Instructions for Exam2Practice:

All Tests that you need to write are in the following 3 files:

1. AssemblyTests.s
2. Pointers.c
3. Structs.c

As you can probably guess, one deals with Assembly, one with structs, and another with pointers. I am going to include the function prototypes below as well as the structs you will use. All struct prototypes are in Constants.h. I encourage you to start on Pointers.c, then structs, then finally assembly. You might notice a few functions it would be beneficial to call later. The grade should total 100 if you manage to pass all of the test cases. The cases are not particularly rigorous, and there is no way to force you not to hardcode the solutions, but that would deprive you of a valuable learning experience. So please follow the directions. Thank you

A few reminders that might make your life easier:

1. AAPCS guarantees that you can use R0-R3 inside the function, but if you call another function and had data stored in there it might be destroyed by that function.
2. If you BL inside an ASM routine, your LR previous address is gone so you might want to push it somewhere
3. You can permanently modify an array by passing a pointer to memory. If you plan on calling a routine to sort some data, your data will be modified. Make sure you are allowed to do so. In this lab you are always allowed to modify your inputs but this is not always the case.
4. Local Variables in ASM are allocated on the stack. It might be beneficial to practice them on some of the ASM routines
5. Sorting is something that you did in 306 and probably don’t remember all that well. I encourage you to practice it as it has shown up on past exams and will help you create a mental picture of what is going on. I would recommend memorizing at least one sorting technique. The technique doesn’t have to be fancy, something like bubble sort is fine.
6. Calling an ASM routine from a C function is not tested in these projects because it is used so often in Lab 7 so it should be fresh on your minds. I encourage you to take a look at it again.
7. To use an ASM routine in a C function make sure you EXPORT the location in your ASM file
8. To call a C function from another file make sure you IMPORT it if its an ASM file or #include the header file

The prototypes for all the functions are on the following pages. If you have any questions feel free to email me at [mebarondeau@utexas.edu](mailto:mebarondeau@utexas.edu) or ask on Piazza.

Assembly Tests:

//This function is passed a pointer to a Mealy FSM and you must return the nextState

//There is output in the Mealy FSM but you are not required to do anything with it

//This function needs to be AAPCS compliant

**uint8\_t NextState( state\_t \*Mealy, uint8\_t currentState, uint8\_t input );**

//This function is passed 3 pointers to three numbers

//Your job is to dereference the pointers and sum their values together

//Finally you must return the sum in an AAPCS compliant way

**uint32\_t AddingPointersInASM(uint16\_t \*num1, uint16\_t \*num2, uint16\_t \*num3);**

//This function is similar to AddingPointersInASM, but multiply instead of add

**uint32\_t MultiplyingPointersInASM(uint16\_t \*num1, uint16\_t \*num2, uint16\_t \*num3);**

//This function is particularly fun

//You are given a pointer to an array and your job is to modify the output Array

//You will use the parsing Offset to grade every ith element from the array

//For instance input = “abababababab\0” and your parsing offset is 4

//The your output will be “aaa”

//For your convenience you are given the size of the array

**void ParsingArrayInASM(uint8\_t \*inputArray, uint8\_t sizeOfArray, uint8\_t parsingOffset, uint8\_t \*outputArray);**

//This function is used to break apart data

//You will be fed a 32 bit number and your job is to break it into 4 1 byte segments

//Once the segments have been broken up, multiply them together and return result

**uint32\_t BreakApartDataASM(uint32\_t input1);**

//This function is the easiest

//Pass a pointer to the assembly routine and return the value at the address

**int32\_t ReturnValueatPointer(uint32\_t \*pointer);**

//This function is passed four pointers and you need to return their result multiplied together

//The trick to this function is to not use the LDR instruction at all

//Call other functions to do the work for you but remember AAPCS

//As a challenge use Local Variables to store intermediate values, not registers

**int32\_t FourPointersASM(int32\_t \*ASMPointer1, int32\_t \*ASMPointer2, int32\_t \*ASMPointer3, int32\_t \*ASMPointer4);**

//This function is passed an unsorted array of numbers

//You can either sort it yourself in ASM if you wish or call a function to sort it for you

//Any form of sorting is acceptable

**uint32\_t rangeUnsortedASM(uint8\_t \*unsortedArray);**

//This function is passed the pointer to a struct!

//Your job is to find the range of student scores

//It might be beneficial to sort the array first

**uint32\_t ASMcallC(score\_t \*studentArray);**

Pointers.C Tests:

//This function is passed three pointers

//Dereference the values, multiply them together and return the result

**uint32\_t PointerMultiplication(uint16\_t \*num1, uint16\_t \*num2, uint16\_t \*num3);**

//Similar to the first except you are adding this time

**uint32\_t PointerAddition(uint16\_t \*num1, uint16\_t \*num2, uint16\_t \*num3);**

//This function is passed the pointers to two arrays that are null terminated

//Add all the elements in each array together

//For instance if array1 is “1, 3, 5” and array2 is “3, 5, 7”

//Then return 24

//Be sure to think about how this would change if the variables were not 8 bits

**uint32\_t SumOfCharArray(uint8\_t \*array1, uint8\_t \*array2);**

//This is one of your most important functions

//Sort an array who you get a pointer to

//Implement a sort however you wish

//Think about how this would change if it were 16 bits or passed as a normal array

**void sortArray(uint8\_t \*array);**

//When passed a pointer to an array, return the number of elements in the array

//Array is null terminated

**uint8\_t lengthOfArray(uint8\_t \*array);**

//Return the length of the array that is passed

//Array is null terminated

**uint8\_t lengthOfArray2(uint8\_t array[]);**

//Sort an array of characters called AlphabetArray

//Once the array has been sorted, index through the array

//Return the element located at indexToReturn

//Think about how this function would change if we used pointers

**char alphabetSort(unsigned char AlphabetArray[], uint8\_t indexToReturn);**

Structs.C Tests:

//You are passed a pointer to an array of state\_t which implements a Mealy FSM

//Use the input and current state to return what the Mealy should output

**uint8\_t GetOutputFromMealy( state\_t \*Mealy, uint8\_t input, uint8\_t currentState);**

//You are passed a pointer to an array of state\_t which implements a Mealy FSM

//Use the input and current state to return what the next state in the machine should be

**uint8\_t getNextState( state\_t \*Mealy, uint8\_t input, uint8\_t currentState);**

//You are passed a pointer to a score\_t struct(show below)

//Sort the scores in increasing order

//Might be beneficial to call some other functions your wrote

**void sortInternalField(score\_t \*input1);**

//You are passed a pointer to a student\_t Array and the number of students in the array

//Find the range of scores

//I encourage you to sort the array and then find the min and max rather than iterating through

//But both approaches will solve the problem

**uint32\_t topScorer(student\_t \*inputArray, uint8\_t numStudents);**

struct state {

uint8\_t output[4];

uint8\_t nextState[8];

};

typedef struct state state\_t;

struct scoreStruct {

uint8\_t scores[10];

};

typedef struct scoreStruct score\_t;

struct Student {

uint32\_t studentIDNumber;

uint16\_t scoreExam1;

uint16\_t scoreExam2;

};

typedef struct Student student\_t;