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.

# 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;

}

1. 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?
2. 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) {

}
3. What other line needs to change for the above program to compile correctly after your changes to drive\_car?

# 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.

1. What does the following code output? Note: %p formats the argument as a pointer in *hex*.
printf(“%p\n”, &i\_array[5]);
2. 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;
3. 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;
 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_;
 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_;
 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_;
 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_;
}

# 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) |

1. Perform 2’s complement binary addition on the following two 8-bit integers.
2. Convert each two’s complement binary value into decimal (right column)
3. Was there overflow in the two’s complement addition? How can you tell this?
4. Strike out the invalid basic C types from the following list: (the first one is done for you)

|  |  |
| --- | --- |
| * 1. ~~bigint~~
 | * 1. char
 |
| * 1. long double
 | * 1. nibble
 |
| * 1. long float
 | * 1. long
 |
| * 1. binary
 | * 1. short int
 |

# Question 4: C operators and pointers

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

1. 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;

1. 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);

# 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 |

# Question 6: Converting x86 assembly into C

1. 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 \_\_\_\_\_\_\_\_\_\_\_\_\_ |

# 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;
 }
}

1. Describe what usefulFunc1 does in one sentence. (Hint: If you need more than one sentence re-think your answer.)
2. Describe what usefulFunc2 does in one or two sentences. (Hint: If you need more than one or two sentences re-think your answer.)

# Blank Space for extra work

# x86 Cheat Sheet

