CST 250
Project 3 Report

Team Members:

John Abbott

Chris Ijams

David McDonald

November 1, 2015

Introduction

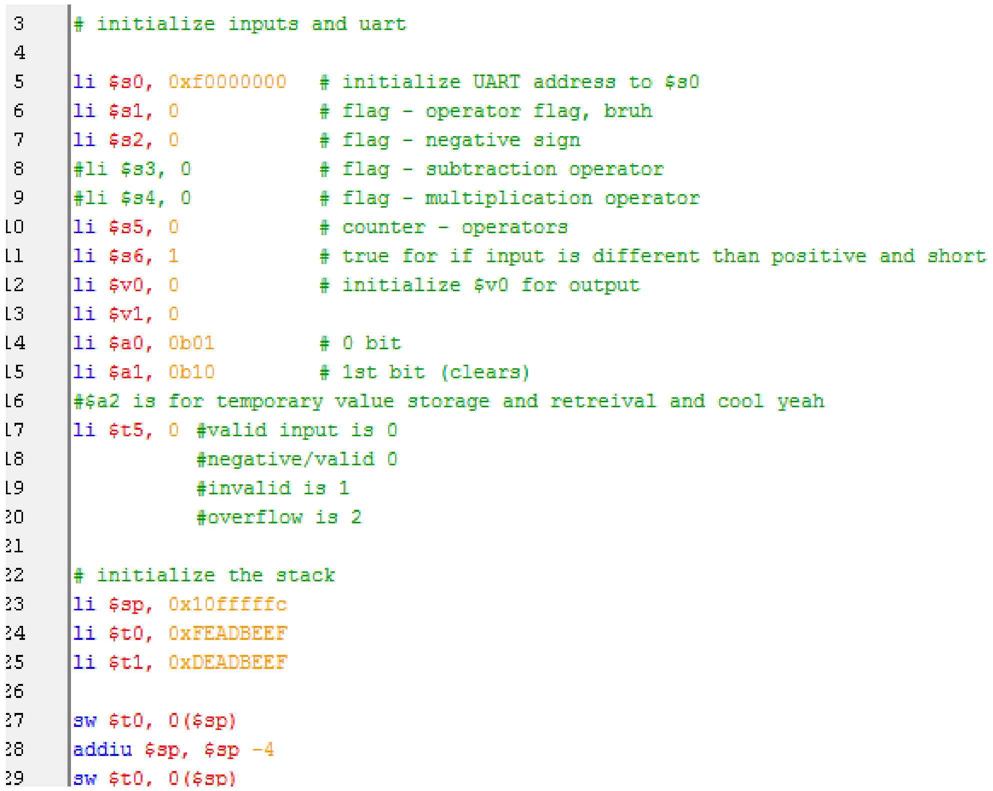
This is an implementation of a simple calculator that uses the UART for both input and output. It is capable of handling addition, subtraction and multiplication of both positive and negative numbers. It uses multiple checks for invalid characters, syntax and input and output overflow conditions and outputs appropriate error messages if any of these are encountered.

Project Approach

Our approach changed over the life of the project. We initially planned to write the entire project as a group. In the beginning, we met in person every other day or so to create a flow chart and discuss program logic. It was clear early on that David's command of PLP was far superior to the others and although we considered using Chris Mar's project 2 solution as a basis for this project, we eventually agreed that David's project two would be a better choice. Chris Ijams tried to help with solving coding issues but ultimately made few if any contributions that ended up in the final code. John contributed code such as the calculation functions and the stack initialization section, as well as making some adjustments such as using the $a2 register for temporary values instead of multiple dedicated registers as had been true of the original code as well as numerous tweaks to ensure or improve various functions. David and John made all the design decisions. David made the majority of coding contributions and John handled the majority of the administrative tasks including creating/maintaining the BitBucket repository and managing communications with team members and faculty. Chris Ijams was enthusiastic but wasn't able to contribute very much to the design or implementation. This division of responsibilities was not by design and shows an area for improvement for project leadership. It is clear that part of the design process must include a plan for the intelligent division of labor and responsibilities.

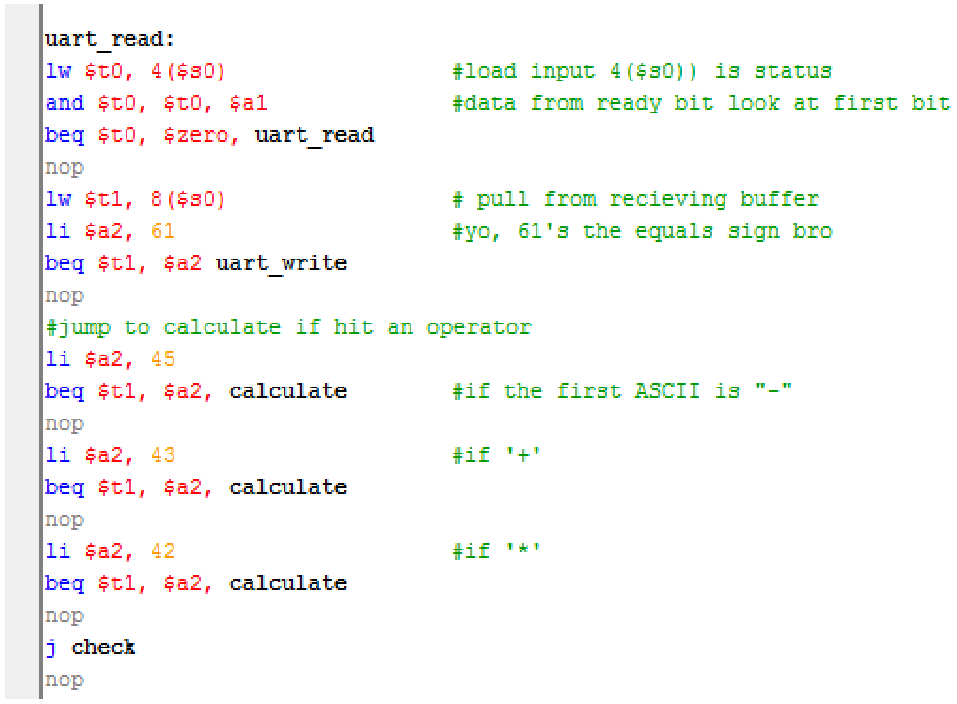
Program Description

The first part of the main initializes the inputs and the UART. The keep track of operators, we use a series of flags and an operator counter, as well as a flag for negative signs. We use $v1 and $v0 to store values and $v1 also stores a running total.



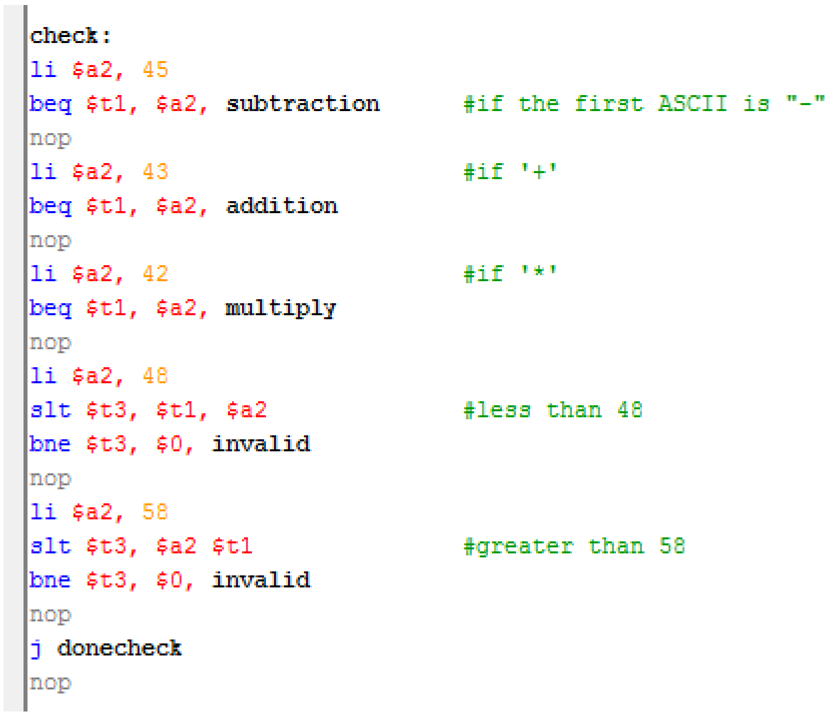
```
# initialize inputs and uart

li $s0, 0xf0000000   # initialize UART address to $s0
li $s1, 0            # flag - operator flag, bruh
li $s2, 0            # flag - negative sign
#li $s3, 0           # flag - subtraction operator
#li $s4, 0           # flag - multiplication operator
li $s5, 0            # counter - operators
li $s6, 1            # true for if input is different than positive and short
li $v0, 0            # initialize $v0 for output
li $v1, 0
li $a0, 0b01         # 0 bit
li $a1, 0b10         # 1st bit (clears)
#$a2 is for temporary value storage and retreival and cool yeah
li $t5, 0  #valid input is 0
           #negative/valid 0
           #invalid is 1
           #overflow is 2

# initialize the stack
li $sp, 0x10fffffc
li $t0, 0xFEADBEEF
li $t1, 0xDEADBEEF

sw $t0, 0($sp)
addiu $sp, $sp -4
sw $t0, 0($sp)
```

The next function reads values from the UART. It loops until input is received. If it receives the '=' sign, it branches to the write function. If it receives a '-', '+', or '*', it branches to the **calculate** function:



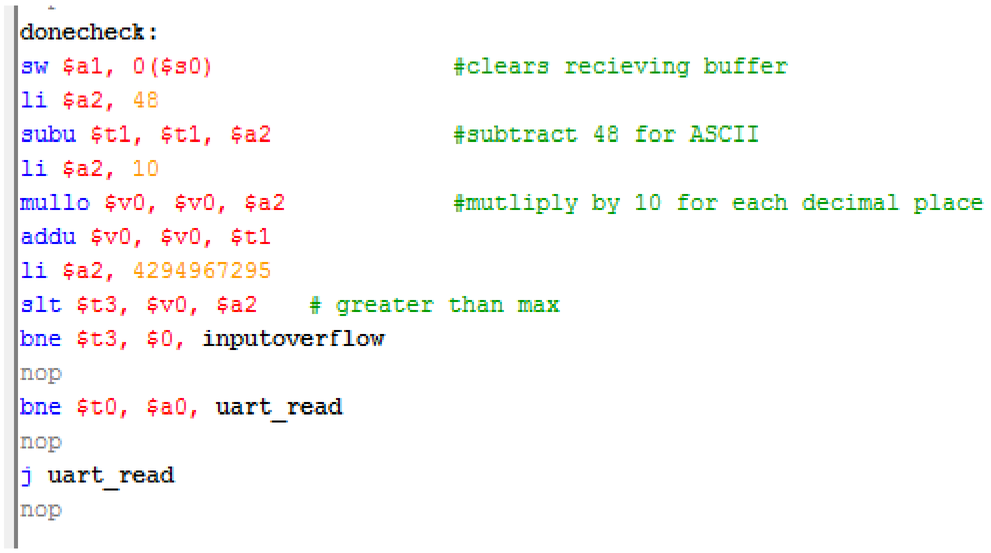
```
uart_read:
lw $t0, 4($s0)                #load input 4($s0)) is status
and $t0, $t0, $a1             #data from ready bit look at first bit
beq $t0, $zero, uart_read
nop
lw $t1, 8($s0)                # pull from recieving buffer
li $a2, 61                    #yo, 61's the equals sign bro
beq $t1, $a2 uart_write
nop
#jump to calculate if hit an operator
li $a2, 45
beq $t1, $a2, calculate       #if the first ASCII is "-"
nop
li $a2, 43                    #if '+'
beq $t1, $a2, calculate
nop
li $a2, 42                    #if '*'
beq $t1, $a2, calculate
nop
j check
nop
```

If it encounters anything else, it branches to the **check** function. Here it checks for '-', '+' and '*' and branches to **subtraction**, **addition** and **multiplication** functions in response. It then checks for characters above and below the range of the numerals in ASCII and branches to a function for handling **invalid** characters if any are encountered.



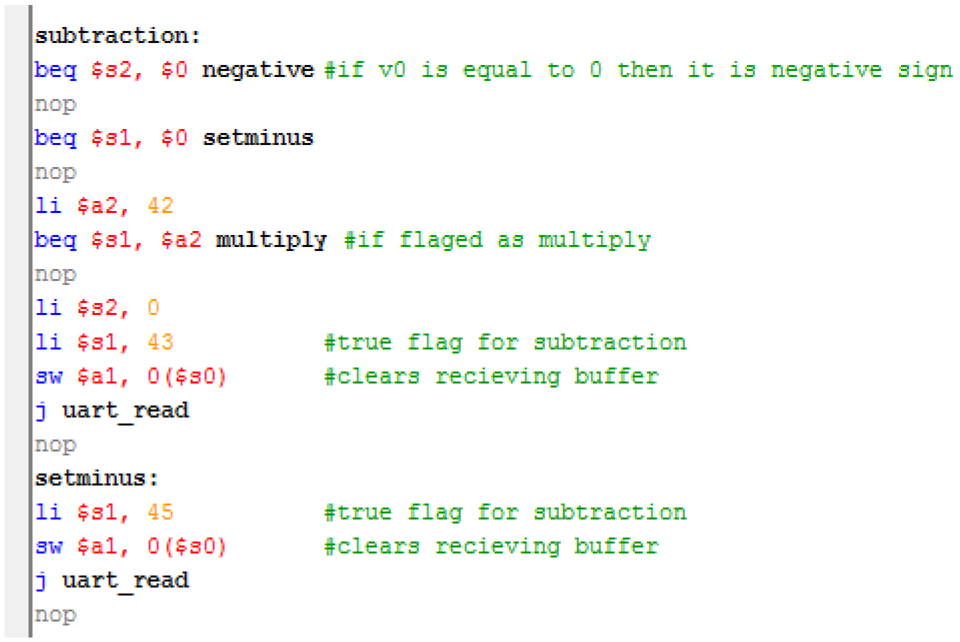
```
check:
li $a2, 45
beq $t1, $a2, subtraction      #if the first ASCII is "-"
nop
li $a2, 43                     #if '+'
beq $t1, $a2, addition
nop
li $a2, 42                     #if '*'
beq $t1, $a2, multiply
nop
li $a2, 48
slt $t3, $t1, $a2              #less than 48
bne $t3, $0, invalid
nop
li $a2, 58
slt $t3, $a2 $t1               #greater than 58
bne $t3, $0, invalid
nop
j donecheck
nop
```

It then continues to the **donecheck** function where numerical values are loaded into the $v0 register where previous values are shifted left and new values are added. It checks for input overflow and branches to that function if found. It otherwise returns to the **uart_read** function.



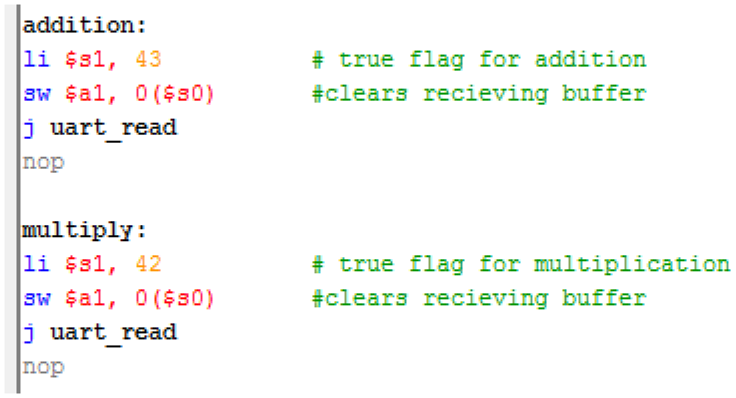
```
donecheck:
sw $a1, 0($s0)                 #clears recieving buffer
li $a2, 48
subu $t1, $t1, $a2             #subtract 48 for ASCII
li $a2, 10
mullo $v0, $v0, $a2            #mutliply by 10 for each decimal place
addu $v0, $v0, $t1
li $a2, 4294967295
slt $t3, $v0, $a2    # greater than max
bne $t3, $0, inputoverflow
nop
bne $t0, $a0, uart_read
nop
j uart_read
nop
```

The **subtraction** function is called in the **check** function when a '-' is received. It checks to see if the "negative flag" is set to 0, if so, it branches to the **negative** function where it checks the "multiply flag" and keeps the value if that flag is flipped. If the "operator flag" is zero, it branches to the **setminus** function where the operator is set to '-'. If the "operator flag" is set to 42, it branches to the **multiply** function.



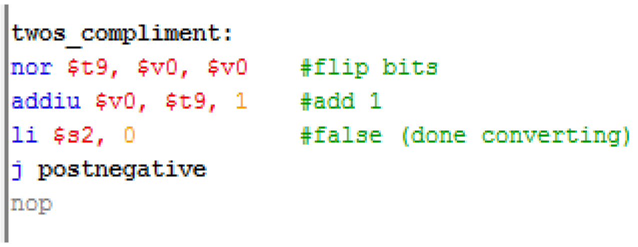
```
subtraction:
beq $s2, $0 negative #if v0 is equal to 0 then it is negative sign
nop
beq $s1, $0 setminus
nop
li $a2, 42
beq $s1, $a2 multiply #if flaged as multiply
nop
li $s2, 0
li $s1, 43          #true flag for subtraction
sw $a1, 0($s0)      #clears recieving buffer
j uart_read
nop
setminus:
li $s1, 45          #true flag for subtraction
sw $a1, 0($s0)      #clears recieving buffer
j uart_read
nop
```

The **addition** and **multiply** functions are similar. They set the "operator flag" to either 43 or 42, clear the receiving buffer and return to the **uart_read** function.



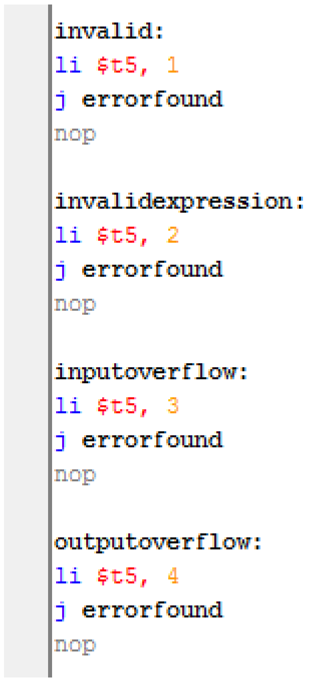
```
addition:
li $s1, 43          # true flag for addition
sw $a1, 0($s0)      #clears recieving buffer
j uart_read
nop

multiply:
li $s1, 42          # true flag for multiplication
sw $a1, 0($s0)      #clears recieving buffer
j uart_read
nop
```

The **twos_compliment** function converts the value in $v0 by flipping the bits using a nor, and adding 1. It then jumps to the **postnegative** function which branches to the appropriate calculation depending on which operator flag is flipped.



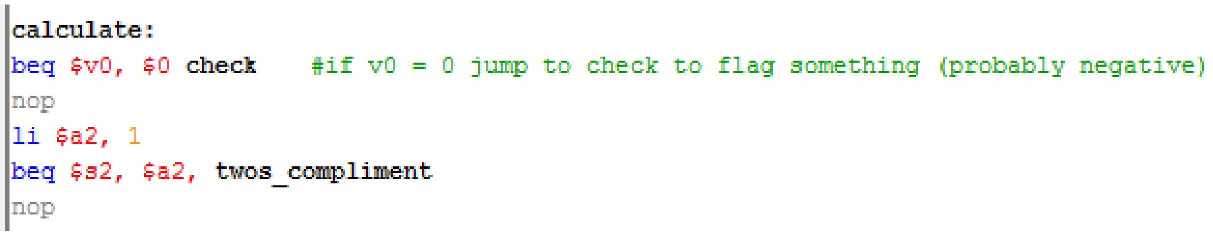
```
twos_compliment:
nor $t9, $v0, $v0     #flip bits
addiu $v0, $t9, 1     #add 1
li $s2, 0             #false (done converting)
j postnegative
nop
```

The **invalid**, **invalidexpression**, **inputoverflow** and **outputoverflow** functions all set the $t5 flag. This value is later passed to the uart asm file to trigger the output of the appropriate error message.



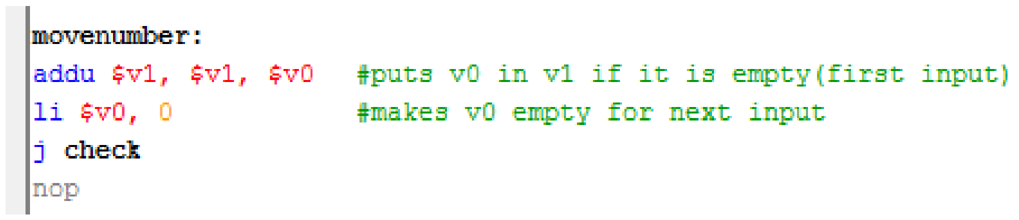
```
invalid:
li $t5, 1
j errorfound
nop

invalidexpression:
li $t5, 2
j errorfound
nop

inputoverflow:
li $t5, 3
j errorfound
nop

outputoverflow:
li $t5, 4
j errorfound
nop
```

The **calculate** function branches to the **check** function if $v0 hasn't received a value yet. If the "negative flag" is tripped, it branches to **twos_compliment**.



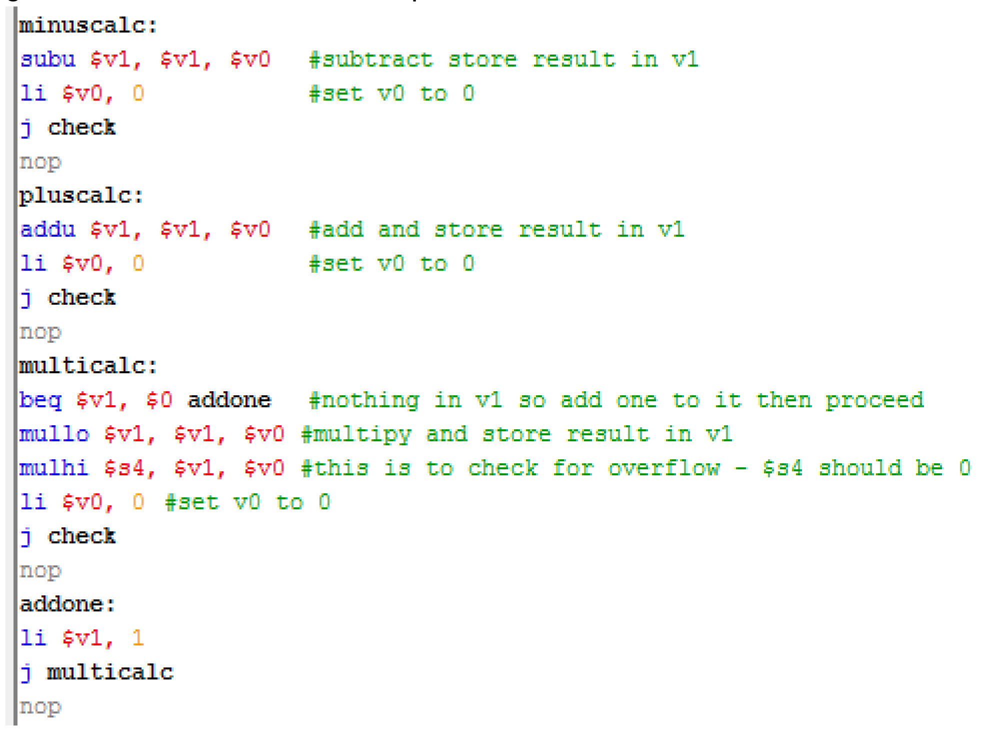
```
calculate:
beq $v0, $0 check    #if v0 = 0 jump to check to flag something (probably negative)
nop
li $a2, 1
beq $s2, $a2, twos_compliment
nop
```

The **movenumber** function moves a value from $v0 to $v1 if $v1 is empty.



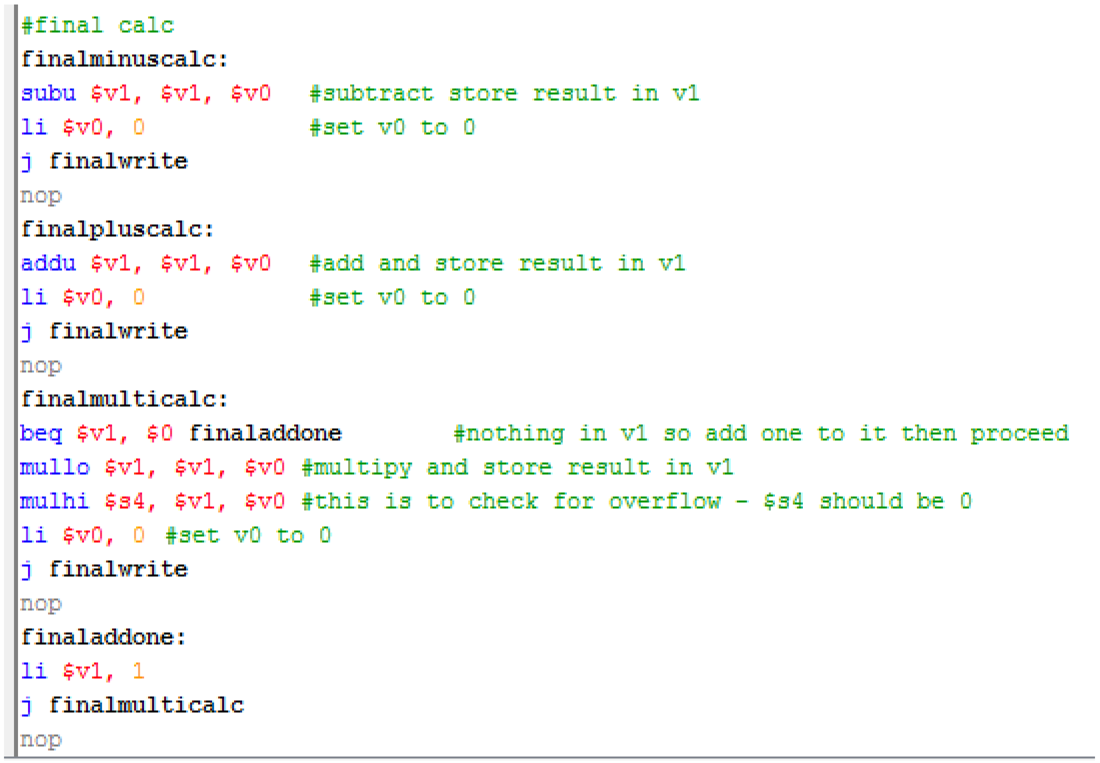
```
movenumber:
addu $v1, $v1, $v0   #puts v0 in v1 if it is empty(first input)
li $v0, 0            #makes v0 empty for next input
j check
nop
```

The calculators functions are fairly straight forward. Each one does as its title implies such as performing subtraction, additions or multiplication.



```
minuscalc:
subu $v1, $v1, $v0   #subtract store result in v1
li $v0, 0            #set v0 to 0
j check
nop
pluscalc:
addu $v1, $v1, $v0   #add and store result in v1
li $v0, 0            #set v0 to 0
j check
nop
multicalc:
beq $v1, $0 addone   #nothing in v1 so add one to it then proceed
mullo $v1, $v1, $v0 #multipy and store result in v1
mulhi $s4, $v1, $v0 #this is to check for overflow - $s4 should be 0
li $v0, 0  #set v0 to 0
j check
nop
addone:
li $v1, 1
j multicalc
nop
```

The “final” calculators are copies of the other calculators except that they include a call to finalwrite which sends the data to the uart asm to be written to the uart.



```
#final calc
finalminuscalc:
subu $v1, $v1, $v0   #subtract store result in v1
li $v0, 0            #set v0 to 0
j finalwrite
nop
finalpluscalc:
addu $v1, $v1, $v0   #add and store result in v1
li $v0, 0            #set v0 to 0
j finalwrite
nop
finalmulticalc:
beq $v1, $0 finaladdone        #nothing in v1 so add one to it then proceed
mullo $v1, $v1, $v0 #multipy and store result in v1
mulhi $s4, $v1, $v0 #this is to check for overflow - $s4 should be 0
li $v0, 0 #set v0 to 0
j finalwrite
nop
finaladdone:
li $v1, 1
j finalmulticalc
nop
```

Output functions are handled by the uart asm.

Lessons Learned

One of the main things we learned was that discussion of a problem with peers and teaching each other are surprisingly effective at solidifying one's own knowledge. We also quickly discovered that problems with seemingly simple solutions can be very difficult to implement in code and may require multiple iterations of design and implementation before they work as expected.

Face to face meetings would have been much better spent using some remote communication device if it had the ability to share the computer screen. Next time we will do better research in this area. We could have used a much better way to flow chart the design as well and ended up working from photographs of flow charts written on a whiteboard.

I would also have made a much quicker assessment of each team member's level of knowledge, abilities and motivations so that each person's strengths could be better capitalized on. Because this wasn't done well, a lot of time and resources were wasted.

Conclusion

This project was complicated and difficult. There were a lot of issues ended up in one of us finding a solution by accident or having to constantly ask for help from faculty. The reference material we needed to solve problems was not available or was very hard to find.

This project appears to really have been about how to approach a seemingly insurmountable problem. The most important takeaway from this is that the first stages of a project like this should be intensive design work that involves breaking down the problem into as small and as manageable pieces as possible so that each piece and be assigned to an owner and conquered without feeling the weight of the whole task.