Core Assembly Manual

Version 0.1

Core Architecture

Core Society is a digital organism simulator where programs live in a rectangular *Grid of Cores*, not unlike a cellular automata, forming a two dimensional space with periodic boundary conditions.

Each core has 256 words of memory and an instruction pointer that always points to one of the words in memory. When the core gains execution focus that word is interpreted as an instruction. Some instructions require additional parameters which will be read from subsequent words. At the end the instruction pointer will be set to a new location, typically the word right after the last read while executing the instruction.

A core can be programmed using the "Core Assembly Language" defined in this document.

Example

The following program copies itself to adjacent cores. When there are no empty adjacent cores it increases the score in an infinite loop.

`[1]`	Set the current address to 1. (a core has 255 addressable words)
`0`	Put the number 0 at current address. This changes the current address to 2.
`eval_target:`	Label the current address (2) *eval_target*.
`TGT 1`	Take the number at address 1, interpret it as direction, and use the adjacent core in that direction as the target for targeted instructions.
`RDW 0 2`	Put the number at address 2 in the target core at address 0 in the executing core.
`IFS 0`	If there is some number other than zero at address 0 execute next instruction.
`NXT next_target`	Set the instruction pointer to the address labeled *next_target*
`//Copy Code`	This is a comment and does nothing.
`SIP 10`	Set the instruction pointer of the target core to 10
`SET 0 !1`	Put number 1 into address 0
`copy:`	Label the following address *copy*
`STW ~0 ~0`	Treat the number at address 0 as an address. Transfer the number at this address in the target core to this address in the executing core.
`INC 0`	Increase the number at address 0 by one.

```
IFL 0 !17         Execute the following instruction only if the number at address 0 is below 17
NXT copy          Set the instruction pointer to the address labeled copy
SIP 2             Set the instruction pointer of the target core to 2

next_target:      Label the following address next_target.
INC 1             Increase the number at address 1 by one.
IFL 1 !5          Execute the following instruction only if the number at address 1 is below 5.
NXT eval_target   Set the instruction pointer to the address labeled eval_target

work:             Label the following address work
ISC !F            Increase score by 0xF (decimal 15) which will cost 15 energy.
NXT work          ...and continue to add score by setting the IP to the last instruction.
```

Syntax

A program is made up multiple lines that denote a memory address, value, label, instruction or comment.

Address

A number followed by ‘:’ requests that the next parsed instruction, label or value is to be placed at this address.

Label

An identifier followed by ‘:’ labels the address where the next parsed instruction or value will be placed. Labels can be used instead of numbers to identify addresses in instruction parameters.

Value

A single hexadecimal number is placed directly into memory at the current address.

Instructions

Instructions are identified by a 3 character Mnemonic. Most instructions have to be followed by one or two parameters that (depending on the instruction) either need to denote numerals or addresses.

Addresses (**A**)

Addresses are denoted by a hexadecimal number. Each core has 256 addressable words so each word is uniquely identified by a hexadecimal number between 0x0 and 0xFF. When a number is prefixed with a '~' the final address is resolved by indirection: Not the given address but the value stored in it's associated word is used as passed as an address parameter to the instruction.

`INC F`	`INC ~F`
the value at address F is increased by one	*the value at address F is the address at which to increase the value*

Numerals (**N**)

Numerals are denoted by prefixing a hexadecimal number with '!'. If no prefix exists a number is treated as an address. When an address is passed to an instruction expecting a numeric parameter the value at the given address is read and used as a parameter.

`ISC !F`	`ISC F`
add 0xF to the score	*add the number at address F to the score*

Two-parametric instructions always occupy two words, the other instructions occupy only one.

Instruction Set

0x0000 NOP

"No Op"

Does nothing.

0x1000 INC A

"Increase"

Increases the number at address A by one.

0x1200 DEC A

"Decrease"

Increases the number at address A by one.

0x1400 NOT A

“Not”

Reverses each bit at address A.

0x1600 RND A

“Random”

Sets the number at address A to a random value.

0x1800 SQT A

“Square Root”

Sets the number at address A to its square root.

0x2000 SET A N

“Set”

Sets the number at address A to the value of N.

0x2400 RDW A_1 A_2

“Read Word”

Sets the number at address A_1 to the number at the target core’s address A_2.

0x2800 STW A N

“Store Word”

Sets the number at the target core’s address A to the value of N.

0x3000 ADD A N

“Add”

Increases the number at address A by N.

0x3400 SUB A N

“Sub”

Decreases the number at address A by N.

`0x3800 MUL A N`
“Multiply”
Multiplies the number at address A by N.

`0x3C00 DIV A N`
“Divide”
Divides the number at address A by N.

`0x4000 MOD A N`
“Modulo”
Sets the number at address A to the remainder of its division by N.

`0x4400 MAX A N`
“Max”
Sets the number at address A to N if N is bigger.

`0x4800 MIN A N`
“Min”
Sets the number at address A to N if N is smaller.

`0x5000 AND A N`
“And”
Each of the 16 bits of the number at address A are compared to the 16 bits of N using the logical AND operation. The 16 resulting bits are stored in address A.

`0x5400 IOR A N`
“Inclusive Or”
Each of the 16 bits of the number at address A are compared to the 16 bits of N using the logical inclusive OR operation. The 16 resulting bits are stored in address A.

`0x5800 XOR A N`
“Exclusive Or”
Each of the 16 bits of the number at address A are compared to the 16 bits of N using the logical exclusive OR operation. The 16 resulting bits are stored in address A.

`0x6000 IFZ A`

"If Zero"

Only executes the next instruction if the number at address A is zero.

`0x6200 IFS A`

"If Set"

Only executes the next instruction if the number at address A is not zero.

`0x6400 IFR A`

"If Random"

Only executes the next instruction if the number at address A is smaller than a random number between 0x0 and 0xFFFF.

`0x7000 IFE A N`

"If Equal"

Only executes the next instruction if the number at address A is equal to N.

`0x7400 IFN A N`

"If Not Equal"

Only executes the next instruction if the number at address A is not equal to N.

`0x7800 IFG A N`

"If Greater"

Only executes the next instruction if the number at address A is greater than N.

`0x7C00 IFL A N`

"If Less"

Only executes the next instruction if the number at address A is smaller than N.

`0x8000 JMP N`

"Jump"

Adds N to the Instruction Pointer. If IP exceeds 0xFF it wraps around.

0x8200 NXT A

"Next"

Sets the Instruction Pointer to address A.

0x8400 SIP A

"Set Instruction Pointer"

Sets the Instruction Pointer of the target core to address A.

0x9000 QIP A

"Query Instruction Pointer"

Sets the number at address A to the value of the target core's Instruction Pointer.

0x9200 QUE A

"Query Unbound Energy"

Sets the number at address A to the value of the target core's unbound energy.

0x9400 QSE A

"Query Shield Energy"

Sets the number at address A to the value of the target core's shield energy.

0x9600 QCE A

"Query Charge Energy"

Sets the number at address A to the value of the target core's charge energy.

0x9800 QTC A

"Query Target Core"

Sets the number at address A to a value N based on the target core's current target. 0 = Up, 1 = Right, 2 = Down, 3 = Left, 4 = Self.

0xA000 TGT N

"Target"

Sets a new target core based on N. 0 = Up, 1 = Right, 2 = Down, 3 = Left. Other values of N are mapped to the executing core. (n > 3 = Self).

0xB000 RSV N

“Reserve”

Executing the RSV instructions costs N energy. After the N energy have fully charged they are added to the core’s unbound energy level. Unbound energy caps at 255. This can be used to prepare the uninterrupted execution of multiple consecutive instructions.

`0xB200 ISE N`

“Increase Shield Energy”

Executing the ISE instructions costs N energy. After the N energy have fully charged they are added to the core’s shield energy level. Shield energy caps at 255. A shielded core can not be modified by adjacent cores.

`0xB400 DSE N`

“Decrease Shield Energy”

Decreases the core’s shield energy level by N. A shielded core can not be modified by adjacent cores.

`0xB600 BST N`

“Boost”

Executing the BST instructions costs N energy. After the N energy have fully charged they are added to the target core’s unbound energy level. Unbound energy caps at 255. This can be used to allow the target core to execute additional instructions.

`0xB800 WKN N`

“Weaken”

Executing the WKN instructions costs N energy. **N may not exceed 0xF**. After the N energy have fully charged they are removed from the target core’s shield energy. Any excess energy further reduces the target core’s unbound energy level. This can be used to make the target core vulnerable to write operations and (if unshielded) to prevent it from executing instructions.

`0xC000 ISC N`

“Increase Score”

Executing the ISC instructions costs N energy. After the N energy have fully charged they are added to the global score. Score is typically used to measure a solutions performance in Scenarios.

`0xC200 DSC N`

“Decrease Score”

Executing the DSC instructions costs N energy. After the N energy have fully charged they are removed from the global score. Score is typically used to measure a solutions performance in Scenarios.