
# The Cuneiform Tablets of 2015

Long Tien Nguyen, Alan Kay

This paper was presented
at the Onward! Essays track
at SPLASH 2015 in Pittsburgh,
PA, October 29, 2015

VPRI Technical Report TR-2015-004

Viewpoints Research Institute, 1025 Westwood Blvd 2nd flr, Los Angeles, CA 90024 t: (310) 208-0524

# The Cuneiform Tablets of 2015

Long Tien Nguyen
University of California, Los Angeles
Viewpoints Research Institute
Los Angeles, CA, USA
long@nguyen.bz

Alan Kay
University of California, Los Angeles
Viewpoints Research Institute
Los Angeles, CA, USA
alan.kay@vpri.org

## Abstract

We discuss the problem of running today's software decades, centuries, or even millennia into the future.

***Categories and Subject Descriptors*** K.2 [*History of Computing*]: Software; K.4.0 [*Computers and Society*]: General

***Keywords*** History; preservation; emulation

## 1. Introduction

In the year 1086 AD, King William the Conqueror ordered a survey of each and every shire in England, determining, for each landowner, how much land he had, how much livestock he had, and how much money his property was worth [2]. The result of this survey, known as the Domesday Book, is the oldest public record in England and gives an extremely detailed glimpse into the sociopolitical structure of England in the 11th century, a time of great change and upheaval [8].

In the year 1984 AD, the BBC, to commemorate the 900th anniversary of the Domesday Book, commissioned the creation of the BBC Domesday Project, an interactive multimedia system intended to be a Domesday Book for the 20th century. Over 1 million people from all over the United Kingdom participated in creating this historical record for posterity. Each contributor was free to choose whatever he or she wished to record; the total sum of the contributions created a snapshot of everyday British life in 1980s. The data of the Domesday Project was stored onto two LV-ROM discs, a custom optical disc format based on the LaserDisc designed specifically for the Domesday Project. The LaserDisc format itself was cutting-edge at the time, as it was the first commercially-available optical disc technology. The Domesday Project's software system was designed to run on an Acorn BBC Master computer, modified with custom hardware designed specifically for the BBC. The Domesday Project was completed and published in the year 1986 AD [10] [36].

In the year 2002 AD, 16 years after the BBC Domesday Project was completed, the LV-ROM technology and the BBC Master computer had long been obsolete. LV-ROM readers and BBC Masters had not been manufactured for many years, and most existing readers and computers had been broken, lost, or discarded. Though the LV-ROM discs containing the Domesday Project were (and still are, as of 2015 AD) intact and readable, the rapid disappearance of hardware that could read the LV-ROM discs and execute their contents jeopardized the survival of the Domesday Project [29]. To save the Domesday Project, the University of Michigan partnered with the University of Leeds to initiate the CAMiLEON Project[1] to develop strategies and techniques for digital preservation, and use them for preserving the Domesday Project [13]. The CAMiLEON staff obtained a broken Domesday Project device, repaired it, and used it to extract the contents of the LV-ROM discs. Then the CAMiLEON staff developed an emulator for the BBC Master computer and the LV-ROM disc reader that would execute the contents of the LV-ROM discs, enabling the user to experience the Domesday Project on a modern computer [28]. Most of the documentation and source code of the Domesday Project had been lost, and the CAMiLEON staff was forced to reverse engineer the hardware and the contents of the LV-ROM discs [9].

Though they successfully produced a prototype of a working emulator for Microsoft Windows that executed the Domesday Project, the CAMiLEON project folded after its funding ended; the CAMiLEON website ironically went offline not soon afterwards. The Domesday Project website also went offline after one of its key people, Adrian Pearce, suddenly passed away [12]. Most of the fruits of the CAMiLEON Project were not released to the public; the images from *one* of LV-ROM discs were uploaded to the BBC website and the papers from the website were successfully

[1] Creative Archiving at Michigan and Leeds: Emulating the Old on the New

saved by the Internet Archive. However, an accurate emulation of the original multimedia experience is not available to the public, and lies somewhere in the decaying private hard drives of the former CAMiLEON staff; its future yet lies in jeopardy. By contrast, the Domesday Book of 1086 AD, stored in the National Archives of the United Kingdom in Kew, is in excellent condition and can be viewed by any visitor with special permission [29]. The full text (in Latin) and translation of the Domesday Book can be found online on several websites. Reproductions are widely available in print in countless libraries and bookstores. It may be that our descendants shall know more of England in 1086 AD than of England in 1986 AD. In this essay, we will try to explore the ways we can prevent our era from becoming, in the words of Vint Cerf, a Digital Dark Age.

## 2. Prehistory

The authors of this essay first met in October of 2011, when the first author attended a seminar led by the second author for the introductory computer science course for freshmen at UCLA. In this lecture, the second author gave a personal tour of the field of computer science and its history, and explored the relationship between computer science and mathematics, science, engineering, and art. We discussed the immature state of computer science today, comparable to the state of civil engineering during Ancient Egypt, when the pyramids were built. The second author opined that a significant barrier to progress in computer science was the fact that many practitioners were ignorant of the history of computer science: old accomplishments (and failures!) are forgotten, and consequently the industry reinvents itself every 5-10 years.

This interaction got us interested in preserving computing history for posterity, a problem that is catastrophically overlooked by the institutions that are supposed to be preserving today's heritage. For example, the second author has repeatedly attempted to persuade, with little success, the Library of Congress to create a concerted and systematic effort to preserve computer software and a catalog of computer architectures (consisting of architecture specifications in English and "reference implementations"). The Computer History Museum, run by "hardware guys," has an extensive and impressive collection of vintage hardware of all kinds and from all eras, but it sits there as a collection of lifeless beige boxes.

Our first experience at historical preservation took place in August of 2013. At the time, the Kleinrock Internet History Center (KIHC) at UCLA started work on the restoration of the systems that powered the very first ARPANET node at UCLA: the SDS Sigma 7 computer that served as the host and the Interface Message Processor (IMP) that served as the router for the host. However, the project ran into several difficulties: first, the 44-year old Sigma 7 and IMP were in poor condition, and it was difficult to determine what repairs were necessary to bring the machines back to life. Secondly, and more importantly, the software for both the Sigma 7 and IMP was missing, and the 40-year old backup tapes that were found did not work (although later one of the original engineers on the IMP, Dave Walden, managed to find a complete copy of the IMP source code in his personal archive). The project was put on indefinite hiatus.

Our experiences show that more attention is given to hardware than to software, in spite of hardware being merely a "lifeless body" without the "immortal soul" of software to bring it to life. The "essence" of the ARPANET, and of many other systems, survives even if all the machines that ran the original software are gone, as long as the software still survives. After the KIHC project was shelved, we did some thinking about what it would take to allow a program to be usable in the far future, when the hardware "bodies" that the program once inhabited are long dead and gone.

## 3. Storage Media

The first problem we have to worry about is the deterioration of storage media. As we have mentioned earlier, the 40-year-old backup tapes for the Sigma 7 and IMP machines were no longer readable. All storage media currently in use have a fairly short lifespan, which varies widely depending on various factors, including storage conditions and usage patterns. For example, the National Media Laboratory reports that CD-Rs can last from 5 to an estimated 100 years, and magnetic tape can last from 10 to 50 years, depending on storage conditions [1]. Therefore, the preservation of digital data using existing storage media requires constant maintenance of the digital archive by transferring data from an old medium to a new medium every few years. This approach, in our opinion, is too risky, since it requires the institution doing the preservation to continuously exist and receive funding.

A better solution was developed by the Long Now Foundation in conjunction with the Los Alamos National Laboratory and Norsam Technologies. Microscopic text is etched onto a disk made out of nickel; the resulting "Rosetta disk" can be read with a microscope at 1000x magnification [24]. A Rosetta disk is estimated to last from 2,000 to 10,000 years; a 3-inch disk can contain about 350,000 pages of text [23]. Other institutions are also developing extremely long-term data storage media [14]. As we are computer scientists, we shall leave this "piece of the puzzle" to the materials scientists, and assume that they have developed such a "Rosetta disk" for us. Now we must figure out what to put on it.

## 4. The Simplest Approach

At first, the reader may think, why can't we just put the program we want to send to the future on the disk, plus the documentation for the machine that the program runs on? The "archaeologist of the future" would read the documentation, write an emulator for the machine, and run the program. Unfortunately, this naive scheme makes the archaeologist of

the future do a lot of work, because real computer architectures have a lot of complexity. Say, for example, we wanted to emulate a bog standard "Windows on Intel x86" computer. The Software Developer's Manual [18] for the Intel x86 architecture has a total of 3603 (!) pages, and that's just the CPU alone! Even 8-bit microcomputers from the 1980s have a lot of complexity. For example, the Commodore 64 Programmer's Reference Guide [6] is 514 pages long. The archaeologist of the future would have to read and understand the whole document, and then write and debug an emulator, taking into account all the edge cases and quirks of the behavior of a real-world computer. Instead, we should do most of the "heavy lifting" ourselves, and design a scheme such that the archaeologist of the future can get the program working in a single afternoon. But what should we do today, so that the archaeologist of the future only needs to do a "fun afternoon's hack?" We were inspired by several ideas.

## 5. Some Food for Thought

Guy Steele begins his keynote lecture Growing a Language [33] at OOPSLA 1998 with a very limited vocabulary. He adds more complex words to his vocabulary by defining them in terms of words that already exist in his vocabulary. These new words, in turn, can be used to define even more complex words down the line. This scheme vividly makes the case for programming languages that consist of small kernels that can be flexibly extended by their users.

The idea of building a language in this manner is not new. In the 1950s, Hans Freudenthal devised a universal language to communicate with aliens from outer space called LINCOS [11]. In his language, he tried to move from visual patterns to mathematics and then attempted to build everything from logic, like Russell and Whitehead attempted in their Principia Mathematica [31]. Freudenthal makes the assumption that any aliens that are sufficiently intelligent to be able to receive radio messages from space must also have discovered mathematics. Freudenthal starts with the concept of a natural number, which can be represented in a self-evident manner as a series of repeated radio "beeps" separated by periods of silence. 0 is represented as no beeps, 1 is represented as one beep, 2 is represented as two beeps, and so on. Thus, his "message to aliens" starts by defining the equals, less-than, greater-than, and arithmetic symbols for natural numbers. Each new concept is defined in terms of concepts previously defined; Freudenthal goes on to define real numbers, variables, propositional logic, and first-order logic. Then, he defines the concepts of time, behavior, and physics. Of course, we are not doing anything as ambitious as sending a message to aliens, so our design does not need to be so sophisticated.

While Freudenthal was designing LINCOS, another compelling example of a powerful universal language with a very compact explanation and bootstrap was Lisp by John McCarthy [27]. In the original formulation of Lisp, McCarthy showed how a powerful programming language could be built up starting from only 9 primitives: *atom*, *car*, *cdr*, *cons*, *eq*, *quote*, *cond*, *lambda*, and *label*. Using only these 9 primitives, a fully-functioning interpreter for Lisp could be written in Lisp itself. A thought the second author had back then was perhaps every piece of digital media should be prefaced by the simplest explanation possible of a virtual machine that could run the rest of the media, and that Lisp could be an excellent candidate.

Lisp, along with Simula [7], arguably the first truly object-oriented programming language, became the foundation of thinking about object-oriented programming, especially dynamic late-bound object-oriented programming, and the seeds of these ideas were incorporated into the Smalltalk systems [22] at Xerox PARC in the 1970s. A feature of the revolutionary Xerox Alto personal computer [34] was that it required a very powerful but very compact software system to make it into what would be considered a very early form of "modern interactive personal computing." As further personal computers were developed at PARC, Smalltalk was made portable by having each "bag of bits," called an "image," that constituted an entire system also carry the microcode for each machine Smalltalk was to run on [22]. When an image was brought to a particular machine, the appropriate microcode was extracted and executed to create a Smalltalk virtual machine[2]. All the rest of the "bag of bits" was completely machine independent. For example, the microcode needed for the Alto was 1024 instructions of 32 bit words: 4 kilobytes. One of the versions of Smalltalk done for the portable NoteTaker computer ran on an off-the-shelf chip (an early version of the 16-bit 8086) and the bootstrap was done with 6 kilobytes of 8086 machine code playing the role of the "microcode." All other parts of the system were machine independent, and the system plus interesting media could all fit into less than 512 kilobytes [22].

This led to the somewhat interesting conclusion that one could store media as a "process" if the system that created it were carried along, and if the "microcode" for the system were also supplied.

Furthermore, if the "microcode" were thought through a little more carefully, it would be much more compact to describe what it did than to describe how to make a Lisp. The idea then would be to have a Lisp-like system, or in this case Smalltalk, in the rest of the bag, and the real beginning part of each bag should be the simplest description of the simplest machine that would bring the rest of the bag to life. An important motivation back in the 1970s, which carries through to the present essay, was that a good goal would be to entice the programmers of the future into revitalizing a piece of media by showing them that it could be "an afternoon's hack." Thus we wanted to avoid a lengthy description in the

[2] Very much modeled after the hardware of the Burroughs B5000 computer [3], which introduced a HLL "machine-code" with bytecodes, descriptors (protected pointers), etc.

beginning, and we wanted to avoid requiring the archaeologist of the future to have to do a lot of work before seeing whether it would be worthwhile. The neat thing would be that the tools to modify, extend, and improve the system could be bundled along with the system as well.

It's worth looking a little deeper into this approach, which is at least as "psychological" as it is technical. One question is whether any of the formal semantics techniques popular today will survive well enough so that they could be referred to. The second author began his computing career in 1961, and more than 50 years later, none of the formal semantics techniques from the 1960s has survived well enough to be used without reteaching. Teaching formal semantics to the archaeologist of the future using English and perhaps logic would be a hefty document, and for most programmers would not be regarded as a "fun afternoon's hack," or even a "fun month's hack." We fear that today's favorite formal semantics techniques, such as denotational semantics, may not survive even 50 years hence. This is also, sadly, the case for Lisp (though it is still worth understanding deeply, just because it was so beautiful and powerful). But most programmers today don't understand just how to bootstrap a Lispish kind of system.

Most programmers today haven't written a machine code program and likely have a bit of a fuzzy idea of what machine code is like. However, universal machines can be both simple and tiny with respect to features, and it might be possible to explain the semantics in a single tempting page. The idea of values, labeling them, transforming them, looping, sequencing, and so forth, will probably hang on, and these could be viable contexts for a one-page description. Note that defining a simple machine with memory locations, simple operations, and a simple instruction format, is counter to perennial desires for abstraction and generality. However, our aim to be both very specific and very brief, and this is quite justifiable in part because the internals of our media document can be brought to full life from following just such a simple one-page description.

In summary, our "message to the future" should consist of two parts: a "one-page description" of a simple virtual machine, and a program that will run on the simple virtual machine.

## 6. Alternative Views

Now, we could also just describe how a NAND gate works and describe how the virtual machine works in terms of NAND gates! This could be a list of logic equations or a schematic diagram. An example of such a project is Brian Silverman's Visual 6502 [20]. He depackaged a 6502 CPU, extracted the schematics from photographs of the die, and wrote a gate-level simulator for the schematics, resulting in a working virtual 6502. Though a NAND gate is easy to understand, a schematic of NAND gates would be harder to use, because the archaeologist of the future would have to transcribe the schematics, and either write a logic simulator or build a hardware circuit (using whatever technology she will have in the future) to run the program. However, a schematic diagram could serve as an alternative specification of the virtual machine.

Why would we need alternative specifications? To guarantee that a "message to the future" is still comprehensible to future generations, we should write it in several ways. Ancient Egyptian hieroglyphics were decoded thanks to the Rosetta Stone, which contained the same text written in three scripts: Ancient Egyptian hieroglyphics, Demotic script, and Ancient Greek. Because the Ancient Greek language was known to modern scholars, they were able to compare the Egyptian text and the Greek text to figure out the meaning of the hieroglyphs. Likewise, if we specify the simple virtual machine several different ways, it will increase the likelihood of the archaeologist of the future being able to understand how it works. Our first specification of the simple virtual machine uses English prose and diagrams. Our assumption is that basic mathematical and computational concepts, such as binary numbers, Boolean logic, arithmetic, and Turing machines will still be known. It seems to us that if knowledge of these concepts were lost then civilization has collapsed to the point where there would be no computers at all! We could then use the logic equations or schematic diagram, as described in the previous paragraph, as a second specification of our simple virtual machine. This could be used to verify the first specification. Figuring out other ways to describe the simple virtual machine is an open problem that we haven't solved yet. Perhaps there is a more accessible formal semantics technique that is yet to be invented.

Currently, we only have one specification of the simple virtual machine, but we are working on figuring alternative specifications for the next version of our system.

## 7. The Cuneiform System

Inspired by the ideas above, we set out to build a system for software preservation, which we christened Cuneiform. It works as follows: in the present day, a preservationist using the Cuneiform tools preserves a software program for posterity. Once a program has been packaged with Cuneiform, an archaeologist of the future spends a fun afternoon "hacking" on an emulator that brings the packaged program back to life. The packaged programs will look and behave exactly as they did in the past. Furthermore, the archaeologist can do more than just execute the packaged program: the tools bundled with the program allow the archaeologist to view the program's documentation, inspect the program's behavior, and even modify or adapt the program.

A program packaged using Cuneiform consists of three parts: the label, the header, and the program itself. We describe the parts in the rest of this essay.

## 8. The Label and the Header

Our packaged program is recorded as a sequence of bits, also called a bitstream, on a suitable digital storage medium. As we mentioned earlier, we are not solving the problem of designing a long-lasting digital storage medium in this essay; we assume that such a medium already exists, such as the Rosetta disk described above.

Suppose we have a Rosetta disk, and the bitstream of the packaged program is stored on the obverse side. On the reverse side is a label with the following information:

1. A short description of the preserved program.
2. Instructions on how to read the data from the storage medium. These instructions are specific to the storage medium used.
3. Instructions on how to interpret the bitstream.

Because our label should be readable with the naked eye, it cannot contain too much information. Therefore, we propose the following scheme. The bitstream begins with a header consisting of a two-dimensional, black-and-white bitmap containing a document that explains how to interpret the rest of the bitstream. Each bit in the header represents one pixel, where 0 is white and 1 is black. The label just needs to explain how to interpret the header as a bitmap in a few sentences plus a simple diagram. (The width and height of the document is specified in the label.) A sample label is shown in figure 1.

## 9. The Chifir Virtual Machine

As we have mentioned earlier, real computer architectures have too much complexity, and we shouldn't make the archaeologist of the future implement an emulator (or even worse, a hardware version) of a real machine. Fortunately, any Turing-complete machine can emulate any other Turing-complete machine. Therefore we propose the following scheme, illustrated in figure 2, which can be described as "one emulator running on top of another emulator." In the present, we have a program that is written for a specific computing platform, which we labeled Original Platform. At preservation time, we write an emulator for the original platform that will run the program to be preserved. The original platform emulator is designed to run on a simple virtual machine, so simple that it can be described in one page. The header with the black-and-white bitmap contains the complete one-page specification for this virtual machine. Then, in the future, an archaeologist will read this specification and, in a fun afternoon, use it to hack together an emulator for this simple virtual machine. The archaeologist, of course, will implement this simple virtual machine emulator using whatever programming language and computing platform she will have in the future, which we labeled as Future Platform in figure 2.

The only two requirements for the design of this simple virtual machine are:

1. It can be described in a single Letter or A4-sized page using English and diagrams. A "one-pager" has a nice psychological quality of compactness and elegance to it; we were inspired by the half-page Lisp metacircular evaluator in the Lisp 1.5 manual [27].
2. It can be implemented in a single afternoon by a reasonably competent programmer.

We think that trying to design a "universal" virtual machine to serve as the simple virtual machine is a bad idea, because trying to ensure compatibility with the entire design space of computer architectures will make the resulting "universal virtual machine" very complicated. In our opinion, this is the mistake of van der Hoeven et al.'s Universal Virtual Computer for software preservation [15]. They tried to make the most general virtual machine they could think of, one that could easily emulate all known real computer architectures easily. The resulting design [25] has a segmented memory model, bit-addressable memory, and an unlimited number of registers of unlimited bit length. This Universal Virtual Computer requires several dozen pages to be completely specified and explained, and requires far more than an afternoon (probably several weeks) to be completely implemented.

For our prototype implementation, we preserved Smalltalk-72 [32], famous as the very first Smalltalk system. Smalltalk-72 ran on the Xerox Alto, whose architecture was a modified Data General NOVA. For the simple virtual machine, we experimented with several architectures, including stack machines, accumulator machines, and two-address machines. Our current design, which sacrifices fast execution and efficient memory usage for extreme simplicity, is a three-address machine which we called Chifir[3]. Our simple virtual machine is somewhat tailored to the Data General NOVA architecture and could also be used for other 16-bit minicomputer architectures of the 1960s and 1970s. A computing platform that is radically different from the Data General NOVA would probably need another design for its simple virtual machine.

A *word* in the Chifir machine has a length of 32 bits. The Chifir machine has the following state:

1. A memory $M$ that consists of $2^{11}$ = 2,097,152 words.
2. A program counter $PC$ with a length of 32 bits.

A Chifir instruction consists of an opcode $O$ and three operands $A$, $B$, and $C$. An operand may only refer to a memory location. All instructions must take three operands, however, some instructions may not use all three operands.

There are 15 instructions, shown in table 1. The notation *M[X]* denotes the $X$th word of memory. The operator $\leftarrow$ represents assignment, where the contents of the operand on the left are replaced with the contents of the operand on the right. Arithmetic and comparison instructions treat words as unsigned integers from 0 to $2^{32}$-1.

[3] A highly-caffeinated Russian tea.

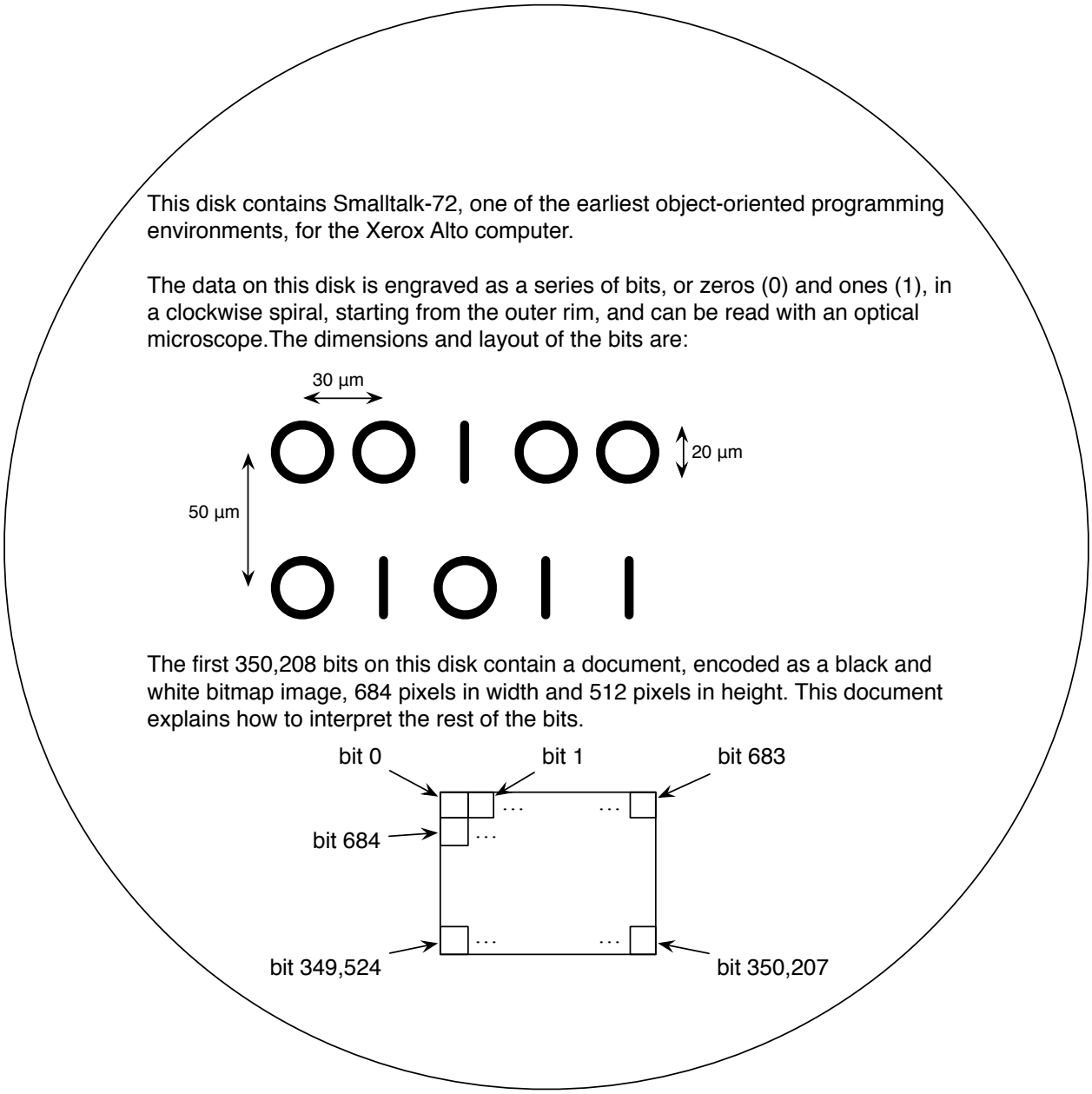



**Figure 1.** A sample label

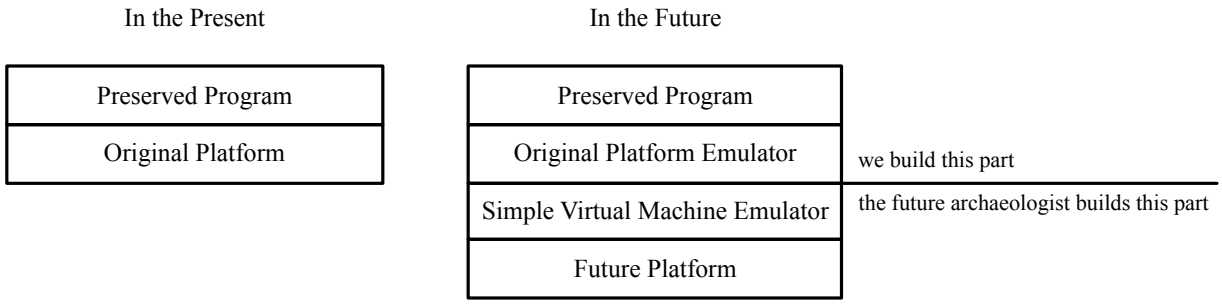



**Figure 2.** The Cuneiform architecture

A sample header, containing a Chifir virtual machine specification, is shown in figure 3.

As we have mentioned before, Chifir is just an incarnation of the idea of the simple virtual machine, tailored to emulate a specific platform. Since the Xerox Alto does not have signed integers, floating point numbers, or many other features, Chifir does not instructions for signed integer or floating point arithmetic. To emulate other platforms radically different from the Xerox Alto, we would need a different kind of simple virtual machine.

## 10. After the Afternoon Hack

Perhaps the archaeologist of the future will want to do more with the preserved program than just run it. Enthusiasts of today recreate historical machines such as the Antikythera mechanism and the Difference Engine. We can imagine that enthusiasts of the future may want to build an Atari 2600 or a PlayStation, to see how their ancestors entertained themselves. Therefore, we can include the complete documentation and specification of the computing platform inside the packaged program, along with some sort of reader application. Once the archaeologist of the future writes their "afternoon hack" and gets the preserved system running, their interest may be sufficiently piqued to try a more time-consuming project.

The archaeologist may also want to study how the preserved program works. Therefore, we should bundle the source code and development environment for the preserved program if these are available. However, the source code is not available for most software, and we should make it easy for the archaeologist to debug and reverse engineer the preserved program. She may also want to extract parts of the preserved program, such as the bitmaps from a HyperCard stack or the music from a video game, and we should make it easy for her to build the tools to do so.

Finally, it is conceivable that the archaeologist may want to modify the preserved program to adapt it to future needs, extend its functionality, or even create a "remix." A modern-day example of such a remix is Malek Jandali's Echoes from Ugarit, a piano arrangement of the Hurrian Songs [21]. Dating from 1400 BC, the Hurrian Songs are the oldest surviving collection of written music in the world. We would like to encourage the Malek Jandalis of the future to improve on our preserved program in ways we could have never imagined.

It is difficult to imagine what kinds of tools the archaeologists of the future may have, but we can help "interface" their futuristic tools with our present-day tools. After the Antikythera mechanism was discovered, it took us a few decades to figure out what its purpose was and how it worked. Imagine how much easier it would have been if the Greeks had tightly bundled a detailed manual and maintenance tools with the mechanism! We bundle a self-contained development environment with the preserved program. This development environment also runs on the Chifir virtual machine used to host the original platform emulator. When the archaeologist of the future launches the Cuneiform package (after writing the "afternoon hack"), she is presented with a choice of going directly to the preserved program or to the development environment. The archaeologist of the future would then be able to use the development environment to read the documentation, study and play with the source code, and most importantly, write tools to interface the preserved program and its components with the tools she will have in the future.

## 11. A Crutch for Preservationists

Though we could take an existing emulator and port it to our Chifir virtual machine, we found this approach to be unsatisfactory because most emulators are tightly coupled to a specific computing platform. Since emulation causes a slowdown of 2 to 4 orders of magnitude, emulators must take advantage of all the specifics and quirks of a certain platform in order to run fast enough on today's machines. For example, SheepShaver, an emulator for PowerPC Macintoshes [4], is tightly coupled to Unix systems on x86. It uses a JIT compiler to translate PowerPC instructions to x86 instructions and implements a custom memory manager on top of libsigsegv for performance reasons. This custom memory manager is a nightmare to debug and modify, where most changes cause Heisenbugs that sometimes disappear when a debugger is attached. To port SheepShaver, we would need to retarget a C or C++ compiler to our Chifir virtual machine, remove the JIT compiler and implement a naive PowerPC interpreter, and port the POSIX environment, libc, SDL, and other libraries

The rest of the bits on this disk contain a program that emulates a Smalltalk-72 system, as well as a development environment that will allow you to inspect, modify, or extend the Smalltalk-72 system and its emulator. To run this program, you need to write an emulator for a virtual computer described in this document.

The state of the virtual computer consists of **M** and **PC**.
**M** is a memory consisting of 2,097,152 words, where each word is 32 bits wide. The rest of the bits on this disk, starting from bit 350,208, are broken up into 32-bit words, where the most-significant bit is encountered first. These 32-bit words are loaded into memory starting at memory address 0.
**PC** is a program counter, 32 bits wide, which contains the memory address of the currently executing instruction. Execution begins at memory address 0.

An instruction consists of 4 words, where the 1st word is the opcode and the remaining 3 words are operands. Each operand specifies a memory address. Some instructions do not use all 3 operands.

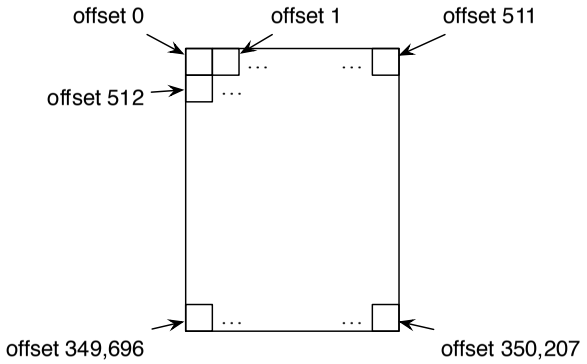
Arithmetic operations treat words as unsigned integers from 0 to $2^{32}$-1. If the result of an arithmetic operation cannot fit in 32 bits, then only the lowest 32 bits are preserved.

**A**, **B**, and **C** denote the first, second, and third operands, respectively.
The notation **M[X]** denotes the **X**th word of memory.
The symbol ← represents assignment, where the contents of the operand on the left are replaced with the contents of the operand on the right.

| Opcode | Semantics |
|---|---|
| 1 | PC ← M[A] |
| 2 | If M[B] = 0, then PC ← M[A] |
| 3 | M[A] ← PC |
| 4 | M[A] ← M[B] |
| 5 | M[A] ← M[M[B]] |
| 6 | M[M[B]] ← M[A] |
| 7 | M[A] ← M[B] + M[C] |
| 8 | M[A] ← M[B] – M[C] |
| 9 | M[A] ← M[B] × M[C] |
| 10 | M[A] ← M[B] ÷ M[C] |
| 11 | M[A] ← M[B] modulo M[C] |
| 12 | If M[B] < M[C], then M[A] ← 1, else M[A] ← 0 |
| 13 | M[A] ← NOT(M[B] AND M[C]) |
| 14 | Refresh the screen |
| 15 | Get one character from the keyboard and store it into M[A] |

**PC** is incremented by 4 after each instruction is executed, except for instructions with opcode 1 and 2.

The virtual computer has a black and white bitmap display. Each pixel is represented by a single 32-bit word, where 0 represents a black pixel and $2^{32}$-1 represents a white pixel. The bitmap display buffer starts at memory address 1,048,576 and is 512 pixels in width and 684 pixels in height, laid out as shown in the diagram below:




To get the memory address of a pixel, add the offset for that pixel to the starting memory address 1,048,576. The screen is refreshed when the screen refresh instruction (opcode 14) is executed. This instruction blocks until the screen has been refreshed.

The virtual computer has a keyboard. To read a character from the keyboard, the keyboard instruction (opcode 15) is executed. If a character from the keyboard is available, it immediately returns, storing the character code into **M[A]**; otherwise, it blocks until a key is pressed. The character codes for the keyboard are listed below:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 8 | Back | 43 | + | 57 | 9 | 71 | G | 85 | U | 99 | c | 113 | q |
| 10 | Up | 44 | , | 58 | : | 72 | H | 86 | V | 100 | d | 114 | r |
| 13 | Enter | 45 | - | 59 | ; | 73 | I | 87 | W | 101 | e | 115 | s |
| 32 | Space | 46 | . | 60 | < | 74 | J | 88 | X | 102 | f | 116 | t |
| 33 | ⇧ | 47 | / | 61 | = | 75 | K | 89 | Y | 103 | g | 117 | u |
| 34 | Hand | 48 | 0 | 62 | > | 76 | L | 90 | Z | 104 | h | 118 | v |
| 35 | # | 49 | 1 | 63 | Right | 77 | M | 91 | [ | 105 | i | 119 | w |
| 36 | () | 50 | 2 | 64 | Smile | 78 | N | 92 | \ | 106 | j | 120 | x |
| 37 | Eye | 51 | 3 | 65 | A | 79 | O | 93 | ] | 107 | k | 121 | y |
| 38 | ○ | 52 | 4 | 66 | B | 80 | P | 94 | ↑ | 108 | l | 122 | z |
| 39 | ‘ | 53 | 5 | 67 | C | 81 | Q | 95 | ← | 109 | m | 123 | { |
| 40 | ( | 54 | 6 | 68 | D | 82 | R | 96 | ‾ | 110 | n | 124 | ’ |
| 41 | ) | 55 | 7 | 69 | E | 83 | S | 97 | a | 111 | o | 125 | } |
| 42 | ✱ | 56 | 8 | 70 | F | 84 | T | 98 | b | 112 | p | 126 | ? |

**Figure 3.** A sample header

**Table 1.** Chifir instruction set

| Opcode | Semantics |
|---|---|
| 1 | PC ← M[A] |
| 2 | If M[B] = 0, then PC ← M[A] |
| 3 | M[A] ← PC |
| 4 | M[A] ← M[B] |
| 5 | M[A] ← M[M[B]] |
| 6 | M[M[B]] ← M[A] |
| 7 | M[A] ← M[B] + M[C] |
| 8 | M[A] ← M[B] - M[C] |
| 9 | M[A] ← M[B] × M[C] |
| 10 | M[A] ← M[B] ÷ M[C] |
| 11 | M[A] ← M[B] modulo M[C] |
| 12 | If M[B] <M[C], then M[A] ← 1, else M[A] ← 0 |
| 13 | M[A] ← NOT(M[B] AND M[C]) |
| 14 | Refresh the screen |
| 15 | Get one character from the keyboard and store it in M[A] |

that the emulator needs. Our experience is that writing a new emulator is easier than trying to decouple the workings of an existing emulator from the platform it runs on. Since we assume that computers of the future will be much faster, we don't care about optimizing our emulator's performance. We just build a naive implementation, which is far simpler than existing emulators, with their crazy optimizations and techniques.

The Universal Virtual Computer [15] that we mentioned above has no higher-level language. To write a program for the UVC, the preservationist must write individual UVC instructions in a special assembly language. This is a tedious and error-prone task, as several generations of assembly language programmers may attest. To make the job easier, we developed a high-level language specifically designed for writing emulators for the original platform. Our language is an expression-oriented, imperative programming language where the only data types are bit vectors and arrays of bit vectors. It features a compact and readable notation for manipulating bit vectors and incorporates ideas from APL [19], $\lambda$-RTL [30], and Verilog [35].

As we mentioned earlier, if the archaeologist of the future wants to modify the emulator or build a hardware replica, she will need to understand how the original platform and its emulator work. For this purpose, we include the source code of the original platform emulator and the compiler for our emulator-writing language.

An emulator written in our emulator-writing language aims to be an "executable and debuggable specification" that "separates meaning from optimization." These two ideas were guiding principles of the STEPS Project at the Viewpoints Research Institute [26], that aimed to reinvent personal computing in 20,000 lines of readable and maintainable code. To achieve such an ambitious goal, the STEPS group developed many high-level, domain-specific languages. Code written in these domain-specific languages specified "what" to do, not "how" to do it, unlike code written in an imperative language like C or C++. To improve the performance of the domain-specific language, optimization details would be written separately from the high-level program instead of obscuring the meaning of the high-level program, thus separating the meaning from optimization. As our emulator-writing language is still an imperative language that is an incremental improvement over writing an emulator in C or C++, we still have a long way to go until we truly have a language for "executable specifications."

## 12. Current Incarnation

We chose Smalltalk-72 for our prototype implementation due to its familiarity to us, as one of the authors was one of the original developers; furthermore, Smalltalk-72 is historically notable as one of the earliest object-oriented programming languages. Our emulator-writing language compiler was written in OMeta/JavaScript and compiled down to the Chifir virtual machine described in figure 3 and 4. We tested our design by writing emulators for the Chifir virtual machine in several languages, including C and JavaScript; it did not take more than a few hours to write each emulator.

For our development environment, we chose Smalltalk-76 [17], an image-based, object-oriented programming language and system that was a predecessor to modern Smalltalks. Like modern Smalltalks, it used bytecodes for portability, but is much smaller and simpler than modern Smalltalks; a functioning Smalltalk-76 interpreter is given in the appendix of the Smalltalk-76 paper cited above. Our current implementation is very slow due to having a bytecode interpreter, but we plan to improve it by translating Smalltalk-76 bytecodes directly to Chifir.

## 13. Future Incarnations

We have a few ideas for what we want to do next.

Firstly, we need to do extensive testing of our system. Our plan is to give our "preserved program" to a large number of programmers and tell them to "solve this puzzle." This group of programmers should have different skill levels and be familiar with different programming languages and frameworks. We should observe them "solve the puzzle" and note what difficulties they face trying to write the "afternoon hack." If they find it too difficult or take far more than an afternoon we should take that into account when designing the next incarnation of Cuneiform.

Secondly, we plan to add a set of "unit tests" to our system to simplify the process of writing the "afternoon hack." These unit tests would execute when the preserved program is fed into the "afternoon hack," and check if the "afternoon hack" had been implemented correctly. For each opcode, an instruction with that opcode would be executed and its result would be compared with the correct value (thus also testing the comparison instruction). Designing a good set of unit tests is tricky, because there is a potential for false negatives (an incorrect emulator can be written so that it passes all the tests). Nevertheless, even a rudimentary set of unit tests would be helpful.

Finally, we'd like to develop alternative specifications for our virtual machine, to serve as a "Rosetta Stone." As we mentioned earlier, our current idea is to have the archaeologist of the future write a logic simulator, and our "program" could be a set of logic equations.

## 14. Final Remarks

The scheme we described in this essay is but one proposal, and is in no way, shape, or form a complete and definitive technical solution to the challenging and underexplored problem of ensuring the future of our digital heritage. And underexplored it is. Whereas library and archival science is currently just waking up to the idea of preserving digital artifacts, the computer industry does not pay much attention to its past, preferring to focus on the next big thing instead. Thus, software preservation has mostly been the domain of hobbyists and enthusiasts and mostly focused on video games and 8-bit microcomputers of the 1980s. Most traditional libraries and library scientists are focused on preserving static documents instead of interactive media; the use of software emulation is still controversial within the fields of library and archival science [5]. The Computing History Museum focuses on preserving hardware, but not software. The Internet Archive and `textfiles.com` are some of the few institutions who preserve software, but they mostly focus on preserving the bits themselves, without focusing on ensuring that future generations can still make use of the preserved bits.

There have been preservation efforts previous to our work, but we find them to be unsatisfactory. The aforementioned CAMiLEON project [16] proposes to preserve emulators written in a subset of C along with the preserved program, but we believe that C is too dependent on the underlying hardware to be practical. Furthermore, many popular languages in the past have since become completely unknown; we fear that the same may happen to a language even as well-established as C. When the second author began his computing career in 1961, the most popular language was Algol, yet Algol has fallen completely into disuse in 2015. The aforementioned Universal Virtual Computer proposes a two-layered emulation approach similar to ours, where the original platform emulators will run on a Universal Virtual Computer, but this Universal Virtual Computer is too complex to be implemented in an afternoon.

Computing is useful, profitable, and fun, but we should spend a little bit of effort to consider more than the immediate present and do some serious long-term thinking. As Stewart Brand, cofounder of the Long Now Foundation, said:

> "Civilization is revving itself into a pathologically short attention span. The trend might be coming from the acceleration of technology, the short-horizon perspective of market-driven economics, the next-election perspective of democracies, or the distractions of personal multi-tasking. All are on the increase. Some sort of balancing corrective to the short-sightedness is needed-some mechanism or myth which encourages the long view and the taking of long-term responsibility, where 'long-term' is measured at least in centuries."

## References


[1] P. Z. Adelstein. Permanence of Digital Information. In *Proceedings of XXXIV International Conference of the Round Table on Archives*, XXXIV CITRA, pages 1-7, Budapest, Hungary, 1999.

[2] Anonymous. *The Anglo-Saxon Chronicle*.

[3] R. S. Barton. A New Approach to the Functional Design of a Digital Computer. In *Papers Presented at the May 9-11, 1961, Western Joint IRE-AIEE-ACM Computer Conference*, pages 393-396, New York, NY, USA, 1961. ACM.

[4] C. Bauer. SheepShaver: an open source PowerMac emulator. URL `http://sheepshaver.cebix.net/`.

[5] D. Bearman. *Reality and Chimeras in the Preservation of Electronic Records*. Corporation for National Research Initiatives, 1999.

[6] Commodore Business Machines, Inc. *Commodore 64 Programmer's Reference Manual*. Howard W. Sams & Co., Inc., 1982.

[7] O. J. Dahl and K. Nygaard. *SIMULA: A Language for Programming and Description of Discrete Event Systems*. Norwegian Computing Center, 1966.

[8] H. C. Darby. *Domesday England*. Cambridge University Press, 1986.

[9] J. Darlington, A. Finney, and A. Pearce. Domesday Redux: The Rescue of the BBC Domesday Project Videodiscs. *Ariadne*, 36, July 2003.

[10] A. Finney. The BBC Domesday Project - November 1986. URL `http://www.atsf.co.uk/dottext/domesday.html`.

[11] H. Freudenthal. *Lincos: Design of a Language for Cosmic Intercourse, Part 1*. North Holland Publishing Company, 1960.

[12] D. Grant. Domesday Preservation Group. URL `http://web.archive.org/web/20100724025051/http://www.domesday1986.com/index.html`.

[13] M. Hedstrom, C. Rusbridge, P. Wheatley, et al. Creative Archiving at Michigan and Leeds Emulating the Old on the New. URL `http://web.archive.org/web/20060909234230/http://www.si.umich.edu/CAMILEON/index.html`.

[14] Hitachi. Successful read/write of digital data in fused silica glass with a recording density equivalent to Blu-ray Disc. URL `http://www.hitachi.com/New/cnews/month/2014/10/141020a.html`.

[15] J. R. van der Hoeven, R. J. van Diessen, and K. van der Meer. Development of a Universal Virtual Computer (UVC) for Long-Term Preservation of Digital Objects. *Journal of Information Science*, 31(3): 196-208, 2005.

[16] D. Holdsworth and P. Wheatley. Emulation, Preservation and Abstraction. URL `http://sw.ccs.bcs.org/CAMiLEON/dh/ep5.html`.

[17] D. H. H. Ingalls. The Smalltalk-76 Programming System Design and Implementation. In *POPL '78 Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '78, pages 9-16, New York, NY, USA, 1978. ACM.

[18] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation, 2011.

[19] K. E. Iverson. *A Programming Language*. John Wiley and Sons, 1962.

[20] G. James, B. Silverman, and B. Silverman. Visualizing a classic CPU in action: the 6502. In *ACM SIGGRAPH 2010 Talks*, SIGGRAPH '10, pages 26:1-26:1, New York, NY, USA, 2010. ACM.

[21] M. Jandali. *Echoes from Ugarit*. Audio CD. CD Baby, 2009.

[22] A. C. Kay. The Early History of Smalltalk. *History of Programming Languages - II*, 1993.

[23] K. Kelly. Very Long-Term Backup. URL `http://blog.longnow.org/02008/08/20/very-long-term-backup/`.

[24] The Long Now Foundation. The Rosetta Project: Technology. URL `http://rosettaproject.org/disk/technology/`.

[25] R. A. Lorie and R. J. van Diessen. UVC: A Universal Virtual Computer for Long-term Preservation of Digital Information. IBM Research Report RJ 10338 (A0502-006), 2005.

[26] A. C. Kay, D. H. H. Ingalls, Y. Ohshima, I. Piumarta, and A. Raab. Proposal to NSF Granted on August 31, 2006. VPRI Research Note RN-2006-02, 2006.

[27] J. McCarthy, P. W. Abrahams, D. J. Edwards, T. P. Hart, and M. I. Levin. *Lisp 1.5 Programmer's Manual*. The MIT Press, 1962.

[28] P. Mellor. CAMiLEON: Emulation and BBC Domesday. URL `http://www.iconbar.com/articles/CAMiLEON_Emulation_and_BBC_Domesday/index937.html`.

[29] R. McKie and V. Thorpe. Digital Domesday Book lasts 15 years not 1000. URL `http://www.theguardian.com/uk/2002/mar/03/research.elearning`.

[30] N. Ramsey and J. W. Davidson. Specifying Instructions' Semantics Using $\lambda$-RTL (Interim Report). University of Virginia Technical Report CS-97-31, 2003.

[31] B. Russell and A. N. Whitehead. *Principia Mathematica*. Cambridge University Press, 1910, 1912, 1913.

[32] J. F. Schoch, An Overview of the Programming Language Smalltalk-72. *ACM SIGPLAN Notices*, 14(9):64-73, 1979.

[33] G. L. Steele Jr. Growing a Language. *Higher-Order and Symbolic Computation*, 12:221-236, 1999.

[34] C. P. Thacker, E. M. McCreight, B. W. Lampson, R. F. Sproull, and D. R. Boggs. Alto: A Personal Computer. Xerox PARC Technical Report CSL-79-11, 1979.

[35] D. Thomas and P. Moorby. *The Verilog Hardware Description Language*, 5th ed. Springer, 2002.

[36] P. Wheatley. Digital Preservation and BBC Domesday. In *Electronic Media Group Annual Meeting of the American Institute for Conservation of Historic and Artistic Works*, pages 1-9, Portland, Oregon, 2004.