We'll be using regular calls and a "Conditional Call" called "ce" or **C**all if **E**qual, like jmps calls can use results from a compare (cmp).

```
…Sub:
 sub #Param1,#Param2
call Out
ret

Out:
 out 3,#Param1
ret
```

Now this program is pretty useful... but to be really useful as a general calculator it needs the basic 4 functions... so far we only have 2! So we need to "ADDZ MAOR FUNKSHUNS!" So let's "plug in" **mul** (Multiply) and **div** (Divide)!

Since the program is modularized we only need to add a pair of code bits that will do our bidding and then just tweak the code slightly.

```
…Add: //no change here!
 add #Param1,#Param2 //Remember math functions save result in 1st parameter of the instruction.
call out
ret

Sub: //Getting warmer!
 sub #Param1,#Param2
call out
ret

Mul: //add Mul with the code
 mul #Param1,#Param2
call Out
ret

Div: //add Div with its code!
 div #Param1,#Param2
call Out
ret

Out: //We don't need to adjust this one!
 out 3,#Param1 //since results are stored in Param1
ret  //we don't have to change anything here. Whoa!
```

Let's pretend that there are only 4 registers. In this program, we'll take inputs, report the states of the ports AND output an average.

```
…Output: //Remember, we still have a ret to do!
 pop edx //Remember the order that you pushed values onto the stack...
```

```
 pop ecx //when you pop them, they'll be in reverse order...
 pop ebx //so pay attention to that!
 pop eax //Now we have restored the register states!
 mov port0,eax //Output the port states
 mov port1,ebx
 mov port2,ecx
 mov port3,edx
 mov port4,#Avg //Output the results of the Averaging
 push #Savstk //Put the return address back on TOP of the stack!
ret //Bingo!
```