Full Name:_____

Quiz I, Spring 2018 Date: 2/27

Instructions:

- Quiz I takes 60 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/25)	2 (xx/25)	3 (xx/25)	4 (xx/25)	Total (xx/100)

1 Machine representation, bitwise operation (25 points):

Answer the following multiple-choice questions. Circle *all* answers that apply. Each problem is worth 5 points.

A. Which of the following values is the closest to 1 million?

1. `1<<10`
2. `1<<20`
3. `1<<30`
4. `0x00100000`
5. `0x01000000`
6. `0x10000000`

B. Which of the following expression clears the most significant bit of the unsigned int variable `x`?

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

C. Which of the following expression evalutes to 0 *if and only if* the value of int variable `x` is 0?

1. `x & x`
2. `x & 0x00000000`
3. `x | 0xffffffff`
4. `x | 0x00000000`
5. None of the above

D. What is the output of the code snippet below (running on a Little-Endian machine)?

```
long long x = -2;
int *y;
y = (int *)&x;
printf("%d %d\n", y[0], y[1]);
```

1. -1 -1
2. -2 -2
3. -1 -2
4. -2 -1
5. Segmentation fault
6. None of the above

E. What is the output of the code snippet below (running on a Little-Endian machine)?

```
float f = -16.0;
char *p;
p = (char *)&f;
printf("%d\n", *p);
```

1. 16
2. -16
3. 0
4. some positive number
5. some negative number

2 Basic C and Assembly (25 points)

A. Given variable declaration `char **p;` what is the type of the expression `*p`?

1. `char**`
2. `char*`
3. `char`
4. `void*`
5. None of the above

B. Given variable declaration `char *p;` what is the type of the expression `&p`?

1. `char**`
2. `char*`
3. `char`
4. `void*`
5. None of the above

C. What is the output of the code snippet below (running on a Little-Endian machine)?

```
void foo(int *p) {
   p++;
   (*p)++;
}
int main() {
   int a[3] = {1, 2, 3};
   int *p;
   p = a;
   foo(p);
   foo(p);
   printf("%d %d %d\n", a[0], a[1], a[2]);
}
```

1. 1 4 3
2. 1 3 3
3. 2 3 3
4. 1 3 4
5. 1 2 3
6. None of the above

D. Suppose register %rdi holds variable x and register %eax holds variable y. Given the following assembly instructions, what could be the *potential* types of x and y respectively?

```
movq (%rdi), %rsi
movl (%rsi), %eax
```

1. `int*, int`
2. `int**, int*`
3. `int**, int`
4. `long long*, long long`
5. `long long**, long long*`
6. `long long**, long long`
7. `char*, char`
8. `char**, char*`
9. `char**, char`

E. Suppose register %rdi holds variable x and register %rsi holds variable y. Which of the following assembly snippet implements y = 8*x and leaves x unchanged?

1. `leaq (,%rdi,8), %rsi`
2. `leaq (%rdi,%rdi,7), %rsi`
3. `movq (,%rdi,8), %rsi`
4. `movq (%rdi,%rdi,7), %rsi`
5. `movq %rdi, %rsi`
   `shlq`[1] `$3, %rsi`
6. `shlq $3, %rdi`
   `movq %rdi, %rsi`

[1]The `shlq` instruction shifts a register (specified in the second operand) to the left by the amount specified in the first operand.

3 C basics (25 points):

When working on Lab1's *avgcsv* program, Ben Bitdiddle has implemented a helper function that splits a C string into an array of fields according to a given delimeter character.

Below is the skeleton of Ben's helper function `split` and his tester program.

```
//split breaks input string s into multiple substrings based on the delimeter character.
//The resulting substrings are stored in the array argument "fields"
//split returns the number of substrings stored.
1: int split(char *s, char delimeter, char **fields)
2: {
3:     char *start_field;
4:     start_field = s;
5:     int n = 0;
6:     while ____________________________________ {
7:         if (((*s) == delimeter) && start_field != NULL) {
8:            fields[n++] = start_field;
9:            start_field = NULL;
10:        } else if ((*s)!= delimeter && start_field == NULL) {
11:           start_field = s;
12:        }
13:        ____________________________
14:    }
15:    if (start_field != NULL) {
16:       fields[n++] = start_field;
17:    }
18:    return n;
19: }

20: int main()
21: {
22:    char **fields;
23:    char test_string[100] = "10;11;12";
24:    int n = split(test_string, ';', fields);
25:    for (int i = 0; i < n; i++) {
26:       printf("%s\n", fields[i]);
27:    }
28: }
```

(a) (5 points) Please complete line 6 and 13 in Ben's implementation of `split`.

(b) (5 points) When Ben runs his program, it outputs "Segmentation fault". What is the reason for the segmentation fault?

1. Line 8 and 16 cause a compilation error because one cannot use `char **` type for array access.
2. `split` has out-of-bound array access on character array `s`.
3. `split` tries to write to array `fields` that has not been allocated space.
4. `split` tries to write to array `s` which only allows read-only access.
5. None of the above.

(c) (5 points) Please fix Ben's program to elimininate the segmentation fault. (Hint: the fix involves very little code changes)

(d) (5 points) After fixing the segmentation fault issue, Ben expects the program output to be:

```
10
11
12
```

However, the program's actual output is

```
10;11;12
11;12
12
```

Please fix Ben's program to produce the expected output (you may directly modify Ben's code in the previous page if you can do so clearly.)

(e) (5 points) Suppose we replace line 23 and 24 of Ben's program to

```
L1: int x = 0x623b61;
L2: int n = split((char *)&x, ';', fields);
```

(Note: We have attached the ASCII table in the appendix)

What is the value of variable `n` after finishing line 24 when running on a little-endian machine?

What is the value of variable `n` after finishing line 24 when running on a big-endian machine?

4 Assembly (25 points):

Examine the following code skeleton and the corresponding assembly. (Question starts on the next page.)

```
L1: unsigned int mystery(__________  array,  int n)
{
    unsigned int result;
    for (int i = 0; i < n; i++) {









    }
    return result;
}

int main()
{
L2: _________  array[3] = {1, 5, 7};
    int ret = mystery(array, 3);
    printf("ret is %d\n", ret);
}
```

The corresponding assembly for the `mystery` function is:

```
mystery:
      movl $0, %edx
      movl $0, %eax
      jmp .L2
.L4:
      movslq %edx, %rcx
      movl (%rdi,%rcx,4), %ecx
      cmpl %ecx, %eax
      ja .L3
      movl %ecx, %eax
.L3:
      addl $1, %edx
.L2:
      cmpl %esi, %edx
      jl .L4
```

In the above assembly, `movslq` copies the 4-byte source operand into the lower order 4-bytes of the 8-byte destination operand. The higher order 4 bytes of the destination operand are filled with 1s if the most-significant-bit of the source operand is 1, and filled with 0s otherwise. Appendix II contains the lecture note information on the jmp instruction.

(a) (5 points) Suppose at the time of entering function `mystery`, register `%rdi` contains the function's first argument `array` and register `%esi` contains the function's second argument `n`. Based on the assembly for `mystery`, please deduce the type information and fill in the blanks at L1 and L2.

(b) (5 points) Where is the local variable `i` in `mystery` stored?

1. register `%edx`
2. register `%ecx`
3. register `%eax`
4. memory at address given by `%rdi+4*%rcx`
5. None of the above.

(c) (10 points) Complete the `mystery` function (Note that when the `mystery` function finishes, register `%eax` should hold the return value of the function.)

(d) (5 points) What is the output of the program?

—END of Quiz I—-

Appendix: ASCII

NAME

The following table contains the 128 ASCII characters encoded in octal, decimal, and hexadecimal

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

Appendix: Family of Jmp instructions

```
jX label
```

- If condition X is met, jump to the label

Example:

```
cmpq %rdi, %rsi
jg .L1
```

The above code jumps to label .L1 if the content of %rsi is greater than that of %rdi when interpreting both as signed numbers.

Below is the lecture notes on various jump instructions:

jX	Condition	Description
je	ZF	Equal/Zero
jne	~ZF	Not Equal/Not Zero
js	SF	Negative
jns	~SF	Nonnegative
jg	~(SF^OF)&~ZF	Greater (Signed)
jge	~(SF^OF)	Greater or Equal (Signed)
jl	(SF^OF)	Less (Signed)
jle	(SF^OF)|ZF	Less or Equal (Signed)
ja	~CF&~ZF	Above (unsigned)
jb	CF	Below (unsigned)