SASM: The Simple Assembler
Reference Manual

Fernando Mário de Oliveira Filho
March 11th, 2015.

SASM, the Simple Assembler, is an assembler for the Simple Assembly Language, a simplified assembly language working on an imaginary and simplified computer. This computer is very simple, capable only of manipulating integers. The assembly language is also very simple, and avoids many complications of real-world assembly languages for real-world processors, like using registers, different kinds of instructions to do the same thing, etc. Its intended purpose is to help students who are learning to program to think in terms of assembly languages, without having to deal with the many technicalities of real-world architectures.

1. A simple computer

Our computer has a random-access memory comprised of memory cells addressed from 0. Given the address of a cell, it is possible to access its contents. Each cell holds an integer. This simple computer is capable of operating on these memory cells by moving data, performing additions, multiplications, etc.

SASM simulates this simple computer in your computer. SASM is written in `C` and uses variables of type `int` to represent the memory of our simple computer. In most systems these variables will have 32 bits, but this size might differ.

2. Instructions

SASM instructions describe the instructions that our simple computer is able to perform. The following notation will be used for an instruction:

`mnc` *op1* *op2*

Here, `mnc` is the *mnemonic* of the instruction, a simple string that represents it. It could be for instance `add` or `mov`; we will see a table soon. Then we have the *operands*, *op1* and *op2*. Some instructions require no operands, others require one operand or two operands, but not more.

Operands can be of four types: integer literal, memory address, indirect memory address, or label. An *integer literal* is simply an integer given in the usual base 10 notation, for example `345` or `-287`; it represents the number given.

A *memory address* refers to a memory cell of our simple computer by means of its address. Memory addresses are nonnegative integers preceded by a dollar sign `$`. For instance, `$0` refers to the memory cell with address 0 and `$345` refers to the memory cell with address 345. When we say that we take the value of a memory address operand, we mean that we take the value stored in the corresponding memory cell. When we say that we set the value of a memory address operand, we mean that we store a value in the corresponding memory cell.

An *indirect memory address* refers to a memory cell whose address is itself in another memory cell. An indirect memory address is a nonnegative integer preceded by a percent sign `%`. The indirect address `%`n represents the memory cell whose address is given in memory cell n. For instance, we might want to refer to the memory cell whose address is in memory cell 17. This we represent by `%17`. If cell 17 contains the number 43 when `%17` is resolved, then we access cell 43. Again, if we say that we take the value of an indirect memory address operand, we mean that we take the value stored in the corresponding memory cell, and similarly when we say that we set the value of the operand.

Let us see a simple example of the three first types of operands. The instruction `mov` takes two operands. It moves the contents of the first operand to the second operand. The

first operand can be a number, a memory address, an indirect memory address, or a label. The second operand can be a memory address or an indirect memory address; it cannot be a number, say, because we cannot move something to a number! Now, to place 5 in cell 17 we do:

```
mov 5 $17
```

To place the contents of cell 12 in the cell whose address is in cell 10, we do:

```
mov $12 %10
```

Finally, a *label* refers to an instruction in your program, so that jump commands can change program flow. Instructions are numbered starting from 0, so a label is actually just a number. So that you don't have to number your instructions or keep track of instruction numbers, the assembler allows you to give names to instructions using labels. This is done by preceding an instruction by a name followed by a colon:

```
label_name:  mov 1 $2
```

Here, `label_name` is the name of the label that represents this instruction. For instance, to jump to this instruction, you could use the `jmp` instruction as follows:

```
jmp label_name
```

Since labels are just numbers, you can move their values to memory positions, and then jump to an instruction whose number is contained in a memory position. For instance you could do:

```
name:  mov  name  $0
       jmp  $0
```

Labels can be composed of letters (capital or not), numbers, and the underscore symbol. The first character of a label must be either a letter or the underscore symbol. Labels are case-sensitive.

Here is a list of all instructions supported by SASM with the types of every operand. When an operand can be a memory address, it can also be an indirect memory address. We call these operands *addresses* for short.

Instruction	**Description**
`add` *op1 op2*	Sets first operand to the value of the first operand plus the second operand. First operand has to be an address; second is either an address or number.
`and` *op1 op2*	Sets first operand to zero if either operand is zero, otherwise sets the first operand to a nonzero number (this is the logical and). First operand must be an address; second operand may also be a number.
`dec` *op1*	Decrements first operand by 1; first operand has to be an address.
`div` *op1 op2*	Sets first operand to the value of the integer division of the first operand by the second operand. First operand has to be an address; second can be also a number. Integer division works the same as in `C` (in ANSI `C99`, the result is the integer part of the quotient).
`inc` *op1*	Increments the first operand by 1; first operand has to be an address.
`jgez` *op1 op2*	If first operand is greater or equal than zero, jumps to the instruction whose number is in the second operand. First operand must be an address; second must be a label or an address.
`jgz` *op1 op2*	Same as `jgez`, but jumps if greater than zero.
`jlez` *op1 op2*	Same as `jgez`, but jumps if less or equal than zero.

`jlz` *op1 op2*	Same as `jgez`, but jumps if less than zero.
`jmp` *op1*	Jumps to the instruction whose number is in the first operand. First operand must be a label or address.
`jnz` *op1 op2*	Same as `jgez`, but jumps if first operand is nonzero.
`jz` *op1 op2*	Same as `jgez`, but jumps if first operand is equal to zero.
`mod` *op1 op2*	Sets first operand to the remainder of the integer division of the first operand by the second operand. First operand must be an address; second operand can also be a number. The behavior is the same as that of the `%` operator from `C`.
`mov` *op1 op2*	Moves contents of the first operand to the second. First operand can be a number, address, or label; second operand must be an address.
`mul` *op1 op2*	Sets the first operand to the result of the product of the first operand and the second operand. First operand must be an address; second operand may also be a number.
`nop`	Does nothing.
`not` *op1*	If first operand is zero, sets it to a nonzero number; otherwise sets it to zero (this is logical negation). First operand must be an address.
`or` *op1 op2*	Sets first operand to zero if both operands are zero, otherwise sets it to a nonzero number (this is the logical or). First operand must be an address; second operand may also be a number.
`prt` *op1*	Prints first operand to the standard output. The operand can be either a number or an address.
`read` *op1*	Reads an integer from the standard input and stores it in the first operand. The operand must be an address.
`ret`	Terminates execution.
`sub` *op1 op2*	Sets the first operand to the value of the first operand minus the second operand. First operand must be an address; second operand may also be a number.

Rule of thumb. When an instruction takes two operands, it places the result in the first operand, which has to be a memory address. The exception is `mov`.

3. The syntax of a SASM program

A SASM program should obey the following rules. Blank lines are ignored, and the number of spaces (one or more) before, after, and between words is irrelevant. Everything beginning with a semicolon `;` is also ignored; these are comments.

Each instruction (instructions are as given on the table of the previous section) should come in a line by itself. The line can be labeled, as explained in the previous section, but the label is optional and only needed if one wants to refer to this instruction later.

As an example, consider Program 1, which reads a number from the standard input and prints its factorial to the standard output.

```
          read   $0
          mov    1       $1
begin:    jz     $0      end
          mul    $1      $0
          dec    $0
          jmp    begin
end:      prt    $1
```

Program 1. Computing the factorial of a number.

There is only one other thing you have to know about a SASM program, and that is the `def` directive. We use memory addresses to store information, but it can become cumbersome to remember which address holds what. For Program 1 we used only two addresses, but more complicated programs use more. So SASM allows you to associate names to numerical values. This can be done with the `def` directive, which is *not* an instruction. The syntax is:

`def` *name value*

where *name* is a string and *value* an integer. Notice that, since `def` is not an instruction, a line like the one above cannot be preceded by a label. This line is not part of your program: it just tells SASM that whenever you type the name it should be replaced by the value. The name given in a `def` directive has to follow the same rules as a label.

Instead of typing an integer as part of an operand, you can then type `(name)` and obtain the value. For instance, Program 1 could be rewritten to become the more pleasant Program 2.

```
def  n    0
def  res  1
          read   $(n)
          mov    1        $(res)
begin:    jz     $(n)     end
          mul    $(res)   $(n)
          dec    $(n)
          jmp    begin
end:      prt    $(res)
```

Program 2. A more pleasant factorial computation.

There are many more examples of short SASM programs with the SASM distribution. Take a look!

4. Running SASM

After you've written a program, you will want to run it. SASM is a simulator; it won't give you executable machine code, but it will simulate your program.

A file called `README`, that comes with the SASM distribution, explains how to compile SASM. Then it can be run as follows, assuming that it is in the system's path:

```
sasm filename [memory_size]
```

where `filename` is the name of a pure-text file containing the SASM program to be run. If `memory_size` is provided, it should be a positive integer. Then SASM will allocate that many memory cells for the simulation. By default, a total of 1 million cells is allocated.

F.M. de Oliveira Filho
Instituto de Matemática e Estatística
Universidade de São Paulo
fmario@gmail.com