moves a better option than using ordinary branch instructions?
    a. When the body of the if statement contains function calls.
    b. When the body of the if statement contains side effects.
    c. When the body of the if statement contains many instructions.

4. X86_64 contains an instruction which performs an indirect jump -- **jmp*** --, which jumps to a location specified in a memory table. This instruction can sometimes be used to perform switch-case-statement control flow. When is performing switch statements using indirect jumps a better option than using ordinary branch instructions?
    a. When there are more fall-through cases than non-fall-through cases.
    b. When there are few instructions in the case statements.
    c. When there are many instructions in the case statements.
    d. When the first case value is zero.
    e. When the range of case values is contiguous.

5. Suppose that the next C language standard contains an 11-bit unsigned integer datatype. What would be a valid reason (or reasons) to reject this proposal?
    a. There are already larger and smaller datatypes (e.g. 8-bit and 16-bit), so it's not useful.
    b. Modern ISA use byte-addressable memory, so accessing arrays of contiguous 11-bit integers would require extra instructions to extract the number.
    c. For a two's complement number to be well-defined, its size must be a multiple of 2.
    d. Having an 11-bit number would make it impossible to satisfy the datatype casting rules in the C standard.

6. For what C datatypes is the concept of "endianness" irrelevant?
    a. char
    b. unsigned char
    c. string (i.e. array of char)
    d. int
    e. float

7. What kind of control flow is contained in this assembly function? (see figure below)
    a. Loop (e.g. while/for)
    b. Conditional Branch (if/else)
    c. Indirect Branch (switch/case)

```
0000000000000000 <func>:
   0:   89 f0           mov    %esi,%eax
   2:   0f bf d0        movswl %ax,%edx
   5:   8d 0c 3a        lea    (%rdx,%rdi,1),%ecx
   8:   83 f9 02        cmp    $0x2,%ecx
   b:   7e 0b           jle    18 <func+0x18>
   d:   39 fa           cmp    %edi,%edx
   f:   7f 04           jg     15 <func+0x15>
  11:   01 ff           add    %edi,%edi
  13:   eb ed           jmp    2 <func+0x2>
  15:   89 f8           mov    %edi,%eax
  17:   c3              retq
```

Answer Table (list any correct answers)

1	
2	
3	
4	
5	
6	
7	

Question 2. Baba is Struct (8 Pts)

Consider the following structure, union, and array definitions:

```
typedef struct {
  int baba;
  short flag[5];
  float* keke;
  char key;
} noun;

typedef union {
  int hot;
  short shut[5];
  float* stop;
  char open;
} property;

noun you[5];
property me[5];
```

```
// Note: Typedef here just means
that were are defining a struct
type that we can use later in the
array definition.

// Create an “arrays of structs”
and an “array of unions”
```

Test is

1. If &you == 0 (i.e. if the address of you[0] is zero), at what address is you[1]?

2. If &me == 0 (i.e. if the address of me[0] is zero), at what address is me[1]?

3. If we access “property.shut[1]”, that will also access half of the bits in “property.hot”. Will that be accessing the least significant bits of “hot” or the most significant?

4. Consider all the primitive data types within the arrays “you” and “me”. Which of these are guaranteed to have their addresses aligned to a multiple of their size?

win

Question 3. Notable Floats (5 Pts)

The following table shows a number of “interesting” values for floating-point numbers, along with their encoding into sign, exp, and frac fields.

Pattern	sign	exp	frac
A	0	00..00	00..00
B	0	00..00	00..01
C	0	00..01	11..11
D	0	00..01	00..00
E	0	01..11	00..00
F	0	11..10	11..10
G	0	11..11	00..00
H	0	11..11	11..11

Match the following definitions to the interesting numbers above (write “A”, “B”, etc. in the box)

0. Zero	
1. One	
2. Smallest Non-zero Denorm	
3. Smallest Possible Normalized Number	
4. Not a Number (NaN)	

Note: These patterns don’t specify the number of bits, but that won’t matter for answering the question. Assume the same strategy for representation of denormalized numbers, NaNs and infinities as other IEEE 754 standard numbers.

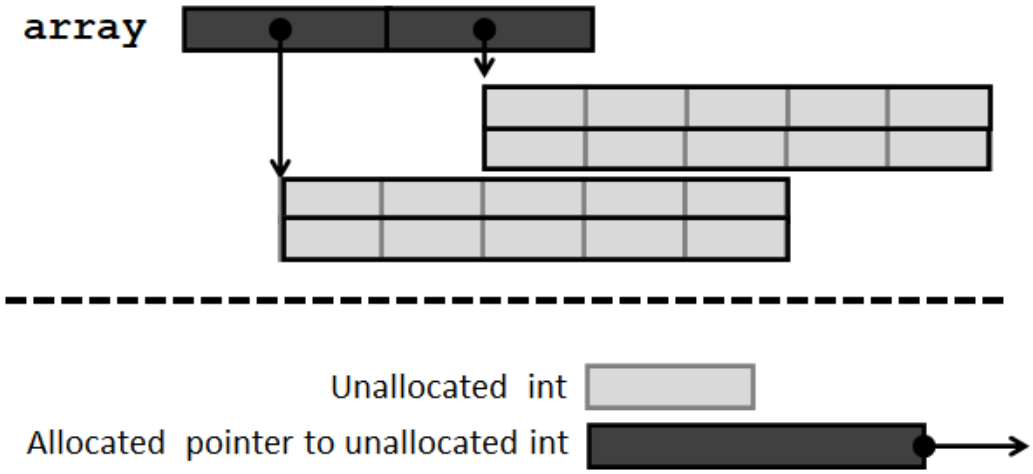
Question 4. Array Interpretation (8 pts)

1. For the following datatype definitions, answer the following questions: (6pts)

- **sizeof(array):** What is the size of “array” in terms of number of bytes?
  For this problem, we are only talking about the memory allocated for the variable “array”, and not any other supporting data structures.
- **How many dereferences?:** List the number of memory dereferences that it would take to access an integer for that datatype. In other words, how many times do you have to access memory *total* (how many loads), to eventually access a single integer. Include the load of the integer itself.

	Array Declaration	sizeof(array)	How many dereferences?
	int* array	8	1
1	int array[3][2]		
2	int (*array)[5]		
3	int *array[5]		

2. What is the array declaration for the following array, represented visually below? (2pts)



4	Datatype for array:	

Question 5. Imaginary Stack Allocator (10pts)

Part 1: Suppose we have two functions, FuncP and FuncQ. FuncP calls FuncQ, and the stack frames of both functions are depicted below.

All functions require a "prologue" and "epilogue" to manage the stack. The prologue allocates stack space, and usually appears at the beginning of the function. The epilogue deallocates stack space, and usually appears at the end of the function.

Based on the stack frame, write the prologue and epilogue for FuncQ. Don't use more instructions than you need to.

		Stack Frames	
Prologue for FuncQ:		Arg. 8	FuncP
		Arg. 7	
		Return Address	
		Old r12	FuncQ
		Old rbp	
Epilogue for FuncQ:		Unused (16 bytes)	
		Space for C array int array[4]	
	rsp->		

Part 2: FuncR is another function, and its assembly is shown below. Fill in the prologue and epilogue for this function too!

Prologue for FuncR:	
Epilogue for FuncR:	

```
___missing_FuncR_prologue___
movq    %rdi, %rbx
movq    $33, 8(%rsp)
movl    $1234, %esi
leaq    8(%rsp), %rdi
call    FuncS
addq    %rbx, %rax
___missing_FuncR_epilogue___
```

Question 6. Tricky Switch (7 pts)

Source Code	Compiled Assembly

`int func(int x, int y, int r) {`
`  switch (x) {`
`  case 0: ___Blank 0___`
`  case 1: ___Blank 1___`
`  case 2: ___Blank 2___`
`  case 3: ___Blank 3___`
`  case 4: ___Blank 4___`
`  case 5: ___Blank 5___`
`  }`
`  return ___Blank 6___;`
`}`
	`func(int, int, int):`
	`  cmpl  $5, %edi`
	`  ja  .L9`
	`  movl  %edi, %edi`
	`  jmp  *.L4(,%rdi,8)`
	`.L4:  (hint, this is the`
	`  .quad  .L8  jump table!)`
	`  .quad  .L7`
	`  .quad  .L6`
	`  .quad  .L5`
	`  .quad  .L5`
	`  .quad  .L3`
	`.L7:`
	`  addl  $6, %edx`
	`.L6:`
	`  leal  (%rdx,%rdx,4), %eax`
	`  ret`
	`.L5:`
	`  leal  (%rdx,%rsi,2), %eax`
	`  ret`
	`.L3:`
	`  movl  %edx, %eax`
	`  xorl  %esi, %eax`
	`  ret`
	`.L8:`
	`  movl  $-4, %eax`
	`  Ret`
	`.L9:`
	`  movl  %edx, %eax`
	`  ret`

Reverse engineer the assembly code on the previous page to figure out what each case of the switch-case statement is doing. Don't forget about "break" statements!

Blank 0	
Blank 1	
Blank 2	
Blank 3	
Blank 4	
Blank 5	
Blank 6	

Question 7. Floating Point Mystery (4pts)

A long time ago, we used to put floating-point questions on the datalab. I found a solution to one of these problems lying around, but can't figure out what it's doing anymore:

```
unsigned mystery_function(unsigned uf) {
  unsigned sign = uf>>31;
  unsigned exp = uf>>23 & 0xFF;
  unsigned frac = uf & 0x7FFFFF;
  if (exp == 0) {
    frac = 2*frac;
    if (frac > 0x7FFFFF) {
      frac = frac & 0x7FFFFF;
      exp = 1;
    }
  } else if (exp < 0xFF) {
    exp++;
    if (exp == 0xFF) {
      frac = 0;
    }
  }
  return (sign << 31) | (exp << 23) | frac;
}
```

1. In what cases will this function return the same thing as the input argument? (1pt)

2. What does the above function do? (3pts)

Question 8. Two Birds, One Bomb (6pts)

```
Dump of assembler code for function phase_5:
   0x00005555555551c8 <+0>:     sub    $0x8,%rsp
   0x00005555555551cc <+4>:     mov    $0xa,%edx
   0x00005555555551d1 <+9>:     mov    $0x0,%esi
   0x00005555555551d6 <+14>:    callq  0x555555555070 <strtol@plt>
   0x00005555555551db <+19>:    mov    %rax,%rsi
   0x00005555555551de <+22>:    lea    0x2e5b(%rip),%rdi        # 0x555555558040 <nodes>
   0x00005555555551e5 <+29>:    callq  0x55555555519e <recurse>
   0x00005555555551ea <+34>:    cmp    $0x21,%eax
   0x00005555555551ed <+37>:    je     0x5555555551fe <phase_5+54>
   0x00005555555551ef <+39>:    mov    $0x0,%eax
   0x00005555555551f4 <+44>:    callq  0x55555555517e <explode_bomb>
   0x00005555555551f9 <+49>:    add    $0x8,%rsp
   0x00005555555551fd <+53>:    retq
   0x00005555555551fe <+54>:    mov    $0x0,%eax
   0x0000555555555203 <+59>:    callq  0x555555555169 <phase_defused>
   0x0000555555555208 <+64>:    jmp    0x5555555551f9 <phase_5+49>

Dump of assembler code for function recurse:
   0x000055555555519e <+0>:     test   %rdi,%rdi
   0x00005555555551a1 <+3>:     je     0x5555555551c2 <recurse+36>
   0x00005555555551a3 <+5>:     push   %rbx
   0x00005555555551a4 <+6>:     mov    0x10(%rdi),%ebx
   0x00005555555551a7 <+9>:     cmp    %esi,%ebx
   0x00005555555551a9 <+11>:    jge    0x5555555551b8 <recurse+26>
   0x00005555555551ab <+13>:    mov    0x8(%rdi),%rdi
   0x00005555555551af <+17>:    callq  0x55555555519e <recurse>
   0x00005555555551b4 <+22>:    add    %ebx,%eax
   0x00005555555551b6 <+24>:    pop    %rbx
   0x00005555555551b7 <+25>:    retq
   0x00005555555551b8 <+26>:    mov    (%rdi),%rdi
   0x00005555555551bb <+29>:    callq  0x55555555519e <recurse>
   0x00005555555551c0 <+34>:    jmp    0x5555555551b4 <recurse+22>
   0x00005555555551c2 <+36>:    mov    $0x0,%eax
   0x00005555555551c7 <+41>:    retq

(gdb) x/21gx &nodes
0x555555558040 <nodes>:     0x0000555555558058  0x0000555555558070  0x000000000000000a
0x555555558058 <nodes+24>:  0x0000555555558088  0x00005555555580a0  0x0000000000000005
0x555555558070 <nodes+48>:  0x00005555555580b8  0x00005555555580d0  0x000000000000000c
0x555555558088 <nodes+72>:  0x0000000000000000  0x0000000000000000  0x0000000000000001
0x5555555580a0 <nodes+96>:  0x0000000000000000  0x0000000000000000  0x0000000000000007
0x5555555580b8 <nodes+120>: 0x0000000000000000  0x0000000000000000  0x000000000000000b
0x5555555580d0:<nodes+144>: 0x0000000000000000  0x0000000000000000  0x0000000000000011
```

So, it turns out that I need a new phase_5 for the bomb lab, because the old phase_5 is waaaay too easy to cheat on … therefore, I made this phase_5 for next year, with a similar kind of flavor.

Please help me make sure this new phase_5 has no errors by solving it for me. I've printed out a couple functions and some memory values. Next year's class will surely thank you. ;)

Note that "strtol" just converts an ascii string (specified by a char input) into an integer.*

1. What datastructure is this problem concerned with? Please be specific. (1 pt)

2. What string passed to "phase_5(char* input)" will defuse the phase? (5 pts)