

# **The God Machine**

By Mike Hahn

Copyright © 2008 Vecsworld.com

Date Last Edited: 26-Nov-2008

# Table of Contents

<b>The God Machine .....</b>	<b>1</b>
<u>Introduction .....</u>	<u>1</u>
<u>God Computer .....</u>	<u>2</u>
<u>God Computer for Dummies .....</u>	<u>3</u>
<u>Looping and Branching.....</u>	<u>4</u>
<u>Pico-Code .....</u>	<u>4</u>
<u>High-Level Language .....</u>	<u>5</u>
Data Structure.....	5
Operators .....	6
Statements .....	7
Method Declarations.....	7
Class Declarations.....	7
Sample Source File.....	8
<b>Appendixes.....</b>	<b>9</b>
<u>Appendix A - Pico-Code Buffer .....</u>	<u>9</u>

# The God Machine

## Introduction

[ [Home](#) ]

I'm about half-way finished the design specs for my God Computer (GC). The GC is a theory that the universe in which we live is actually a computer simulation, written by intelligent digital life forms who live in a digital universe of their own. That digital universe is a relatively simple computer whose vast memory is organized as a 2D array of bits. When I'm finished my design specs (so far I've designed the high level and the low level; the middle level has yet to be designed), I intend to write a Windows-based simulation of the GC. When that's done I'm going to upload the simulation to this web site, and publicize the GC on 2 forums: a philosophy forum and a science/computer science forum.

I maintain that the design I have come up with is just about the most concise and elegant possible design for a mathematical, digital computer that exists in almost pure software, independent of real-world transistors, electrons, and logic gates. Despite its simplicity, my design is scalable, so it can be used to simulate our universe, with its subatomic particles numbering up to  $10^{200}$ . There are billions of possible computer designs. Perhaps they all exist simultaneously in a vast digital universe. Every clock tick, for every possible computer design, a new memory is generated, identical to the memory from the previous clock tick, and if that memory is treated as one long integer, it is incremented. Then that same memory becomes dynamic, and is executed by the current computer design. The vast majority of memory-computer design pairs don't actually do anything interesting. But a select few actually work. And a tiny fraction of those that work generate digital life forms, and each life form has its own copy of digital DNA, and they evolve and become ever more complex, and eventually achieve sentience. Then these sentient beings write a computer simulation of our universe, starting with the big bang.

This theory of mine is compatible with all the world's major religions. Take Christianity, for instance. Assume that Christianity is the one true religion. The creators of our universe (a computer simulation) somehow detected that intelligent life has evolved on earth. So they created a database of all humans, living or dead. Those humans who pray and worship Jesus are granted special status by our creators. After every human dies, our creators save a digital copy of the human's brain (the soul), and a Boolean, yes/no decision is made: either that soul is saved, or it isn't. Those souls which are saved go to a place called Heaven, and dwell there for all eternity.

I realize my GC idea sounds like science fiction. But it's the only Creation theory I know of that does away with the need for a mystical Creator that has existed since the beginning of time. There are just so many countless GCs out there in that digital universe, and at least one of them clicked, so here we are.

# God Computer

The memory of the God Computer (GC) consists of a 2D array of bits, which is X columns wide by Y rows tall.

$X = 2$  raised to the 16th power = 65,536 columns

$Y = 2$  raised to the power of  $(X - 16)$  rows

$Y = \text{approx. } 10$  raised to the 20,000th power

The array of bits is divided into 2 vertical sections. The left-hand section is 16 columns wide. The right-hand section is 65,520 columns wide. The top 4 rows are devoted to global data: the locations of 4 cursors (current bits). The first half of the top row (Row 1) contains various housekeeping data. The second half contains the pico-code buffer. Row 2 contains the row no. of the current microcode instruction. Rows 3 and 4 contain the row nos. of the primary and secondary data cursors. The left-hand section of Rows 3 and 4 contain the column nos. of those cursors.

The microcode computer instructions are contained in the left-hand section. Each instruction is 4 bits long, giving a maximum of 16 different instructions, and each row contains one instruction. The instructions are normally executed one at a time, from top to bottom.

Here are the 15 basic microcode instructions (this is subject to change):

0000 U - UP	Up	1000 R - READ	Read
0001 D - DOWN	Down	1001 W - WRTE	Write
0010 P - PREV	Previous	1010 C - CURS	Cursor Toggle
0011 N - NEXT	Next	1011 V - VERT	Vertical Section
0100 T - TOP	Top	1100 X - XOR	XOR
0101 L - LEFT	Left	1101 J - JUMP	Jump
0110 E - ESC	Escape	1110 K - UJMP	Unconditional Jump
		1111 S - STOP	Stop

- The Up/Down instructions move the current cursor up/down. If the current cursor is at the top/bottom of the section, it wraps around to the opposite edge of the section.
- The Previous/Next instructions move the current cursor left/right. If the current cursor is at the leftmost/rightmost column of the section, it wraps around to the opposite edge of the section.
- The Top/Left instructions move the current cursor to the top/left edge of the current vertical section.
- The Escape instruction changes the meaning of the next instruction. If the current section is the left-hand section, and the next instruction is Up/Down, the secondary cursor skips vertically to the next one bit, changing the row no. of both primary and secondary cursors. If the current section is the right-hand section, and the next instruction is Down, the distance to skip downward equals 2 raised to the power of the column no. If the next instruction is Next, the direction to skip is from left to right, and the distance to skip equals 2 raised to the power of the row no. If the next instruction is Up, the distance to skip (from left to right) equals 2 raised to the power of the current memory size. If the next instruction is Escape, the row no. of the left-hand section is set equal to the row no. of the right-hand section.
- The Read command copies the bit at the current cursor to the leftmost bit of Row 1. The Write command is like the Read command, except it copies that bit in the opposite direction.
- The Cursor Toggle instruction toggles the cursor between primary and secondary.
- The Vertical Section instruction toggles between the left-hand and right-hand sections.
- The XOR command performs a NOT operation on the leftmost bit of Row 1.

- The Jump command skips the next instruction if the leftmost bit of Row 1 is true (equals one). The Unconditional Jump command always skips the next instruction.
- The Stop command indicates that the current section of microcode should terminate.

There exist at least 3 levels of code. The lowest level is the microcode. The next higher level of code (call it the milli-code) probably has 16 or more possible instructions (I haven't designed it yet). Each line of code in the milli-code has an op-code, and the top section of the lower right quadrant takes the current milli-code op-code, treating it as an address, and looks up the location in the microcode to branch to for the current op-code.

## God Computer for Dummies

In my theory, the original universe is digital in nature (all ones and zeros). It has 3 dimensions of space, and one time dimension. The X-axis is the set of all possible computer designs. The Y-axis is the set of all natural numbers: 1, 2, 3, etc. The Z-axis is the set of all bits in a particular computer. This string of bits parallel to the Z-axis is initialized to Y.

Let's let X = some possible computer design, let's say it's computer design no. 123 (the 123rd computer design). Let Y = some extremely large number, expressed in binary notation, e.g. 1011100101001110100... (several thousand bits long). This means that the memory of computer design #123 is initialized to Y. With the particular computer I'm currently designing, the no. of bits in its memory = 2 raised to the power of (2 raised to the 16th power) = approx. 10 raised to the 20,000th power. Expressed in decimal notation, this no. of bits (its memory size) would be a one followed by 20,000 zeros. I haven't actually completely done away with the need for a Creator. A digital universe it seems to me is much simpler (and easier to create) than our own universe (even though our universe contains less than N subatomic particles, where N = 10 raised to the 200th power).

So, getting back to the computer I am designing, let's say it's computer design #123, and it's memory is initialized to Y. If we are very lucky, this particular string of bits parallel to the Z-axis (where X = 123 and Y = some large no. expressed in binary notation), will not only function as a working computer, but will also give rise to digital life forms, each life form with its own digital DNA. These life forms will evolve in complexity until they rival the human brain in complexity, at which point they will become sentient. Then these sentient digital life forms will modify a large portion of the memory of the digital computer in which they live, in order to create a computer simulation of our universe (running on computer design #123), complete with the big bang. This computer simulation would have to model up to N subatomic particles, where N = 10 raised to the 200th power (N = the no. of subatomic particles in our universe).

The code I am writing (the God Machine project) is a computer simulation of computer design #123, whose memory is initialized to Y (some large string of ones and zeros). I have to determine the value of Y, and the inner workings of computer design #123. The purpose of this code is to prove that computer design #123 actually works, and at least has the potential to give rise to digital life forms. I won't actually program the code that generates those life forms, but the computer simulation of computer design #123 will actually work, and enable human programmers to develop all sorts of computer programs running on the "hardware" of computer design #123.

# Looping and Branching

In the microcode, looping and branching is accomplished with the J, K, U and D commands, along with the P and N commands. At the outermost level in a given block of microcode, the current secondary bit in the left-hand section is in column 4. The U command is used for looping, and the D command is used for branching. The destination row of a jump (caused by the Escape command followed by U or D) is flagged with a one bit in the current column in the left-hand section. In the case of nested loops/if statements, the current column of innermost code blocks is greater than 4.

## Simple If Statement

Col 0-3	Col 4-7
1101 - J	0000
0110 - E	0000
0001 - D	0000
...	...
XXXX	1000

## Loop Statement

XXXX	1000
...	...
1100 - X	0000
1101 - J	0000
0110 - E	0000
0000 - U	0000

## If-Elseif-Else Statement

1101 - J	0000	
0110 - E	0000	
0001 - D	0000	
...	...	if block
0011 - N		
0110 - E	0000	
0001 - D	0000	
XXXX	1000	
...	...	
1101 - J	0000	
0110 - E	0000	
0001 - D	0000	
...	...	elseif block
0011 - N		
0110 - E	0000	
0001 - D	0000	
XXXX	1000	
...	...	else block
1110 - K	0000	
0010 - P	0100	

## Pico-Code

The Pico-Code is the blueprint for the GC's hardware. Even though each of the 16 possible microcode instructions theoretically takes the same amount of time (one clock-tick), it conceptually executes a unique block of pico-code. The pico-code hardware consists of an (almost) infinitely long tape of ones and zeros, plus a set of 16 registers and a 16-bit program counter (PC). Each register is 65,536 bits long. Register 0 is the accumulator (ACC). Register 1 is the data register (DAT). All pico-code instructions are stored in a buffer of size 32 KB. Each pico-code instruction is a multiple of 4 bits long. See Appendix A for the contents of the pico-code buffer.

## Pico-Code Instructions

Name	Op Code	Length	Description
DTX	0000	20	DAT = X; X is a signed 16-bit value
DTA	0001	20	DAT = 2 raised to the X-th power; X is a 16-bit value
DSH	0010	4	DAT = DAT >> 1
ACD	0011	4	ACC = ACC + DAT
ACZ	0100	4	ACC = 0
MOV	1000	12	REG X = REG Y; X,Y are 4-bit values
JMP	1001	20	PC = X; X is a 16-bit value
BRZ	1010	20	PC = X (if ACC = 0)
BRN	1011	20	PC = X (if ACC < 0)
JIN	1100	8	PC = REG Y; Y is a 4-bit value
TRD	1101	8	DAT[0] = Tape[REG Y] (read a bit from the tape)
TWR	1110	8	Tape[REG Y] = DAT[0] (write a bit to the tape)
RET	1111	4	Return or Halt

## High-Level Language

The high-level language included with the GC is a simplified version of Java, with a Lisp-like syntax. The user interface is a bare-bones, command prompt style setup, where the user enters expressions, and the expressions are evaluated and printed on the next line. When in insert mode, the user enters lines of text into the current source file. To exit insert mode, the user presses Enter twice in a row. To edit existing lines of text in the current source file, the user must use old-fashioned, line editor commands such as:

```
ch <s1>,<s2>
```

The ch command changes substring s1 into substring s2 on the current line.

## Data Structure

All data directly accessible from the high-level language is stored in data structures, each of which is called a qnode, which consists of 2 adjacent rows in the right-hand section. Each row is 65,520 bytes long (actually only 4096 bytes long in the Windows version of the GC, to save memory space). In each row, the first bit (leftmost bit) is 0 if this row is a pointer, and 1 if this row contains data. If the row contains data, the next bit determines if the row is a set, and if not, the next 3 bits determine the type of data, the next bit determines if the row is an operator, and if not, the next 16 bits determine the array length (0 to 65,000). If the row is an operator, the next 16 bits determine the op-code. If the array length is 0 and the data type is char, the row equals the null string. A pointer of zero is referred to as nil.

## Data Types

Length	Type
1	boolean
1	byte
2	short
4	int
8	long
8	float
2	char
N/A	void

## Operators

Display	Description	Types	Example
+	Addition	Numeric	+ a 48
-	Subtraction		
*	Multiplication		
/	Division		
%	Modulus	Integer	% num 10
-	Negation	Numeric	- count
+	Concatenation	char	+ s1 s2
==	Equals	boolean	== x y
<>	Not Equal		
<	Less Than		
>	Greater Than		
<=	Less Than or Equal To		
>=	Greater Than or Equal To		
=	Assignment	All	= x expr1
not	Not	boolean, Integer	not flag
and	And		and p q
or	Or		
xor	Exclusive Or		
:	Get Array Element	All, Integer	: arr 8
:=	Set Array Element		:= arr 14 expr
..	Dot Operator	Object	.. obj fld
?	If-Then	boolean, All	? flag x y
<<	Left Shift	Integer	<< n 3
>>	Right Shift		
+	Union	set	
-	Difference		
*	Intersection		
<=	Inclusion		
in	Membership		
++	Increment	Integer	++ count
--	Decrement		-- count
new	New Object	Object	new obj
atomic	Is an Atom	All	atomic q
pchild	Get Child from Qnode		pchild q
pnext	Get Next from Qnode		pnext q
cons	Construct Qnode		cons p q

## Statements

Name	Description	Example
if	if-elseif-else	if flag1 (block1) flag2 (block2) true (block3)
while	while loop	while flag (block)
do-while	do-while loop	do-while (block) flag
for	for loop	for i (m n) (block)
=	assignment	= x expr
++	increment	++ count
--	decrement	-- count
call	call procedure call method	p1 arg1 arg2 .. obj meth1 .. obj (meth1 arg1 arg2)
break	exit loop	
continue	skip to bottom of loop	
return	exit procedure exit function	return return expr
try	exception handling	try (block1) (excl evar1 (block2)... ) (block3)
throw	throw exception	throw obj

## Method Declarations

Name	Description	Example
proc	procedure	(proc (modifs) name (parms) locals (block))
func	function	(func (modifs) type name (parms) locals (block))
cons	constructor	(cons (modifs) class (parms) locals (block))
<b>tokens</b>		
modifs	modifier list	public, private, final, static
name	name of method	
type	function return type	
class	name of class	
parms	parameter list	int n; char ch; boolean flag
locals	local variable list	see above
block	statement list	stmt1; stmt2; stmt3;

## Class Declarations

Name	Description	Example
class	class declaration	(class (modifs) name (extends)(implements) body
interface	interface declaration	(interface (modifs) name (ext-interf) interf-body
<b>tokens</b>		
modifs	modifier list	public, private, final, static
name	name of class/interface	
extends	extends clause	extends class-name
implements	implements clause	implements interf1 interf2
ext-interf	ext-interf clause	extends interf1 interf2
body	class body	(fields)(methods)
interf-body	interface body	(fields)(method-hdrs)
fields	field declarations	((modifs) type fld1; (modifs) type fld2)
methods	method declarations	
method-hdr	method header	(proc (modifs) name (parms)) (func (modifs) type name (parms))

## Sample Source File

```
package (pkg1 pkg2)
(
  import (pkg3 pkg4 class1)
  import (pkg5 interf1)
  import (pkg6 pkg7 *)
)
class-decl1
class-decl2
interf-decl1
interf-decl2
```

# Appendixes

## Appendix A - Pico-Code Buffer

### Pico-Code Instructions

Name	Op Code	Length	Description
DTX	0000	20	DAT = X; X is a signed 16-bit value
DTA	0001	20	DAT = 2 raised to the X-th power; X is a 16-bit value
DSH	0010	4	DAT = DAT >> 1
ACD	0011	4	ACC = ACC + DAT
ACZ	0100	4	ACC = 0
MOV	1000	12	REG X = REG Y; X,Y are 4-bit values
JMP	1001	20	PC = X; X is a 16-bit value
BRZ	1010	20	PC = X (if ACC = 0)
BRN	1011	20	PC = X (if ACC < 0)
JIN	1100	8	PC = REG Y; Y is a 4-bit value
TRD	1101	8	DAT[0] = Tape[REG Y] (read a bit from the tape)
TWR	1110	8	Tape[REG Y] = DAT[0] (write a bit to the tape)
RET	1111	4	Return or Halt

**Note:** the following code is in an early stage of development and is subject to change.

Micro	Addr.	Op	Data	Description
XOR	0008	ACZ		ACC = 0
	000C	TRD	0	DAT[0] = Tape[0]
	0010	ACD		ACC = ACC + DAT[0] = DAT[0]
	0014	BRZ	8048	IF ACC = 0 GOTO 8048
	0028	ACZ		ACC = 0
	002C	DTX	0000	DAT = 0
	0040	TWR	0	Tape[0] = 0
	0044	RET		
	8048	DTX	0001	DAT = 1
	005C	TWR	0	Tape[0] = 1
	0060	JMP	0000	GOTO NXT
QUAD	0200	DTA	0010	DAT = 2 <sup>16</sup>
		ACZ		
		ACD		ACC = 2 <sup>16</sup>
		MOV	1C	DAT = REG C
		ACD		
		MOV	C0	REG C = REG C + 2 <sup>16</sup>
		MOV	2C	
		DTX	4	
		MOV	04	
		DTX	1	
		MOV	31	REG 3 = 1
	0220	TRD	2	DAT[0] = Tape[2]
		TWR	3	Tape[3] = DAT[0]
		DTX	1	
		MOV	02	
		ACD		

		MOV	20	REG 2++
		MOV	03	
		ACD		
		MOV	30	REG 3++
		DTX	FFFF	
		MOV	04	
		ACD		
		MOV	40	REG 4--
		BRZ	0000	IF ACC = 0 GOTO NXT
		JMP	0220	
NXT	0000	DTA	0010	DAT = 2 <sup>16</sup>
		ACZ		
		ACD		ACC = 2 <sup>16</sup>
		MOV	1C	DAT = REG C
		ACD		
		MOV	C0	REG C = REG C + 2 <sup>16</sup>
		JMP	0100	GOTO DECO
DECO	0100	DTA	0014	DAT = 2 <sup>20</sup>
		ACZ		
		ACD		
		MOV	20	REG 2 = 2 <sup>20</sup>
		DTA	0004	
		ACD		ACC = 2 <sup>20</sup> + 16
		MOV	F0	Reg. F:11,1 = ACC
		DTX	0008	
		MOV	61	REG 6 = 8
	8500	TRD	C	DAT[0] = Tape[C] = [10,0]
		ACZ		
		ACD		ACC = DAT
		BRZ	9000	IF ACC = 0 GOTO 9000
		MOV	16	DAT = 8,4,2,1 – must init REG 6 = 8 ok
		MOV	91	REG 9 = DAT
		DSH		DAT >> 1
		MOV	61	REG 6 = 4,2,1,0
		MOV	06	ACC = REG 6
		BRZ	9100	
		MOV	12	
		DSH		
		MOV	21	REG 2 >> 1 = 2 <sup>19</sup> ,18,17,16
		MOV	02	ACC = REG 2
		MOV	1F	DAT = REG F
		ACD		ACC = REG F + REG 2
		MOV	F0	REG F = ACC
	9000	MOV	0C	
		DTX	0001	
		ACD		ACC++
		MOV	C0	REG C++
		JMP	8500	
	9100	MOV	0F	do a JIN on 16 bits to left of Tape[F]...
		DTX	FFF0	
		ACD		ACC = REG F – 16
		MOV	70	REG 7 = start of tape
		MOV	37	save start of op-code

		ACZ		
		MOV	20	SUM = 0
		DTX	16	
		MOV	61	REG 6 = 16
	9200	TRD	7	DAT[0] = Tape[7]
		MOV	91	REG 9 = curr bit
		MOV	12	REG 2 = SUM
		ACZ		
		ACD		
		ACD		ACC = 2*REG 2
		MOV	19	DAT = curr bit
		ACD		ACC = ACC + curr bit
		MOV	20	REG 2 = SUM
		MOV	06	
		DTX	FFFF	
		ACD		ACC = REG 6 - 1
		MOV	60	REG 6--
		BRZ	9300	
		MOV	07	
		DTX	1	
		ACD		
		MOV	70	REG 7++
		JMP	9200	
	9300	MOV	73	restore REG 7 to col. 0
		DTX	FFFC	
		MOV	0C	
		ACD		
		MOV	C0	REG C = REG C - 4; back to col. 0
		JIN	2	do the JIN