**ECE 260: Fundamentals of Computer Engineering – Lab #6  
More MIPS Procedures**

**Name \_\_\_\_\_Le Quient Lewis II\_\_\_\_\_ Lab Partner(s) \_\_\_N/A\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**1. Introduction**: This lab provides students with additional practice implementing MIPS procedures using the *jal* and *jr* instructions, modifying the stack, and utilizing registers **$a0**-**$a3** and **$v0**-**$v1**.

**2. Background**: In the previous lab you were required to implement several procedures in MIPS assembly. This lab is a continuation of the previous lab and will give you some additional experience implementing MIPS procedures.

For your convenience, the conventions for calling procedures are included here:

* The **CALLER** will:
  + Save all temp registers that it wants to survive subsequent procedure calls into its stack frame  
    (**$t0**-**$t9**, **$a0**-**$a3**, and **$v0**-**$v1**)
  + Pass the first 4 arguments to a CALLEE in registers **$a0**-**$a3** — save subsequent arguments on stack, in reverse order
  + Call CALLEE procedure, using a *jal* instruction which places the return address in register **$ra**
    - If this CALLER is also a CALLEE, you must save **$ra** before using *jal*
  + Access CALLEE procedure’s return values in registers **$v0**-**$v1** after CALLEE returns
  + Restore all temp registers that were saved prior to calling CALLEE
    - Be sure to grab return value from CALLEE prior to restoring any saved **$v0**-**$v1** from stack or you will overwrite the CALLEE’s return value
* If needed the **CALLEE** will:
  + Allocate a stack frame with space for saved registers, local variables, and spilled args
  + Save any “preserved” registers that it will use/overwrite: **$ra**, **$sp**, **$fp**, **$gp**, **$s0**-**$s7**
  + If CALLEE has local variables -or- needs access to args on the stack, save CALLER’s frame pointer and set **$fp** to 1st entry of CALLEE’s stack
  + EXECUTE procedure
  + Place return values in **$v0**-**$v1**
  + Restore saved registers including those that were preserved for CALLER
  + Restore **$sp** to its original value
  + Return to CALLER with *jr* **$ra**

Recall that, by convention, certain registers must be preserved across procedure calls. These registers are indicated in the MIPS Green Sheet and were presented in lecture. Further, recall that any registers that are not preserved cannot be assumed to hold the same value after a procedure returns, i.e. the value of **$t0** may be different after a procedure call.

This lab will help you further understand procedures and practice their implementation within MIPS.

**3. Writing Procedures**

**3.1 Writing a Recursive Binary Search**

In this part, you will need to implement a MIPS assembly program that performs a recursive binary search on an array. A binary search is an efficient method for searching ordered data with O(log n) average performance. The binary search algorithm searches an input array for a specified *key* value. The algorithm recursively splits the input array in half such that the portion of the array that is searched gets smaller and smaller as the function recurses deeper and deeper. Parameters named *min* and *max* provide the lower and upper bounds of the search at each level of recursion. The return value of the binary search function is either the array index of the located key value, or -1 to indicate that the key value was not found in the array.

A C implementation of the binary search algorithm is shown in Figure 1. You can read more about the binary search algorithm and its various implementations here: <https://en.wikipedia.org/wiki/Binary_search_algorithm>. Figure 2 shows a “*main*” function to illustrate how the “*binary\_search*” method should be called.

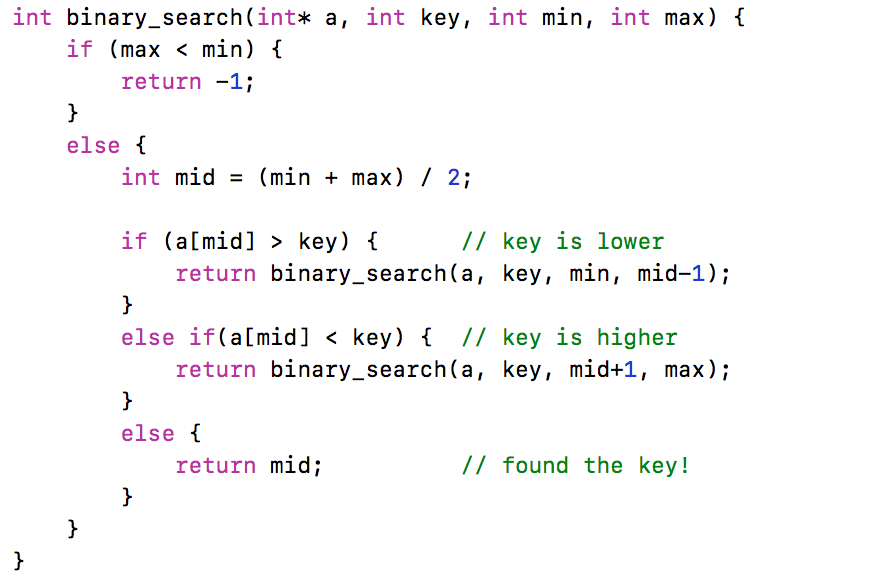


Figure 1: Binary Search Procedure

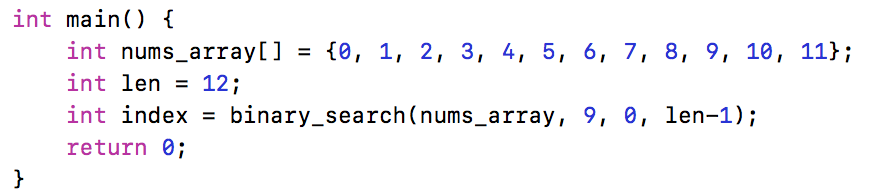


Figure 2: Searching for the Value 9 Within an Array Using the Binary Search Procedure

1) In MARS, open the file called lab06\_part1.asm. In this file, you will need to write an assembly program to implement the “*binary\_search*” procedure. You’ll see the .data segment has already been populated with an array called *nums\_array* and a length variable called *len*. Additionally, you’ll see that registers **$s0**, **$s1**, and **$s2** have been populated with the value for *len*, the base address of *nums\_array*, and a key for which to search. You should change the value of register **$s2** (the key value) in the *initialize* section when testing your code.

2) Under the “*ece260\_main*” label, set up the argument registers and make the call to your “*binary\_search*” procedure.

3) Implement your “*binary\_search*” procedure under the label provided. Start with the base case. You should find that there is no need to manipulate the stack when executing the base case. After completing the base case, work on the recursive case. Because the “*binary\_search*” procedure is a CALLER, you’ll need to save (and later restore) the **$ra** register on the stack. However, that should be the only value that needs to be stored on the stack. Because of the way the C code is structured, no values need be saved between recursive calls to “*binary\_seach*”. That means that you can avoid using **$sX** registers and use exclusively **$tX** registers for all computation. The only local variable is *mid*, and it is never used after a call to “*binary\_search*” returns. Therefore, you can store *mid* in a **$tX** register with no worries.

4) After your “*binary\_search*” procedure returns to “*ece260\_main*”, move the final return value into the **$s3** register. The provided unit tests will look for your final value there.

5) Test your program to verify that it operates as expected. The table below provides a list of key values that you should test on your program. When testing code, it is EXTREMELY important to test more than just test a few cases. You should identify as many different test cases as possible and test that they all produce the correct output. Each key value in the table below has a short caption that describes the case that the key value is intended to test. Complete the table by passing each key value to your function by modifying the **$s2** register in the *initialize* section.

Table 4: Expected values for Binary Search algorithm

|  |  |  |
| --- | --- | --- |
| **Key Value** | **Expected Index** | **Computed Index** |
| **29**  key is in array; never recurses | 10 | 10 |
| **11**  key is in left half of array | 5 | 5 |
| **53**  key is in right half of array | 24 | 24 |
| **2**  key is 0th index of array | 0 | 0 |
| **71**  key is last index of array | 28 | 28 |
| **10**  key not in array; search left half | -1 | -1 |
| **60**  key not in array; search right half | -1 | -1 |
| **1**  key not in array; search key is less than smallest value | -1 | -1 |
| **99**  key not in array; search key is bigger than largest value | -1 | -1 |

**Test Your Code**

You should test your code with different initial values to ensure that it satisfies the requirements described above. Then, test your code using the supplied unit tests. To run the supplied unit tests, open a Cygwin shell and type the following:

cd h:  
cd ECE260  
cd Lab06\_More\_MIPS\_Procedures  
make test\_part1

You will see output that indicates if your code passed or failed the included unit tests. If your code did NOT pass the units tests, address any errors and try running the unit tests again.

**3.2 Writing a Recursive Selection Sort (CHALLENGING!)**

In this part, you will need to implement a MIPS assembly program that performs a selection sort on an array of integers stored in memory. The selection sort algorithm divides the input array into two sections, a sorted section and an unsorted section. The sorted section resides in the lower indices of the array. The unsorted section resides in the upper indices of the array. Initially, the entire input array is considered unsorted. At each step, the algorithm selects the smallest value from the unsorted section and swaps it with the smallest index of the unsorted section. The size of the sorted section then grows to include that newly sorted index. Consequently, the size of the unsorted section shrinks. This process continues until there are no more indices left in the unsorted section and the complete array is sorted.

A C implementation of a recursive implementation of the selection sort algorithm is shown in Figure 3. Note that the recursive selection sort utilizes two utility procedures, one to swap array elements and another to find the index of the minimum value in the unsorted section of the array. You can read more about the selection sort algorithm (and see some helpful animations) here: <https://en.wikipedia.org/wiki/Selection_sort>. Figure 4 shows a “*main*” function to illustrate how the “*recursiveSelectionSort*” method should be called.

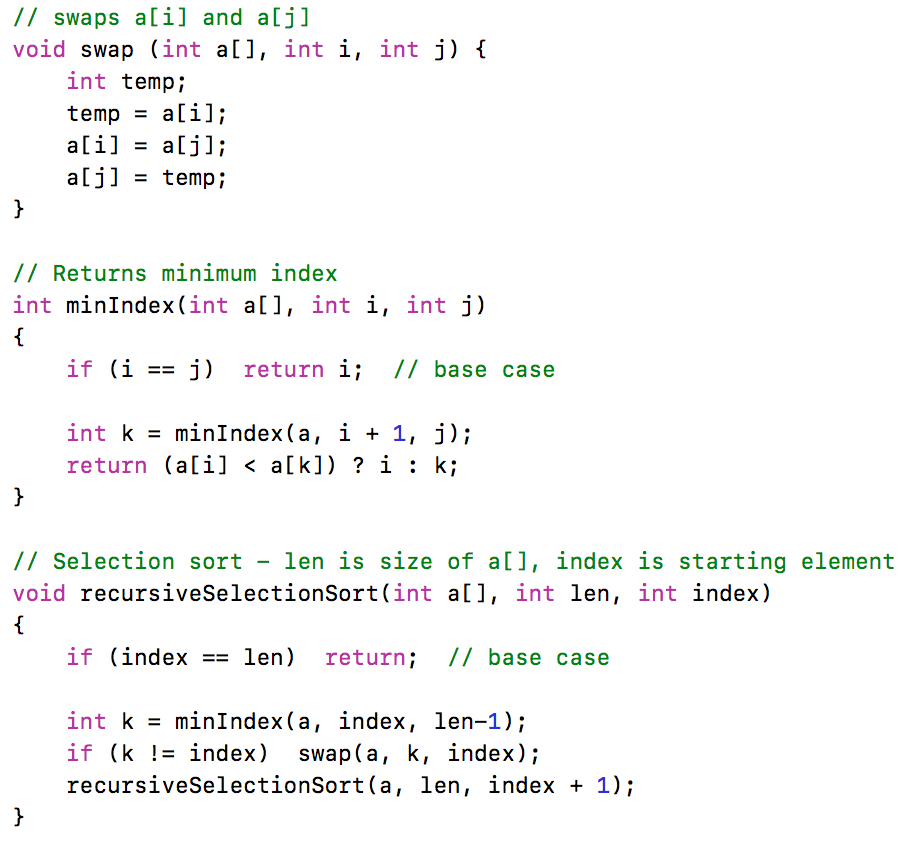
****

Figure 3: Recursive Selection Sort Procedure

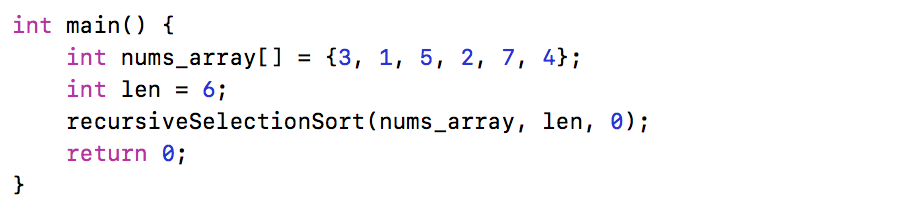
****

Figure 4: Driver Code for Recursive Selection Sort Procedure

Before getting started, it should be noted that this part is a bit more challenging than all of the other procedures that you have written. To help make things a bit easier, you should write **and test** each of the utility procedures first. Only after you have passed all the tests for the utility procedures should you move on to the actual “*recursiveSelectionSort*” procedure. A separate set of unit tests have been provided for each of the utility procedures.

1) In MARS, open the file called lab06\_part2.asm. You’ll see the .data segment has already been populated with an array called *nums\_array* and a length variable called *len*. Additionally, you’ll see that registers **$s0** and **$s1** have been populated with the value for *len* and the base address of *nums\_array*. You’ll also see labels for the various procedures you need to write. You may need to create additional labels within some of your procedures.

2.a) Implement your “*swap*” procedure under the label provided. **Hint:** refer to the lecture notes on “Supporting Nested Procedures”. The “*swap*” procedure takes three arguments. Pass the base address of an array in register **$a0** and pass the indices to swap in registers **$a1** and **$a2**. The “*swap*” procedure has no return value, it simply swaps the values at *a[i]* and *a[j]*. The “*swap*” procedure is a leaf-procedure so it is recommended that you use **$tX** registers within the body of this procedure. You should not need to do any stack manipulation in the “*swap*” procedure.

2.b) Write some temporary driver code to test your “*swap*” procedure (you should delete this later). Under the “*ece260\_main*” label, set up the argument registers with the base address of *nums\_array* and two indices of your choosing. Make the call to your “*swap*” procedure to see if it operates as expected.

2.c) Unit tests have been provided that will allow you to do some automated testing on the “*swap*” procedure before continuing. Do not move onto the next step until you are certain that your “*swap*” procedure operates correctly. To run the supplied unit tests for the “*swap*” procedure, open a Cygwin shell and type the following:

cd h:  
cd ECE260  
cd Lab06\_More\_MIPS\_Procedures  
make test\_part2\_swap

You will see output that indicates if your procedure passed or failed the included unit tests. If your code did NOT pass the units tests, address any errors and try running the unit tests again.

3.a) Implement your “*minIndex*” procedure under the label provide. The “*minIndex*” procedure takes three arguments. Pass the base address of an array in register **$a0**. Pass two indices, a lower bound and an upper bound, in registers **$a1** and **$a2** respectively. The “*minIndex*” procedure returns the index of the smallest element between the lower and upper bounds (inclusive). Although the “*minIndex*” procedure is not a leaf-procedure, it does not compute any data that needs to be saved between recursive calls. Therefore, you should be able to avoid using **$sX** registers. There are, however, at least two pieces of data that you will likely want to store on the stack – the **$ra** and **$a1** registers.

3.b) Write some temporary driver code to test your “*minIndex*” procedure (you should delete this later). Under the “*ece260\_main*” label, set up the argument registers with the base address of *nums\_array* and two indices of your choosing. Make the call to your “*minIndex*” procedure to see if it operates as expected.

3.c) Unit tests have been provided that will allow you to do some automated testing on the “*minIndex*” procedure before continuing. Do not move onto the next step until you are certain that your “*minIndex*” procedure operates correctly. To run the supplied unit tests for the “*minIndex*” procedure, open a Cygwin shell and type the following:

cd h:  
cd ECE260  
cd Lab06\_More\_MIPS\_Procedures  
make test\_part2\_minIndex

You will see output that indicates if your procedure passed or failed the included unit tests. If your code did NOT pass the units tests, address any errors and try running the unit tests again.

4.a) Implement your “*recursiveSelectionSort*” procedure under the label provide. The “*recursiveSelectionSort*” procedure takes three arguments. Pass the base address of an array in register **$a0**. Pass the length of that array in register **$a1** and a starting element in register **$a2**. The “*recursiveSelectionSort*” procedure has no return value. However, it will use the “*swap*” procedure to sort the contents of the input array. Just like with the “*minIndex*” procedure, you should be able to implement the “*recursiveSelectionSort*” procedure without using any **$sX** registers. However, you will need to save the return address and all of the arguments onto the stack. You will need to restore the argument registers more than once in the “*recursiveSelectionSort*” procedure.

4.b) Delete (or comment out) any driver code that you wrote for your “*swap*” and “*minIndex*” procedures. Now, write some driver code for your “*recursiveSelectionSort*” procedure. Under the “*ece260\_main*” label, set up the argument registers with the base address of *nums\_array* in register **$a0**, the length of *nums*\_array in register **$a1**, and a 0 in register **$a2**. Make the call to your “*recursiveSelectionSort*” procedure to see if it operates as expected.

4.c) Unit tests have been provided that will allow you to do some automated testing on the “*recursiveSelectionSort*” procedure before continuing. Do not move onto the next step until you are certain that your “*recursiveSelectionSort*” procedure operates correctly. To run the supplied unit tests for the “*recursiveSelectionSort*” procedure, open a Cygwin shell and type the following:

cd h:  
cd ECE260  
cd Lab06\_More\_MIPS\_Procedures  
make test\_part2\_sort

You will see output that indicates if your procedure passed or failed the included unit tests. If your code did NOT pass the units tests, address any errors and try running the unit tests again.

**Test Your Code**

Although you have already tested that various procedures above, you must still test to ensure that all of your procedures work together. Test all of your code together using the supplied unit tests. To run the supplied unit tests, open a Cygwin shell and type the following:

cd h:  
cd ECE260  
cd Lab06\_More\_MIPS\_Procedures  
make test\_part2

You will see output that indicates if your code passed or failed the included unit tests. If your code did NOT pass the units tests, address any errors and try running the unit tests again.

**4. Submission**

When you have finished your lab, demo your program for your instructor. Write your answers to the above questions electronically in this document. To submit your lab assignment, make sure your files have all been saved (*including this file*). In a Cygwin window type the commands:

cd h:  
cd ECE260  
cd Lab06\_More\_MIPS\_Procedures  
make submit

Enter your Marmoset username and password (which you should have received by email). Note that your password will not be echoed to the screen. Make sure that after you enter your username and password, you see a message indicating that the submission was successful. Log into [Marmoset](https://cs.ycp.edu/marmoset/login) via the web to check the files you submitted to ensure they are correct.

**DO NOT MANUALLY ZIP YOUR PROJECT AND SUBMIT IT TO MARMOSET.  
YOU MUST USE THE make submit COMMAND.**