서강대학교
SOGANG UNIVERSITY

College of Engineering

Department of
Computer Science and Engineering

Introduction to Computer Systems

Exam, Spring 2022

Seoul, April 26, 2022

Student Number:	☐☐☐☐☐☐☐☐
First Name:	
Last Name:	
Date of Birth:	
Program of Study:	

Task	1.1 & 1.2	1.3	1.4	2.1	2.2	2.3	3	$\sum$
Maximal Score	11	8	7	6	6	6	13	57
Attained Score								
Correction								

Remarks:

- The duration of the exam is **75 minutes**.
- Please write down your name and student id at the top of **every page**.

Good Luck!

Task 1: Representing and Manipulating Information

1.1 General Understanding

a) How do computer systems store, recognize, and interpret information? 2 Points

The computer system stores data in bits. By understanding the context of the bits, the computer system recognizes its data type and interpret it by using data representation.

b) Explain four steps of the compilation system for C codes. 2 Points

Preprocessor: merge multiple files into one file
Compiler: C $\rightarrow$ Assembly
Assembler: Assembly $\rightarrow$ binary code
Linker: linking multiple binary codes

1.2 Byte Ordering

In C code, we assign "int" variable v with 18259.

a) Compute the hexadecimal representation of the value of variable v. 2 Points

$18259 = 0x4753$

b) If the address of variable v is 100, write down byte information at 100, 101, 102, and 103 when using big endian and little endian. 2 Points

Little endian
100 :0x53
101 :0x47
102 :0x00
103 :0x00
Big endian
103 :0x53
102 :0x47
101 :0x00
100 :0x00

c) We run the following code with the memory layout addressed in b). Write down the screen result for both byte orderings, little endian and big endian, while referring to the ascii table attached in the next page. 3 Points

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

www.alpharithms.com

Figure 1: Ascii table

```
char *p;
p = 100;
for ( ; p < 104; p++ )
    if ( *p > 0 )
        printf("%c", *p);
    else printf("*");
```

little endian: SG** big endian: **GS

1.3 Integer Arithmetics

let x be a two-byte binary representation of signed integer as $1111110101110110_{(2)}$.

a) Compute the decimal value of x. 2 Points

$$1111110101110110_{(2)} = -(0000001010001010_{(2)}) - (2 + 8 + 128 + 512) = -650 \quad (1)$$

b) There is function $byteint(x)$ that keeps only lower 8 bits in signed integer representation. Compute a decimal value of $byteint(x >> 4) >> 2$. 2 Points

$1111110101110110 >> 4 = 1111111111010111$
$byteint(1111110101110110 >> 4) = 11010111$
$byteint(1111110101110110 >> 4) >> 2 = 11110101 = -11$

c) Compute a decimal value of $byteint(x >> 2) >> 4$. 2 Points

$1111110101110110 >> 2 = 1111111101011101$
$byteint(1111110101110110 >> 2) = 01011101$
$byteint(1111110101110110 >> 2) >> 4 = 00000101 = 5$

d) Comparing results of b) and c) with $x >> 6$, which operation do people prefer to the other and why? 2 Points

People prefer (b) to (c) because (b) still keeps the meaning.

1.4 Floating Arithmetics

We have 8-bit floating representation $float8$ that has 1 sign bit, 3 exp bits, and 4 frac bits. The bias of this representation is $2^{3-1} - 1 = 3$. We have two $float8$ variables x and y that have 2.625 and 24.

a) Compute binary codes of x and y. 2 Points

y cannot be represented by 8-bit floating representation. However, if we treat it as a normalized value,

$$x = 2.625 = 10.101 = 1.0101 * 2^1 \quad (2)$$

$$y = 24 = 11000 = 1.1 * 2^4 \quad (3)$$

$$\text{binary representation}_x = 01000101 \quad (4)$$

$$\text{binary representation}_y = 01111000 \quad (5)$$

(6)

b) compute $(x + y) - y$ and $x + (y - y)$ and explain a reason that they are different. 5 Points

Since y is NAN, all computation returns NAN. But, if we consider y as a normalized value,
$(x + y) = (01000101) + (01111000)$
$(x + y)_{frac} = 0.0010(1) + 1.1000(0) = 1.1010(1) = 1.1011$
$((x + y) - y)_{frac} = 1.1011 - 1.1000 = 0.0011$
$((x + y) - y) = 01001000$
$x + (y - y) = x = 01000101$
During alignment, tail of x are deleted.

Task 2: Machine-Level Programming

2.1 XOR Swap

Lee creates JH64 architecture that supports following instructions.

movq src, dst: move src to dst.
xorq src, dst: compute xor operation of src and dst and store the result in dst. Both operands should be registers.

```
void swap_xor ( long *x, long *y ){
    *x ^= *y;
    *y ^= *x;
    *x ^= *y;
}

void swap_tmp ( long *x, long *y ){
    long tmp = *x;
    *x = *y;
    *y = tmp;
}
```

a) Let us compile both function swap_xor() and swap_tmp(). Pointer variable x and y are stored in $\%rdi$ and $\%rsi$. 4 Points

```
# swap_xor
movq ($rdi), %rax # %rax = *x
movq ($rsi), %rbx # %rbx = *y
xorq %rbx, %rax   # %rax = *x ^ *y
movq %rax, ($rdi) # *x = %rax
movq ($rdi), %rax # %rax = *x
xorq %rax, %rbx   # %rbx = *x ^ *y
movq %rbx, (%rsi) # *y = %rbx
movq (%rsi), %rbx # %rbx = *y
xorq %rbx, %rax   # %rax = *x ^ *y
movq %rax, ($rdi) # *x = %rax

# swap_tmp
movq ($rdi), %rax
movq ($rsi), %rbx
movq $rax, (%rsi)
movq $rbx, (%rdi)
```

b) Which swap function would be preferred and why? 2 Points

Function swap_tmp takes less instructions so it would be preferred by people.

2.2 Procedure

The compiler translates the following C function called ”factxy” into the byte code. The next table shows corresponding assembly instructions with byte address. Please answer the following questions.

```
long factxy ( long x, long y ){

  if ( x == 0 )
    return 1;

  return y * factxy ( x-1, y+1 );

}
```

Byte address	Assembly instruction
0x114d	factxy: testq %rdi, %rdi
0x1150	(a) .L8
0x1152	movq $1, %rax
0x1157	ret
0x1158	.L8: (b)
0x1159	movq %rsi, %rbx
0x115c	subq $1, %rdi
0x1160	addq $1, %rsi
0x1164	callq factxy
0x1169	imulq %rbx, %rax
0x116d	(c)
0x116e	ret

1. What branch instruction should we use at (a)? 2 Points

```
jne .L8
```

2. The current code shown above works but returns incorrect results. Explain a reason and fix the problem by filling (b) and (c). 2 Points

```
(b):    pushq %rbx
(c):    popq %rbx
```

3. If we call function $factxy(3, 3)$ in the main procedure, draw a status of the stack frame when the processor reaches 0x1157 and before performing the instruction. Assume that %esp is 10000 (decimal for the sake of convenience) and %rbx is 100 right after we call procedure ”factxy” at first. 2 Points

Byte address	Value
10000	a return address to the main procedure
9992	
9984	
9976	
9968	
9960	
9952	
9944	
9936	
9928	

Byte address	Value
10000	a return address to the main procedure // x = 3
9992	100
9984	0x1169 // x = 2
9976	3
9968	0x1169 // x = 1
9960	4
9952	0x1169 // x = 0
9944	
9936	
9928	

2.3 Transpose 2D array

We want to implement a function that transposes a 2D (nested) source array and stores it on a target 2D (nested) array. To this end, function $transpose()$ takes four arguments: long pointers dst and src that point to the first element of the source array and the target array (e.g. &a[0][0] and &b[0][0]), and the number of rows and columns rows and cols as shown in the code. For example, transpose (&a[0][0], &b[0][0], rows, cols)

```
void transpose ( long *src, long *dst, long rows, long cols){
        for(long i=0;i<rows;i++){
                for(long j=0;j<cols;j++){
                        dst[ (1) ] = src [ (2) ];
                }
        }
}
```

1. Fill (1) and (2) to store the transposed source array to the target array. 2 Points

```
void transpose ( long *src, long *dst, long rows, long cols){
        for(long i=0;i<rows;i++){
                for(long j=0;j<cols;j++){
                        dst[ j * rows + i ] = src [ i * cols +
                           j ];
                }
        }
}
```

2. Fill (a), (b), (c), and (d) in the assembly code with correct branch instructions or labels. 4 Points

```
transpose:
        movl $0, $r10d
        jmp (a)
.L3:
        assembly code of dst [ (1) ] = src [ (2) ]
        addq $1, %rax
.L4:
        cmpq %rcx, %rax
        (b)
        addq $1, %r10
.L2:
        cmpq %rdx, %r10
        (c)
        movl $0, %eax
        jmp (d)
.L6:
        ret
```

```
transpose:
        movl $0, $r10d
        jmp .L2
```

```
.L3:
        movq %r10, %r9
        imulq %rcx, %r9
        addq %rax, %r9
        movq %rax, %r8
        imulq %rdx, %r8
        addq %r10, %r8
        movq (%rsi, %r9, 8), %r9
        movq %r9, (%rdi, %r8, 8)
        addq $1, %rax
.L4:
        cmpq %rcx, %rax
        jl .L3
        addq $1, %r10
.L2:
        cmpq %rdx, %r10
        jge .L6
        movl $0, %eax
        jmp .L4
.L6:
        ret
```

Task 3: 64-bit Adder

We implement 64-bit integer addition and subtraction in the programming assignment. To this end, we define structure data type ”int64” inlcuding two unsigned integer 32-bit variables to store 64-bit information and implement addtion and two’s complement.

1. Complete the structure type ”int64”. 2 Points

```
struct int64 {

    XXX;

};
typedef struct int64 int64;
```

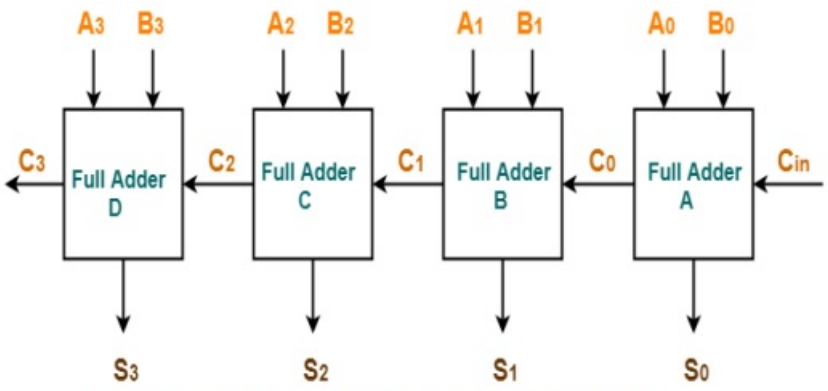
```
struct int64 {

    unsigned int hp, lp;

};
typedef struct int64 int64;
```

2. There is a function ”getBits” that returns a bit at an index ind of a ”int64” variable a. For example, $getBit(2, 1)$ returns 1 and $getBit(2, 0)$ returns 0. The index ind is counted from the right most bit and the function returns 0 or 1. Complete the function getBit. 2 Points

```
char getBit(int64 a, int ind) {

    if (XXX1)
        return XXX2;
    else return XXX3;

}
```

```
char getBit(int64 a, int ind) {

    if (ind < 32)
        return (a.lp & (1 << ind)) >> ind;
    else return (a.hp & (1 << (ind - 32))) >> (ind - 32);

}
```

3. In order to compute 64-bit addition, we mimic how a 64-bit adder compute the reuslt. The 64-bit adder comprises of 64 1-bit full adders and concatenates them by passing the carry bit of each adder to the next adder. In the program, we set that A_i, B_i, and C_{i-1} as a_bit_i, b_bit_i, and $carry\_bit\_in$ during computing a 1-bit adder at index i. Find the bitwise expression to calculate the ’carry_bit_out’ from variables above. 2 Points



In this picture, C3 will be 'carry_bit_out'.

```
carry_bit_out = XXX;
```

```
carry_bit_out = (carry_bit_in & (a_bit_i ^ b_bit_i)) | (a_bit_i
    & b_bit_i);
```

4. Write how to get the sign bit of the ”int64” variable using the above getBit function. And write a conditional statement for overflow by using variables sign_a and sign_b. 2 Points

```
char sign_c = XXX1;

if (XXX2)
    g_overflow = 1;
```

```
char sign_c = getBit(result, 63);

if ((sign_a & sign_b & (!sign_c)) || ((!sign_a) & (!sign_b) &
   sign_c))
    g_overflow = 1;
```

5. Complete the code to compute the 2's complement. 3 Points

```
int64 complement64(int64 x) {

    int64 result;
    result.hp = 0;
    result.lp = 0;

    result.hp = XXX1;
    result.lp = XXX2;

    int64 one;
    one.hp = XXX3;
    one.lp = XXX4;
    result = add64(result, one);

    return result;
}
```

```
int64 complement64(int64 x) {

    int64 result;
    result.hp = 0;
    result.lp = 0;

    result.hp = ~x.hp;
    result.lp = ~x.lp;

    int64 one;
    one.hp = 0;
    one.lp = 1;
    result = add64(result, one);

    return result;
}
```

6. Run the program and write down the the result of addition with following input: 2 Points

A : FFFF FFFF FFFF FF00
B : 8000 0000 0000 0010

```
    Overflow (or Negative Overflow)
```