CST 250

Final Project

Hieu Pham

- Project Requirement
- Reference Information
- Implementation
- Results
- Challenges

Project Requirement

- Review the information for Gnome Sorting algorithm.
- Implement the Gnome Sorting function.
- This sorting function needs to perform an in-place sort on an array in memory (only memory within the array is modified while sorting).

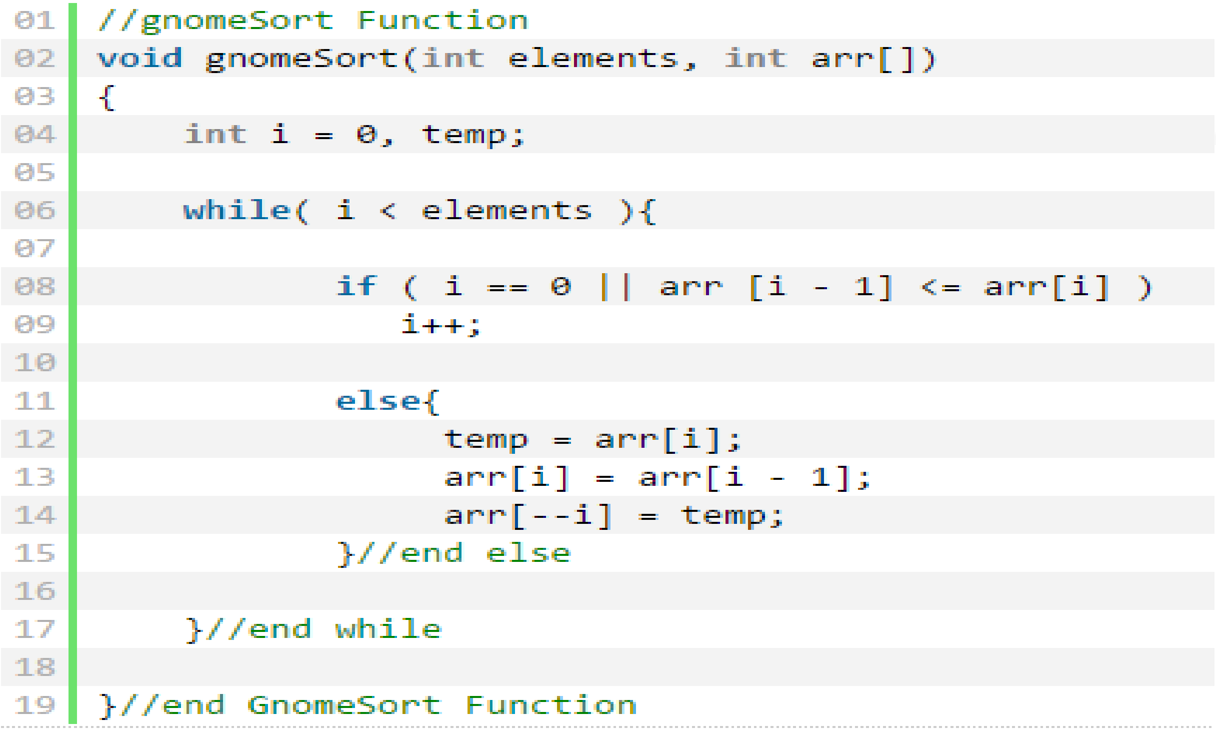
Reference Information

- There are common places in the Internet where Gnome Sorting algorithm information can be found. I visited the following sites during this project:
  - https://en.wikipedia.org/wiki/Gnome_sort
  - http://buffered.io/posts/sorting-algorithms-the-gnome-sort/
  - https://en.wikibooks.org/wiki/Algorithm_Implementation/Sorting/Gnome_sort

- From a high-level perspective, the gnome sort involves comparison and exchange mechanism similar to the bubble sort.

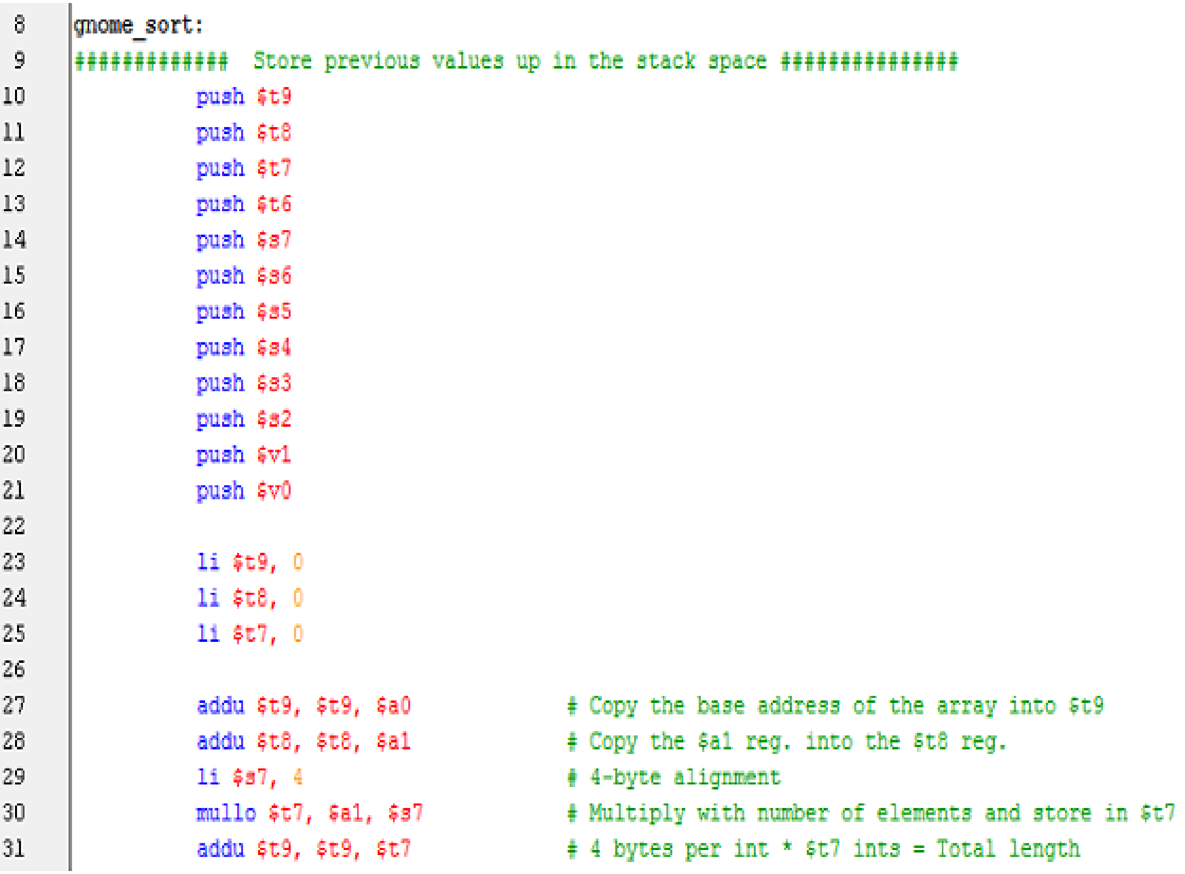
Implementation

PSEUDO_CODE

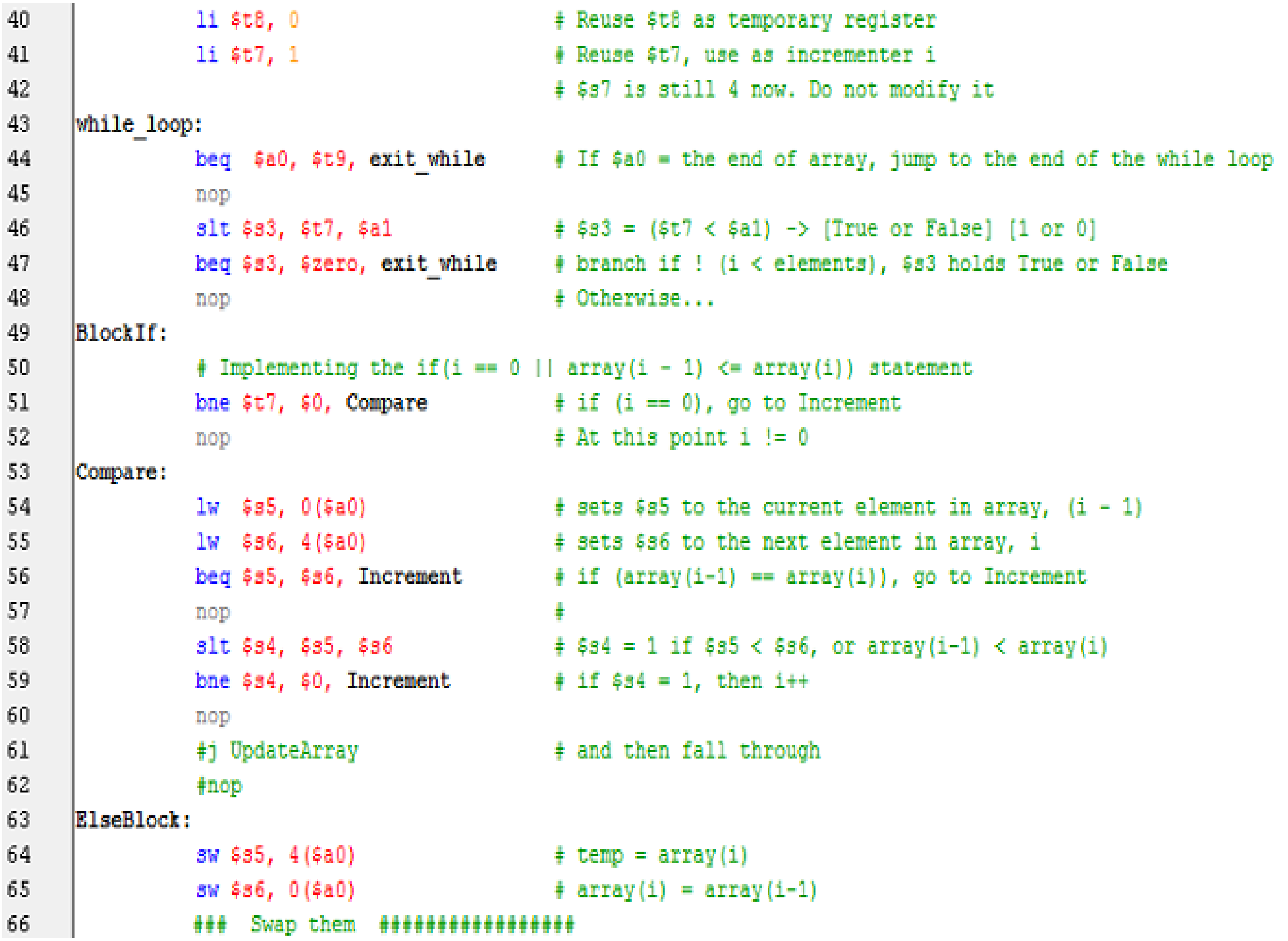


```
//gnomeSort Function
void gnomeSort(int elements, int arr[])
{
    int i = 0, temp;

    while( i < elements ){

           if ( i == 0 || arr [i - 1] <= arr[i] )
              i++;

           else{
                temp = arr[i];
                arr[i] = arr[i - 1];
                arr[--i] = temp;
           }//end else

    }//end while

}//end GnomeSort Function
```

Implementation

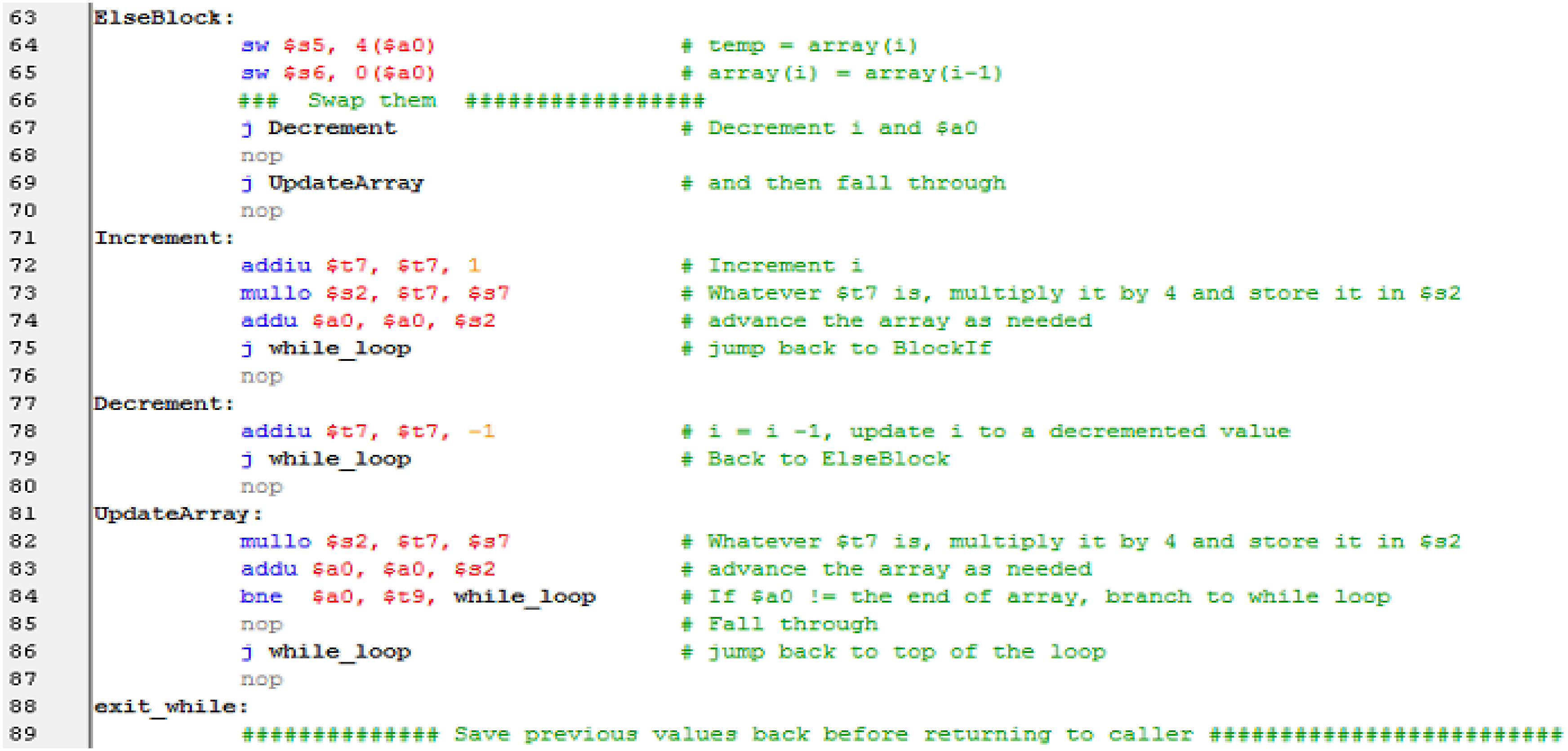


```
gnome_sort:
#############  Store previous values up in the stack space ###############
            push $t9
            push $t8
            push $t7
            push $t6
            push $s7
            push $s6
            push $s5
            push $s4
            push $s3
            push $s2
            push $v1
            push $v0

            li $t9, 0
            li $t8, 0
            li $t7, 0

            addu $t9, $t9, $a0              # Copy the base address of the array into $t9
            addu $t8, $t8, $a1              # Copy the $a1 reg. into the $t8 reg.
            li $s7, 4                       # 4-byte alignment
            mullo $t7, $a1, $s7             # Multiply with number of elements and store in $t7
            addu $t9, $t9, $t7              # 4 bytes per int * $t7 ints = Total length
```



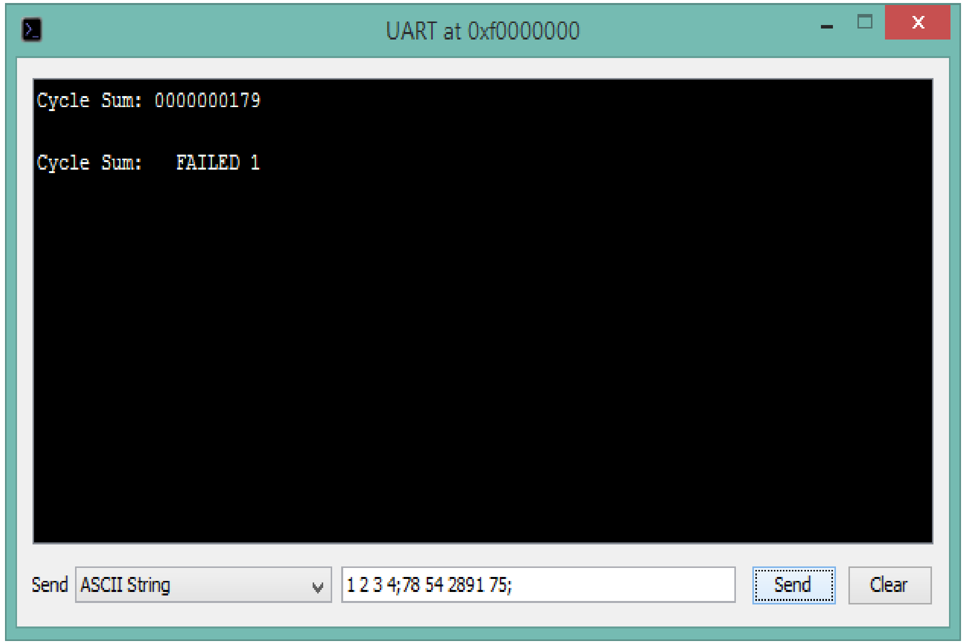
```
            li $t8, 0                       # Reuse $t8 as temporary register
            li $t7, 1                       # Reuse $t7, use as incrementer i
                                            # $s7 is still 4 now. Do not modify it
while_loop:
            beq  $a0, $t9, exit_while       # If $a0 = the end of array, jump to the end of the while loop
            nop
            slt $s3, $t7, $a1               # $s3 = ($t7 < $a1) -> [True or False] [1 or 0]
            beq $s3, $zero, exit_while      # branch if ! (i < elements), $s3 holds True or False
            nop                             # Otherwise...
BlockIf:
            # Implementing the if(i == 0 || array(i - 1) <= array(i)) statement
            bne $t7, $0, Compare            # if (i == 0), go to Increment
            nop                             # At this point i != 0
Compare:
            lw  $s5, 0($a0)                 # sets $s5 to the current element in array, (i - 1)
            lw  $s6, 4($a0)                 # sets $s6 to the next element in array, i
            beq $s5, $s6, Increment         # if (array(i-1) == array(i)), go to Increment
            nop                             #
            slt $s4, $s5, $s6               # $s4 = 1 if $s5 < $s6, or array(i-1) < array(i)
            bne $s4, $0, Increment          # if $s4 = 1, then i++
            nop
            #j UpdateArray                  # and then fall through
            #nop
ElseBlock:
            sw $s5, 4($a0)                  # temp = array(i)
            sw $s6, 0($a0)                  # array(i) = array(i-1)
            ###  Swap them  #################
```

Implementation



```
ElseBlock:
            sw $s5, 4($a0)                  # temp = array(i)
            sw $s6, 0($a0)                  # array(i) = array(i-1)
            ###  Swap them  #################
            j Decrement                     # Decrement i and $a0
            nop
            j UpdateArray                   # and then fall through
            nop
Increment:
            addiu $t7, $t7, 1               # Increment i
            mullo $s2, $t7, $s7             # Whatever $t7 is, multiply it by 4 and store it in $s2
            addu $a0, $a0, $s2              # advance the array as needed
            j while_loop                    # jump back to BlockIf
            nop
Decrement:
            addiu $t7, $t7, -1              # i = i -1, update i to a decremented value
            j while_loop                    # Back to ElseBlock
            nop
UpdateArray:
            mullo $s2, $t7, $s7             # Whatever $t7 is, multiply it by 4 and store it in $s2
            addu $a0, $a0, $s2              # advance the array as needed
            bne  $a0, $t9, while_loop       # If $a0 != the end of array, branch to while loop
            nop                             # Fall through
            j while_loop                    # jump back to top of the loop
            nop
exit_while:
            ############## Save previous values back before returning to caller #########################
```

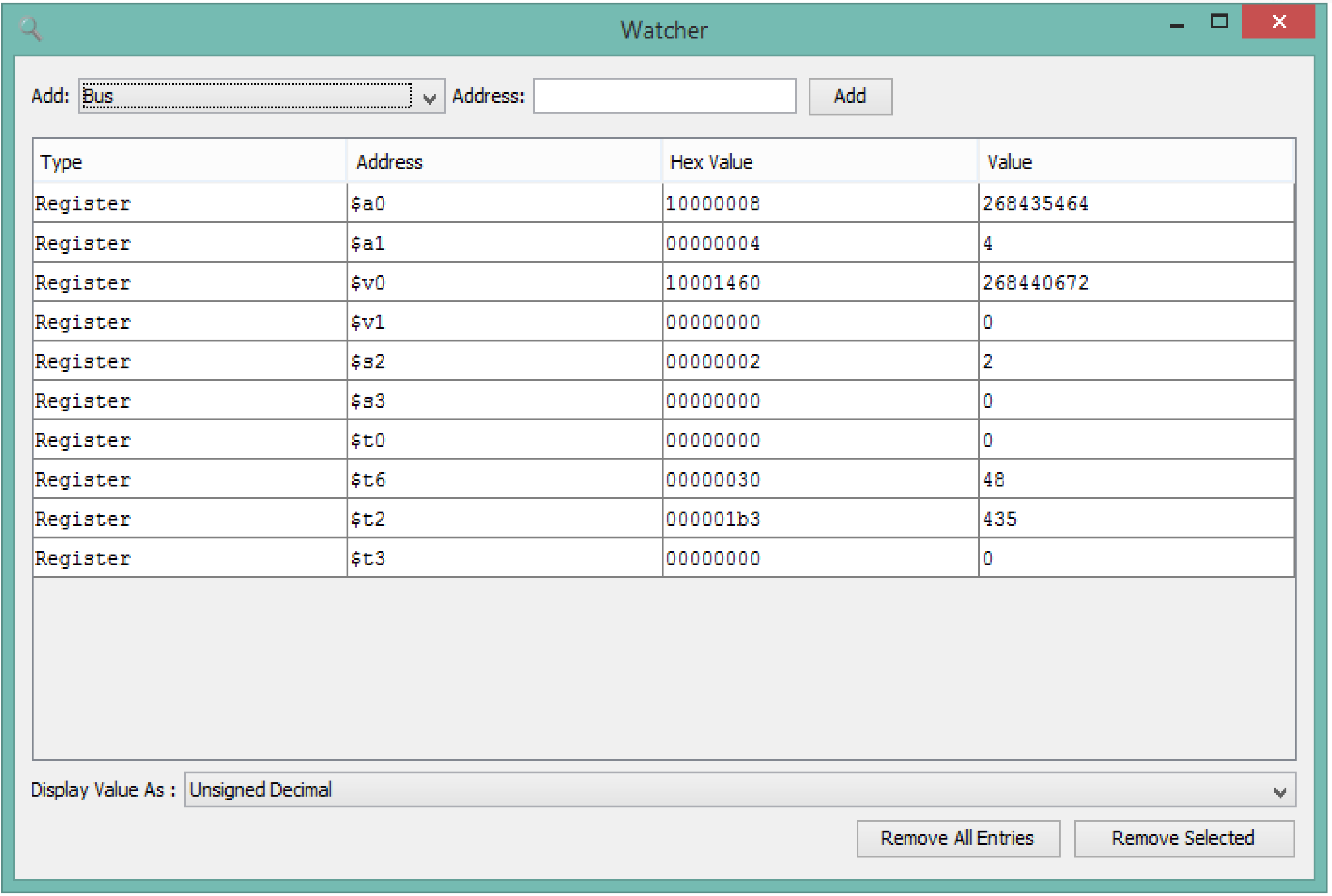
Results



UART at 0xf0000000

```
Cycle Sum: 0000000179

Cycle Sum:   FAILED 1
```

Send: ASCII String — 1 2 3 4;78 54 2891 75;



Watcher

Add: Bus Address:

Type	Address	Hex Value	Value
Register	$a0	10000008	268435464
Register	$a1	00000004	4
Register	$v0	10001460	268440672
Register	$v1	00000000	0
Register	$s2	00000002	2
Register	$s3	00000000	0
Register	$t0	00000000	0
Register	$t6	00000030	48
Register	$t2	000001b3	435
Register	$t3	00000000	0

Display Value As : Unsigned Decimal

Challenges

The only challenge I encountered during this exercise was converting the pseudo-code to the PLP MIPS assembly instruction flow. For example, converting the while loop statement such as follows:

```
while(I < elements) {
        [loop body]
}
```

required a new thought process in MIPS assembly instruction arrangement to break the mechanics of while loop into blocks represented by labels for jumping back and forth to mimic the actual while loop logic. Those who come from a high-level programming language domain will find how cumbersome the translation process is. However, it can be done. It took a little effort to think at low-level to get pass the challenges.

I can conclude that assembly language is very honest in its self-respecting place where it, too, obeys its Programmer’s instruction one mnemonic at a time.