The Iota Architecture and Programming Model

Version 1.0

October, 2012

David R. Miller
dave@millermattson.com

Table of Contents

Overview

The Iota machine is a tiny programming language and abstract processing engine designed for educational use for experimentation with automatic program generation. It has the following properties:

- Any bit pattern in Iota memory is a legal Iota program – there are no instruction faults
- All operations address existing memory – there are no memory faults
- Iota source code consists of a string of characters, one character per instruction

A corollary is that any string of random characters is the source code for a legal, executable Iota program. This allows, for example, a genetic algorithm to use an Iota program in source form as its genome, evolving the source code string until the Iota program produces the desired output.

Architecture

The Iota abstract machine is a Von Neumann architecture where all memory locations are accessed through a program counter (**PC**) or stack pointer (**SP**). The machine architecture can be characterized as a processor with three special-purpose registers (**PC**, **SP**, and **NZ** flag), $\boldsymbol{L}$ memory locations ("words") $\boldsymbol{W}$ bits wide, and an instruction set of about 40 opcodes. All possible states of memory and registers comprise legal Iota programs – there is no possibility of an illegal instruction or a memory fault.

Allowing all possible bit patterns to result in legal (but not necessarily interesting) programs simplifies experimentation with automatic program generation. AI programs, such as neural nets and genetic algorithms, can regard Iota programs as arbitrary bit patterns, evaluate them with an Iota simulator, and compare the results against some fitness metric, without concern for syntactic program correctness.

The Iota language is similar to a rudimentary assembly language. Iota source code can be expressed using either long or short mnemonics. Here is a simple Iota program that copies its input to its output as long as the data is nonzero, shown with long mnemonics with added comments:

```
LOOP          ; loop until the ENDL opcode
IN            ; read one char from stdin and push it on the stack
BNZ           ; skip the next instruction if nonzero
HALT          ; HALT
OUT           ; pop a word from the stack and write it to stdout
ENDL          ; loop forever
```

All memory addressing is through the stack pointer (**SP**) and is implied in each instruction, so there are no operands specified in the source code.

Using the short mnemonics, each Iota instruction can be represented by a single character, and is the preferred format for entering and saving programs. The sample program above is represented in source code form using short mnemonics as the string:

"`LIzHO]`".

An Iota abstract machine contains $\boldsymbol{L}$ memory locations, $\boldsymbol{W}$ bits wide, where

$$1 \le \boldsymbol{L} \le 2^{32}$$
$$6 \le \boldsymbol{W} \le 32$$

This Iota architecture specifies a set of possible abstract machines, where the *rank* of a machine is determined by unique combinations of $\boldsymbol{L}$ and $\boldsymbol{W}$. An Iota program runs differently on machines of different ranks. For example, an Iota program that runs correctly on an abstract machine of rank $\boldsymbol{L}$=16, $\boldsymbol{W}$=16 will not necessarily produce the same results on a machine with rank $\boldsymbol{L}$=20, $\boldsymbol{W}$=8.

Memory and addressing model

The $\boldsymbol{L}$ words of memory are addressed as locations 0 through $\boldsymbol{L}$ – 1. The Stack Pointer (**SP**) seamlessly wraps around, treating the entire address space as a circular buffer. Address calculations relating to the Program Counter (**PC**) (e.g., branch targets) are performed using unsigned 32-bit arithmetic and then adjusted if necessary to fit in the range [0..$\boldsymbol{L}$). When the PC would have been advanced to address A where A > $\boldsymbol{L}$, the address is immediately readjusted to the address modulo $\boldsymbol{L}$. This implies that it is impossible to address a non-existent memory location.

Each memory location can store unsigned values in the range [0..$2^{\boldsymbol{W}}$). The memory locations are all writable. Instruction opcodes are stored in memory by their numeric values as defined below.

Execution model

After machine reset, the three Iota registers have these initial values, and are described in more detail in a later section.

PC=0
SP=0
NZ=false

For each simulation cycle, if the value at the memory location indexed by **PC** is a defined opcode number, the opcode is executed in simulation as described in the tables below. If the memory location indexed by **PC** is not a defined opcode number, it is executed as if it were a NOP opcode. This implies there are no instruction faults.

Input and Output

The Iota instruction set has one IN and one OUT instruction. These are character-oriented, regardless of the value of $\boldsymbol{W}$. Each time an IN instruction is executed, one character is consumed from a user-specified input string or stream and pushed onto the Iota stack, narrowing or widening the value to $\boldsymbol{W}$ bits. Each OUT instruction pops the top of the Iota stack, casts the value to a (char) type, and sends it to the abstract output stream.

Source Code Representation

For Iota machines of rank $\boldsymbol{W} \leq 8$, the preferred source code format is a string of characters of length $\leq \boldsymbol{L}$. When loaded into Iota memory, the source string S is interpreted as follows:

```
for each character C in S:
      if C is a defined short mnemonic:
            store the opcode by opcode number
      else
            store the character value narrowed to W bits
```

When a short-mnemonic source code disassembly is generated by an Iota abstract machine of any rank, the listing will render each memory value N as a single character as follows:

```
if N is an assigned opcode number:
      show the short opcode mnemonic
else if N is representable as a single character:
      show the literal character
else:
      show a ';' (NOP) opcode
```

PC – Program Counter

The program counter **PC** contains a value in the range [0..***L***). Values wrap around the address space, such that when the instruction execution reaches the end of memory, execution by default continues at location zero. After machine reset, the **PC** starts at zero, and by default, is incremented by one after each instruction is executed. If any branching or looping opcode results in a calculated address outside the range [0..***L***), the address is changed to the calculated address modulo ***L*** so that it always refers to a valid memory location.

SP – Stack Pointer

The stack pointer **SP** contains a value in the range [0..***L***). Values wrap around, so that the entire memory becomes a circular buffer for the **SP**. After machine reset, the **SP** starts at zero, and always points at the last item stacked (the "top" of the stack). The **SP** grows toward lower addresses. If the **SP** changes to an address outside the range [0..***L***), it is changed to the value modulo ***L*** so that it always refers to a valid memory location. This implies that the first item pushed onto the stack after machine reset will be written to the top of memory at location ***L***-1, and the new **SP** will contain the value ***L***-1 and will grow downward from there.

NZ – Non-Zero Flag

The non-zero flag **NZ** reflects the non-zeroness of the most recent instruction that read or wrote memory, as defined in the details below. **NZ** = true means that the result was nonzero. The **NZ** flag can be tested with the BZ and BNZ instructions. On machine reset, **NZ** is initialized to false.

Math

Values in memory are regarded as unsigned integers in the range [0..2^{W}). The arithmetic opcodes evaluate their operands as if using 32-bit unsigned integers with the results truncated to the least significant ***W*** bits. See individual opcodes below for more details.

Example Program

Fibonacci sequence generation

For an Iota machine of rank ***L***=32, ***W***=16, the following Iota program generates and outputs the low order bytes of the Fibonacci sequence, starting at 1, 2, 3, 5, 8, ...

```
“;;.TS+SObT704O.TT+TS+;;SOT^;;.TD”
```

The following is a disassembly of the starting state:

```
+----------- memory address
|  +-------- memory contents
|  | +------ short mnemonic
|  | |  +--- long mnemonic
0 00 ; NOP
1 00 ; NOP
2 10 . INC
3 24 T TARGET
4 0c S SWAP
5 0e + ADD
6 0c S SWAP
7 04 O OUT
8 23 b BRAP
9 24 T TARGET
a 2b 7 SKIP7
b 0d 0 PUSH0
c 28 4 SKIP4
d 04 O OUT
e 10 . INC
f 24 T TARGET
10 24 T TARGET
11 0e + ADD
12 24 T TARGET
13 0c S SWAP
14 0e + ADD
15 00 ; NOP
16 00 ; NOP
17 0c S SWAP
18 04 O OUT
19 24 T TARGET
1a 14 ^ XOR
1b 00 ; NOP
1c 00 ; NOP
1d 10 . INC
1e 24 T TARGET
1f 06 D DUP
```

An execution trace through the first few output numbers is shown below. In each instruction cycle, the line beginning with "T" shows the cycle number, the registers, and the 32 memory locations, followed by a one-line disassembly of the next instruction to be executed. The memory location referenced by **SP** is highlighted in red. The location referenced by **PC** is highlighted in blue. The output data is highlighted in green. Note that this Iota program was not designed; it was evolved with a genetic algorithm.

```
T00: PC=00 SP=00 NZ=F mem[]=00 00 10 24 0c 0e 0c 04 23 24 2b 0d 28 04 10 24 24 0e 24 0c 0e 00 00 0c 04 24 14 00 00 10 24 06
00: 00 ; NOP
T01: PC=01 SP=00 NZ=F mem[]=00 00 10 24 0c 0e 0c 04 23 24 2b 0d 28 04 10 24 24 0e 24 0c 0e 00 00 0c 04 24 14 00 00 10 24 06
01: 00 ; NOP
T02: PC=02 SP=00 NZ=F mem[]=00 00 10 24 0c 0e 0c 04 23 24 2b 0d 28 04 10 24 24 0e 24 0c 0e 00 00 0c 04 24 14 00 00 10 24 06
02: 10 . INC
T03: PC=03 SP=00 NZ=T mem[]=01 00 10 24 0c 0e 0c 04 23 24 2b 0d 28 04 10 24 24 0e 24 0c 0e 00 00 0c 04 24 14 00 00 10 24 06
03: 24 T TARGET
T04: PC=04 SP=00 NZ=T mem[]=01 00 10 24 0c 0e 0c 04 23 24 2b 0d 28 04 10 24 24 0e 24 0c 0e 00 00 0c 04 24 14 00 00 10 24 06
04: 0c S SWAP
T05: PC=05 SP=00 NZ=T mem[]=00 01 10 24 0c 0e 0c 04 23 24 2b 0d 28 04 10 24 24 0e 24 0c 0e 00 00 0c 04 24 14 00 00 10 24 06
05: 0e + ADD
T06: PC=06 SP=1f NZ=T mem[]=00 01 10 24 0c 0e 0c 04 23 24 2b 0d 28 04 10 24 24 0e 24 0c 0e 00 00 0c 04 24 14 00 00 10 24 01
06: 0c S SWAP
T07: PC=07 SP=1f NZ=T mem[]=01 01 10 24 0c 0e 0c 04 23 24 2b 0d 28 04 10 24 24 0e 24 0c 0e 00 00 0c 04 24 14 00 00 10 24 00
07: 04 O OUT
Output =====> 0
T08: PC=08 SP=00 NZ=F mem[]=01 01 10 24 0c 0e 0c 04 23 24 2b 0d 28 04 10 24 24 0e 24 0c 0e 00 00 0c 04 24 14 00 00 10 24 00
08: 23 b BRAP
```

T09: PC=04 SP=00 NZ=F mem[]=01 01 10 24 0c 0e 0c 04 23 24 2b 0d 28 04 10 24 24 0e 24 0c 0e 00 00 0c 04 24 14 00 00 10 24 00
04: 0c S SWAP
T10: PC=05 SP=00 NZ=F mem[]=01 01 10 24 0c 0e 0c 04 23 24 2b 0d 28 04 10 24 24 0e 24 0c 0e 00 00 0c 04 24 14 00 00 10 24 00
05: 0e + ADD
T11: PC=06 SP=1f NZ=T mem[]=01 01 10 24 0c 0e 0c 04 23 24 2b 0d 28 04 10 24 24 0e 24 0c 0e 00 00 0c 04 24 14 00 00 10 24 02
06: 0c S SWAP
T12: PC=07 SP=1f NZ=T mem[]=02 01 10 24 0c 0e 0c 04 23 24 2b 0d 28 04 10 24 24 0e 24 0c 0e 00 00 0c 04 24 14 00 00 10 24 01
07: 04 O OUT
Output =====> 1
T13: PC=08 SP=00 NZ=T mem[]=02 01 10 24 0c 0e 0c 04 23 24 2b 0d 28 04 10 24 24 0e 24 0c 0e 00 00 0c 04 24 14 00 00 10 24 01
08: 23 b BRAP
T14: PC=04 SP=00 NZ=T mem[]=02 01 10 24 0c 0e 0c 04 23 24 2b 0d 28 04 10 24 24 0e 24 0c 0e 00 00 0c 04 24 14 00 00 10 24 01
04: 0c S SWAP
T15: PC=05 SP=00 NZ=T mem[]=01 02 10 24 0c 0e 0c 04 23 24 2b 0d 28 04 10 24 24 0e 24 0c 0e 00 00 0c 04 24 14 00 00 10 24 01
05: 0e + ADD
T16: PC=06 SP=1f NZ=T mem[]=01 02 10 24 0c 0e 0c 04 23 24 2b 0d 28 04 10 24 24 0e 24 0c 0e 00 00 0c 04 24 14 00 00 10 24 03
06: 0c S SWAP
T17: PC=07 SP=1f NZ=T mem[]=03 02 10 24 0c 0e 0c 04 23 24 2b 0d 28 04 10 24 24 0e 24 0c 0e 00 00 0c 04 24 14 00 00 10 24 01
07: 04 O OUT
Output =====> 1
T18: PC=08 SP=00 NZ=T mem[]=03 02 10 24 0c 0e 0c 04 23 24 2b 0d 28 04 10 24 24 0e 24 0c 0e 00 00 0c 04 24 14 00 00 10 24 01
08: 23 b BRAP
T19: PC=04 SP=00 NZ=T mem[]=03 02 10 24 0c 0e 0c 04 23 24 2b 0d 28 04 10 24 24 0e 24 0c 0e 00 00 0c 04 24 14 00 00 10 24 01
04: 0c S SWAP
T20: PC=05 SP=00 NZ=T mem[]=02 03 10 24 0c 0e 0c 04 23 24 2b 0d 28 04 10 24 24 0e 24 0c 0e 00 00 0c 04 24 14 00 00 10 24 01
05: 0e + ADD
T21: PC=06 SP=1f NZ=T mem[]=02 03 10 24 0c 0e 0c 04 23 24 2b 0d 28 04 10 24 24 0e 24 0c 0e 00 00 0c 04 24 14 00 00 10 24 05
06: 0c S SWAP
T22: PC=07 SP=1f NZ=T mem[]=05 03 10 24 0c 0e 0c 04 23 24 2b 0d 28 04 10 24 24 0e 24 0c 0e 00 00 0c 04 24 14 00 00 10 24 02
07: 04 O OUT
Output =====> 2
T23: PC=08 SP=00 NZ=T mem[]=05 03 10 24 0c 0e 0c 04 23 24 2b 0d 28 04 10 24 24 0e 24 0c 0e 00 00 0c 04 24 14 00 00 10 24 02
08: 23 b BRAP
T24: PC=04 SP=00 NZ=T mem[]=05 03 10 24 0c 0e 0c 04 23 24 2b 0d 28 04 10 24 24 0e 24 0c 0e 00 00 0c 04 24 14 00 00 10 24 02
04: 0c S SWAP
T25: PC=05 SP=00 NZ=T mem[]=03 05 10 24 0c 0e 0c 04 23 24 2b 0d 28 04 10 24 24 0e 24 0c 0e 00 00 0c 04 24 14 00 00 10 24 02
05: 0e + ADD
T26: PC=06 SP=1f NZ=T mem[]=03 05 10 24 0c 0e 0c 04 23 24 2b 0d 28 04 10 24 24 0e 24 0c 0e 00 00 0c 04 24 14 00 00 10 24 08
06: 0c S SWAP
T27: PC=07 SP=1f NZ=T mem[]=08 05 10 24 0c 0e 0c 04 23 24 2b 0d 28 04 10 24 24 0e 24 0c 0e 00 00 0c 04 24 14 00 00 10 24 03
07: 04 O OUT
Output =====> 3
T28: PC=08 SP=00 NZ=T mem[]=08 05 10 24 0c 0e 0c 04 23 24 2b 0d 28 04 10 24 24 0e 24 0c 0e 00 00 0c 04 24 14 00 00 10 24 03
08: 23 b BRAP
T29: PC=04 SP=00 NZ=T mem[]=08 05 10 24 0c 0e 0c 04 23 24 2b 0d 28 04 10 24 24 0e 24 0c 0e 00 00 0c 04 24 14 00 00 10 24 03
04: 0c S SWAP
T30: PC=05 SP=00 NZ=T mem[]=05 08 10 24 0c 0e 0c 04 23 24 2b 0d 28 04 10 24 24 0e 24 0c 0e 00 00 0c 04 24 14 00 00 10 24 03
05: 0e + ADD
T31: PC=06 SP=1f NZ=T mem[]=05 08 10 24 0c 0e 0c 04 23 24 2b 0d 28 04 10 24 24 0e 24 0c 0e 00 00 0c 04 24 14 00 00 10 24 0d
06: 0c S SWAP
T32: PC=07 SP=1f NZ=T mem[]=0d 08 10 24 0c 0e 0c 04 23 24 2b 0d 28 04 10 24 24 0e 24 0c 0e 00 00 0c 04 24 14 00 00 10 24 05
07: 04 O OUT
Output =====> 5
T33: PC=08 SP=00 NZ=T mem[]=0d 08 10 24 0c 0e 0c 04 23 24 2b 0d 28 04 10 24 24 0e 24 0c 0e 00 00 0c 04 24 14 00 00 10 24 05
08: 23 b BRAP
T34: PC=04 SP=00 NZ=T mem[]=0d 08 10 24 0c 0e 0c 04 23 24 2b 0d 28 04 10 24 24 0e 24 0c 0e 00 00 0c 04 24 14 00 00 10 24 05
04: 0c S SWAP
T35: PC=05 SP=00 NZ=T mem[]=08 0d 10 24 0c 0e 0c 04 23 24 2b 0d 28 04 10 24 24 0e 24 0c 0e 00 00 0c 04 24 14 00 00 10 24 05
05: 0e + ADD
T36: PC=06 SP=1f NZ=T mem[]=08 0d 10 24 0c 0e 0c 04 23 24 2b 0d 28 04 10 24 24 0e 24 0c 0e 00 00 0c 04 24 14 00 00 10 24 15

Instruction Set Summary

These are the opcodes of the Iota instruction set, shown with their single-character mnemonics and long mnemonics.

System and stack

```
;       NOP     No-operation
R       RESET   Reset
H       HALT    Halt program execution
I       IN      Input from stdin
O       OUT     Output to stdout
p       POP     Pop the stack
D       DUP     Duplicate the top of the stack
C       PUSHPC  Push the current PC
c       POPPC   Pop the stack and set PC
Y       POPSP   Pop the stack and set SP
G       SPTGT   Set the SP to the next TARGET
P       PUSHNZ  Push the NZ flag
S       SWAP    Swap the top two stacked words
```

Math and logic

```
0       PUSH0   Push a zero onto the stack
+       ADD     Add the top two stacked words
-       SUB     Subract the top two stacked words
.       INC     Increment the top of the stack
,       DEC     Decrement the top of the stack
*       MUL     Multiply the top two stacked words
/       DIV     Divide the top two stacked words
^       XOR     Bitwise exclusive OR
&       AND     Bitwise logical AND
|       OR      Bitwise logical OR
(       SHL     Logical shift left the top stacked word
)       SHR     Logical shift right the top stacked word
~       NOT     Bitwise invert the top stacked word
```

Conditionals

```
Z       BZ      Branch on zero
z       BNZ     Branch on not-zero
=       BEQ     Branch on equal
>       BGT     Branch on greater than
{       BLT     Branch on less than
}       BGE     Branch on greater or equal
```

Unconditionals

```
L       LOOP    Loop until the following ENDL
]       ENDL    End of LOOP
B       BRAN    Branch to next TARGET opcode
b       BRAP    Branch to previous TARGET opcode
T       TARGET  Branch target for BRAN, BRAP
1       SKIP1   Skip over the next instruction
2       SKIP2   Skip over the next two instructions
3       SKIP3   Skip over the next three instructions
4       SKIP4   Skip over the next four instructions
5       SKIP5   Skip over the next five instructions
6       SKIP6   Skip over the next six instructions
7       SKIP7   Skip over the next seven instructions
8       SKIP8   Skip over the next eight instructions
9       SKIP9   Skip over the next nine instructions
```

Instruction Set Details:

In the pseudo-code below, “trunc ***W***” represents a narrowing operator to ***W*** bits.

Temporaries such as “Temp” and “Op0” are abstractions for exposition only and do not exist in the machine model.

System and stack

long mnemonic	mnemonic	numeric	description
NOP	**;**	0 (see note)	No-operation, do-nothing

The NOP opcode may be encoded in memory by the value zero, or by any value not assigned to another opcode. During execution, all unassigned opcode values are mapped to the NOP instruction.

PC = **PC** + 1 mod ***L***
SP = no change
NZ = no change

long mnemonic	mnemonic	numeric	description
RESET	**R**	1	Reset

PC = 0
SP = 0
NZ = false

long mnemonic	mnemonic	numeric	description
HALT	**H**	2	Halt program execution

Causes program execution to stop.

long mnemonic	mnemonic	numeric	description
IN	**I**	3	Input a char from stdin, push it onto the stack

SP = **SP** – 1 mod ***L***
SP** = getchar() trunc ***W
PC = **PC** + 1 mod ***L***
NZ = true if the result stacked is nonzero, else false

long mnemonic	mnemonic	numeric	description
OUT	**O**	4	Pop a word from the stack, output to stdout

If the value on the top of the stack is outside the range of a char, it will be truncated to a char as it is output. This is inconsequential for Iota machines of rank $\boldsymbol{W} \leq 8$.

putchar((char)***SP**)
SP = **SP** + 1 mod $\boldsymbol{L}$
PC = **PC** + 1 mod $\boldsymbol{L}$
NZ = true if the character output is nonzero, else false

long mnemonic	mnemonic	numeric	description
POP	**p**	5	Pop a word from the stack

SP = **SP** + 1 mod $\boldsymbol{L}$
PC = **PC** + 1 mod $\boldsymbol{L}$
NZ = true if the item popped is nonzero, else false

long mnemonic	mnemonic	numeric	description
DUP	**D**	6	Duplicate the last stacked value

Temp = ***SP**
SP = **SP** – 1 mod $\boldsymbol{L}$
***SP** = Temp
PC = **PC** + 1 mod $\boldsymbol{L}$
NZ = true if the value duplicated is nonzero, else false

long mnemonic	mnemonic	numeric	description
PUSHPC	**C**	7	Push the PC onto the stack

SP = **SP** – 1 mod $\boldsymbol{L}$
***SP** = **PC** trunc $\boldsymbol{W}$
PC = **PC** + 1 mod $\boldsymbol{L}$
NZ = no change

long mnemonic	mnemonic	numeric	description
POPPC	**c**	8	Pop the **PC** from the stack

PC = ***SP** mod $\boldsymbol{L}$
SP = **SP** + 1 mod $\boldsymbol{L}$
NZ = no change

long mnemonic	mnemonic	numeric	description
POPSP	**Y**	9	Pop the **SP** from the stack

SP = ***SP** mod ***L***
PC = **PC** + 1 mod ***L***
NZ = no change

long mnemonic	mnemonic	numeric	description
SPTGT	**G**	10	Set the **SP** to the next TARGET opcode

A search for the subsequent TARGET opcode is done at the time the SPTGT instruction is encountered, from the SPTGT instruction to memory location ***L*** - 1. The search does not wrap around. If no TARGET opcode is found, or if the PC is already at ***L*** - 1, the SPTGT is executed as if it were a NOP instruction.

If a subsequent TARGET opcode is found:
    SP = address of the TARGET opcode
else
    SP = no change
PC = **PC** + 1 mod ***L***
NZ = no change

long mnemonic	mnemonic	numeric	description
PUSHNZ	**P**	11	Push the **NZ** flag

SP = **SP** – 1 mod ***L***
***SP** = **NZ**
PC = **PC** + 1 mod ***L***
NZ = no change

long mnemonic	mnemonic	numeric	description
SWAP	**S**	12	Swap the top two items on the stack

Temp = ***SP**
SP** = *(**SP** + 1 mod ***L)
*(**SP** + 1 mod ***L***) = Temp
PC = **PC** + 1 mod ***L***
NZ = no change

Math and logic

long mnemonic	mnemonic	numeric	description
PUSH0	**0**	13	Push a zero onto the stack

SP = **SP** – 1 mod ***L***
*__SP__ = 0
PC = **PC** + 1 mod ***L***
NZ = false

long mnemonic	mnemonic	numeric	description
ADD	**+**	14	Add the top two stacked words, push the result

SP = SP – 1 mod ***L***
__SP__ = ((__SP__ + 2 mod ***L***) + *(__SP__ + 1 mod ***L*****)**) trunc ***W***
PC = **PC** + 1 mod ***L***
NZ = true if the result is nonzero, else false

long mnemonic	mnemonic	numeric	description
SUB	**-**	15	Subtract the top two stacked words and push the result

SP = SP – 1 mod ***L***
__SP__ = ((__SP__ + 2 mod ***L***) - *(__SP__ + 1 mod ***L*****)**) trunc ***W***
PC = **PC** + 1 mod ***L***
NZ = true if the result is nonzero, else false

long mnemonic	mnemonic	numeric	description
INC	**.** (period)	16	Increment the item at the top of the stack

*__SP__ = (*__SP__**)** + 1 trunc ***W***
PC = **PC** + 1 mod ***L***
NZ = true if the result is nonzero, else false

long mnemonic	mnemonic	numeric	description
DEC	**,** (comma)	17	Decrement the item on the top of the stack

*__SP__ = (*__SP__**)** – 1 trunc ***W***
PC = **PC** + 1 mod ***L***
NZ = true if the result is nonzero, else false

long mnemonic	mnemonic	numeric	description
MUL	*	18	Multiply the top two stacked words and push the result

SP = **SP** – 1 mod ***L***
SP** = (*(**SP** + 2 mod ***L) * (*(**SP** + 1 mod ***L***)) trunc ***W***
PC = **PC** + 1 mod ***L***
NZ = true if the result is nonzero, else false

long mnemonic	mnemonic	numeric	description
DIV	/	19	Pop two words, divide, push the quotient and remainder

If the divisor is zero, the quotient will be the maximum possible word value, and the remainder zero.

Op0 = *(**SP** + 1 mod ***L***)
Op1 = ***SP**
if Op1 is zero, change Op0 to the maximum value and Op1 to 1
*(**SP** + 1 mod ***L***) = quotient of Op0 / Op1 trunc ***W***
***SP** = remainder of Op0 / Op1
PC = **PC** + 1 mod ***L***
NZ = true if the quotient is nonzero, else false

long mnemonic	mnemonic	numeric	description
XOR	^	20	Bitwise XOR the top two stacked words and push the result

SP = **SP** - 1 mod ***L***
SP** = (*(**SP** + 2 mod ***L) XOR *(**SP** + 2 mod ***L***)) trunc ***W***
PC = **PC** + 1 mod ***L***
NZ = true if the result is nonzero, else false

long mnemonic	mnemonic	numeric	description
AND	&	21	Bitwise AND the top two stacked words and push the result

SP = **SP** - 1 mod ***L***
SP** = (*(**SP** + 2 mod ***L) AND *(**SP** + 2 mod ***L***)) trunc ***W***
PC = **PC** + 1 mod ***L***
NZ = true if the result is nonzero, else false

long mnemonic	mnemonic	numeric	description
OR	**|**	22	Bitwise OR the top two stacked words and push the result

SP = **SP** - 1 mod ***L***
SP** = (*(**SP** + 2 mod ***L) OR *(**SP** + 2 mod ***L***)) trunc ***W***
PC = **PC** + 1 mod ***L***
NZ = true if the result is nonzero, else false

long mnemonic	mnemonic	numeric	description
SHL	**(**	23	Logical shift left

SP** = ***SP** << 1 trunc ***W
PC = **PC** + 1 mod ***L***
NZ = true if the result is nonzero, else false

long mnemonic	mnemonic	numeric	description
SHR	**)**	24	Logical shift right

SP** = ***SP** >> 1 trunc ***W
PC = **PC** + 1 mod ***L***
NZ = true if the result is nonzero, else false

long mnemonic	mnemonic	numeric	description
NOT	**~**	25	Bitwise NOT

SP** = NOT ***SP** trunc ***W
PC = **PC** + 1 mod ***L***
NZ = true if the result is nonzero, else false

Conditionals

long mnemonic	mnemonic	numeric	description
BZ	**Z**	26	Branch if zero (NZ flag is false)

Skips one opcode if **NZ** is false.

if **NZ** is false:
    PC = **PC** + 2 mod ***L***
else
    PC = **PC** + 1 mod ***L***
SP = no change
NZ = no change

long mnemonic	mnemonic	numeric	description
BNZ	**z**	27	Branch if nonzero (NZ flag is true)

Skips one opcode if **NZ** is true.

if **NZ** is true:
    PC = **PC** + 2 mod ***L***
else
    PC = **PC** + 1 mod ***L***
SP = no change
NZ = no change

long mnemonic	mnemonic	numeric	description
BEQ	**=**	28	Compare top two stacked words, branch if equal

if *(**SP** + 1 mod ***L***) .eq. ***SP**
    PC = **PC** + 2 mod ***L***
else
    PC = **PC** + 1 mod ***L***
SP = no change
NZ = no change

long mnemonic	mnemonic	numeric	description
BGT	**>**	29	Compare top two stacked words, branch if greater than

if *(**SP** + 1 mod ***L***) > ***SP**
    PC = **PC** + 2 mod ***L***
else
    PC = **PC** + 1 mod ***L***
SP = no change
NZ = no change

long mnemonic	mnemonic	numeric	description
BLT	**{**	30	Compare top two stacked words, branch if less than

if *(**SP** + 1 mod ***L***) < ***SP**
    PC = **PC** + 2 mod ***L***
else
    PC = **PC** + 1 mod ***L***
SP = no change
NZ = no change

long mnemonic	mnemonic	numeric	description
BGE	**}**	31	Compare top two stacked words, branch if greater than or equal

if *(**SP** + 1 mod ***L***) ≥ * **SP**
    PC = **PC** + 2 mod ***L***
else
    PC = **PC** + 1 mod ***L***
SP = no change
NZ = no change

Unconditionals

long mnemonic	mnemonic	numeric	description
LOOP	**L**	32	Repeat the following instructions up to the next ENDL

PC = **PC** + 1 mod ***L***
SP = no change
NZ = no change

long mnemonic	mnemonic	numeric	description
ENDL	**]**	33	End of LOOP

Execution resumes at the instruction following the preceding LOOP opcode. A search for the preceding LOOP opcode is done at the time the ENDL instruction is encountered, from the current **PC** to location 0. The search does not wrap around. If no LOOP opcode is found, or if the **PC** is already at location 0, the ENDL is executed as if it were a NOP instruction.

If there is a preceding LOOP instruction:
    PC = location of LOOP opcode + 1
else:
    PC = **PC** + 1 mod ***L***
SP = no change
NZ = no change

long mnemonic	mnemonic	numeric	description
BRAN	**B**	34	Branch to the next TARGET opcode

A search for the subsequent TARGET opcode is done at the time the BRAN instruction is encountered, from the BRAN instruction to memory location ***L*** - 1. The search does not wrap around. If no TARGET opcode is found, the BRAN is executed as if it were a NOP instruction. If the TARGET is found at memory location ***L*** – 1, execution will resume at location 0.

If there is a subsequent TARGET instruction:

PC = (location of TARGET opcode + 1) mod ***L***

else:

PC = **PC** + 1 mod ***L***

SP = no change
NZ = no change

long mnemonic	mnemonic	numeric	description
BRAP	**b**	35	Branch to the previous TARGET opcode

A search for the previous TARGET opcode is done at the time the BRAP instruction is encountered, from the BRAP instruction to memory location 0. The search does not wrap around. If no TARGET opcode is found or if the **PC** is already at location 0, the BRAP is executed as if it were a NOP. instruction.

If there is a prior TARGET instruction:

PC = location of TARGET opcode + 1

else:

PC = **PC** + 1 mod ***L***

SP = no change
NZ = no change

long mnemonic	mnemonic	numeric	description
TARGET	**T**	36	Branch target for BRAN and BRAP

See SPTGT, BRAN, and BRAP instructions for the semantics. The TARGET opcode is just a marker, and is executed as if it were a NOP.

PC = **PC** + 1 mod ***L***
SP = no change
NZ = no change

long mnemonic	mnemonic	numeric	description
SKIP1	**1**	37	Skip one instruction

PC = **PC** + 2 mod ***L***
SP = no change
NZ = no change

long mnemonic	mnemonic	numeric	description
SKIP2	**2**	38	Skip the next two instructions

PC = **PC** + 3 mod ***L***
SP = no change
NZ = no change

long mnemonic	mnemonic	numeric	description
SKIP3	**3**	39	Skip the next three instructions

PC = **PC** + 4 mod ***L***
SP = no change
NZ = no change

long mnemonic	mnemonic	numeric	description
SKIP4	**4**	40	Skip the next four instructions

PC = **PC** + 5 mod ***L***
SP = no change
NZ = no change

long mnemonic	mnemonic	numeric	description
SKIP5	**5**	41	Skip the next five instructions

PC = **PC** + 6 mod ***L***
SP = no change
NZ = no change

long mnemonic	mnemonic	numeric	description
SKIP6	**6**	42	Skip the next six instructions

PC = **PC** + 7 mod ***L***
SP = no change
NZ = no change

long mnemonic	mnemonic	numeric	description
SKIP7	**7**	43	Skip the next seven instructions

PC = **PC** + 8 mod ***L***
SP = no change
NZ = no change

long mnemonic	mnemonic	numeric	description
SKIP8	**8**	44	Skip the next eight instructions

PC = **PC** + 9 mod ***L***
SP = no change
NZ = no change

long mnemonic	mnemonic	numeric	description
SKIP9	**9**	45	Skip the next nine instructions

PC = **PC** + 10 mod ***L***
SP = no change
NZ = no change

Iota Simulator Reference Implementation

iota(1)

NAME

iota – command line simulator for running programs in the Iota tiny programming language.

SYNOPSIS

iota [--version] [-L arch-length] [-W arch-width] [-k max-cycles]
[-I data-input-string] [-S] [-T] program-source

COPYRIGHT

GPLv3 or later.

DESCRIPTION

This is a command-line simulator of the Iota tiny architecture and programming language. The source code for an Iota program is a string of characters, where each character is one complete instruction. For example, the Iota program 'Gp..OOOOOOOOOOOOHTFello World!', when executed, will output the character string "Hello World!".

The Iota programming language was designed for experimentation with genetic algorithms that evolve procedural solutions to specific problems. All possible bit patterns are legal Iota programs, and all character strings form legal Iota source code. All Iota programs execute deterministically without the possibility of instruction or memory faults. See the link at the author information below for more information.

OPTIONS

-h | --help
Print the command line synopsis.

--version
Print the Iota simulator and language version numbers.

-L arch-length
The number of locations in the Iota simulated memory, in the range 1 to 2^32. Default is the length of the program string. Iota memory is initialized to all NOP opcodes (numeric value zero).

-W arch-width
The width, in bits, of Iota simulated memory, in the range 6 to 32. Default is 8 bits. Iota opcodes are encoded to fit in 6 bits width.

-k max-cycles
Set an upper limit on the number of Iota instructions executed, in the range 1 to 2^32. Default is 200.

-I data-input-string
This string supplies the characters for the Iota IN opcode. Each IN instruction reads one character from this string. After the last character has been consumed, subsequent IN instructions will continue to read the same last character. Iota programs have no way of detecting the end of the input data stream.

-S

Print an Iota memory dump and disassembly after loading the program.

-T

Print a trace of the Iota program execution, showing the state of memory and registers after each instruction.

program-string

The source code for the Iota program to be executed.

EXIT STATUS

The program exits with 0 if the Iota program ran successfully and stopped at a HALT instruction, or if it stopped after executing max-cycles (-k option) instructions. The program exits with a nonzero exit code for any error.

FILES

iota-machine.h is the header file with the interface and implementation of the simulator. iota.cpp is the command line driver.

HISTORY

Ver. 1.0 of the Iota language and simulator was released October, 2012.

AUTHOR

David R. Miller <dave@millermattson.com>

More information is available at http://millermattson.com/dave

Copyright Information