Full Name:_____

Mock Midterm Practice (Fall2022)

Instructions:

- Quiz II takes 70 minutes. Read through all the problems and complete the easy ones first.
- This exam is **closed book**, except that you may bring a single doube-sided page of prepared note.

1 (xx/32)	2 (xx/20)	3 (xx/28)	4 (xx/10)	Total (xxx/90)

This exam assumes 64-bit x86 hardware (little Endian) unless otherwise mentioned.

1 Machine Representation (32 points, 4 points each):

Multiple choices. Circle *all* answers that apply.

A. Suppose register %rbx corresponds to some C variable `v`. Given instruction `addq $4, %rbx`, what are *all* the potential types of `v` and the corresponding C statement of the instruction?

1. type: `int` , C statement: `v +=4;`
2. type: `unsigned int` , C statement: `v +=4;`
3. type: `long` , C statement: `v +=4;`
4. type: `unsigned long`, C statement: `v +=4;`
5. type: `int *` , C statement: `v +=4;`
6. type: `int *` , C statement: `v++;`
7. type: `long *` , C statement: `v +=4;`
8. type: `long *` , C statement: `v ++;`
9. none of the above

B. Suppose %rdi and %rsi corresponds to C variables `x` and `y`, respectively. Given instruction `movq (%rdi, %rsi, 8), %rax`, what are the most likely types of `x` and `y`, respectively?

1. `long` and `long`
2. `long *` and `long`
3. `long *` and `long *`
4. `int` and `int`
5. `int *` and `int`
6. `int *` and `int *`
7. none of the above

C. Which of the following machine instructions change the value of register `%rsp`?

1. `ret`
2. `pushq %rax`
3. `popq %rbx`
4. `call`
5. `movq %rax, (%rsp)`
6. none of the above

D. Which of the following instructions read from or write to memory?

1. `ret`
2. `pushq %rax`
3. `popq %rbx`
4. `call`
5. `addq %rax, %rbx`
6. none of the above

E. Suppose `%rsi` corresponds to C variable `y` of some pointer type. Which of the following instructions dereference the pointer `y`?

1. `leaq (%rsi), %rax`
2. `movq (%rsi), %rax`
3. `movq %rsi, %rax`
4. `subq %rax, (%rsi)`
5. `subq %rax, %rsi`
6. none of the above

F. Consider the following recursive function:

```
void foo(int c) {
   if (c <= 0)
       return;
   foo(c-1);
}
```

What is the minimum stack size (in bytes) needed in order to execute `foo(128)` successfully[1]?

1. 8096
2. 1024
3. 512
4. 128
5. any value

G. Which of the following statements on segmentation faults are true?

1. Performing any out-of-bounds array access will result in an immediate segmentation fault.
2. Segmentation faults only occur when an instruction tries to write to memory.
3. Dereferencing a null pointer will always result in a segmentation fault.
4. Performing pointer arithmetic will always result in a segmentation fault.
5. none of the above

H. Suppose *local variable* `a` is defined as `int a[16];` Which of the following statements are true?

1. `a` takes up space on the stack.
2. `a` takes up space on the heap.
3. `subq $64, %rsp` allocates space for `a` and `addq $64, %rsp` de-allocates space for `a`.
4. `addq $64, %rsp` allocates space for `a` and `subq $64, %rsp` de-allocates space for `a`.
5. none of the above

[1] You should assume that the compiler does not perform tail call optimization to avoid allocating a stack frame per function call.

2 Basic C (20 points, 4 points each)

Answer the following multiple-choice questions. Circle *all* answers that apply. Each is 5 points.
A. In the following code, what's the most likely outcome of line 4?

```
1: int a = 5;
2: int *p;
3: p = (int *)a;
4: printf("%d\n", *p);
```

1. It will produce a compilation error.
2. It will cause a segmentation fault.
3. It will print 5.
4. It will print some memory address.
5. None of the above

B. In the following code, what should be at line 3 in order for character 'd' to be printed at line 4?

```
1: char *s = "abcdef";
2: char v;
3: v = _____;
4: printf("%c\n", v);
```

1. `v = s+3;`
2. `v = *(s+3);`
3. `v = s[3];`
4. `v = *(++s);`
5. `v = ++s;`
6. None of the above

C. What is the output of the following code snippet?

```
char a[5] = {0, 0, -1, -1, -1};
int *p;
p = (int *)(a + 1);
printf("%d\n", *p);
```

1. a positive number
2. a negative number
3. 0
4. Segmentation fault
5. Compilation error

D. Suppose `x` is of type `unsigned int`. Which of the following statement computes 0 if and only if the i-th bit of x (starting from the left) is zero? (The 0-th bit corresponds to the most significant bit).

1. `x & 0x80000000`
2. `(x << i) >> i`
3. `x & (1 << (31-i))`
4. `x | (0x80000000 >> i)`
5. `x & (0x7fffffff >> i)`
6. None of the above

E. Suppose `x` is of type `unsigned int`. Which of the following statement sets the i-th bit of x (starting from the left) to be one? (The 0-th bit corresponds to the most significant bit).

1. `x &= (1 << i)`
2. `x |= (1 << i)`
3. `x |= (1 << (31-i))`
4. `x &= (1 << (31-i))`
5. `x &= ~(1 << (31-i))`
6. None of the above

3 C to assembly (28 points):

Ben Bitdiddle wrote the following `swap` function to swap two long integers. Its corresponding assembly code is also shown below.

```
1  void
2  swap(long *a, long *b)
3  {
4     long tmp;
5     tmp = *a;
6     *a = *b;
7     *b = tmp;
8  }
```

```
1  swap:
2  movq (%rdi), %rax
3  movq (%rsi), %rdx
4  movq %rdx, (%rdi)
5  movq %rax, (%rsi)
```

A. (2 points) Where are function arguments `a` and `b` stored at, respectively?

Answer: _____

B. (4 points) For each C statement, which is its corresponding assembly instruction (or set of assembly instructions)? Fill in the line number of the assemly instructions that correspond to each C statement in the table below.

C line number	Assembly line number
5	
6	
7	
6	

Ben Bitdiddle then implemented the `reverse_array` function to reverse the elements in an array of long integers. Note that `reverse_array` uses his previously implemented `swap` as a helper function. Ben's `reverse_array` C function and its corresponding assembly are shown below:

```
// sz is the number of elements in array a
1  void reverse_array(long *a, long sz)
2  {
3      long s = 0;
4      long e = sz-1;
5      while (s < e) {
6        swap((long *)a[s], (long *)a[e]);
7        s++;
8        e--;
9    }
10 }
```

```
1   reverse_str:
2         pushq   %r12
3         pushq   %rbp
4         pushq   %rbx
5         movq    %rdi, %r12
6         leaq    -1(%rsi), %rbx
7         movq    $0, %rbp
8   .L3:
9         cmpq    %rbx, %rbp
10        jge     .L6
11        movq    (%r12,%rbx), %rsi
12        movq    (%r12,%rbp), %rdi
13        call    swap
14        addq    $1, %rbp
15        subq    $1, %rbx
16        jmp     .L3
17 .L6:
18        popq    %rbx
19        popq    %rbp
20        popq    %r12
21        ret
```

C. (4 points) Where are the C variables `s` and `e` stored at, respectively?

Answer: _____

D. (2 points) Which line of assembly performs computes the condition that determines whether the loop in `reverse_array` should terminate or continue?

Answer: _____

E. (2 points) Which line(s) of assembly copy the values to be the arguments used by `swap` function?

Answer: _____

F. (2 points) Which C variable does `%r12` contain in the body of the loop?

1. `a`
2. `s`
3. `e`
4. `sz`
5. None of the above

G. (4 points) Suppose `%rsp` is `0x00007fffffffff20` *before* executing the first instruction of `reverse_str` (aka line 2 of assembly). What is the value of `%rsp` *immediately before* and *immediately after* executing line 13 of assembly?

Value of `%rsp` immediately before: _____

Value of `%rsp` immediately after: _____

Finally, Ben Bitdiddle writes the following program to test the correctness of `reverse_array`.

```
int
main()
{
  long a[4] = {1, 2, 3, 4};
  reverse_array(a, 4);
  assert(a[0] == 4);
  assert(a[1] == 3);
  assert(a[2] == 2);
  assert(a[3] == 1);
  return 0;
}
```

H. (4 points) Ben found out that running his program results in segmentation fault, and the offending instruction is the first instruction of `swap`, aka `movq (%rdi), %rax`. If Ben is to print out the value of `%rdi` in GDB, what is the value? Answer: _____

I. (4 points) Please fix Ben's bug. You may directly write your corrections on the given C functions.

4 More C Programming (10 points):

Please complete the following code to implement a function `hex2int` that converts a hex string to its integer value (by interpreting the bit pattern represented by the hex string as 2's complement).

```
// return the integer value of the ASCII hex digit c
// you may assume hex digits are always given in lower case
char hex2digit(char c) {









}

int hex2int(char *s) {
    assert(strlen(s) == 8); //assume the string always contains exactly 8 hex "digits"









}

void main() {
  char *s;
  s = "ffffffff";
  int x = hex2int(s);
  assert(x == -1);
  s = "0000000f";
  x = hex2int(s);
  assert(x == 15);
}
```

1. (5 points) Please complete the helper function `hex2digit` that converts a hex character to an integer value in the range $[0, 15]$. You may assume that hex characters are always in lower case.
2. (5 points) Complete `hex2int` to convert a hex string representing an integer in 2's complement to the corresponding integer. The expected return value of this function is demonstrated using two example inputs in `main`. You may assume that the hex string is always exactly 8 characters long and in lower case.

—END of Quiz II—-

5 Appedix: X86 Cheatsheet

5.1 Registers

x86 registers are 8-bytes. Additionally, the lower order bytes of these registers can be independently accessed as 4-byte, 2-byte, or 1-byte register. The register names are:

8-byte register	Bytes 0-3	Bytes 0-1	Byte 0 (lowest order byte)
%rax	%eax	%ax	%al
%rbx	%ebx	%bx	%bl
%rcx	%ecx	%cx	%cl
%rsi	%esx	%si	%sil
%rdi	%edi	%di	%dil

...the rest is omitted...

5.2 Instructions

Instruction suffixes:

"byte" (`b`)	1-byte
"word" (`w`)	2-bytes
"doubleword" (`l`)	4-bytes
"quardword" (`q`)	8-bytes

Complete memory addressing mode: A memory operand of the form `D(Rb, Ri, S)` accesses memory at address `D + val(Rb) + val(Ri)*S`, where val(Rb) and val(Ri) refer to the value of registers Rb and Ri respectively, D is a constant, and S is a constant of value 1, 2, 4, or 8.

Sign extension and zero extension:

movzlq S, D	copy 4-byte-sized S to 8-byte-sized D and fill in the higher order 4 bytes of D with zero byte
movslq S, D	copy 4-byte-sized S to 8-byte-sized D and sign extend the higher order 4 bytes of D, i.e. fill with 0s if S's sign bit is zero and fill with 1s if S's sign bit is one.

Basic Arithmatic instructions that you might not remember:

sal / shl k, D	Left shift destination D by k bits
sar	Arithmatic right shift destination D by k bits
shr	Logical right shift destination D by k bits

Jump instructions:

Jump instruction following `cmp S, D`:

jmp	Unconditional jump
je	Jump if D is equal to S
jne	Jump if D is not equal to S
jg	Jump if D is greater than S (signed)
jge	Jump if D is greater or equal than S (signed)
jl	Jump if D is less than S (signed)
jle	Jump if D is less or equal than S (signed)
ja	Jump if D is above S (unsigned)
jae	Jump if D above or equal S(unsigned)
jb	Jump is D is below S (unsigned)
jbe	Jump if D is below or equal S (unsigned)

5.3 Calling convention

Argument Passing:

Which argument	Stored in register
1	%rdi
2	%rsi
3	%rdx
4	%rcx
5	%r8
6	%r9
7 and up	passed on stack

Return value (if any) is stored in %rax

Caller save registers: %rax, %rcx, %rdx, %rdi, %rsi, %r8-11

Callee save registers: %rbx, %rbp, %r12-15

6 Appendix: ASCII

The following table contains the 128 ASCII characters.

Oct	Dec	Hex	Char	Oct	Dec	Hex	Char
000	0	00	NUL '\0'	100	64	40	@
001	1	01	SOH (start of heading)	101	65	41	A
002	2	02	STX (start of text)	102	66	42	B
003	3	03	ETX (end of text)	103	67	43	C
004	4	04	EOT (end of transmission)	104	68	44	D
005	5	05	ENQ (enquiry)	105	69	45	E
006	6	06	ACK (acknowledge)	106	70	46	F
007	7	07	BEL '\a' (bell)	107	71	47	G
010	8	08	BS '\b' (backspace)	110	72	48	H
011	9	09	HT '\t' (horizontal tab)	111	73	49	I
012	10	0A	LF '\n' (new line)	112	74	4A	J
013	11	0B	VT '\v' (vertical tab)	113	75	4B	K
014	12	0C	FF '\f' (form feed)	114	76	4C	L
015	13	0D	CR '\r' (carriage ret)	115	77	4D	M
016	14	0E	SO (shift out)	116	78	4E	N
017	15	0F	SI (shift in)	117	79	4F	O
020	16	10	DLE (data link escape)	120	80	50	P
021	17	11	DC1 (device control 1)	121	81	51	Q
022	18	12	DC2 (device control 2)	122	82	52	R
023	19	13	DC3 (device control 3)	123	83	53	S
024	20	14	DC4 (device control 4)	124	84	54	T
025	21	15	NAK (negative ack.)	125	85	55	U
026	22	16	SYN (synchronous idle)	126	86	56	V
027	23	17	ETB (end of trans. blk)	127	87	57	W
030	24	18	CAN (cancel)	130	88	58	X
031	25	19	EM (end of medium)	131	89	59	Y
032	26	1A	SUB (substitute)	132	90	5A	Z
033	27	1B	ESC (escape)	133	91	5B	[
034	28	1C	FS (file separator)	134	92	5C	\ '\\'
035	29	1D	GS (group separator)	135	93	5D	]
036	30	1E	RS (record separator)	136	94	5E	^
037	31	1F	US (unit separator)	137	95	5F	_
040	32	20	SPACE	140	96	60	`
041	33	21	!	141	97	61	a
042	34	22	"	142	98	62	b
043	35	23	#	143	99	63	c
044	36	24	$	144	100	64	d
045	37	25	%	145	101	65	e
046	38	26	&	146	102	66	f
047	39	27	'	147	103	67	g
050	40	28	(	150	104	68	h
051	41	29	)	151	105	69	i
052	42	2A	*	152	106	6A	j
053	43	2B	+	153	107	6B	k
054	44	2C	,	154	108	6C	l
055	45	2D	-	155	109	6D	m
056	46	2E	.	156	110	6E	n
057	47	2F	/	157	111	6F	o
060	48	30	0	160	112	70	p
061	49	31	1	161	113	71	q
062	50	32	2	162	114	72	r
063	51	33	3	163	115	73	s
064	52	34	4	164	116	74	t
065	53	35	5	165	117	75	u
066	54	36	6	166	118	76	v
067	55	37	7	167	119	77	w
070	56	38	8	170	120	78	x
071	57	39	9	171	121	79	y
072	58	3A	:	172	122	7A	z
073	59	3B	;	173	123	7B	{
074	60	3C	<	174	124	7C	|
075	61	3D	=	175	125	7D	}
076	62	3E	>	176	126	7E	~
077	63	3F	?	177	127	7F	DEL

7 Appendix III: strstr, strcpy, strncpy, strlen

```
STRSTR(3)                Linux Programmer's Manual                                STRSTR(3)

NAME
       strstr - locate a substring

SYNOPSIS
       #include <string.h>

       char *strstr(const char *haystack, const char *needle);

DESCRIPTION
       The  strstr()  function finds the first occurrence of the substring needle in the string haystack.
The terminating null bytes ('\0') are not compared.

RETURN VALUE
       The strstr() function returns a pointer to the beginning of the substring in haystack, or NULL
if the substring is not found.

-------------------------------------------------------------------------------------------------------
STRCPY(3)                 Linux Programmer's Manual                                 STRCPY(3)

NAME
       strcpy, strncpy - copy a string

SYNOPSIS
       #include <string.h>

       char *strcpy(char *dest, const char *src);
       char *strncpy(char *dest, const char *src, size\_t n);

DESCRIPTION
       The  strcpy() function copies the string pointed to by src, including the terminating null byte
('\0'), to the buffer pointed to by dest.  The strings may not overlap, and the destination string dest
must be large  enough to receive the copy.

       The strncpy() function is similar, except that at most n bytes of src are copied.  Warning: If there
is no null byte among the first n bytes of src, the string placed in dest  will  not  be null-terminated.

       If  the length of src is less than n, strncpy() writes additional null bytes to dest to ensure that
a total of n bytes are written.

RETURN VALUE
       The strcpy() and strncpy() functions return a pointer to the destination string dest.

-------------------------------------------------------------------------------------------------------
STRLEN(3)                  Linux Programmer's Manual                                 STRLEN(3)

NAME
     strlen - calculate the length of a string

SYNOPSIS
       #include <string.h>

       size_t strlen(const char *s);

DESCRIPTION
       The strlen() function calculates the length of the string pointed to by s, excluding the terminating
null byte ('\0').

RETURN VALUE
       The strlen() function returns the number of characters in the string pointed to by s.
```

Q1
A 3 4 6
B 2
C 1 2 3 4
D 1 2 3 4
E 2 4
F 2
G 3
H 1 3

Q2
A 2
B 2,3
C 2
D 3
E 3

Q3

(a) %rdi, %rsi
(b) 5 --> 2
    6 --> 3, 4
    7 --> 5

(c) s is %rbp
    e is %rbx

(d) line 9

(e) line 11 and 12

(f) 1

(g) before: 0x00007fffffffff08
    after: 0x00007fffffffff00

(h) 0x0000000000000001

(i) swap(&a[s], &a[e]);

Q4

```
char hex2digit(char c)
{
   if (c >= '0' && c <= '9') {
      return c - '0';
   } else if (c >= 'a' && c <= 'f') {
      return c - 'a' + 10;
   }
   assert(0);
}
```



```
int hex2int(char *s) {
   unsigned result = 0;
   for (int i = 0; i < 8; i++) {
      result = result << 4 + hex2digit(s[i]);
   }
   return (int)result;
}
```