Introduction to the Avida Digital Life Platform Teacher Information

The Virtual CPU Structure

The virtual CPU, which is the default "body" or "hardware" of the organisms, contains the following set of components, which are further illustrated in the diagram, below:

- A **memory** that consists of a sequence of instructions, each associated with a set of flags to denote if the instruction has been executed, copied, mutated, etc. Memory is treated as circular, such that execution loops back to the first instruction after the last instruction has been executed.
- An **instruction pointer** (IP) that indicates the next site in the memory to be executed.
- Three **registers** that can be used by the organism to hold data currently being manipulated. These are often operated upon by the various instructions and can contain arbitrary 32-bit integers.
- Two **stacks** that are used for storage. The organism can theoretically store an arbitrary amount of data in the stacks, but for practical purposes we currently limit the maximum stack depth to 10.
- An **input buffer** and an **output buffer** that an organism uses to receive information and return the processed results.
- A **Read-Head**, a **Write-Head** and a **Flow-Head** that are used to specify positions in the CPU memory. A copy command reads from the Read-Head and writes to the Write-Head. Jump-type statements move the IP to the Flow-Head

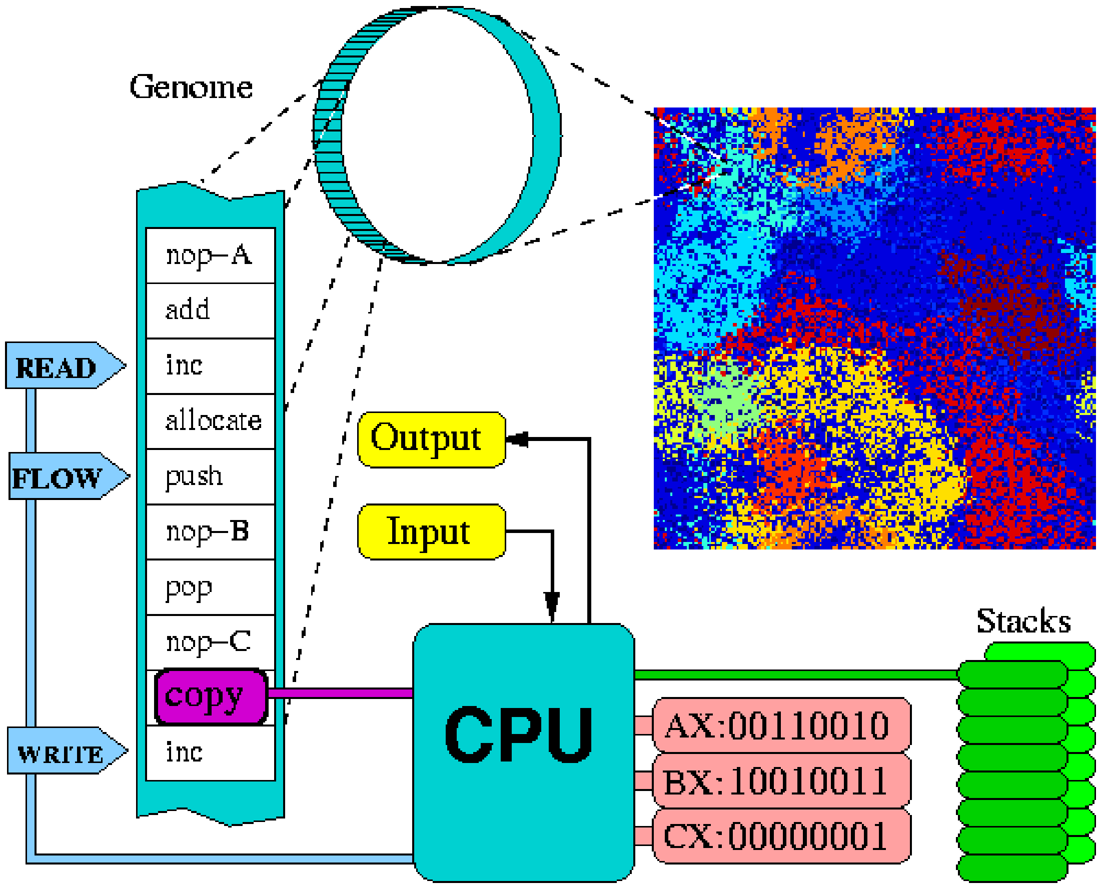

Image source: Avida, http://avida.devosoft.org/documentation/page/Default_Ancestor_Guided_Tour

Instruction Set Configuration

The instructions were created with three things in mind:

1. To be as complete as possible (both in a "Turing complete" or "computationally universal" sense—that is, it can compute any computable function—and, more practically, to ensure that simple operations only require a few instructions).
2. For each instruction to be as robust and versatile as possible; all instructions should take an "appropriate" action in any situation in which they can be executed.
3. To have as little redundancy as possible between instructions. (Several instructions have been implemented that are redundant, but such combinations are typically not be turned on simultaneously for a run.)

One major concept that differentiates this virtual assembly language from its real-world counterparts is in the additional uses of `nop` instructions (no-operation commands). These have no direct effect on the virtual CPU when executed, but often modify the effect of any instruction that precedes them. Think of them as purely regulatory genes. The default instruction set has three such `nop` instructions: `nop-A`, `nop-B`, and `nop-C`.

The remaining instructions can be separated into three classes. The first class is those few instructions that are unaffected by nops. Most of these are the "biological" instructions involved directly in the replication process. The second class of instructions is those for which a `nop` changes the head or register affected by the previous command. For example, an `inc` command followed by the instruction `nop-A` would cause the contents of the `AX` register to be incremented, while an `inc` command followed by a `nop-B` would increment `BX`.

The notation we use in instruction definitions to describe that a default component (that is, a register or head) can be replaced due to a nop command is by surrounding the component name with `?`'s. The component listed is the default one to be used, but if a `nop` follows the command, the component it represents in this context will replace this default. If the component between the question marks is a register, then a subsequent `nop-A` represents the `AX` register: `nop-B` is `BX`, and `nop-C` is `CX`. If the component listed is a head (including the instruction pointer), then a `nop-A` represents the Instruction Pointer, `nop-B` represents the Read-Head, and `nop-C` is the Write-Head. Currently, the Flow-Head has no `nop` associated with it.

The third class of instructions are those that use a series of `nop` instructions as a template (label) for a command that needs to reference another position in the code, such as `h-search`. If `nop-A` follows a search command, it scans for the first complementary template (`nop-B`) and moves the Flow-Head there. Templates may be composed of more than a single `nop` instruction. A series of nops is typically abbreviated to the associated letter and separated by colons. Thus the sequence "`nop-A nop-A nop-C`" would be displayed as "`A:A:C`."

Instruction Set Reference

Abbreviation	Instruction	Instruction Definition
a	nop-a	No-operation instruction; modifies other instructions
b	nop-b	No-operation instruction; modifies other instructions
c	nop-c	No-operation instruction; modifies other instructions
d	if-n-equ	Execute next instruction only-if ?BX? does not equal its complement
e	if-less	Execute next instruction only if ?BX? is less than its complement
f	pop	Remove a number from the current stack and place it in ?BX?
g	push	Copy the value of ?BX? onto the top of the current stack
h	swap-stk	Toggle the active stack
i	swap	Swap the contents of ?BX? with its complement
j	shift-r	Shift all the bits in ?BX? one to the right
k	shift-l	Shift all the bits in ?BX? one to the left
l	inc	Increment ?BX?
m	dec	Decrement ?BX?
n	add	Calculate the sum of BX and CX; put the result in ?BX?
o	sub	Calculate the BX minus CX; put the result in ?BX?
p	nand	Perform a bitwise NAND on BX and CX; put the result in ?BX?
q	IO	Output the value ?BX? and replace it with a new input
r	h-alloc	Allocate memory for an offspring
s	h-divide	Divide off an offspring located between the Read-Head and Write-Head
t	h-copy	Copy an instruction from the Read-Head to the Write-Head and advance both
u	h-search	Find a complement template and place the Flow-Head after it
v	mov-head	Move the ?IP? to the same position as the Flow-Head
w	jmp-head	Move the ?IP? by a fixed amount found in CX
x	get-head	Write the position of the ?IP? into CX
y	if-label	Execute the next instruction only if the given template complement was just copied
z	set-flow	Move the Flow-Head to the memory position specified by ?CX?

Avida Tasks (logic functions that can be rewarded)

Task	Description
not	This task is triggered when an organism inputs a 32-bit number, toggles all of the bits, and outputs the result. This is typically done either by nanding (by use of the nand instruction) the sequence to itself, or negating it and subtracting one. The latter approach only works since numbers are stored in twos-complement notation.
nan	This task is triggered when two 32-bit numbers are input, the values are “nanded” together in a bitwise fashion, and the result is output. Nand stands for “not and.” The nand operation returns a zero if and only if both inputs are one; otherwise it returns a one.
and	This task is triggered when two 32-bit numbers are input, the values are “anded” together in a bitwise fashion, and the result is output. The and operation returns a one if and only if both inputs are one; otherwise it returns a zero.
orn	This task is triggered when two 32-bit numbers are input, the values are “orn” together in a bitwise fashion, and the result is output. The orn operation stands for or-not. It returns true if for each bit pair one input is one *or* the other one is zero.
oro	This task is triggered when two 32-bit numbers are input, the values are “ored” together in a bitwise fashion, and the result is output. It returns a one if either the first input *or* the second input is a one, otherwise it returns a zero.
ant	This task is triggered when two 32-bit numbers are input, the values are “andn-ed” together in a bitwise fashion, and the result is output. The andn operation stands for and-not. It only returns a one if for each bit pair one input is a one *and* the other input is not a one. Otherwise it returns a zero.
nor	This task is triggered when two 32-bit numbers are input, the values are “nored” together in a bitwise fashion, and the result is output. The nor operation stands for not-or and returns a one only if both inputs are zero. Otherwise a zero is returned.
xor	This task is triggered when two 32-bit numbers are input, the values are “xored” together in a bitwise fashion, and the result is output. The xor operation stands for “exclusive or” and returns a one if one, but not both, of the inputs is a one. Otherwise a zero is returned.
equ	This task is triggered when two 32-bit numbers are input, the values are equated together in a bitwise fashion, and the result is output. The equ operation stands for “equals” and returns a one if both bits are identical, and a zero if they are different.

See additional information at http://devolab.msu.edu/documentation/.