CS 211: Midterm 1: 100 points

Instructor: Prof. Santosh Nagarakatte

Full Name Here:
RUID:

Question	Max Points	Points
1	20	
2	25	
3	15	
4	15	
5	25	

Problem 1: (20 points)

Answer "True" or "False" to these questions. If the answer is "False", you need to provide a reason to state why the answer is "False" (you will not get points otherwise).

1. Von-Neumman architecture is a stored-program computer. x86 is a Von-Neumann machine. True or False?

   Answer: True

2. Hardware understands the simplified C program generated by the compiler and executes it. True or False?

   Answer: False. Hardware only understands machine codes *i.e.*, 0's and 1's.

3. A integer array (*e.g.*, int a[100] in C). The start of the array is at address 0x100. Then the end of the array is at address 0x140. True or False?

   Answer: False. 0x100 + 400 = 256 + 400 = 256 + 400 = 0x290

4. The endianness of machine (big endian or little endian) needs to be considered when you want to store a character (char in C) in memory. True or False?

   Answer: False. Endiannes is important only when the datatype has multiple bytes.

5. ISA exposes the caches to the programmer. True or False?

   Answer: False. Cache is micro-architecture. It is hidden from the programmer.

6. Program counter (PC) stores the instruction that the hardware is going to execute next. Automatic increment that occurs to the PC after the instruction has completed execution is the only way to change it. True of False?

   Answer: False. Jmp is another way to change the PC.

7. Assembly programmer's view of memory is equivalent to a linear array of bytes in C. True or False?

   Answer: True.

8. 0x1111 is the minimum number that you can present in 2's complement representation of an integer using 4-bit signed presentation. True or False?

   Answer: False. 0x1000 is the minimum value.

9. *%ah* register is the higher 8-bits of *%eax* register. True or False?

   Answer: False. *%ah* is the higher 8-bits of *%ax* register.

10. Primary purpose of scaled index addressing mode is to represent memory accesses with pointers. True or False?

    Answer: False. scaled index addressed mode is for accessing structures and arrays.

Problem 2: C Programming (25 points)

1. (5 points) Consider the code fragment in C. Consider that array begins at address 0x1000.

```
int array[100];
char array2[100];

int* ptr = array;
char* ptr2 = array2;

ptr = ptr + 20;
ptr2 = ptr2 + 20;
```

   What will be addresses pointed to by *ptr* and *ptr*2 in hex? Explain the reason for your answer.

   Answer: ptr = 0x1050. The pointer ptr is a integer pointer. ptr + 20 will access the 20th element of the array which is at location 0x1000 + 20 * 4 = 0x1050.

   ptr2 = 0x1014. The pointer ptr2 is a char pointer. ptr + 20 will access the 20th element of array2 which is at location 0x1000 + 20 = 0x1014.

2. (5 points) What is wrong with the following code?

```
char *strcpy(char *s) {

   char *d;
   int i, len;
   len = strlen(s);
   for (i=0; i<len; i++) {
       d[i] = s[i];
   }
   d[i] = \0;
   return d;
}
```

   Answer: The programmer forgot to allocate memory with memory allocation routines (*e.g.* malloc). The program will likely experience segmentation fault.

3. (15 points) You are given a singly linked list which is terminated by NULL. Complete the following C function to delete a single node from the linked list which matches the key. Fill in the blanks (....).

```
 /* structure for the linked list */
struct node{
   int value;
   struct node* next;
};

/* start pointer points to the begining of the list. prev points to
the previous node that is being examined */

struct node* delete_node(struct node* start, int key){

 struct node* curr = start;
 struct node* prev = start;

 Answer: curr
 while ( .....  != NULL){

    /* check if head is the element to be deleted */

    Answer: key
    if(curr->value == .....){

        /* check if curr points to the beginning of the list */

        if(curr == prev){

           /* update the start pointer, free the node, and return */
           start = curr->next;
           free(curr);
        }
        else{
           Answer: curr->next;
           prev->next = ............;

           Answer: curr
           free ( ..........  );

        }
        return start;
    }
    prev = curr;
    /* check the next node */

    Answer: curr->next
    curr = ............;
```

```
 } /* While loop ends here */

 return start;
} /* function ends here */
```

Problem 3: Data Representation 1 (15 points)

1. (4 points) Convert decimal number 69 into 8-bit binary representation and hex representation.

   Answer:
   Binary:0100 0101
   hex : 0x0045

2. (4 points) Give two reasons for using IEEE-754 representation for floating point numbers instead of fixed point representation.

   Answer:
   (1) Represent very small numbers. If you use a fixed point representation, the smallest value is obtained when all the bits are devoted to numbers after the decimal point.
   (2) Represent very large numbers. With the exponent, IEEE-754 representation can represent both positive and negative numbers.

3. (7 points) Represent -2.325 in IEEE-754 32-bit floating representation. Clearly indicate the sign, exponent and mantissa bits.

   - sign:
   - exponent:
   - mantissa:
   - show work below to arrive at the above answer:

     Answer:

     0.325 * 2 = 0.65 * 2 = 1.3 -1 = 0.3 * 2 = 0.6 * 2 = 1.2 -1 = 0.2 * 2 = 0.4 * 2 = 0.8 * 2 = 1.6 -1 = 0.6 * 2 = 1.2 -1 = 0.2 ....

     10.0101 (0011) repeats

     1.00101 (0011) repeats 4 times 00 x 2^1.

     Exponent = 1+ 127 = 128 Sign = 1 Mantissa = 00101 0011 0011 0011 0011 00 for a total of 23 bits

Problem 3: Data Representation 2 (15 points)

1. (8 points) Consider you are designing **6-bit signed integer** representation. What are the maximum and minimum values you can represent?

   - Max value in two's complement representation: $2^5 - 1$.
   - Min value in two's complement representation: -2^5
   - Max value in one's complement representation: $2^5 - 1$
   - Min value in one's complement representation: -$2^5 - 1$

2. (5 points)You obtain the representation for a negative number in two's complement method by taking one's complement of a number and then adding one. Why does it work? What is the key idea behind two's complement representation?

   Answer: The key idea is to treat the sign bit as a value with a negative sign. In a n-bit representation, the value of the sign bit is -2^{n-1}.

   When you have a positive number x. By taking two's complement you want to get $-x$.

   In a positive number x, the sign bit is 0. When you take the one's complement of the number, you get:

   $-2^{n-1} + 2^{n-1} - 1 - x$

   On simplification you get $-x - 1$. When you add $+1$ to the number (as with two's complement: take one's complement and add one), you arrive at $-x$.

   This provides a unique representation for zero. Further all the number calculations are according to powers of two unlike one's representation for negative numbers.

3. (2 points) The binary number 1011 is (-5) in 4-bit signed two's complement representation. Sign extend it to 8-bit representation.

   Answer: 1111 1011

Problem 4: Assembly Programming (25 points)

1. (10 points) Consider you have a structure with two fields:

```
struct node{
 int value;
 struct node * next;
};
...
ptr = (struct node *)malloc(...);
ptr->value = 40;
ptr->next = NULL;
```

   The variable *ptr* is a pointer to a node and is resident in register *%eax*. What do the following operations do?

   - *mov* $(\%eax), \%ebx$
     - Explain the operation with respect to C code above:
       Answer: It is moving the $ptr->value$ into *%ebx*. Hence *%ebx* will be 40 after the operation.
     - What will be value of *%ebx*: 40.
   - *mov* $4(\%eax), \%eax$.
     - Explain the operation with respect to C code above:
       Answer: It is moving the $ptr->next$ to *%eax*. hence *%eax* will be NULL (0).
     - What will be the value of *%eax* after the operation: 0

2. (5 points) Are the following operations legal? If no, why?

   (a) *mov* $(\%eax), \%ecx$:
   Answer: Yes

   (b) *mov* $0x8(\%eax), (\%ebp)$:
   Answer: No. Both source and destination operands cannot be memory.

   (c) *mov* $0x80(\%eax, \%ebx, \%ecx), \%edx$:
   Answer: No. Third operand in scaled addressing mode cannot be a register.

3. (6 points) What is the effective address for each of the following:

```
%eax: 15
%edx: 0x40
%ebp: 0x6000
```

   (a) $0x4(\%ebp)$: 0x6004

   (b) $(\%ebp, \%edx)$: 0x6040

   (c) $(\%ebp, \%edx, 4)$: 0x6100

4. (4 points) When you generate assembly code, you often see the following code.

```
xorl %eax, %eax
```

Why does the compiler generate this code? What is another alternative way of performing the same operation?

Answer: zeros the *%eax* register. Another way of doing it is *mov* $0*x*0 *%eax*