Lab 2 (week 4 and 5)

Instructions

- Complete each task and demonstrate the working program to your tutor. Tasks should be demonstrated using AVR Studio's simulator. **Part A of this lab must be marked by the end of the lab session in week 4**. **Part B and Part C of this lab must be marked by the end of the lab session in week 5.**
- Read the lab sheet carefully and go through the theory before you come to the week 4 lab session. Post any doubts in the course forum indicating your lab time so that the corresponding tutors can answer those.
- Part B and C, will lead to the lift emulator you need to develop for your project.

Hints

- Be organised and neat – it will heavily reduce the debugging time
- Put comments
- Have a printed sheet of the AVR instruction set (page 1,10-15 of http://www.cse.unsw.edu.au/~cs2121/AVR/AVR-Instruction-Set.pdf)

Part A – Reverse String (3 Marks)

Implement a program that loads a null-terminated string from program memory and pushes it onto the stack, then writes out the reversed string into data memory. The stack pointer must be initialized before use.

Eg: "abc",0 will be stored in data memory as "cba",0

Part B – Function calls (3 marks)

In this part, you need to modify part D of lab 1 and perform *in-place sorting* using a function. For this, you need to write a function named *insert_request* to implement the *in-place array sorting*. The prototype of the function *insert_request* in C programming is given below.

```
uint8_t insert_request(uint8_t* sorted_array, uint8_t array_size, uint8_t value);
      // assume uint8_t is an unsigned 8-bit number
      // sorted_array is the 16-bit address of sorted array in data memory
      // array_size is the number of elements in the array
      // value is the number that is to be inserted in the sorted array
      // return value is the new size of the array
```

- Arguments to the function: address of the sorted array (array in **data memory**), the number of elements in the array, and the next value to be inserted are the arguments to the function.
- Return value: the return value is the number of elements in the array after the insertion.
- In your assembly implementation of the function, use registers *r8, r9, r10, r11* for passing the arguments and *r24* for the return value.
- Make sure you save the conflict registers on to the stack correctly at the beginning (prologue) of the function (this includes any register that you modify the content inside the function, excluding the return register) and restore them at the end (epilogue) of the function. Refer to the examples in your lecture notes.
- You are not required to store arguments or local variables in the stack.

Now outside the function:

- load the sorted array (1, 2, 5, 7, 8, 12, 20) from program memory to data memory;
- iterate through the queue of numbers (0, 10, 1, 25, 6) in the program memory while calling *insert_request* for each element. A pseudocode of this step is given below.

```
uint8_t queue[] = {0, 10, 1, 25, 6};
uint8_t array_size = initial_array_size;
for (i=0; i<queue_size; i++) {
      array_size=insert_request(sorted_array, array_size, queue[i]);
}
```

- Make sure the stack pointer is initialised before calling the function and find out why.

Part C – Basic lift controller (4 marks)

This part will lead to your final project in this course, the lift emulator.

Extend *insert_request* function above to a basic lift controller (simulation based on data structures and algorithms) that can service the requests for the lift. The lift should be like in a real-life scenario (see the prologue in lab 1). Note that demonstration of the actual movement of the lift is not required yet, and you should only simulate the ordering of the subsequent service requests.

The sorted array now represents the remaining requests which the lift should service. The next value to be inserted represents the next request call entered by a user. Now, the sorting of the array should be performed based on the **current floor** and the **current travel direction of the lift (up or down)** - which you can assume to be arguments. Two examples are given below:

- The lift is currently on floor 3 and it is moving upwards. It is supposed to service the requests for floors (5,7,8,1) respectively – this is the "sorted" array now. A user on floor 10 requests for the lift. When this request is inserted the "sorted" array looks like (5,7,8,10,1). Then another request comes, this time from floor 6 and the array looks like (5,6,7,8,10,1), The next request is from floor 7, but it should not be inserted as 7 is already in the array.
- The lift is on floor 8 and moving down. The list of subsequent floors to visit are (5,3,2,10,11,15). If the button for floor 13 is pressed, it will be inserted into the list such that the new list is (5,3,2,10,11,13,15). If the button for floor 4 is then pressed, the new list will be (5,4,3,2,10,11,13,15).

You need to demonstrate the working code to the tutor using different example cases of the *sorted array (remaining requests)* and *the queue of the subsequent requests*. This can be done by modifying the array definitions in the source code that initially stores in the program memory.