CSE 338 – Computer Organization

Project #1 Report

Group Members:

Cem Güleç - 150117828

Ömer Faruk Çakı - 150117821

On top of our program, we have a `.data` section, all of our memory allocations, prompt strings etc. are declared there.

Then we have a `.text main` section, which is the starting point of our program.

In order to explain the project code further, we need to explain how we implemented the menu first.

Menu:

We designed the menu as a while loop. When user run the program for the first time, welcome message is printed. After that on each loop, menu's choices are printed on the screen and program expect an input from user.

For example, if the input is 1, the program will ask for 1st question's input which is the '*number of iterations*' in this case. After getting corresponding inputs from user, we place necessary inputs in $a registers. After we filling necessary $a registers, program calling the corresponding procedure with `jal Procedure_1`.

This step is similar for the 2nd and 3rd question, just inputs are different and they are forwarded to their own procedures after setting $a registers with input values.

If 4 is selected, program will jump to the `EXIT` label and exits the program. EXIT is a system call and it's available at the end of the file:

```
EXIT:
li $v0, 10 # terminate program
syscall
```

If the selection isn't between 1-4, while loop will continue looping and prompts for selection until a valid menu item is selected.

Question 1:

The procedure parameters as follows:

```
# $a0: <int> = number of iterations
```

The only parameter is the `$a0` and it represents the number of iterations. We save `$a0` to stack by allocating a 4 byte. Because $a register values should be preserved as requirement of this project.

We are counting back to the zero from the value of $a0 in a while loop. A and b values are both initially 1. At each iteration we calculate the new a and b values. There are two loops, one for printing a's sequences and other one printing b's sequences.

After all printing is done, program restores the original value of $a0 from the stack and deallocated the space of the $sp by 4 bytes. Finally, we return back where ww left with `jr $ra`

Question 2:

The procedure parameters as follows:

```
# $a0: <addr> = address of first matrix string
# $a1: <addr> = address of second matrix string
# $a2: <int>  = first dimension
# $a3: <int>  = second dimension
```

At the very beginning of the procedure, first thing we do is saving all four $a registers to stack by allocating 16 bytes of memory from the stack.

The `$a0` register holds the first matrix as a space separated string exactly in the same way of user entered. `$a1` holds the second string as well.

We parse these strings by iterating them 1 byte at a time by getting its value with `lb` (load byte). We store new parsed integers in a memory location we allocated before, `first_matrix_array` and `second_matrix_array` respectively.

Parsing the number largen than 1 digit is a bit tricky process, we do that by running another loop inside and it checks the next char before really moving into it and if it's again a number, we construct the real number by multiplying the previous digit with 10.

At the end of parsing operation there are two arrays available to hold our parsed matrices. Before the matrix multiplication operation, we calculate the second dimension of the second matrix which is initially unknown.

While thinking about the multiplication method, we come up with the following c++ code which we designed considering MIPS.

```
for (int i = 0; i < a2; i++)
{
    for (int j = 0; j < x; j++)
    {
        int res = 0;
        for (int k = 0; k < a3; k++)
        {
            res = res + (A[i][k] * B[k][j]);
        }
        cout << res << " ";
    }
    cout << endl;
}
```

We coded our MIPS instructions by looking at the c++ code we wrote above. Our MIPS code is same implementation of this.

After the multiplication operation is done, we restore our initial `$a` register values and deallocate the stack space by adding the 16 bytes we allocated before. After that we return the return address with `jr $ra`

Question 3:

Before getting into the implementation, first of all, we have reserved space for "buffer"- which will be used in storing our original string read from the user, under the register $t0. Also, address Is saved under the argument register $a0, to reach out this address later on.
Other than that, space and newline are reserved under the registers $t1, $t7 respectively.
We also save $a register to stack by allocating 4 bytes of memory from the stack.
Last of all, we have reserved space for "bufferSmaller"- which will be used in order to store the string after converting letters into smaller ones, under the register $t3. Below clear explanation of which register is used in which purpose is explained:

```
# t0: buffer variable
# t1: spaceChar
# t2: character read / first index
# t3: bufferSmaller variable
# t4: small letter variable / last index
# t5: counter
# t6: last index value
# t7: newLine / first index value
```

Getting into the implementation, first thing we have done is to hold a register $s0 to store value 90. In purpose that, it will check whether ascii value of a character read from the string is small or not.

There are 2 loops that are handling situations based on specific purposes.
First loop handles converting letters into smaller ones. Here, every character read from the address which is held by $t0 is assigned to another register $t2.

Firstly, it is checked whether the character is a new line character or not. If that's the case, we know that we come across to end of conversion mission.

Else, we or the character ascii value with 0x20 hexadecimal value, in order to get corresponding small letter ascii value.

In every iteration, we increment the index value by adding $t0 with 1. This loop calls itself over and over again until facing with the end line character.
Additionally, there is another register $t5 which reserved under the purpose of

holding the counter value. In every iteration, we increment it by one. This counter lets us see the length value of the string.

After completing the conversion, second loop handles checking whether the converted string is palindrome or not. To implement this, we have declared two pointer representation in the algorithm, which are holding first and last indexes of the string- $t2, $t4 respectively. For holding the last index, $t5- mentioned before is used in this purpose.

Here, we check character ascii values which are pointed by $t2 and $t4 added to the memory location address of $a0- which is the base location of the converted string. This gives us the character values from the beginning and the end of the string.

If the values are not the same which violates the rules of being palindrome, we directly jump “endNotPalindrome”- which simply terminates the program and prints out that the corresponding string is not palindrome.

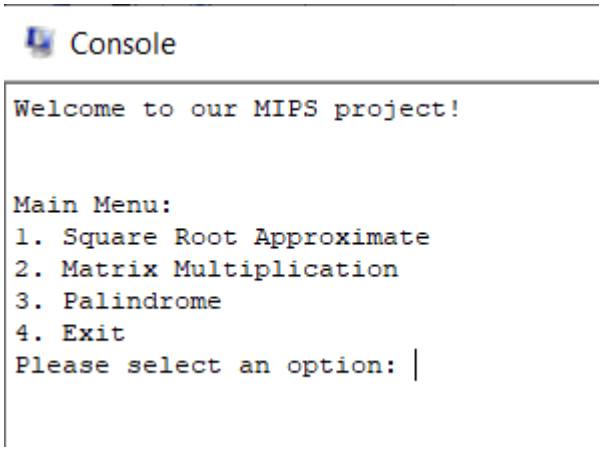
Otherwise, we continue on checking other cases that, we increment register value which was holding first index by one iteratively. Also, we decrement register value which was holding last index by one iteratively.

There are 2 cases that needs to be checked: first one is what happens if two indexes reach to the middle point if the string length is odd. And the other one is what happens if there is no mid-point which means that length of the string is even.

In order to handle this, “bge $t2, $t4, endPalindrome” instruction checks both of the cases at once. If the indexes meet up in the middle point or not- which means they jump over each other to other half and $t2 will be greater than $t4, we will simply jump to the “endPalindrome” which will be terminating the program and printing out that the string is palindrome.

After the palindrome checking operation is done, we restore our initial `$a` register values and deallocate the stack space by adding the 4 bytes we allocated before. After that we return the return address with `jr $ra`

Sample Run:



First of all, programs greets us with the menu which is described in previous part. Takes input from the user and depending on that input proceeds to corresponding operation.

```
Please select an option: 1

Enter the number of iteration for the series: 5
a: 1 3 7 17 41
b: 1 2 5 12 29

Main Menu:
1. Square Root Approximate
2. Matrix Multiplication
3. Palindrome
4. Exit
Please select an option:
```

Once the option 1 is selected, iteration number is asked. Depending on the input given it proceeds with the first procedure implementation. After the implementation memory is refreshed and once again menu is called.

```
Please select an option: 2

Enter the first matrix: 3 2 5 6 4 4 4 4 2 2 1 1

Enter the second matrix: 1 8 7 5 4 2 1 3

Enter the first dimension of first matrix: 3

Enter the second dimension of first matrix: 4

Multiplication matrix:
43 62
52 72
21 31


Main Menu:
1. Square Root Approximate
2. Matrix Multiplication
3. Palindrome
4. Exit
Please select an option:
```

Once option 2 is selected and required inputs are entered program is proceeded with the second procedure and calculated the matrix calculation. After implementation is done memory refreshed and menu is called once again.

```
Please select an option: 3
Enter the input string: Kedi ideK
kedi idek is palindrome

Main Menu:
1. Square Root Approximate
2. Matrix Multiplication
3. Palindrome
4. Exit
Please select an option:
```

An example to a palindrome string.

```
Please select an option: 3
Enter the input string: KeKeMeK
kekemek is not palindrome

Main Menu:
1. Square Root Approximate
2. Matrix Multiplication
3. Palindrome
4. Exit
Please select an option:
```

An example to a not palindrome string.

```
Main Menu:
1. Square Root Approximate
2. Matrix Multiplication
3. Palindrome
4. Exit
Please select an option: 4
Program ends. Bye :)
```

Whenever we want to quit the program pressing 4 is sufficient. After executing this program prints outs the ending string.