

Computer Organization

Spring 2005

Test #1 – Feb 18

Name _____ **Answer Key**

There are 8 pages - make sure you have all of them. The last page includes an ASCII table.
Answer all questions - pay attention to the # of points for each question.
Don't leave anything blank - partial credit is always possible!

Question 1 (20 pts): Complete the following tables by filling in each blank space with the appropriate value.

Decimal	8 bit 2's complement binary	Hex
1	00000001	01
-22	11101010	EA
-103	10011001	99
122	01111010	7A

Decimal	8 bit unsigned binary	Hex
155	10011011	9B
217	11011001	D9
243	11110011	F3

Decimal	IEEE Single Precision (32 bit)		
	Sign (1)	Exponent (8)	Significand (23)
-16.75	1	1000011	00001100000000000000000

Question 2 (25 pts): Short answer questions, 5 points each.

2a (5 pts): A machine uses a 16-bit two's complement representation for integers.

What is the largest integer that can be represented?	<u>32767</u>
What is the smallest integer that can be represented?	<u>-32768</u>
How many different integers are representable?	<u>65536</u>

2b (5pts): Briefly discuss why an add instruction that is part of a stack-based instruction set architecture would be smaller (require fewer bits to encode) than an add instruction in a general-purpose instruction set architecture.

In a stack-based instruction set arch, the add instruction does not include any explicit operands – so there are no bits in the machine code that represent which operands to add.

In a GP register instruction set arch. all register operands must be specified in each instruction. If there are 16 possible registers, an add instructions would need to include 4 bits to specify each registers.

2c (5pts): A machine uses a 16-bit two's complement representation for integers, and little-endian byte-ordering, this means that the least significant byte of an integer is stored at the lower address. Show what the following program will print out:

```
int x;    /* 16 bit signed integer! */
char *p = (char *) & x;
x = 0x0013;
printf("x is %d\n",x);
printf("%d %d\n",p[0],p[1]);
```

x is 19

19 0

2d (5pts): The C code below shows an addition operation involving two 8 bit unsigned integers. Write the missing code that determines whether the result of the addition was correct or not.

```
unsigned char x,y,z;

/* x and y are assigned some values */
z = x + y;
if ( ... you need to write this ...)
    printf("z is Wrong!\n");
```

For unsigned addition, a carry out indicates overflow, but we don't have the carry out bit to check! If the result is smaller than either operand, then the addition was wrong:

```
if ((z < x) || (z < y)) {
```

2e (5pts): What is the output of the following program?

```
void showstring(char *s) {
    while (*s) {
        printf("%d ",*s);
        s++;
    }
    printf("\n");
}
int main() {
    showstring("CompOrg!");
}
```

67 111 109 112 79 114 103 33

Output is: _____

Question 3 (10 pts): This question has two parts, both involve the following C statement:

$$y = a*x*x + b*x + c;$$

Part a (5 pts): Show how this computation might look when using a processor based on a *stack-based instruction set architecture* (generic assembly is fine, with instructions like "push y", "add" and "pop tmp").

```
push a          #
push x
mult            # TOS is a * x
push b
add             # TOS is a * x + b
push x
mult            # TOS is (a * x + b)*x
push c
add             # TOS is (a * x + b)*x + c
pop y
```

Part b (5 pts): Show how the computation might look when using a processor based on an *accumulator based instruction set*. Generic assembly is fine here also, instructions like "add y", "load z" and "store x" are expected.

```
load a          # accum = a
mult x          # accum = a*x
add b           # accum = a*x +b
mult x          # accum = (a*x+b)*x
add c           # accum = (a*x+b)*x+c
store y
```

Question 4 (20 pts): Write a C function named `parity()` with the following prototype:

```
unsigned char parity(unsigned int x);
```

The function should return an unsigned char (8 bit unsigned integer) with the value 0 or 1 as described below. Your function can assume that an unsigned int is represented with 32 bits.

The parity function determines whether the number of bits set to 1 in the representation of x is even or odd. The value of *parity* (x) is 0 if the number of bits set to 1 in the representation of x is even. The return value should be 1 if the number of bits set to 1 in the representation of x is odd. For example, `parity(0)` is 0, `parity(2)` is 1 and `parity(7)` should return 1.

```
unsigned char parity(unsigned int x) {
    int i;
    int cnt=0;

    for (i=0;i<sizeof(x);i++) {
        if ((0x01 << i) & x) {
            cnt = 1-cnt;
        }
    }
    return(cnt);
}
```

Question 5 (15 pts): We want to create a (small) instruction set for a computer that has 4 general purpose registers named R1, R2, R3 and R4, and one dedicated register named RESULT. The individual instructions that the computer needs are as follows:

- increment a register (add 1 to the contents of the register, stores the new value in register RESULT).
- add any two general purpose registers, put the sum in register RESULT.
- multiply any two general purpose registers, put the product in register RESULT.
- copy the value in register RESULT to any one of the 4 general purpose registers.

5a (7 pts): Develop a 6-bit binary *machine code* that encodes these instructions (each instruction must be 6 bits). Provide the details of your encoding (describe your machine code here):

Register encoding: R1=00, R2=01, R3=10, R4=11

Instructions format three 2-bit fields: OP SRC1 SRC2

OP:

00: increment SRC1

01: add SRC1 and SRC2

10: Multiply SRC1 and SRC2

11: COPY from RESULT into SRC1

SRC1, SRC2 are 2 bit register codes.

5b (8 pts): Using your machine code from part a, write a machine code program that computes the following expression and leaves the answer in RESULT. You must assume that initially the value of x is in R1, y is in R2, R3 and R4 are both 0.

$$x(x+2) + 2y$$

Comment your machine code program!

```
# Initially: R1 = x, R2 = y, R3 = 0, R4 = 0

000000  INC R1      # RESULT = X + 1
111000  COPY R3    # R3 = X+1
001000  INC R3     # RESULT = X+2
111000  COPY R3    # R3 = X+2

100010  MULT R1,R3 # RESULT = X * (X+2)
111000  COPY R3    # R3 = X * (X+2)

010111  ADD R2,R2  # RESULT = 2*Y
111100  COPY R4    # R4 = 2 * Y
```

Question 6 (10 pts): The code below includes a C function named `uppercase` that is given a pointer to a C string. `uppercase` is supposed to force all the alphabetic characters in a C string to upper case and return a pointer to the string. This function does not work (there are lots of problems!) Identify the problems (state what the problem are), and then fix it (rewrite the code so that it works). Leave `main()` alone, it's fine the way it is.

An ASCII table is included on the next page (in case you find it helpful).

```
char *uppercase(char *s) {

    while(*s!='0') {
        if ((*s>='a') || (*s<='z'))
            *s = *s-'A'+'Z';
        *s++;
    }
    return(s);
}

/* Leave main alone! */
int main() {
    char s[10] = "Hello";
    printf("String is %s\n",uppercase(s));
    return(0);
}
```

Problems:

comparison `*s!='0'` should be `*s!=0` (or `*s!='\0'`)
if should be `((*s>='a') && (*s<='z'))`
arithmetic should be `*s = *s -'a' + 'A';`
returns a pointer to the end of the string, needs to
return a pointer to the beginning of the string.

```
char * uppercase(char *s) {
    char *p = s;

    while(*s) {
        if ((*s>='a') & (*s<='z'))
            *s = *s-'a'+'A';
        s++;
    }
    return(p);
}
```

ASCII Table (decimal)

0	NUL	32	SP	64	@	96	~
1	SOH	33	!	65	A	97	a
2	STX	34	"	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	\$	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	'	71	G	103	g
8	BS	40	(72	H	104	h
9	HT	41)	73	I	105	i
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	,	76	L	108	l
13	CR	45	-	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o
16	DLE	48	0	80	P	112	p
17	DC1	49	1	81	Q	113	q
18	DC2	50	2	82	R	114	r
19	DC3	51	3	83	S	115	s
20	DC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	v
23	ETB	55	7	87	W	119	w
24	CAN	56	8	88	X	120	x
25	EM	57	9	89	Y	121	y
26	SUB	58	:	90	Z	122	z
27	ESC	59	;	91	[123	{
28	FS	60	<	92	\	124	
29	GS	61	=	93]	125	}
30	RS	62	>	94	^	126	~
31	US	63	?	95	_	127	DEL

Question	Possible	Score
1	20	
2	25	
3	10	
4	20	
5	15	
6	10	
Total	100	

Extra Credit (0 pts): How many bits does it take to represent that number of ways in which emacs is better than vi?