# The help section: a documentation of how to write calM’s assembly

This is the help section where you will find everything about how to write correct calM assembly so we will tackle many subjects such as: instructions, addressing modes, operands, and finally some routines you should follow when writing calM’s assembly on our emulator, then some errors to avoid.

As shown in the documentation we have 3 types of instructions, we cover the most important and most used ones :

* Zero operands instructions, which are: RET, PUSHA, POPA
  + RET:This instruction is used to return control from a subroutine to the calling routine. It pops the top value from the stack and sets the program counter to that value, effectively jumping back to the instruction that called the subroutine.
  + PUSHA:This instruction is used to push the value of the main 8 registers onto the stack
  + POPA:This instruction is used to pop values from the stack and store them in the main 8 registers .
* One operand instructions, where we find two types of instructions:
  + Instructions that can take only registers as operands: they are generally arithmetic and logic instructions operations that operate only one operand, let’s list them: NEG, NOT, SHL, SHR, READ, WRITE, PUSH, POP, ROR, ROL.
  + NEG : it used to negate the value of the operand “multiplying the value by -1”
  + NOT :is used to perform a bitwise logical NOT operation on the operand ,itinverts all the bits of the operand, so that each 0 becomes a 1 and each 1 becomes a 0.
  + SHL:This instruction is used to shift the bits of a register or a memory location to the left by a specified number of bits, with the bits shifted out on the left side discarded and 0's added on the right side.
  + SHR:the same like SHL but it shifts the bits to the right.
  + READ:This instruction is used to read a value from an input device or a memory location and store it in a register.
  + WRITE:This instruction is used to write the value of a register or a memory location to an output device or a memory location.
  + PUSH:This instruction is used to push the value of a register or a memory location onto the top of the stack.
  + POP:This instruction is used to pop the top value from the stack and store it in a register or a memory location.
  + ROR:This instruction is used to rotate the bits of a register or a memory location to the left by a specified number of bits, with the bits shifted out on the left side added on the right side.
  + ROL:This instruction is used to rotate the bits of a register or a memory location to the left by a specified number of bits, with the bits shifted out on the left side added on the right side.
  + Instructions that can only take addresses as operands( which means only one addressing mode is available it’s the direct mode for these instructions of course): they are branching instructions let’s list them: CALL, BE, BNE, BS, BI, BIE, BSE, BRI:
    - CALL:It transfers program control to a subroutine or a function, saving the return address so that the program can resume execution from where it left off after the subroutine completes its operation.
    - BE:It performs a conditional branch to a target address if the zero flag is set. The zero flag is typically set when the result of a previous arithmetic or logical operation is zero.
    - BNE:It performs a conditional branch to a target address if the zero flag is not set.
    - BS: It performs a conditional branch to a target address if the sign flag is set. The sign flag is typically set when the result of a previous arithmetic operation is negative.
    - BI:It performs a conditional branch to a target address if the value of the accumulator is negative (usually used after doing a SUB operation between two operands , here we branch if the first operand is lower than the second) .
    - BIE:It performs a conditional branch to a target address if the value of the accumulator is negative or zero (usually used after doing a SUB operation between two operands , here we branch if the first operand is lower than the second or equal ) .
    - BSE:It performs a conditional branch to a target address if the value of the accumulator is positive (usually used after doing a SUB operation between two operands , here we branch if the first operand is greater than the second or equal).
    - BRI: It performs an unconditional branch to a target address, without any condition or flag checking.
* Two operands instructions, which are: NAND, CMP, MOV, ADD, SUB, MUL, DIV, AND, OR, XOR, NOR:

* + NAND: NAND operation between the two operands and stores the result in the accumulator.
  + CMP:
  + MOV: Move a value from a memory location or source register to a destination register.
  + ADD: This instruction adds the values together and stores the result in the accumulator
  + SUB: subtract the first value from the second value and stores the result in the accumulator
  + MUL: multiply the two operands and stores the result in in the R4:ACC
  + DIV: divide the first value from the second value and store the quotient in the ACC register and the remainder in the R4 register. “Or in ACCR,ACCL”

If it’s a division in 8 bits.

* + AND: AND operation between the two operands.
  + OR: OR operation between the two operands.
  + XOR: XOR operation between the two operands.
  + NOR: NOR operation between the two operands.

The result of those two operations is stored in the ACC register.

Then let’s speak about operands:

In our language there are two types of operands registers and numbers, numbers can be representing addresses or only immediate values, let’s begin with the simplest:

* Registers: we have a total of 9 registers that can be used as operands with a size of 2 bytes each one:

• 4 general purpose registers R1, R2, R3, R4, three of them are divided into two parts high and low ( Most significant byte is high and least significant byte is low they are used mainly in one byte operations when we have operands in one byte of memory ), which are: R1H, R2H, R3H, R1L, R2L, R3L( those can be used also as operands in one byte operations as it’s already mentioned).

• 2 used in the based and Indexed addressing modes: BR, IDR.

• 1 stack register : SR.

• 1 instruction Register: IDR.

• 1 accumulator register: ACC.

* Then numbers when you write numbers in calM’s assembly the number is considered as decimal and as a standard for our machine and language we restrict a max value which is 32 767 and the min value is -32 767 ( which is the biggest numbers you can represent in 2 bytes in binary), then you have the ability to write ADD R1, -5, so the representation of negative numbers is valid, this is it for numbers as immediate operands, for other addressing modes:
  + Direct mode: you just have to add an Asterisk \* after the number to say to the assembler this number has to be considered as an address and I’m doing an operation with direct addressing mode, Ex: MOV R1,10\*

Ps: You can perform an inverted addressing by writing negative addresses, like MOV -1\*,10 this moves 10 to the last memory address.

* + Indirect mode: to keep it simple for you, just add another Asterisk \*

Example: MOV R1,10\*\* , here you are operating on R1 and 10 with an indirect addressing mode

Let’s explain the next ones with examples directly

* Based: For Ex: MOV R1, BR\*+10

What this instruction does is: it takes the content of BR and adds to it 10 then the value we get is an address that we operate on, like this is how based addressing mode works, it’s like a shift but relative to the address “ value ” stored in BR.

* Indexed:
* For Ex: MOV R1, IDR\*+10

What this instruction does is: it takes the content of IDR and adds to it 10 then the value we get is an address that we operate on, like this is how Indexed addressing mode works it’s like a deplacement but relative to the address “ value ” stored in IDR.

* Based Indexed: two ways to do it:
  + MOV R1,IDR\*+BR+10
  + MOV R1,BR\*IDR+10

And here also it remains the same concept as in the two previous modes,

but it’s relative to the sum of the values of BR and IDR considered as and

address.

We have always the presence of the Asterisk \* why? Because we thought it will remind you that this will be considered as an address afterall, it’s true we take immediate values from the registers but after doing the sum with the number ( which it’s shifted with ) we use the result as an address.

* Shift: it can be on one byte or two bytes relative to the size of the operand which we shift with, and it can be like this: MOV R1, 100\*+1

or MOV R1, etiq\*+1 ( in this case etiq is a label I will speak about labels later ), as a production team of calM we don’t advise you to use such addressing modes for now it’s not very stable, sorry we hope we will make it a 100% functional Inshallah.

Since we are speaking about operands: Put in your mind that you can have a

number as first operand for MOV instruction.

You need also to know what are labels because it’s an important concept used

widely in assembly programming, labels are just a sheet of card where in a side you right a name as string and in the other you write a value, this is a label we create a kind of “ variable” that holds a number and you can use it the same way you use the number and the assembler does the work of flipping the card for you, as an Example: LABEL etiq 2 ,this is how you create a label, obviously its name is etiq and the value it’s holding is 2.

Note: it’s name can’t be a name of register or instruction or label itself, and cannot contain special characters.

Then let’s speak about your experience while writing calM code in the our emulator available at the web platform:

First of all don’t worry you can write code in lowercase or uppercase we do the conversion for you.

We have many buttons that help you discover our features:

* execute to execute the code,
* registers and memory to check the content of the memory and registers after execution.
* simulate to play the animation of the execution of the code in the machine.
* Hex to code: to change the hexadecimal code you inserted to calM’s assembly language automatically
* to Hex: to change the calM’s assembly language to hexadecimal code automatically

You can also write comments while coding by inserting the special character: “ ; or # or // or /\*comment\*/ ”

Some errors you can do while writing calM’s assembly code:

Pay attention !

Those instructions for example: NEG, NOT, SHL, SHR, READ, WRITE, PUSH, POP, ROR, ROL can only have registers as operands.

A label cannot have a name as a register and cannot contain special characters.

It’s not all we have, we will always work on improving it, just let us know your feedback.