MIPS Calculator

Introduction:

The calculator takes an input equation from the console and performs the given operations one by one. It supports eight operations in total.

Arithmetic:

- Addition
- Subtraction
- Multiplication
- Division

Logical:

- AND
- OR
- Left shift
- Right shift

The input values can be 32-bit signed integers. The calculator handles all exceptions effectively. It is capable of doing a maximum of seven operations with eight digits in a single equation.

Methodology:

On start, the calculator displays all the possible functions that can be performed and prompts the user to enter an equation that is stored as a string. Once saved, the equation can be processed to separate the operands and the operators. After the parsing process is complete, the operands and the operators are saved into arrays from where they can be fetched to perform calculations. The calculator picks up two operands from the memory and an operand, performs the operation and uses the result as an operand for the next calculation. Once all the operations have been processed, the results are displayed. Once again, the operands and the operators are fetched from the memory and sent to the console to form the original equation and then the final result is displayed along with it.

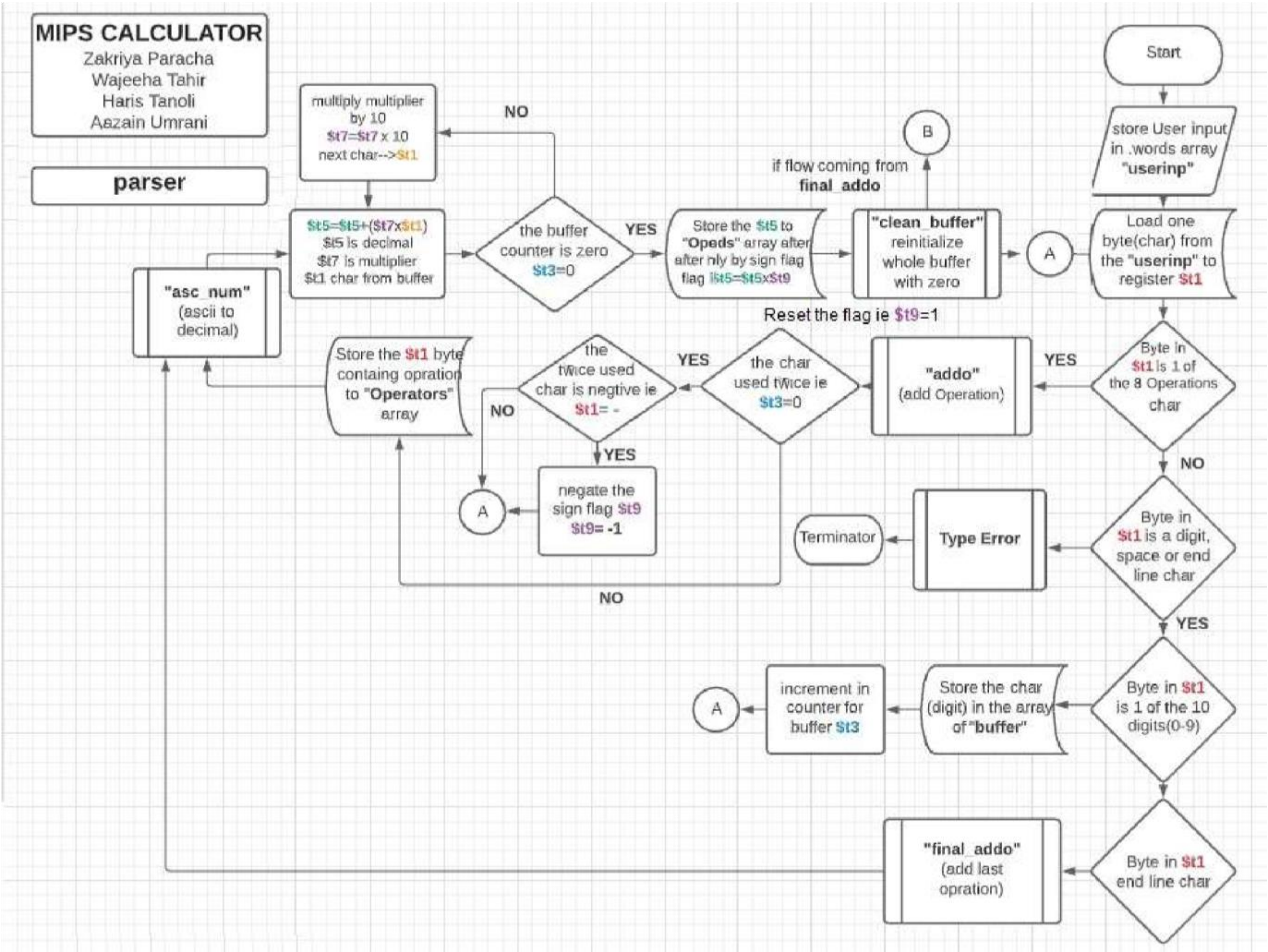
Concepts used:

- Functions
- Arrays
- Loops
- Error handling
- Data parsing and Extraction

Description:

Flow charts:

Parsing:

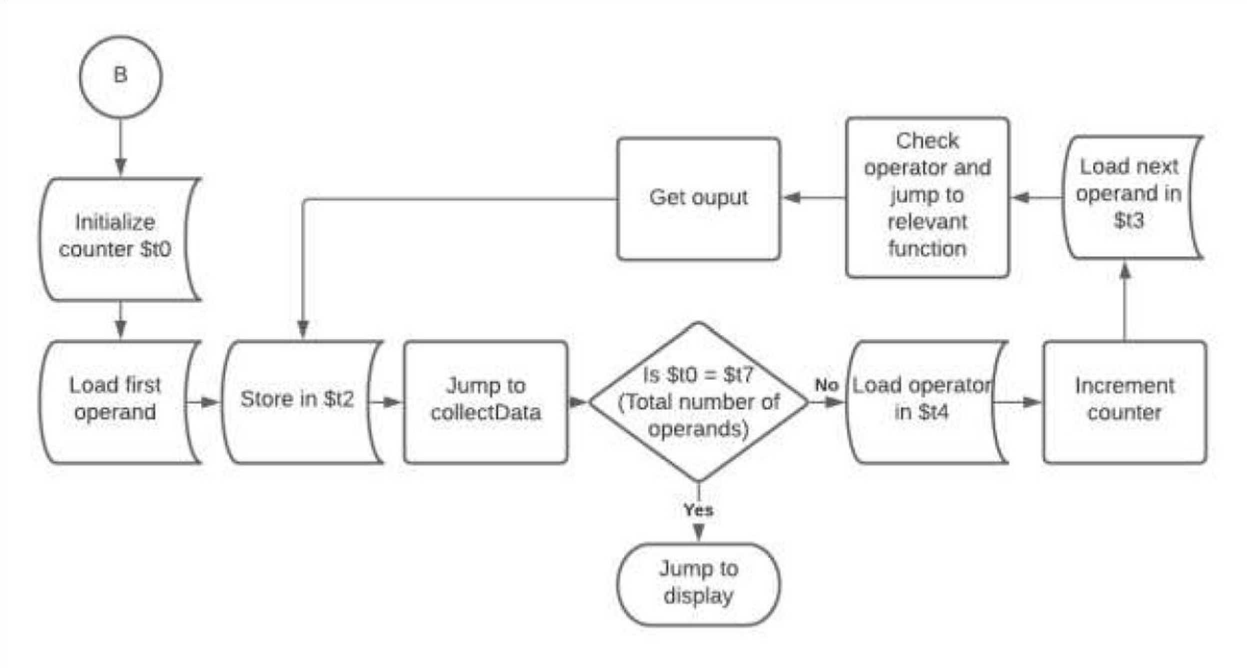

MIPS CALCULATOR
Zakriya Paracha
Wajeeha Tahir
Haris Tanoli
Aazain Umrani
parser
Start
store User input in .words array "userinp"
Load one byte(char) from the "userinp" to register $t1
Byte in $t1 is 1 of the 8 Operations char
YES
"addo" (add Operation)
the char used twice ie $t3=0
YES
the twice used char is negtive ie $t1= -
YES
negate the sign flag $t9 $t9= -1
A
NO
A
NO
Store the $t1 byte containg opration to "Operators" array
"asc_num" (ascii to decimal)
$t5=$t5+($t7x$t1) $t5 is decimal $t7 is multiplier $t1 char from buffer
the buffer counter is zero $t3=0
NO
multiply multiplier by 10 $t7=$t7 x 10 next char-->$t1
YES
Store the $t5 to "Opeds" array after after nly by sign flag flag i$t5=$t5x$t9
"clean_buffer" reinitialize whole buffer with zero
Reset the flag ie $t9=1
if flow coming from final_addo
B
A
NO
Byte in $t1 is a digit, space or end line char
Type Error
Terminator
YES
Byte in $t1 is 1 of the 10 digits(0-9)
Store the char (digit) in the array of "buffer"
increment in counter for buffer $t3
A
Byte in $t1 end line char
"final_addo" (add last opration)

A byte is loaded from the input string, then it’s evaluated on the following bases:

1. If it’s a digit then store it in the buffer, iterate the buffer and return to the main function to fetch the next char.
2. If it’s a character then check if it appears twice, if it’s twice skip the second operator i.e. <<, >>, then negate the **$t9 flag** if there is a negative sign i.e. negative number. Store the operators in ***operators*** array, convert char digits from buffer to integers and store in **$t5,** store the integer in ***operands*** array, clean the buffer and fetch the next char.

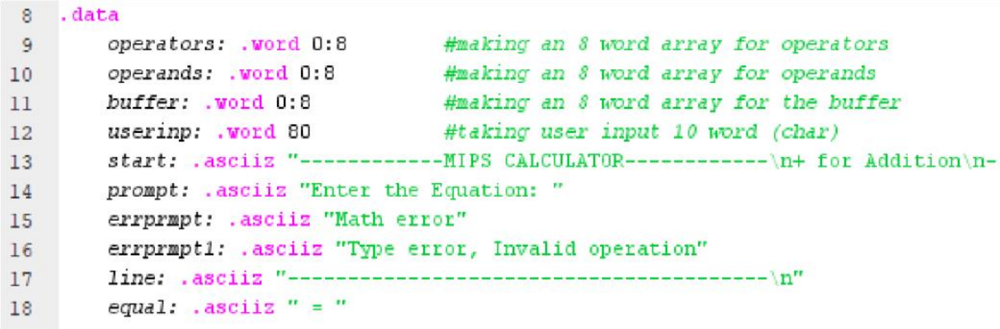
3. If it’s the end line character, store the operand in its array, clean the buffer, clean registers and go to **B** where the values are evaluated.
4. If the character is none of the above mentioned, call the Type Error and End the code.

Calculations:



.data:

1. “userinp” is the input string
2. “operators” is the array which stores the ascii for the operators i.e. (+, -, >>, <<)
3. “operands” is the array which stores the numbers which are being operated on.
4. **“**buffer**”** array is a temporary array which stores the digits while parsing



```
.data
    operators: .word 0:8        #making an 8 word array for operators
    operands: .word 0:8         #making an 8 word array for operands
    buffer: .word 0:8           #making an 8 word array for the buffer
    userinp: .word 80           #taking user input 10 word (char)
    start: .asciiz "------------MIPS CALCULATOR------------\n+ for Addition\n-
    prompt: .asciiz "Enter the Equation: "
    errprmpt: .asciiz "Math error"
    errprmpt1: .asciiz "Type error, Invalid operation"
    line: .asciiz "----------------------------------------\n"
    equal: .asciiz " = "
```

Procedures:

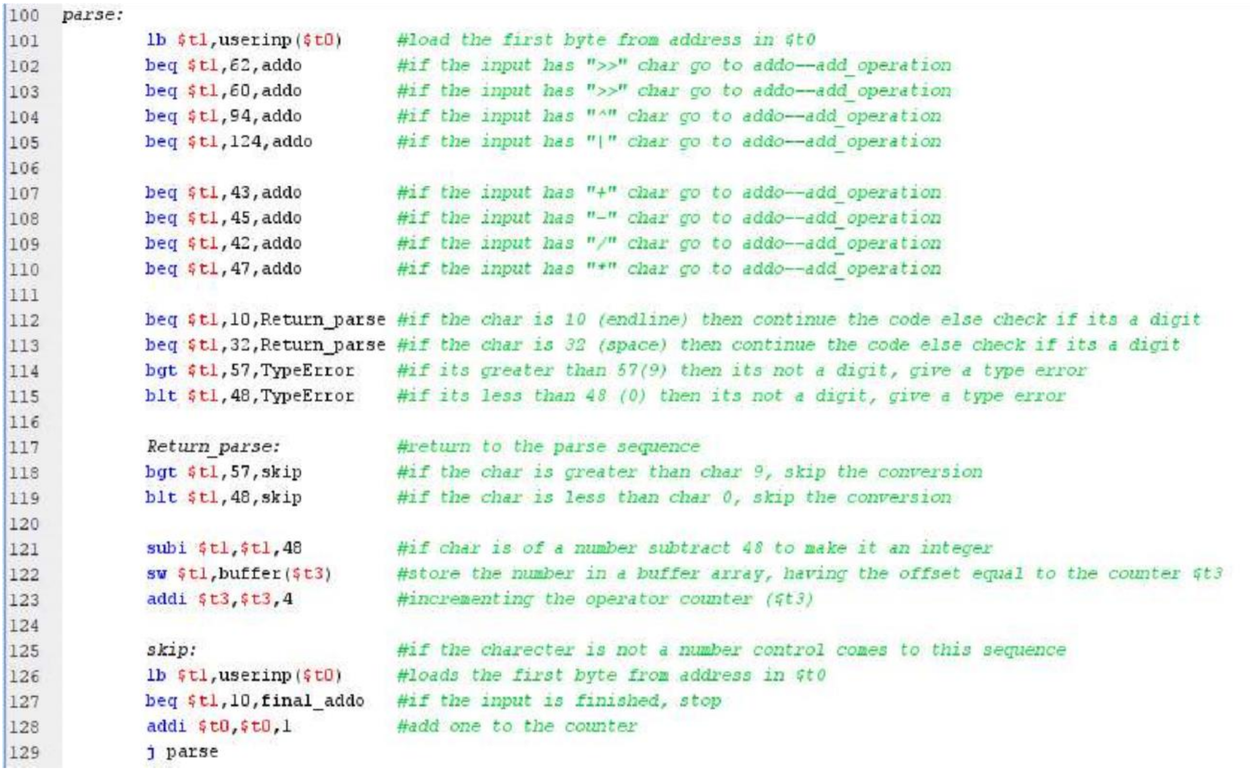
- **main:**

main is the first procedure of the project, it welcomes the user, and prompts them to add the equation.

- **parse:**

The first operation which is called after inputting the string is the *userinp* array, it loads one byte (char) in register *$t1* then sends the control to the *addo* procedure which stores the operator and operands in their own arrays.

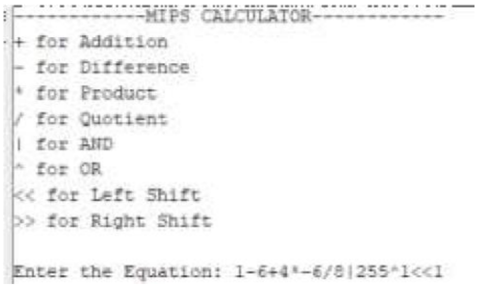
If the char is not an operation, check for the error, if it’s a digit then store it in the buffer and if it’s an end char then go to the *final_addo* which adds the operators and operands and goes back to the *main* function.



```
parse:
        lb $t1,userinp($t0)         #load the first byte from address in $t0
        beq $t1,62,addo             #if the input has ">>" char go to addo--add_operation
        beq $t1,60,addo             #if the input has ">>" char go to addo--add_operation
        beq $t1,94,addo             #if the input has "^" char go to addo--add_operation
        beq $t1,124,addo            #if the input has "|" char go to addo--add_operation

        beq $t1,43,addo             #if the input has "+" char go to addo--add_operation
        beq $t1,45,addo             #if the input has "-" char go to addo--add_operation
        beq $t1,42,addo             #if the input has "/" char go to addo--add_operation
        beq $t1,47,addo             #if the input has "*" char go to addo--add_operation

        beq $t1,10,Return_parse #if the char is 10 (endline) then continue the code else check if its a digit
        beq $t1,32,Return_parse #if the char is 32 (space) then continue the code else check if its a digit
        bgt $t1,57,TypeError        #if its greater than 57(9) then its not a digit, give a type error
        blt $t1,48,TypeError        #if its less than 48 (0) then its not a digit, give a type error

        Return_parse:               #return to the parse sequence
        bgt $t1,57,skip             #if the char is greater than char 9, skip the conversion
        blt $t1,48,skip             #if the char is less than char 0, skip the conversion

        subi $t1,$t1,48             #if char is of a number subtract 48 to make it an integer
        sw $t1,buffer($t3)          #store the number in a buffer array, having the offset equal to the counter $t3
        addi $t3,$t3,4              #incrementing the operator counter ($t3)

        skip:                       #if the charecter is not a number control comes to this sequence
        lb $t1,userinp($t0)         #loads the first byte from address in $t0
        beq $t1,10,final_addo       #if the input is finished, stop
        addi $t0,$t0,1              #add one to the counter
        j parse
```

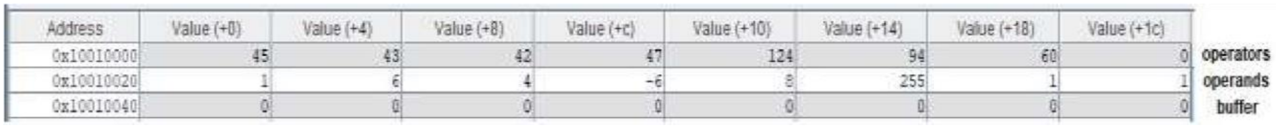
Example:

Input:



```
-------------MIPS CALCULATOR-------------
+ for Addition
- for Difference
* for Product
/ for Quotient
| for AND
^ for OR
<< for Left Shift
>> for Right Shift

Enter the Equation: 1-6+4*-6/8|255^1<<1
```

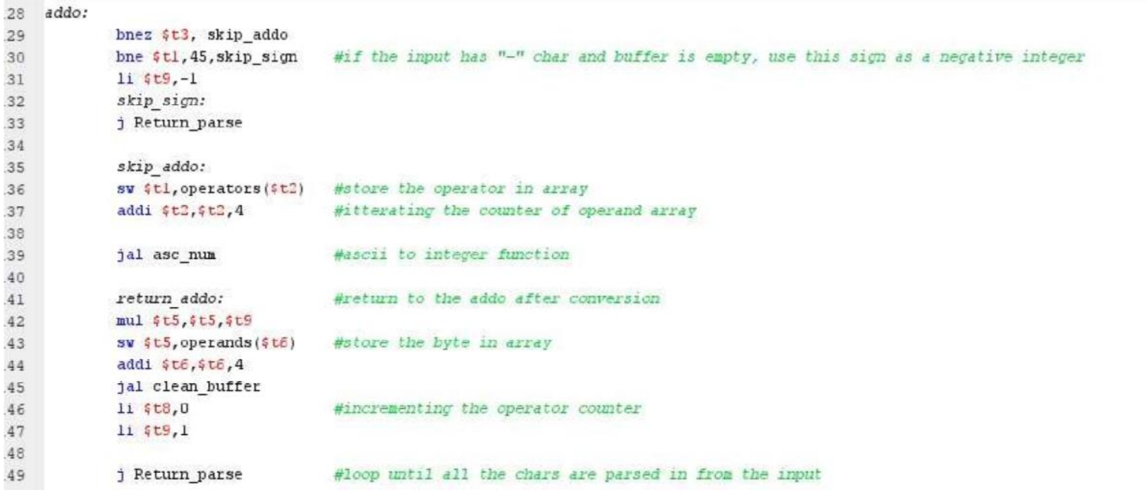
Data Segment: (After Parsing)



Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)	
0x10010000	45	43	42	47	124	94	60	0	operators
0x10010020	1	6	4	-6	8	255	1	1	operands
0x10010040	0	0	0	0	0	0	0	0	buffer

- **addo:**

This function checks for the multi-operators i.e. <<, >>, - and skips the second operator storage, in case of a negative number this loads -1 in sign flag *$t9*. Then it stores the operator's ascii in the array, then it calculates the number using the *asc_num* procedure which converts the digits in the buffer into a number and finally we store it in the operands array. Then we clean the buffer using *clean_buffer* by populating it by 0, and then we return to *parse* to process the next number.

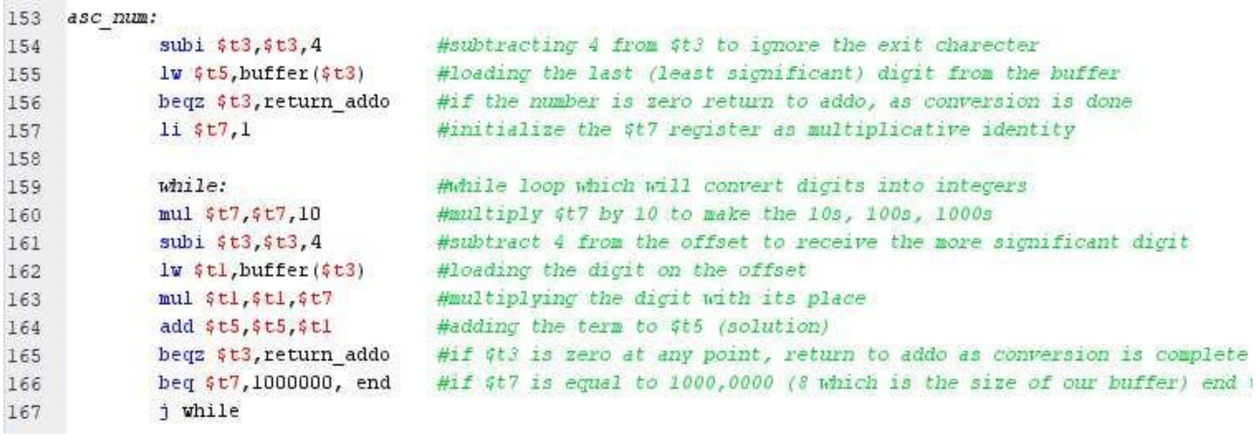


```
addo:
        bnez $t3, skip_addo
        bne $t1,45,skip_sign    #if the input has "-" char and buffer is empty, use this sign as a negative integer
        li $t9,-1
        skip_sign:
        j Return_parse

        skip_addo:
        sw $t1,operators($t2)   #store the operator in array
        addi $t2,$t2,4          #itterating the counter of operand array

        jal asc_num             #ascii to integer function

        return_addo:            #return to the addo after conversion
        mul $t5,$t5,$t9
        sw $t5,operands($t6)    #store the byte in array
        addi $t6,$t6,4
        jal clean_buffer
        li $t8,0                #incrementing the operator counter
        li $t9,1

        j Return_parse          #loop until all the chars are parsed in from the input
```

- **asc_num:**

Converts the digits in the buffer into a single number by multiplying them with the multiplier *$t7* and it uses the buffer counter *$t3*, in reverse to load the digits.

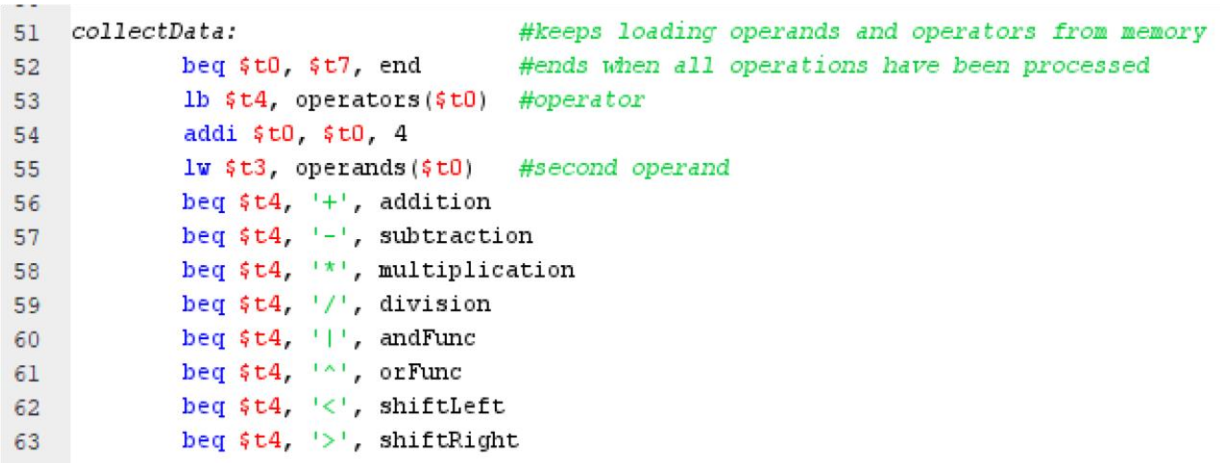
If the $t3 is zero, the number is converted, jump back to the *addo.*



```
asc_num:
        subi $t3,$t3,4              #subtracting 4 from $t3 to ignore the exit charecter
        lw $t5,buffer($t3)          #loading the last (least significant) digit from the buffer
        beqz $t3,return_addo        #if the number is zero return to addo, as conversion is done
        li $t7,1                    #initialize the $t7 register as multiplicative identity

        while:                      #while loop which will convert digits into integers
        mul $t7,$t7,10              #multiply $t7 by 10 to make the 10s, 100s, 1000s
        subi $t3,$t3,4              #subtract 4 from the offset to receive the more significant digit
        lw $t1,buffer($t3)          #loading the digit on the offset
        mul $t1,$t1,$t7             #multiplying the digit with its place
        add $t5,$t5,$t1             #adding the term to $t5 (solution)
        beqz $t3,return_addo        #if $t3 is zero at any point, return to addo as conversion is complete
        beq $t7,1000000, end        #if $t7 is equal to 1000,0000 (8 which is the size of our buffer) end
        j while
```

- **collectData:**

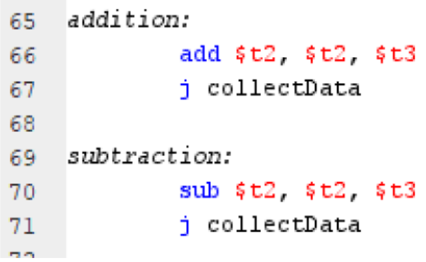
Before calling the function, the first operand is loaded into the $t2 register and the counter $t0 is initialized with zero. The function runs in a loop until all of the operands have been processed. An operator in the form of a character is loaded into the $t4 register and the counter is incremented. The second operand is loaded into $t3. The operator is matched with the given cases and the control jumps to the relevant function. After the processing has been completed, the control is transferred back to the collectData function and now the operands and operators at the next offset value are loaded into the registers. This loop continues until the given number of operands, stored in $t7 have been processed. Once complete, the control shifts to the end function which returns us back to the main function using the return address stored in $ra.



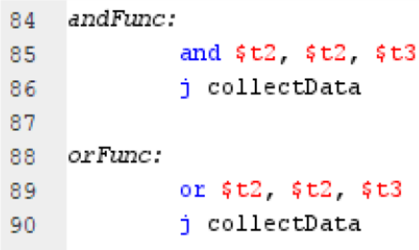
```
collectData:                    #keeps loading operands and operators from memory
        beq $t0, $t7, end       #ends when all operations have been processed
        lb $t4, operators($t0)  #operator
        addi $t0, $t0, 4
        lw $t3, operands($t0)   #second operand
        beq $t4, '+', addition
        beq $t4, '-', subtraction
        beq $t4, '*', multiplication
        beq $t4, '/', division
        beq $t4, '|', andFunc
        beq $t4, '^', orFunc
        beq $t4, '<', shiftLeft
        beq $t4, '>', shiftRight
```

- **addition, subtraction, andFunc, orFunc:**

When control is transferred to any of these functions, they perform addition, subtraction, AND or OR on the registers $t2 and $t3. The result is stored into the $t2 register and it acts as the first operand for the next operation. The control is transferred back to the controlData function.

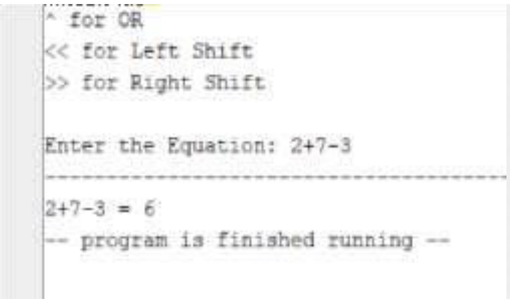


```
addition:
        add $t2, $t2, $t3
        j collectData

subtraction:
        sub $t2, $t2, $t3
        j collectData
```

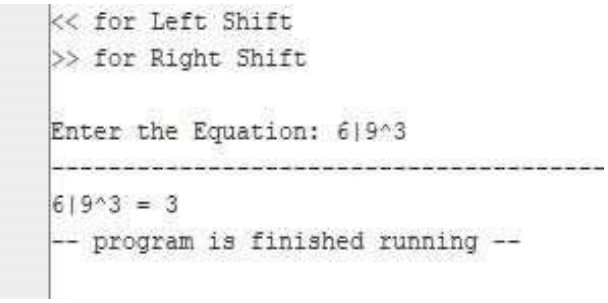


```
andFunc:
        and $t2, $t2, $t3
        j collectData

orFunc:
        or $t2, $t2, $t3
        j collectData
```

Case Output: i.e. 101(6) | 1001(9) ❼ 0001(1) ^ 0011(3) ❼ 0011(3)



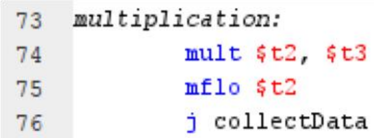
```
^ for OR
<< for Left Shift
>> for Right Shift

Enter the Equation: 2+7-3
----------------------------------------
2+7-3 = 6
-- program is finished running --
```



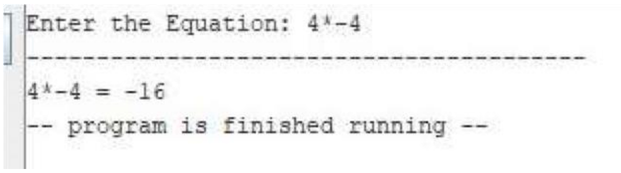
```
<< for Left Shift
>> for Right Shift

Enter the Equation: 6|9^3
----------------------------------------
6|9^3 = 3
-- program is finished running --
```

• Multiplication:

Multiplication is performed on the values loaded in $t2 and $t3 and the lower 32-bits are saved into $t2.



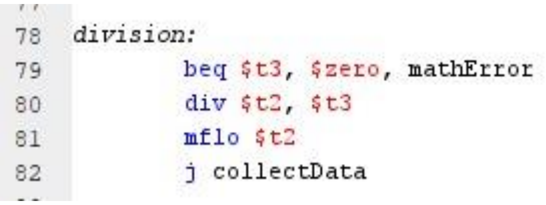
```
multiplication:
        mult $t2, $t3
        mflo $t2
        j collectData
```



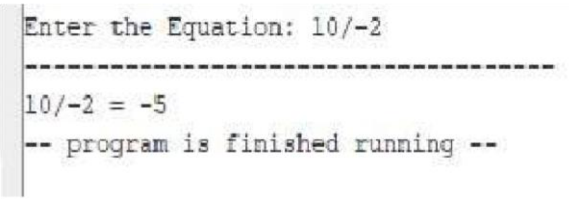
```
Enter the Equation: 4*-4
----------------------------------------
4*-4 = -16
-- program is finished running --
```

• Division:

The function first checks if the denominator, stored in $t3, is zero, if true, it jumps to a function to display a “Math Error” and ends the program. If the denominator is not zero and division is possible, $t2 is divided by $t3 and the answer is stored in $t2.



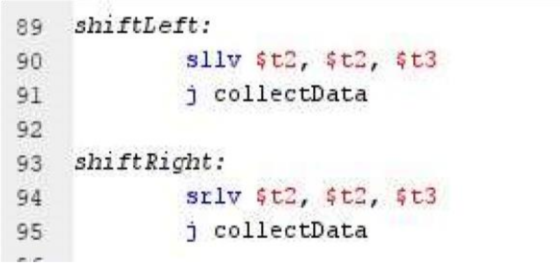
```
division:
        beq $t3, $zero, mathError
        div $t2, $t3
        mflo $t2
        j collectData
```



```
Enter the Equation: 10/-2
----------------------------------------
10/-2 = -5
-- program is finished running --
```

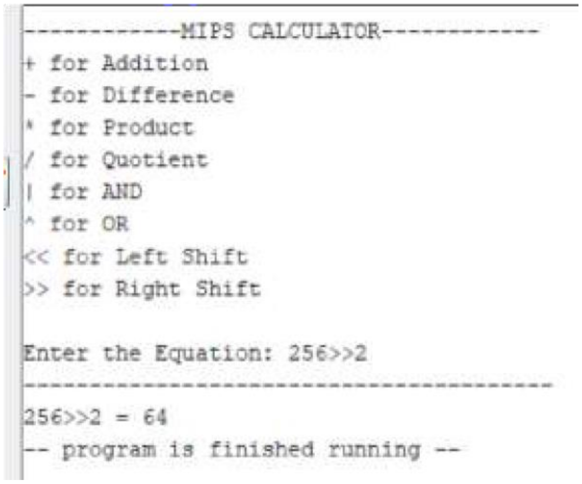
• shiftLeft, shiftRight:

The value stored in $t2 is shifted logically left or right by $t3 bits and the result is stored in $t2. The sll**v** or srl**v** commands were used as the shift amounts were stored in a variable i.e. register.

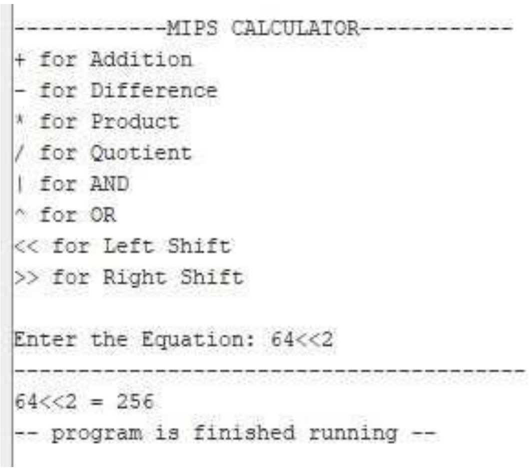


```
shiftLeft:
        sllv $t2, $t2, $t3
        j collectData

shiftRight:
        srlv $t2, $t2, $t3
        j collectData
```

Case Output:



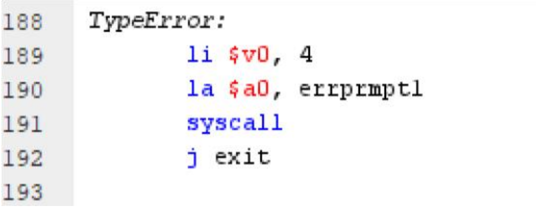
```
------------MIPS CALCULATOR------------
+ for Addition
- for Difference
* for Product
/ for Quotient
| for AND
^ for OR
<< for Left Shift
>> for Right Shift

Enter the Equation: 256>>2
----------------------------------------
256>>2 = 64
-- program is finished running --
```



```
------------MIPS CALCULATOR------------
+ for Addition
- for Difference
* for Product
/ for Quotient
| for AND
^ for OR
<< for Left Shift
>> for Right Shift

Enter the Equation: 64<<2
----------------------------------------
64<<2 = 256
-- program is finished running --
```

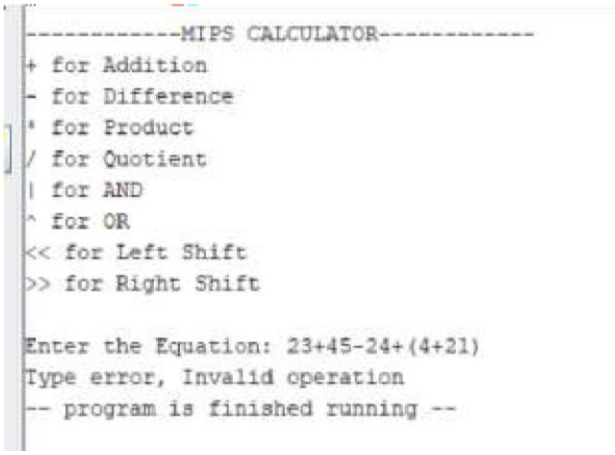
- **TypeError:**

When any of the operators being input is not one of the given 8 operators, an error is prompted.



```
TypeError:
        li $v0, 4
        la $a0, errprmpt1
        syscall
        j exit

```

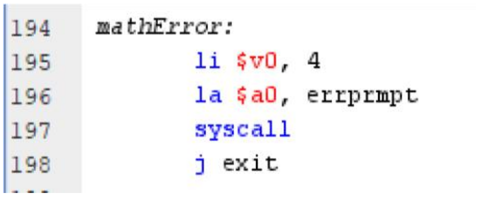
Case Output:



```
------------MIPS CALCULATOR------------
+ for Addition
- for Difference
* for Product
/ for Quotient
| for AND
^ for OR
<< for Left Shift
>> for Right Shift

Enter the Equation: 23+45-24+(4+21)
Type error, Invalid operation
-- program is finished running --
```

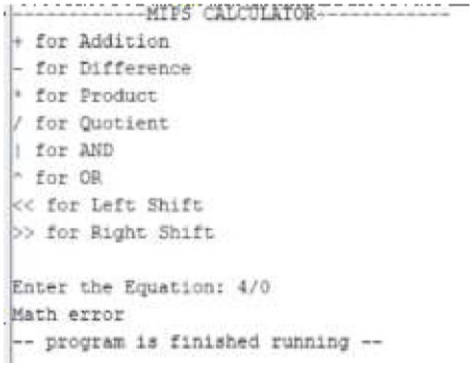
- **mathError:**

When division by zero is detected, the calculator throws an error and ends the program.



```
mathError:
        li $v0, 4
        la $a0, errprmpt
        syscall
        j exit
```

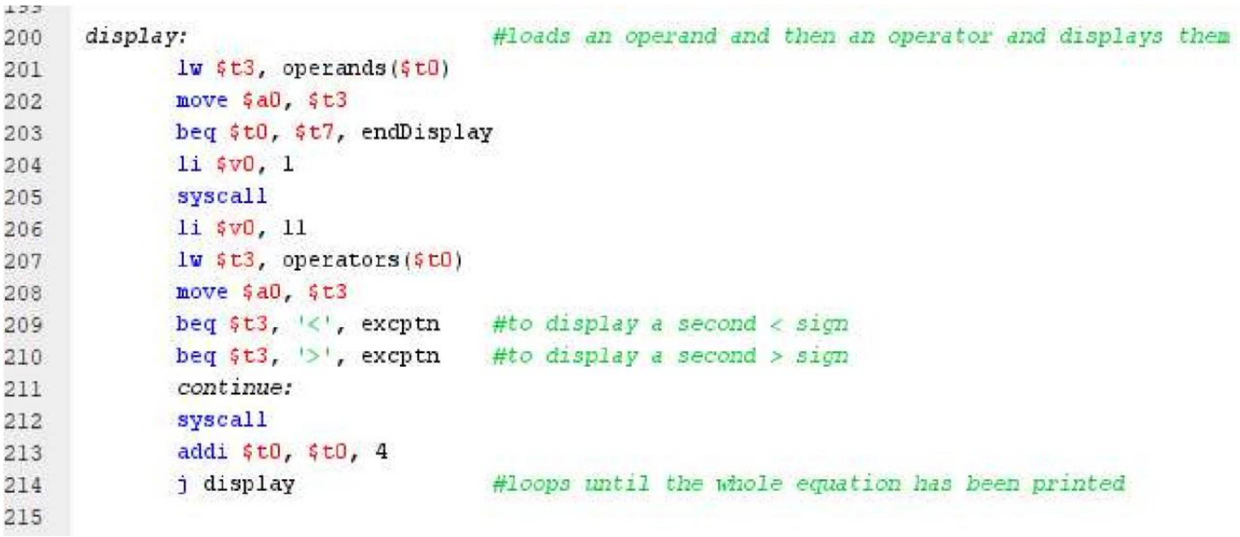
Case Output:



```
------------MIPS CALCULATOR------------
+ for Addition
- for Difference
* for Product
/ for Quotient
| for AND
^ for OR
<< for Left Shift
>> for Right Shift

Enter the Equation: 4/0
Math error
-- program is finished running --
```

- **display:**

The operands and operators are loaded into $t3. The value in $t3 is passed as an argument in $a0 and then a system call is issued with 1 or 11 in $v0 for displaying numbers or characters respectively. The loop continues until the entire equation is printed. Only one character from the symbols of left or right shift are stored in the array hence when a ‘<’ or ‘>’ is encountered in the operators, to maintain uniformity with the original equation, they are printed twice using the **excptn** function.



```
display:                       #loads an operand and then an operator and displays them
       lw $t3, operands($t0)
       move $a0, $t3
       beq $t0, $t7, endDisplay
       li $v0, 1
       syscall
       li $v0, 11
       lw $t3, operators($t0)
       move $a0, $t3
       beq $t3, '<', excptn    #to display a second < sign
       beq $t3, '>', excptn    #to display a second > sign
       continue:
       syscall
       addi $t0, $t0, 4
       j display               #loops until the whole equation has been printed
```

- **endDisplay**:

Once the equation is printed, an equal sign is printed to the front of it and then the final answer that was saved in $t2. The control then returns back to main and ends the program.

- **end**:

Jumps to the address stored in $ra by the jal instruction.

- **exit:**

Issues a system call with 10 in $v0 to end the program.

Features:

- Can solve maximum 8 digits, 8 operands in a single equation. And is scalable.
- Ability to read equation from console with or without spaces
- Negative integers can be used
- Prevents division by zero
- Handles exceptions effectively
- Displays accurate results