

StreASM

A stream processing language for the criminally insane.

StreASM is a stream processing language styled after Assembly instructions. As it does not compile directly to machine code, but rather uses the Assembly syntax and methodology as inspiration, some liberties have been taken in the design of its instruction set to streamline the programmer experience.

1 - Registers

'Registers' are at the core of StreASM, much like variables are in other languages. A register is a globally accessible 32 bit signed integer storage location. Most instructions take a number of registers as parameters and perform some operation (See Table 1). An individual register is signified with a single lowercase or uppercase character (known as the "ident". eg 'a', 'r', 'I'), followed by an integer (known as the "index". eg '1', '10', '-50'). For example, the registers a1, r10 and I-50 are all valid names. Note that ident's are **case sensitive** and therefore r0 and R0 refer to different registers.

All Registers are completely un-initialised on runtime. They will throw an error if accessed before assignment.

1.1 - Registers as Arrays (Indirect Addressing)

An ident can be seen as a parallel to an Array. To support this notion, an additional way of indirectly addressing registers has been implemented, making use of the ident but with a second register for the index. An example of the syntax is r[r0]. This statement will access from the ident r the register with the index of the value of r0. So for example, if r0 were to equal 22, and r22 had the value of 64, r[r0] would evaluate to the value 64.

1.2 - Registers with Nice-Names

Some programmers may wish to identify their registers with names which are more meaningful. The '#DEF' command can be used to alias a 'nice name' String to an individual register. Any subsequent instruction can then accept this 'nice name' in place of the individual register. This functionality provides no additional power, but can help improve human-readability. A '#DEF' command is denoted by '#DEF' followed by the 'nice name', followed by the register to be aliased. An example invocation is given below:

```
#DEF count r0
    MOV count, 1           ;Equivalent to 'MOV r0, 1'
```

It is preferred that all '#DEF' commands occur at the very start of the program for style, but this is not a requirement. Note that unlike instructions, the '#DEF' command is **not indented**.

2 - Labels

In an Assembly-styled language, labels are the key to control flow. A label signifies a specific line of code in the program, and a set of instructions exist (such as JMP and CALL) to move the instruction pointer to execute instructions from that label. A label is denoted by a case-sensitive String followed by a colon (":"), in the first column. An example usage of a label is given below:

```
label:
    JMP label              ;Infinitely Loop
```

Two built-in labels exist, @NEXT and @END, which signify the next instruction and the end of the program respectively. It is possible to jump to a label before or after the current instruction; it is not necessary for the program to have scanned past a label to use it. Attempting to jump to an undefined label will cause an error to be thrown. It is important to note that labels are **case sensitive**.

3 - Instructions

Instructions in StreASM control the values of registers and the program flow. Each instruction is a 2-5 uppercase letter mnemonic and takes a specified number of arguments. A table of instructions and their function can be seen in Table 1.

3.1 - The NXT Instruction

The NXT instruction is an important one, as it is the sole interface for reading inputs and writing outputs. StreASM reads from the `stdin` and writes to the `stdout`, and this is represented faithfully in the instruction. Unlike other instructions, the NXT instruction writes and reads to/from a block of registers. As such, an `ident` is specified to write to/read from rather than a specific register.

3.1.1 - Reading

Using NXT to read from `stdin` is denoted by 'NXT', followed by the `ident` to write to, followed by '`stdin`'. The value of the zeroth register in the `ident` is set to the number of streams that were read. For each stream, 1 to `n`, the register at that index of the `ident` is assigned to the value of that stream entry. For example, consider the case where we read the entry of three streams, having values 7, 28 and 19, respectively, into the `ident i`. This will write to `i` such that:

Register	i-1 (i-2, i-3, ...)	i0	i1	i2	i3	i4 (i5, i6, ...)
Value	Unchanged	3	7	28	19	Unchanged

3.1.2 - Writing

Using NXT to write to `stdout` is denoted by 'NXT', followed by '`stdout`', followed by the `ident` to read from. Before using the NXT instruction, you must move values to the `ident`. By default, StreASM will look at the first 1024 registers in the `ident` to find values to print. If no values are inserted into the `ident`, it simply prints a newline. However, it is possible to indicate the number of streams to be output using the zeroth register, which will make the StreASM look for that many integers (including past the default 1024 limit) to print (or fail with an error if it cannot find that many).

3.2 - Control Flow and Conditionals

All uses of Control Flow require a defined `label`. JMP can be used to perform an unconditional jump. This changes the flow of execution by changing the instruction pointer. In simple terms, this instruction is used to change the line of the program that is being executed. CALL and RET can be used to implement functions. CALL pushes the current instruction pointer to the return stack and jumps to the given label. RET will pop and return to the instruction pointer stored at the top of the return stack. If the return stack is empty, it is equivalent to jumping to `@END`.

The TST series of conditional jump instructions can be used to jump (without modifying the return stack) to one location or another based on a condition specific to the instruction. See Table 1 for a full list.

3.4 - Bit-based Instructions

Bit Set (BS) is used to set or clear a specific bit in a register. BS follows the syntax `BS reg, bit, val`. The instruction will set the `bit` (where bit 0 is the least significant bit) of `reg` to either binary 1 or 0, based on `val`. For example, calling `BS r1, 2, 1` where `r1` is 10 will set `r1` to be 14. TSTB is a conditional jump command follows the same paradigm as BS, in that it tests a bit and chooses its jump based on whether that bit is set or cleared.

4 - Comments

Text following a semi-colon (';') is a comment, and will be ignored by StreASM. It is strongly recommended that you use comments to record the intent of programs written in StreASM. Comments can be written after an instruction on the same line, or on a line of their own. You can use comments before an instruction to stop that instruction from being executed.

Table 1

Mnemonic, Operands	Description
NXT ident, stdin	Read Input, where stdin is the string 'stdin' - see Section 3.1.1, 'Reading'
NXT stdout, ident	Write Output, where stdout is the string 'stdout' - see Section 3.1.2, 'Writing'
MOV dest, src	Replace the value in the dest register with the value in the src register. src may be replaced with a literal value.
BS reg, bit, val	Set or clear a bit in a register - see Section 3.4, 'Bit-based Instructions'
INCR reg	Increment the value in the given register.
DECR reg	Decrement the value in the given register.
ADD dest, src1, src2	Add the values in registers src1 and src2, and put the result in register dest. Either of src1 or src2 may be replaced with a literal value.
SUB dest, src1, src2	Subtract the value in src2 from the value in src1, and put the result in dest. Either of src1 or src2 may be replaced with a literal value.
MUL dest, src1, src2	Multiply the values in registers src1 and src2, and put the result in register dest. Either of src1 or src2 may be replaced with a literal value.
DIV dest, src1, src2	Divide the value in src1 by the value in src2, and put the result in dest. Either of src1 or src2 may be replaced with a literal value.
AND dest, src1, src2	Bitwise AND the values in registers src1 and src2, and put the result in register dest. Either of src1 or src2 may be replaced with a literal value.
NAND dest, src1, src2	Bitwise NAND the values in registers src1 and src2, and put the result in register dest. Either of src1 or src2 may be replaced with a literal value.
OR dest, src1, src2	Bitwise OR the values in registers src1 and src2, and put the result in register dest. Either of src1 or src2 may be replaced with a literal value.
NOR dest, src1, src2	Bitwise NOR the values in registers src1 and src2, and put the result in register dest. Either of src1 or src2 may be replaced with a literal value.
XOR dest, src1, src2	Bitwise XOR the values in registers src1 and src2, and put the result in register dest. Either of src1 or src2 may be replaced with a literal value.
COM dest, src1	Bitwise NOT the value in register src1, and put the result in register dest. src1 may be replaced with a literal value.
CLR reg	Set the value in the given register to 0.
JMP label	Unconditionally jump to the given label.
CALL label	Jump to the given label. The current instruction pointer will be pushed onto the call stack.
RET	Pop and return to the instruction indexed by the value at the head of the call stack. If the stack is empty, the program will exit.
TSTZ reg, if0, ifn0	Test whether the value of reg is equal to 0. If it is, jump to the label if0. Otherwise, jump to the label ifn0.
TSTE reg1, reg2, ifeq, ifneq	Test whether the value of reg1 is equal to the value of reg2. If it is, jump to the label ifeq. Otherwise, jump to the label ifneq.
TSTG reg1, reg2, ifg, ifng	Test whether the value of reg1 is greater than the value of reg2. If it is, jump to the label ifg. Otherwise, jump to the label ifng.
TSTGE reg1, reg2, ifge, ifnge	Test whether the value of reg1 is greater than or equal to the value of reg2. If it is, jump to the label ifge. Otherwise, jump to the label ifnge.
TSTL reg1, reg2, ifl, ifnl	Test whether the value of reg1 is less than the value of reg2. If it is, jump to the label ifl. Otherwise, jump to the label ifnl.
TSTLE reg1, reg2, ifle, ifnle	Test whether the value of reg1 is less than or equal to the value of reg2. If it is, jump to the label ifle. Otherwise, jump to the label ifnle.
TSTB reg, bit, if1, if0	Test a certain bit of the value of reg. If the bit is equal to 1, jump to the label if1. Otherwise, jump to the label if0.

Appendix

Example programs:

Printing out numbers from 0 to 10

```
    MOV r1, 0
main:
    MOV o1, r1
    NXT stdout, o
    INCR r1
    TSTLE r1, 10, main, @END
```

Printing out the first 12 square numbers

```
    MOV r1, 1
main:
    MUL r2, r1, r1
    MOV o1, r2
    NXT stdout, o
    INCR r1
    TSTLE r1, 12, main, @END
```

Reversing a single finite stream

```
    MOV r0, 0
collect:
    NXT i, stdin
    TSTZ i0, reverse, @NEXT
    INCR r0
    MOV r[r0], i1
    JMP collect
reverse:
    TSTZ r0, @END, @NEXT
    MOV o1, r[r0]
    NXT stdout, o
    DECR r0
    JMP reverse
```