Name: _____ Student ID:_____

CS 354 Exam 1

Version A/B

Instructions:

1. Do not open this exam until you are instructed to by the examiners.
2. Fill in your name and UW Student ID on all the pages of this exam.
3. You have 1 hour and 30 minutes to answer all the questions.
4. An x86 cheat sheet has been provided for your use during the exam on the back of the final page.
5. Please check that this exam contains 12 printed pages.
6. All the x86 instructions in this exam follow the AT&T syntax discussed in the class lectures. Your answers must also follow the AT&T syntax.
7. Last page of this exam are intentionally empty and you can use it for your rough work and other calculations.

	Worth	Points Earned
Question 1		_____
Question 2		_____
Question 3		_____
Question 4		_____
Question 5		_____
Question 6		_____
Question 7		_____
Total		_____

TODO:
Fill in above table
Make 2 versions with different question ordering / numbers.

Name: _____ Student ID:_____

Question 1: C programming

Jason writes a simple C program to "simulate" driving a car. Refer to the below code for this question.

```
#include <stdio.h>

typedef struct Car {
  char *make;
  char *model;
  int year;
  unsigned int miles;
} car_t;


/** "Drives" a car some number more miles
 * @param car: The car to drive. A mutable. After this function,
 *             the mileage of this car should be higher.
 * @param miles: The number of miles to drive the car
 * @return: Return the cars mileage after driving the car.
 */
int drive_car(car_t car, unsigned int miles) {
  int new_miles = car.miles + miles;
  car.miles = new_miles;
  return new_miles;
}

int main(int argc, char *argv[]) {
  car_t jasons;
  jasons.make = "Toyota";
  jasons.model = "Camry";
  jasons.year = 1996;
  jasons.miles = 181020;

  int new_mileage = drive_car(jasons, 80); // Drive to Milwaukee

  if (new_mileage != jasons.miles) {
    printf("WHY ARE THESE NOT EQUAL???\n");
  } else {
    printf("That makes more sense!\n");
  }

  return 0;
}
```

Name: _____ Student ID:_____

(A) Refer to the code on the previous page. Jason has made a mistake :(. When he runs the program, he gets a very confusing result. The program outputs:
"WHY ARE THESE NOT EQUAL???".
In a single sentence, why does Jason see this surprising result?

(B) Re-write `drive_car` (including the parameter list) so that Jason isn't so confused and the above program prints "That makes more sense!" Hint: You can do this with exactly the same number of lines.

```
/** "Drives" a car some number more miles
* @param car: The car to drive. A mutable.After this function,
*             the mileage of this car should be higher
* @param miles: The number of miles to drive the car
* @return: Return the cars mileage after driving the car.
*/
int drive_car(__________ car, unsigned int miles) {




}
```

(C) What other line needs to change for the above program to compile correctly after your changes to `drive_car`?

Name: _____ Student ID:_____

Question 2: Pointers and arrays

```
int i_array[10] = {10,11,12,13,14,15,16,17,18,19};
int diff_array[9] = {0,0,0,0,0,0,0,0,0};
char *str = “Hello, all”;
int *i_ptr1, *i_ptr2;
char *cp;
```

Assume: the integer array `i_array` begins at `0x5000` and the string `str` begins at `0x6000`
Note: `sizeof(int)` is 4 and `sizeof(char)` is 1.

(A) What does the following code output? Note: %p formats the argument as a pointer in *hex*.

```
printf(“%p\n”, &i_array[5]);
```

(B) What is the return value of the following code? In layman’s terms, what is this code doing?

```
cp = str;
int i = 0;
while (*cp != ‘\0’) {
    cp++;
    i++;
}
return i;
```

(C) Complete the code below. After the code runs, `diff_array` should be populated with the differences of adjacent values in `i_array`. I.e., after this code runs `diff_array` should be `{1,1,1,1,1,1,1,1,1}`. Hint: You may not need to use all of the blank space below, and you do ***not*** need more blank space.

```
i_ptr1 = i_array;
i_ptr2 = &i_array[1];
_______________;
_______________;
while (i_ptr2 < _____________) {
    int diff = ___________________;
    diff_array[__________] = diff;
    ____________________________________;
    ____________________________________;
    ____________________________________;
    ____________________________________;
}
```

Name: _____ Student ID:_____

Question 3: Types and Data Representation

Fill in the following table. All numbers are 8-bit two's complement integers.

0 1 0 1 1 0 1 0	(B1)
+ 1 1 1 1 0 0 0 0	(B2)
(A)	(B3)

(A) Perform 2's complement binary addition on the following two 8-bit integers.
(B) Convert each two's complement binary value into decimal (right column)
(C) Was there overflow in the two's complement addition? How can you tell this?

(D) Strike out the invalid basic C types from the following list: (the first one is done for you)

~~a. bigint~~	b. char
c. long double	d. nibble
e. long float	f. long
g. binary	h. short int

Name: _____ Student ID:_____

Question 4: C operators and pointers

(A) Fill in the empty spaces by evaluating pointer expressions assuming the memory contents provided on the right side.
Assume:
1) integers are 4 bytes in size
2) the base address of array "arr" is 0x100(hexadecimal)

```
int a = 10;
int *p = &a;
int arr[3] = {2,3,4,5};
int *q = arr;
```

Expression	Value (Show addresses in hexadecimal format)
*p	
q[0]	
&arr[2]	
q + 3	
*(q+1)	

(B) The following "for loop" statement calculates the sum of **even** integers from 1 to 10. Fill in the empty dashes **without using the sum variable**.

```
int sum = 0;
for(int i=1;_____________ ;i++){
       if(i%2==1){

          _____________________________ ;
      }
      sum += i;
}
printf("The sum is %d\n", sum);
```

Name: _____ Student ID:_____

Question 5: Converting C to x86 assembly

Given the following C code (left) and a set of x86 assembly instructions, put the assembly instructions in the correct order so they execute the same operations as the C code. You may not need all of the assembly instructions.

```
// This is an array that has data. The specific data is unimportant.
// This array begins at address 0x5000
int data[...] = {...}
```

```
int result;
int *xp = &data[0];
int *yp = &data[1];
if (*xp > *yp) {
  result = *xp - *yp;
} else {
  result = *yp - *xp;
}
// result is held in register %eax.
```

```
(a) movl (%ebx), %eax
(b) movl 4(%ebx, %esi, 4), %ecx
(c) movl $0x5000, %ebx
(d) movl 4(%ebx), %ecx
(e) movl 0x5000, %ebx
(f) movl %ecx, %eax
(g) subl %ecx, %eax
(h) subl %eax, %ecx
(i) jle .else
(j) jmp .endif
(k) .endif:
(l) .else:
(m) cmpl %ecx, %eax
```

Name: _____ Student ID:_____

Question 6: Converting x86 assembly into C

(A) Fill in the empty dashes on the right side column so that it becomes a C style equivalent of the following x86 assembly instructions.
Assume:

- these instructions are executed one after the other, sequentially
- x is stored in memory at the address %ebp +4
- Register %ecx contains y.
- Register %edx contains z.

`movl 4(%ebp), %eax`	Load x into register _____.
`leal 10(%ecx,%edx,4), %ebx`	t1 (%ebx) = (4 ___ z ____ y ____ 10)
`addl %ebx, %edx`	z = z + _____.
`sall $3, %eax`	_____ = x _____ 3
`imull  %eax, %edx`	z = _____ * _____
`movl %edx, 12(%ebp)`	Store _____ into address _____

Name: _____ Student ID:_____

Question 7: More C Programming and Linked Lists

The following code follows a similar structure as the linked list from assignment/project 1. It contains two functions that operate on linked lists.

```
#include <stdio.h>
#include <stdlib.h>

struct node{
        int data;
        struct node* next;
};

void usefulFunc1(struct node* curr){
    while(curr != NULL){
        printf("%d --> ", curr->data);
        curr = curr->next;
    }
    printf("\n");
}

void usefulFunc2(struct node** head, int val){
    struct node* prev = NULL;
    struct node* curr;
    if(head == NULL) return;
    curr = *head;
    while(curr != NULL){
        if(curr->data == val){
            if(prev != NULL){
                prev->next = curr->next;
                free(curr);
                curr = prev;
            }else{
                *head = curr->next;
                free(curr);
                curr  = *head;
                continue;
            }
        }
        prev = curr;
       curr = curr->next;
    }
}
```

Name: _____ Student ID:_____

(A) Describe what `usefulFunc1` does in one sentence. (Hint: If you need more than one sentence re-think your answer.)

(B) Describe what `usefulFunc2` does in one or two sentences. (Hint: If you need more than one or two sentences re-think your answer.)

Name: _____ Student ID:_____

Blank Space for extra work

Name: _____ Student ID:_____

x86 Cheat Sheet

general purpose registers

```
%eax     (%ax,%ah,%al)
%ecx     (%cx,%ch,%cl)
%edx     (%dx,%dh,%dl)
%ebx     (%bx,%bh,%bl)
%esi
%edi
%ebp [base pointer]
%esp [stack pointer]
```

program counter

```
%eip
[instruction pointer]
```

condition codes (CCs)

```
cf (carry flag)
zf (zero flag)
sf (sign flag)
of (overflowing flag)
```

jump

```
j dst        always jump
je dst       jump if equal/zero
jne dst      … not eq/not zero
js dst       … negative
jns dst      … non-negative
jg dst       … greater (signed)
jge dst      … >= (signed)
jl dst       … less (signed)
jle dst      … <= (signed)
ja dst       … above (unsigned)
jb dst       … below (unsigned)
```

dst is address of code (i.e., jump target)

comparison

```
cmpl src2, src1
   // like computing src1 - src2
cf=1 if carry out from msb
zf=1 if (src1==src2)
sf=1 if (src1-src2 < 0)
of=1 if two's complement
        under/overflow
```

testing

```
testl src2, src1
  // like computing src1 & src2
zf set when src1&src2 == 0
sf set when src1&src2 < 0
```

data movement

```
movl src, dst
```

src or dot can be:
- immediate (e.g., $0x10 or $4)
- register (e.g., %eax)
- memory (e.g., an address)

limits:
- dst can never be an immediate
- src or dot (but not both) can be memory

general memory form:

```
N (register1, register2, C)
```

which leads to the memory address:

```
N + register1 + (C * register2)
```

N can be a large number;
C can be 1, 2, 4, or 8

common shorter forms:

```
N              absolute (reg1=0,reg2=0)
(%eax)         register indirect (N=0,reg2=0)
N(%eax)        base + displacement (reg2=0)
N(%eax,%ebx)   indexed (C=1)
```

example:

```
movl 4(%eax), %ebx
```

takes value inside register %eax, adds 4 to it, and then fetches the contents of memory at that address, putting the result into register %ebx; sometimes called a "load" instruction as it loads data from memory into a register

set

```
sete dst      equal/zero
setne dst     not eq/not zero
sets dst      negative
setns dst     non-negative
setg dst      greater (signed)
setge dst     >= (signed)
setl dst      less (signed)
setle dst     <= (signed)
seta dst      above (unsigned)
setb dst      below (unsigned)
```

`dst` must be one of the 8 single-byte reg (e.g., %al)

often paired with `movzbl` instruction
(which moves 8-byte reg into 32-bit & zeroes out rest)

arithmetic
two operand instructions

```
addl src,dst   dst = dst + src
subl src,dst   dst = dst - src
imull src,dst  dst = dst * src
sall src,dst   dst = dst << src (aka shll)
sarl src,dst   dst = dst >> src (arith)
shrl src,dst   dst = dst >> src (logical)
xorl src,dst   dst = dst ^ src
andl src,dst   dst = dst & src
orl src,dst    dst = dst | src
```

one operand instructions

```
incl dst      dst = dst + 1
decl dst      dst = dst - 1
negl dst      dst = -dst
notl dst      dst = ~dst
```

arithmetic ops set CCs implicitly

```
cf=1 if carry out from msb
zf=1 if dst==0,
sf=1 if dst < 0 (signed)
of=1 if two's complement
     (signed) under/overflow
```