

IBM

KB

the UVC:  
a method for  
preserving  
digital  
documents

proof of concept







KB

010010

the UVC:  
a method for  
preserving  
digital  
documents

0010

proof of concept

Ir. Raymond Lorie

01001011

01000010

Design: Steven L. Stijger  
Published by: IBM Netherlands, Amsterdam  
IBM / KB Long-Term Preservation Study  
Report Series Editor: Dr. Raymond J. van Diessen

Available from:

IBM External Communications	Koninklijke Bibliotheek
PO Box 9999	PO Box 90407
1000 CE Amsterdam	2509 LK The Hague
The Netherlands	The Netherlands

Title: **The UVC: a Method for Preserving Digital Documents - Proof of Concept**

ISBN: 90-6259-157-4  
Author: Ir. Raymond Lorie  
Date: December 2002  
Copyright: IBM / Koninklijke Bibliotheek

*This study was commissioned by the Koninklijke Bibliotheek,  
National Library of the Netherlands*

# 01 IBM / KB 010000010

## long-term preservation study

The National Library of the Netherlands (Koninklijke Bibliotheek, KB) is faced with the problem of preserving large amounts of digital documents for the long term. These documents come from two sources: from media published directly in digital form and from digitizing paper documents. In 2000, the KB and IBM started building an electronic deposit system ("Digital Information Archiving System or DIAS"), the technical core of the infrastructure for KB's e-Deposit for the Netherlands.

From the beginning it was clear that this project could not rely on out-of-the-box solutions alone because up to that time no solution readily addressed both the aspects of large volume and durable storage as well as the long-term preservation requirements. So an IBM / KB Long-Term Preservation Study (LTP Study) was initiated as part of the overall project of developing an electronic deposit system.

The primary objective of the LTP Study was to investigate the functionality required for the long-term preservation (hundreds of years) of the digital information stored in DIAS. This study has resulted in 6 reports: one overview report and five specific reports, each one addressing an important aspect of long-term preservation in its own right.

Participants in the LTP Study:

#### **IBM**

Raymond J. van Diessen  
Raymond Lorie  
Sidney Huiskamp  
Hans Verhoeven

#### **Koninklijke Bibliotheek**

Johan F. Steenbakkens  
Titia van der Werf-Davelaar  
Patricia Alkhoven  
Adriaan Lemmen

#### **RAND Corporation**

Jeff Rothenberg

#### **British Library**

Deborah Woodyard

I would like to thank all the participants for their input and enthusiasm. The results make an important contribution to the development and implementation of dedicated functionality for the long-term preservation of digital information and for guaranteeing long-term access.

Report Series Editor,  
Raymond J. van Diessen

# Titles of the Report Series

## **Number 1: The Long-Term Preservation Study of the DNEP Project - an Overview of the Results**

This report explains the reasons and objectives behind defining the LTP Study as part of the overall project to implement an electronic deposit system. It also provides a quick and general overview of all the study results, which are then elaborated on in the other published reports.

## **Number 2: Authenticity in a Digital Environment**

Authenticity acquires a new meaning in a digital context. Normally objects are physical and their physical characteristics are the main source for defining authenticity. Moreover, authenticity is not a single concept, but involves different aspects that can be associated with an object:

- € A traceable path from the object's origin to its current ownership.
- € Measures and techniques for safeguarding against and/or recognizing modifications.
- € Techniques for establishing the use of original materials.

The problem of digital objects is that in fact they are just conceptual objects. A digital object is a conceptual object to be interpreted (rendered) by executing the digital object in a specific IT infrastructure (hardware & software). This report focuses on defining a framework in which we can define what is actually meant when one speaks of an authentic digital object.

## **Number 3: Preservation Requirements in a Deposit System**

The initial DIAS release only provides basic functionality for preserving and rendering the stored digital objects for the long term. One of the primary responsibilities of the LTP Study is to define the functional requirements of the Preservation Subsystem, which is scheduled for development later. This report identifies requirements of the DIAS Preservation Subsystem so as to provide the services and functions for monitoring the technical environment associated with the digital objects stored in DIAS.

The Preservation Subsystem can be summarized by the following three objectives:

- € Identifying digital objects that are in danger of becoming inaccessible because of changes in technology.
- € Implementing the activities associated with technical preservation.
- € Supplying the requisite technical metadata in order to generate / validate the environments needed during digital object delivery.

## **Number 4: The UVC: a Method for Preserving Digital Documents - Proof of Concept**

Within IBM Research in Almaden, Raymond Lorie was already working on a combined emulation / migration approach to preserve a certain class of digital objects with an approach called the Universal Virtual Computer (UVC).

The main idea consists of archiving a program P along with the data file that decodes the data and returns the information to a future client based on a logical view. The logical view of the data is simple and self-contained enough to be interpreted without any specific software or hardware. Program P is written for the Universal Virtual Computer (UVC) that is general, yet basic enough to continue to be relevant in the future. Given the simplicity of the UVC, it will be relatively easy to write an emulator of the UVC in the future on a real machine of that time. The emulated machine will run the program P and return all data in an easy to understand logical view of the data.

The LTP Study conducted a proof of concept with the KB to test the UVC approach in a library environment. The PDF format was selected because it is the primary data format for electronic publications to be stored in DIAS.

## **Number 5: Managing Media Migration in a Deposit System**

Storage technology obsolescence makes media migration a necessity. Data has to be copied from one storage medium to another on a regular basis. However, the fact that storage technology becomes obsolete is not the only trigger for rewriting previously stored digital objects. All storage media degrade over time and have to be rewritten either on the same medium (refreshing) or on another medium (migration).

Ordinarily media refreshment / migration would be a straightforward process. However, the large amounts of storage associated with an electronic deposit system introduce certain volume-specific requirements. Most electronic deposit systems define their storage capacity needs in several TeraBytes ( $10^{12}$  Bytes). Take a deposit system with 100 TeraBytes of information stored on tape, for example. Let's assume that you want to migrate all this information to an optical storage medium. Current optical storage media have a capacity of around 5 GigaBytes and a write speed of around 4 MegaBytes/second. A quick calculation shows that a complete migration to optical storage would take at least 290 days (100 TeraBytes / 4 MegaBytes per second)!

This report describes the actions to be taken to manage media migration / refreshment effectively within an electronic deposit system, focussing specifically on the media migration issues within DIAS. Potential additional capacity required for media migration might be created by redundancy and parallelism.

## **Number 6: Archiving Web Publications**

More and more Web publications are becoming a primary source of information and will thus be stored as digital objects in DIAS. Web publications have specific characteristics and requirements that DIAS must meet if they are to be archived successfully.

This report investigates the issues and requirements introduced by archiving Web publications and their potential impact on DIAS.



# 01 contents



1/	Summary	1
2/	Introduction	3
3/	Description of the UVC-Based Preservation Method	5
	3.1 What Needs to be Done Today (at Archive Time)	7
	3.2 What Needs to be Done in the Future (at Restoration Time)	11
	3.3 Internal Representation of the Data: Two Options	12
4/	The System Prototype	15
5/	Application: the Archival of PDF Documents	17
	5.1 What Should be Archived	17
	5.2 Preparing the Bit Stream to be Archived	18
	5.3 Recovering the Information (in the Future)	23
6/	A Note on the UVC Convention	25
7/	Evaluation	27
	7.1 The UVC	27
	7.2 The Application: Archiving PDF Documents	27
8/	Future Work	29
	Appendix A: References	33
	Appendix B: Glossary	35
	Appendix C: The UVC Architecture	37
	Appendix D: UVC Assembler	43
	Appendix E: Complete Logical View of a PDF Document	45





# 01 summary



This paper reports on the joint study between IBM and the Koninklijke Bibliotheek (KB) in The Hague; it covers the work done at the IBM Almaden Research Center as a proof of concept for a new long-term preservation technology. In particular, it deals with archiving PDF documents.

Currently, the technology addresses the preservation of data files. The main idea consists of archiving, with the data file, a program P that decodes the data and returns the information to a future client based on a logical view. The definition of the logical view is also archived, along with a program Q which decodes and returns the definition of the logical view based on a fixed logical view. Both P and Q are written for a *Universal Virtual Computer (UVC)*, which is general, yet basic enough to continue to be relevant in the future. Given the simplicity of the UVC, it will be relatively easy to write an emulator for the UVC in the future on the real machine being used at that time. The emulated machine will run programs P and Q and return all data in an easy to understand logical view.

The goal of the joint study was to validate the technology by applying it to the preservation of PDF documents. The PDF format was chosen because of its importance. Saving PDF means saving the page layout, the bookmarks, the few metadata fields embedded in the PDF file and the text itself along with all its presentation attributes. We achieved our goal by using existing tools to create raster images for all pages, and to extract the images, the metadata fields, the bookmarks, and the text with its presentation attributes from the file. The information is packed into a single bit stream, together with all the UVC programs needed to retrieve it from the bit stream and return it to the client in its logical form.

The PDF archiving application is the first real test of the UVC-based approach for long-term preservation. It has been an excellent test bed for the UVC itself, the emulator, the assembler and the program for defining logical views. In particular, decoding a JPEG file posed significant challenges for the UVC; the feedback has been extremely positive.

As far as the UVC architecture is concerned, we did not need to modify the current specifications; we did not need to add any instructions or constructs. The memory model (bit-addressable) and the general load/store/move instructions to/from any bit address proved to be ideal for the kind of bit manipulation required to analyze and build bit streams.

We archived an image for each page of a document. We did not try to compress the image at this time because we want to build and evaluate some specific compression methods in a subsequent phase of the project. We also extracted all textual elements from the document including their presentation characteristics and positions as a page. For some environments, archiving the page layout may be sufficient. However, archives are of greater value if they can be accessed based on content. Without making any decision on how the information should or should not be indexed, we show how the information could at least be made available. To demonstrate this feature, we prepared a demo that illustrates how a future application may want to use the restored data. The

application interface accepts a query to find a specific word or group of words that appears in a document with certain presentation characteristics. The application finds the page, displays it, and highlights the words that match the keywords in the query. This fully tests the correlation between image and text.

We successfully demonstrated that the concepts are quite sound, can be applied to the archiving of PDF documents, and can be implemented.

# 2/

# 01 introduction 1000010

This paper reports on the joint study between IBM and the Koninklijke Bibliotheek (KB) in The Hague; it covers the work done at the IBM Almaden Research Center as a proof of concept for a new long-term preservation methodology. In particular, it deals with the archiving of PDF documents.

The problem that national libraries are facing today is well known. For centuries, paper has been used as the preferred medium for storing text and images. Today, some of the archived objects (books, newspapers, pictures, etc.) are in danger of completely deteriorating. Current technologies suggest that documents should be digitized and saved in digital form. This offers many advantages. First, the object can be copied repeatedly without degradation; its content can be distributed remotely and can be accessed at will. Finally, the physical space needed to store the object becomes smaller and smaller as storage density increases. On the other hand, digitization involves its own challenges: how can we ensure that such documents (digital files) can be preserved for a long time, surviving changes in storage technology, computer hardware, software, formats, etc. [Rothenberg 1995]?

But this is only one aspect of the challenge. Increasingly, national libraries are becoming responsible for safeguarding documents that are generated electronically, rather than digitized. In a full-fledged deposit library system, the library must accept digital documents from the publishers and preserve them indefinitely. In the not-too-distant future, the majority of documents will be digital from the outset and the community has to be prepared to handle them.

Clearly, any proposed technology should be able to handle the archiving of both the electronically generated documents and those obtained through digitizing. In the latter case, an image is the only information available; it is generally accompanied by some application metadata. Assuming we do not try to recover the text by applying optical character recognition, the image and the application metadata constitute all the available information. That information is a subset of what is available in those documents that were digital from the outset. So, by focussing on those documents that are digital from the outset, we really cover the entire spectrum.

The joint study's emphasis is on archiving documents that are static; their content is fixed and does not depend on how or in what environment the document is requested. This is the case for the vast majority of documents handled by national or research libraries today. This means that the following kinds of documents fall outside the scope of the study: interactive multimedia presentations, active CD-ROMs (games, interactive sessions, etc.), dynamic Web pages, and of course, computer programs. This is not to say that archiving these types of documents is not important; but static documents, which are both simpler and more relevant to the immediate needs of a deposit library, seemed to be a good initial choice. In addition, we believe that the technology being developed will naturally be expanded to include dynamic documents, although more research will be needed.

This report is organized as follows. Chapter 3 introduces the technology already proposed in [Lorie 2001a] and [Lorie 2001b]. Chapter 4 identifies the components of a prototype system built at IBM Research, and refers to several appendices covering the more technical aspects of the project. It is not absolutely necessary that one reads the appendices to understand the remainder of the report. However, the Appendix covering the architecture of the virtual machine that forms the core of the proposal is important since it demonstrates not only its relative simplicity, but also its power. Chapter 5 looks at the applicability of the method for preserving PDF documents. Designing a full solution was clearly beyond the scope of this study but we did look at the most relevant aspects of the problem. In particular, we report on our experiment in archiving the images, the textual content with all its presentation attributes, the metadata (mostly the bibliographical information), and a certain amount of additional information that will be needed at the time the document is restored. Chapter 6 summarizes what a potential user needs to know, both when archiving and when restoring the document; this is what we call the UVC Convention. The final two chapters evaluate what has been done and provides suggestions for future work.

*Digitization involves its own challenges: how can we ensure that such documents (digital files) can be preserved for a long time, surviving changes in storage, computer hardware, software, formats, etc.*

# 3/

## description of the UVC-based preservation method

Figure 3.1 illustrates the central idea of the UVC-based preservation system. A vertical line separates what is being done in the present, from what will be done in the future. The information that crosses the line is a preserved bit stream. It contains the data and a program P. The central idea of this method is that P is an executable program for a Universal Virtual Computer (UVC). The UVC is simple enough to continue being relevant for a very long time. In the future, the program P is passed through a UVC interpreter (synonym for emulator) under the control of a Restore application program. As the Restore program repeatedly invokes the interpreter, each iteration returns a new tagged data item. The tags essentially associate a semantic meaning with a data item. They are very similar to the ones used in XML [XML 2002].

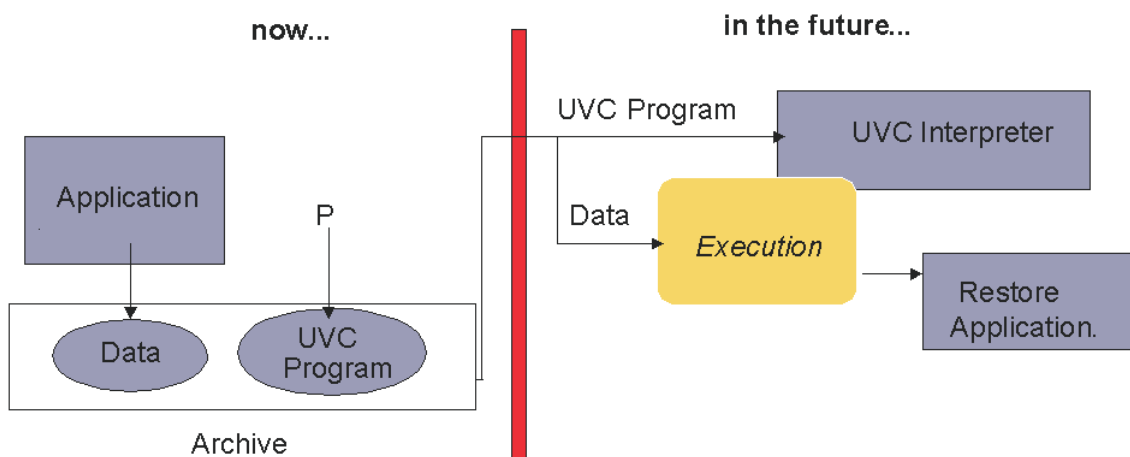


Figure 3.1 / Central mechanism of an UVC-based preservation system

*The central idea of this method is that  $P$  is an executable program for a Universal Virtual Computer (UVC). The UVC is simple enough to continue being relevant for a very long time. In the future, the program  $P$  is passed through a UVC interpreter (synonym for emulator) under the control of a Restore application program.*

Before we continue describing the details of the method, it is worthwhile to place it in the framework of the strategies proposed much earlier in the literature: conversion and emulation.

**Conversion** is the strategy commonly applied at the moment. Each time something changes in the environment that impedes the correct execution of a process, the process needs to be examined; and programs, data structures, or interfaces have to be changed in order to make the programs run again.

**Emulation** consists of developing a software emulator for a year 2000 machine for a future computer, say in 2050. That emulated machine will then be able to run the 2000 programs on the actual data, without requiring any change to the program or data.

The pros and cons of these strategies have been discussed at length in [Lorie 2001c] and [Rothenberg 1999]. The UVC-based methodology makes a distinction between preserving data and preserving the behavior of a program. For data, it utilizes a conversion program capable of decoding the original form of the data into a logical format that will be much easier to understand in the future. But that conversion program is written today (for a UVC machine), not in 2050. It will be executed in 2050 on a UVC emulator written specifically for the 2050 machine. For programs, the UVC-based methodology will rely on a UVC emulator for the 2000 machine written in 2000, and a UVC emulator for the 2050 machine. This clearly differs from the emulation method proposed in [Rothenberg 1995], in that it does not require writing an emulator for a real machine from the past in the future.

*The UVC-based methodology makes a distinction between preserving data and preserving the behavior of a program. For data, it utilizes a conversion program capable of decoding the original form of the data into a logical format that will be much easier to understand in the future. But that conversion program is written today (for a UVC machine), not in 2050.*

### 3.1 What Needs to be Done Today (at Archive Time)

#### Step 1: Define the appropriate logical schema

Figure 3.2 shows the schema for a logical view of the data for a library application. A catalog has a name and contains a set of books (indicated by Book+, where "+" indicates that an arbitrary number of books (>0) is allowed). A book has a number, a set of authors, a title, a year and an editor. All of these values are character strings.

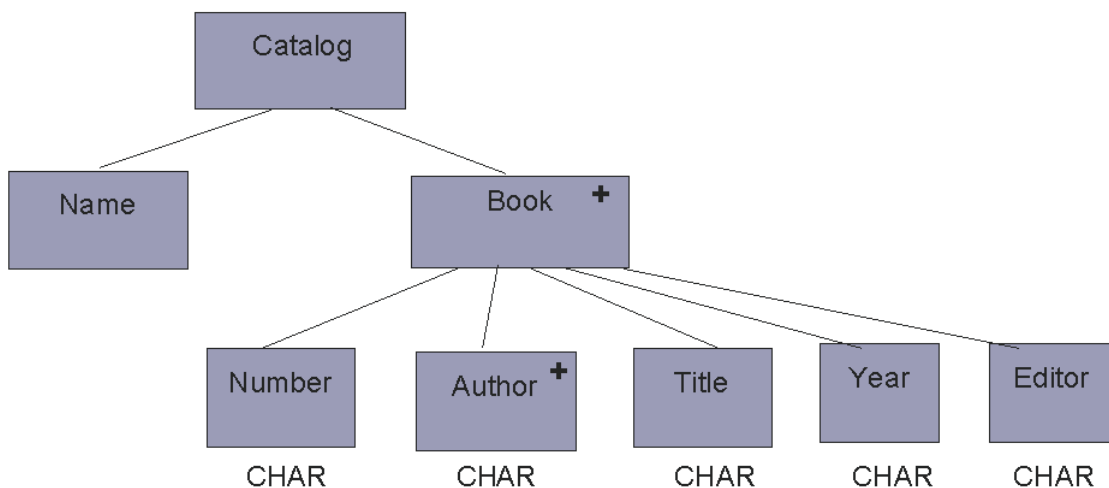


Figure 3.2 / Graphical view of the logical schema

As explained above, executing the UVC program for decoding the data will return tagged data items based on the schema. Starting with the first data item (the catalog itself), the succession of tagged data items will look like this:

```

<Catalog>
  <name> A.B. Morgan Collection
  <book>
    <number> 123456
    <author> Smith, John
    <author> Smith, Mary
    <title> Adventures
    <year> 1988
    <editor> ABC Editions
  </book>
  <book>
    <number> 654321
    <author> Green, John
    <title> My Story
    <year> 2000
    <editor> XYZ Inc.
  </book>
  etc...

```

The tags appear in brackets; for example, <book> is a tag. A tag specifies the semantic of the character string that follows: '123456' is the number of the book, 'ABC Editions' is the editor, etc. If the future user gets these tagged data elements, he will probably fully understand the meaning of each data element and the relationships between them. But this is only true because the application is so simple and the concept of books and related attributes so universally known. In general, this is not the case and the future user will need additional information on the logical structure. This information is needed for preservation (see step 4).

### Step 2: choosing an internal representation

This is part of the normal design of an application. At this point, we assume that the design team does not take long-term archiving into account at all. Many formats are possible; one possibility is to concatenate all fields with separators, where separator (x) indicates the length of the following data element, while separator [y] indicates the number of values for a multi-value field such as author. The data associated with the 'Adventures' book may be internally stored as:

```
(6)123456[2](11)Smith, John(11)Smith, Mary(10)Adventures(4)1988(12)ABC Editions
```

Although the structure is very simple, it requires some documentation; otherwise, it would be impossible to know how to interpret the bit stream, and the information would essentially be lost. A brief text could be included explaining the encoding; but this is true only because this case is so simple. The moment the encoding becomes more complex,

*The method hinges on the fact that the UVC definition will be universally available, so that anyone would be able to implement a UVC emulator on any machine at any time.*

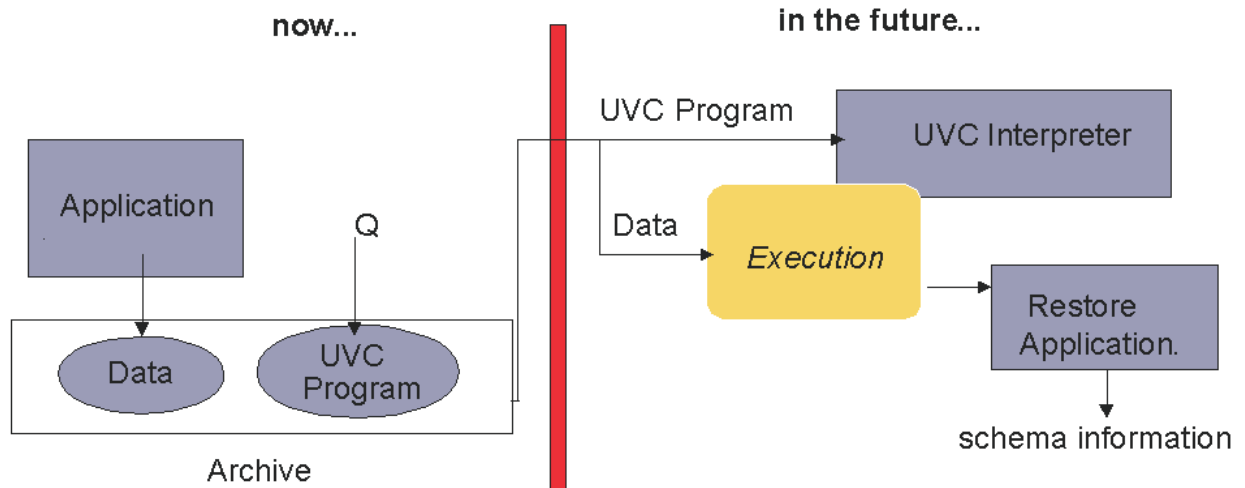


Figure 3.3 / Archiving and restoring schema information

the only practical solution is to associate a program P with the data, with P performing the decoding.

### Step 3: Writing the UVC program for data interpretation

In our proposal, the program P is written for a Universal Virtual Computer (UVC). It specifies the logic required to interpret the bit stream, extracting the multiple data elements from the stream and tagging each of them with one of the tags defined in the schema.

The UVC is a general-purpose computer. Its architecture is obviously influenced by the characteristics of real existing computers. What is important however is that it does not need to be physically implemented. Thus, there is no actual physical cost. For example, the UVC can have a large number of registers; each register can have a variable number of bits plus a sign bit, the sequential memory can also be whatever size is required. Speed is not a real concern since machines will be much faster in the distant future, and an emulation of the UVC on a future machine will run faster - considerably faster - than a machine language program running on today's machines. Once the UVC program is written, it can be tested on a UVC emulator running on one of today's machines.

The method hinges on the fact that the UVC definition will be universally available, so that anyone would be able to implement a UVC emulator on any machine at any time. It may just be a document, written in plain English - or another natural language - and distributed anywhere in the world.

### Step 4: Archiving the schema information

Step 1 discusses the need for defining an appropriate schema for a given application. The schema in Figure 3.2 specifies how the data will appear to the client. Specifically it defines the set of relevant tags. That information must be communicated to the future client and thus needs to be archived, as well. This can be done in a manner similar to that used to archive the data (Figure 3.3; note the resemblance to Figure 3.1).

Again, we need to store an internal representation of the schema information in the bit stream together with a UVC program Q that can be used in the future to decode the bit stream and return it to the caller, element by element, exactly as we do for the data itself. The only difference lies with the schema defining the structure of the restored schema

information. Since the structure of the schema information (see Figure 3.4) is the same for all applications, a schema to read schemata is chosen once and for all, and is a part of the UVC Convention.

A field has a Name (an abbreviated name used as tag), and a Long Name, which provides some additional information. A comment of any length can be associated with the field to explain its role and its relationships to other fields. The attribute field contains one character specifying a cardinality constraint such as <0 or 1>, <more than 1>, etc; a blank attribute stands for <1 and only 1>. The level shows where the field appears in the hierarchical schema being defined. Note in Figure 3.2 that Catalog is at level 0, Name and Book at level 1, the other nodes are at level 2. So, for the Catalog, the schema would be "seen" as:

```

<DOCTYPE> Catalog
  <FIELD>
    <NAME> Name
    <LONG_NAME>
    <COMMENT> The name of the collection
    <ATTRIBUTE>
    <LEVEL> 1
    <TYPE> CHAR
  </FIELD>
  <FIELD>
    <NAME> Book
    <LONG_NAME>
    <COMMENT>
    <ATTRIBUTE> +
    <LEVEL> 1
    <TYPE> CHAR
  </FIELD>
  <FIELD>
    <NAME> Number
    <LONG_NAME>
    <COMMENT> The book's numerical identifier
    <ATTRIBUTE>
    <LEVEL> 2
    <TYPE> CHAR
  </FIELD>
  etc...

```

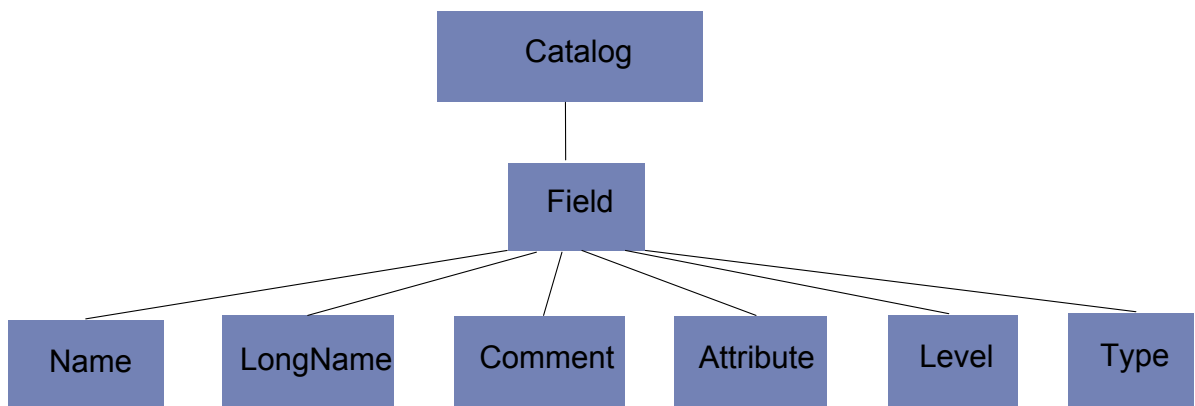


Figure 3.4 / A schema for reading schemata

Steps 1 to 4 are preparatory steps that must be executed once for each type of document. The schema information and the UVC programs will comprise the start of the bit stream. Then, for each instance of this type, the internal representation of the data will be added to the stream. If several documents of the same type are archived in the same bit stream, the schema information and the UVC programs do not need to be replicated for each document.

## 3.2 What Needs to be Done in the Future (at Restoration Time)

### **Step 1: Make sure that an emulator is available on the current machine**

In a full-fledged deployment of a preservation methodology, we can assume that every machine in the future will come with a UVC emulator. Since the UVC is a very simple machine, writing such an emulator is a relatively easy task (for a skilled computer scientist). In addition, one emulator is needed for each machine type; but once the code exists for one machine, it can simply be recompiled or slightly modified to fit another machine. This is also true when migrating an emulator from one machine generation to the next one.

*It is important to note that it is the restore program that reads the data; the UVC does not know anything about I/Os; it simply receives a pointer to the data in an unlimited virtual memory. (The same is true for the UVC object code). The execution will produce the tagged data elements, one by one.*

### **Step 2: Write a restore program to restore the data**

Assuming the emulator exists, a simple application program still needs to be developed. Its purpose is to read the UVC object code and the bit stream into memory and to invoke the emulator to execute the UVC program. It is important to note that it is the restore program that reads the data; the UVC does not know anything about I/Os; it simply receives a pointer to the data in an unlimited virtual memory. (The same is true for the UVC object code). The execution will produce the tagged data elements, one by one. When the emulator is invoked, a few arguments are passed on, in addition to the addresses of the data and the object code; these include the address of a storage space where a data element is returned, the address of a storage space where the tag is returned, an operation code and a return code. The application invokes the emulator repeatedly in order to get all the data elements. The application program can process and/or store each data element returned as it sees fit.

**Step 3: write a restore program to restore the schema**

Generally, one of the operation codes is used to request the data, another to request the schema information. In both cases, the logic - essentially the loop through all elements - is the same. Even if this is the case, two different programs will probably be written, one to get the schema, the other to get the data. Since, for the schema information, the logical view is fixed, a single restore program may actually support all applications. In addition, the future client may choose not to retrieve the schema if he already knows the logical view for the documents being restored. If not, he will only request the schema once for a collection of documents of the same type.

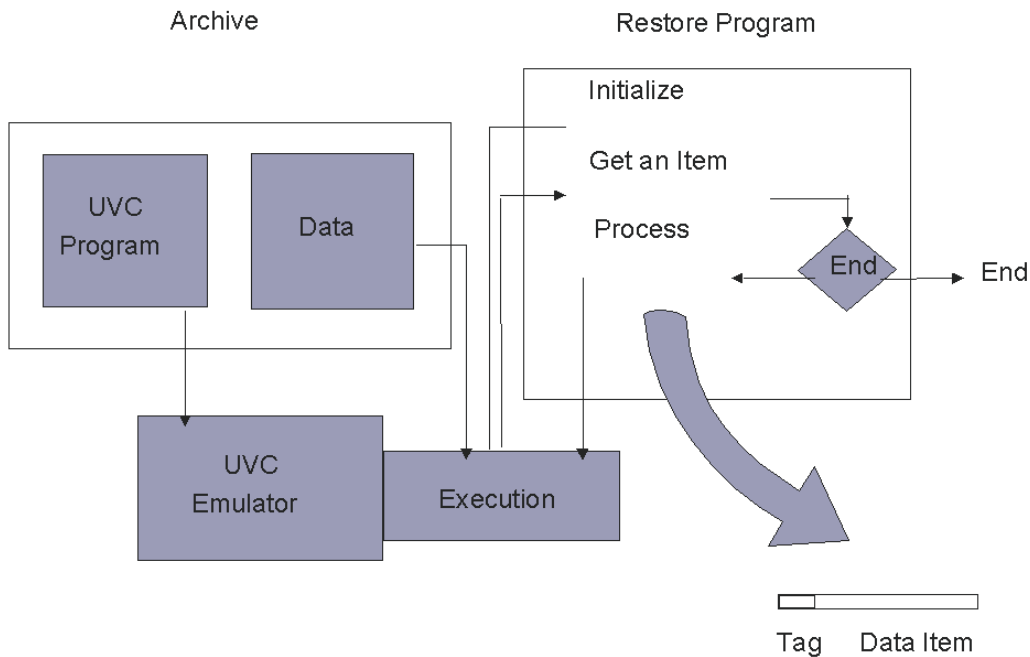


Figure 3.5 / A restore program

### 3.3 Internal Representation of the Data: Two Options

In the simple example considered above, the internal representation of the data was chosen by the initial application. No special concession was made for long-term preservation; the format was not altered in any way to simplify the decoding (and, indirectly, the UVC program). We refer to this particular case as Option 1. But this is not the only possibility. An alternative, Option 2, opts for extracting the relevant data elements from the original file, and then organizing them into a different internal representation. That representation may be simpler, making it easier to write the corresponding decoder (in UVC). Both approaches are being used in our application of the technology to preserve PDF documents (see later in this paper). Figure 3.1 illustrates approach 1; Figure 3.6 illustrates the conversion of the initial data into another representation.

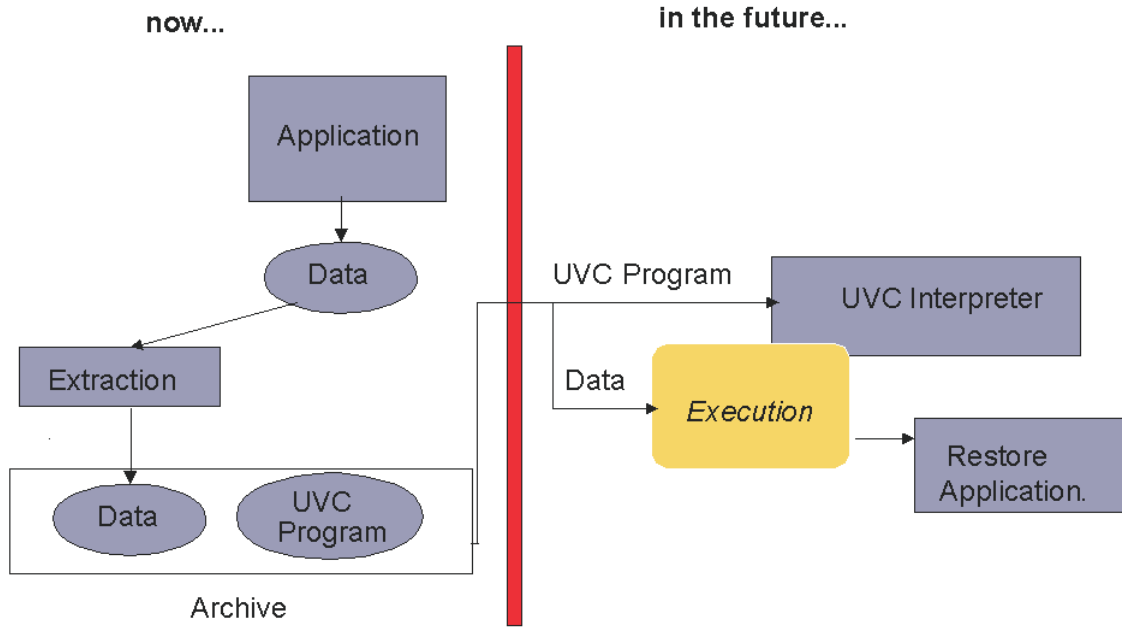


Figure 3.6 / Option 2 - Initial conversion of data into data



# 4/

# the system prototype

Most of the ideas presented above were implemented in 2000 (and enhanced in 2001) as part of an initial prototype complete enough to support the joint project. This prototype contains:

€ **The definition of the UVC (see Appendix C)**

Justifying the choice of a new computer architecture is never easy. However, the UVC is special in the sense that it is a virtual machine that will only (at least in production) be used in the future on much faster processors. This makes it different from the Java VM that needs to support the current Java language extremely efficiently. As a result, the UVC may be freed from some of the current constraints such as fixed size registers, predefined byte size, etc. The UVC is not necessarily minimal; some instructions are actually very powerful (move bits, for example). We also wanted the instruction set to be easy to understand, with few side effects or error conditions. We shall come back to this point in the Evaluation Chapter 7.

€ **An assembler (with some higher level constructs)**

To facilitate the development of an executable UVC program (see Appendix D).

€ **A UVC emulator on an Intel/Windows NT computer**

The emulator supports the features described in Appendix A with one caveat. In the architecture, one will find phrases such as: 'there is an arbitrarily large number of registers', 'an "infinite" bit memory', 'variable size registers', etc. The implementation does not support these unlimited values. We simply chose reasonable values. Specifically a register has 32 bits like the real machine on which the emulator works. We use 100 segments, each with 100 registers and up to 8 megabytes of storage. This is quite enough for the application at hand. The implementation can be seen as a Version 0. Any archived document will include a version number N and any version M with  $M \geq N$  will be able to process the document.

€ **A general mechanism for defining a schema**

In the Chapter 3 on what to do in the present (step 1), we discussed the choice of a schema to define how the data will be seen by a future user. We never specified how to define such a schema. This is because the process for defining a schema is not part of the UVC convention. The schema information can be stored in any representation, and the UVC decoder understands that representation. For the prototype, we implemented a tool that accepts a schema definition and constructs a bit stream using a reasonable internal representation. We also wrote the corresponding UVC decoder. This tool can thus be used by various applications, but is not meant to be used forever.

The input format is very similar to a Data Type Definition (DTD) in XML. For example, the DTD for the Catalog application can be written as follows.

```
ELEMENT Catalog (1, 2+)
ELEMENT 1 [Name]
ELEMENT 2 [Book] (3, 4+, 5, 6, 7)
ELEMENT 3 [Number](CHAR)
ELEMENT 4 [Author](CHAR)
ELEMENT 5 [Title] (CHAR)
ELEMENT 6 [Year] (CHAR)
ELEMENT 7 [Editor](CHAR)
```

The syntax is intuitive and can be "parsed" as follows: a catalog is a hierarchy of items; it has a name and contains a set of books (indicated by 2+, where 2 is a reference to Book and the "+" attribute indicates that an arbitrary number of books (>0) is allowed). A book has a number, a set of authors (4+), a title, a year and an editor. All of these values are character strings. The relationships are expressed as references between the abbreviated names. This way of looking at the data clearly corresponds to the one illustrated in Figure 3.2.

Once we use the same program to save all schemata, we can write a single Restore program to retrieve all schemata. This is an implementation of what was described in 3.2 (step 2). This method cannot be used generally for retrieving the data, although the general logic is the same.

# 5/

## 01 application: 01000010 the archival of PDF documents

As a proof of concept for the methodology described above, the KB chose the PDF file format from Acrobat (Adobe) [Adobe 2000], because of its importance in the publishing community. Implementing a complete operational system was clearly outside the scope of the study. What we did instead is to isolate the main technical problems and show how the proposed technology could handle them.

In what follows, we start by determining which information should be archived and then we describe how to gather it. Finally, we describe an application that shows how a future client can access the archived information and use it for whatever is fashionable in the future.

### 5.1 What Should be Archived

This is an important question because any information that is not appropriately archived today will be lost forever. For a deposit library, the most important aspect of the information is the appearance of the document. Today, many documents are still printed on paper, even if they are stored in electronic form. For these documents, the way they appear to the human eye is all there is. In other words, the e-form is only used for compact storage but not for additional functionality. For such documents, it would seem reasonable to archive only the image. However, as the number of documents increases, the problem of retrieving the right document(s) becomes more and more difficult. Indices on titles or keywords are helpful, but clients who are becoming accustomed to the Web browser functionality, will very quickly demand more. Then it becomes reasonable to assume that the text itself (in ASCII format) should also be made available to a future application (this was also suggested in [Ockerbloom 2001]). It is symptomatic that Acrobat Reader does allow text to be exported in ASCII format (or depending on the version, only permits this to a limited extent), which is just one more reason to provide that capability for the archived document. The same argument can be made for other PDF capabilities such as bookmarks and some formatted fields the PDF file contains and that can be retrieved by Acrobat Reader.

There is another type of information implicitly contained in a PDF document that we do not archive: it is the description of how an individual character is drawn. Since the shape is generated by a mathematical formula in PDF, it looks more like a program execution. Instead, we only save a faithful bit map representation of the shape, at a reasonable resolution. The idea is that we save what a reader perceives today. The fact that the same characters could be drawn nicely later on a much larger scale is secondary. After all, a document published on an 8.5x11" page only needs to be restored to roughly that size; it need not be converted into a poster or a billboard! On the other hand, if a future printer or display has a much higher resolution, it is easy to write the software that would generate an appearance virtually equivalent to the original.

*Since the shape is generated by a mathematical formula in PDF, it looks more like a program execution. Instead, we only save a faithful bit map representation of the shape, at a reasonable resolution. The idea is that we save what a reader perceives today.*

## 5.2 Preparing the Bit Stream to be Archived

Ultimately the bit stream will contain quite a few sections corresponding to the various types of data mentioned above. Each section contains a particular type of information; the details follow.

### **Step 1: Getting the page images**

We save the image of the PDF document, page by page. For the moment, we use a raster image (BMP format). This is only temporary. First, the class of documents we considered were black and white, so the size of an image is manageable; second, compression techniques that only require simple decoding algorithms that the UVC decoder could handle without any problem could be applied. We can get the raster image of a page by invoking a graphical interface called Gsview (interface for AFPL Ghostscript [Aladdin 2001]); it produces the BMP image of the whole document. A simple Java program partitions the file into separate pages.

### **Step 2: Getting the imbedded information**

The information imbedded in the PDF file consists of the text itself with its presentation attributes, the images, the bookmarks, and a few fields of descriptive information.

#### **Step 2.1: Extracting the text and images**

We save the textual information separately from the page images. We do that by saving a sequence of homogeneous elements; homogeneous here means that all presentation attributes such as font, size, embellishment, etc., are the same for all characters. Although we do not need to save the fonts themselves (since the page image already

shows the exact look of the page), we believe that knowing that an element is bold, underlined, italic, small, large, etc. is a relevant part of the information. This is more than presentation; it conveys the importance of a certain word or phrase and may be useful to answer queries such as "find documents with preservation in italics".

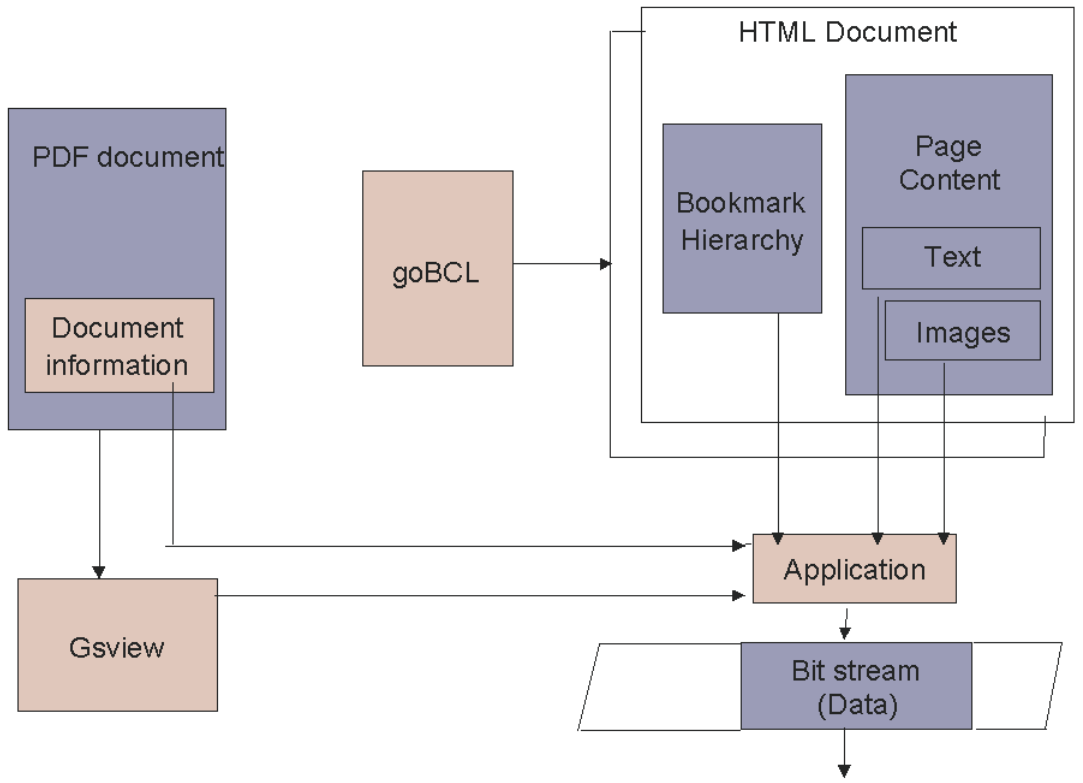


Figure 5.1 / Extracting text from a PDF file

Figure 5.1 illustrates the process. The first step consists of converting the PDF document into another, format, HTML, which is easier to interpret. Tools for such a conversion exist; we use the one provided (on the Web) by BCL-Computers [BCL 2002]. The result of the conversion is a set of HTML files and images. The images are in JPEG format and will be saved as is in the bit stream. The text elements and their presentation attributes are available, piece by piece, in the HTML text, and can be easily extracted.

### Step 2.2: Extracting the descriptive information from the PDF file

The next step consists of extracting the descriptive information associated with the document. The information is extracted directly from the PDF file. Its relative address in the file is easily identifiable. The document information section consists of data related to the creation of the document. Its schema is as follows

ELEMENT 10 [Document\_information] (11?, 13?, 15?, 17?, 19?, 21?, 23?, 25?)  
 ELEMENT 11 [Creator] (CHAR)  
 ELEMENT 13 [Creation\_date](CHAR)  
 ELEMENT 15 [Producer] (CHAR)  
 ELEMENT 17 [Keywords] (CHAR)  
 ELEMENT 19 [Subject] (CHAR)

ELEMENT 21 [Title](CHAR)  
 ELEMENT 23 [Author] (CHAR)  
 ELEMENT 25 [Modification\_date](CHAR)

with the following comments:

- 11 Software used to create the document.
- 13 Document creation date.
- 15 Software used to convert the document to PDF.
- 17 Keywords describing the document contents.
- 19 Subject of the document.
- 21 Title of the document
- 23 Author's name.
- 25 Date last modified.

**Step 2.3: Extracting the bookmark information**

A bookmark establishes a mapping between a name (or a short text) and a certain page in the document. A PDF document may contain a series of bookmarks organized hierarchically. Often the bookmark names are identical to the titles of sections in the document. Although this is not a requirement, we assume it to be true in the example shown in Figure 5.2.

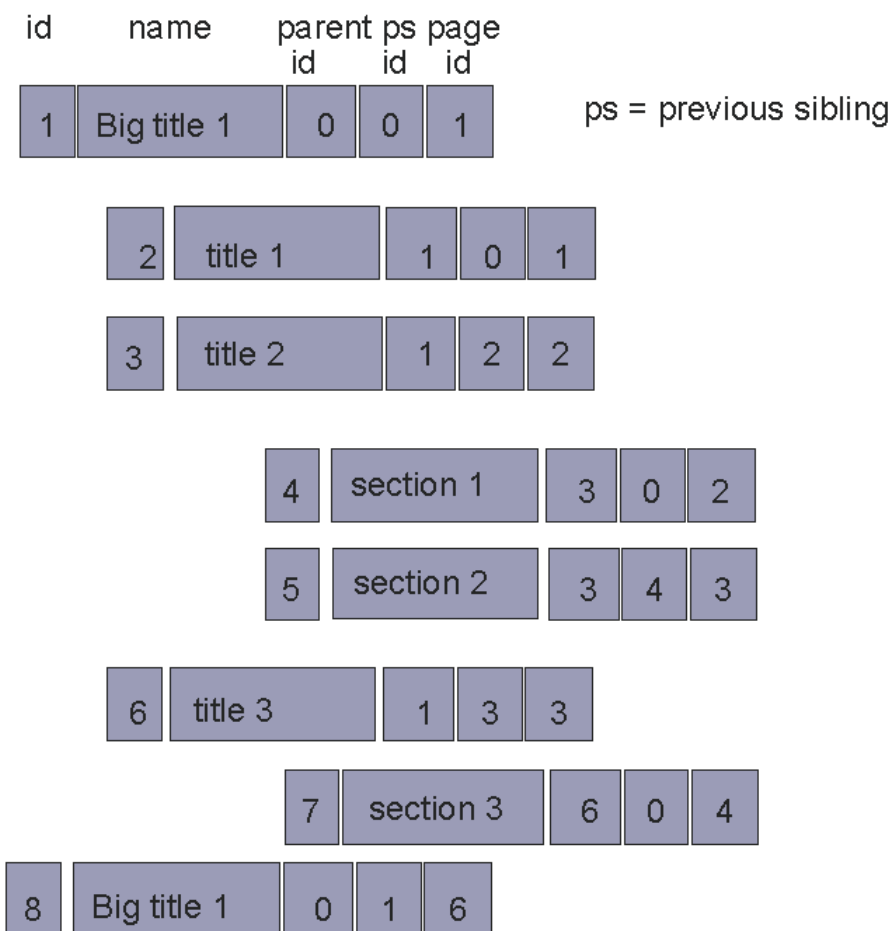


Figure 5.2 / Bookmark structure

The hierarchical bookmark structure can be represented in the logical view by a list of quintuples, as in:

```

ELEMENT 30 [Bookmark_hierarchy] (31+)
ELEMENT 31 [Bookmark] (32, 33, 34, 35, 36)
ELEMENT 32 [Bookmark_id](NUM)
ELEMENT 33 [Bookmark_name] (CHAR)
ELEMENT 34 [Parent_id](NUM)
ELEMENT 35 [Previous_sibling_id](NUM)
ELEMENT 36 [Page_id] (NUM)

```

The five elements in a bookmark quintuple respectively denote the identifier of the bookmark (an integer number), its name (any piece of text), the id of its parent, the id of its previous sibling, and, finally, the identifier of the page associated with the bookmark. In our implementation, a Java program retrieves such data from a small file generated by the BCL PDF to HTML converter. For the example, the sequence of values returned, together with their tags is:

```

1      Big title 0      0      1
2      Title 1  1      0      1
3      Title 2  1      2      2
etc.

```

The quintuple in bold shows that the "Title 2" bookmark has been assigned the id 3; it is under the parent quintuple 1, and follows sibling one (number 2). The bookmark "points" to page 2.

### Step 3: saving information from pages

#### Step 3.1: the text

The way the information is stored internally is not that important; it must be as compact as possible and it must be easy for the UVC decoding program to decode. The schema that defines the logical view is shown below. A page has several attributes: an identification, a height and a width. It contains objects of three types: lines, general images and rectangular images (a rectangle with a single color). A line is made up of segments; a general image has a position on the page (X\_ and Y\_ coordinates), a width and a height. A rectangular image has a pair of coordinates, a width and height, and a color. Note that we have not integrated the view for a general image (see step 5).

```

ELEMENT 38 [Page] (39, 42, 43, 44+, 83*, 144*)
ELEMENT 44 [Line ] (45, 46, 47, 48+)
ELEMENT 48 [Segment] (49, 50, 57, 58, 59, 60, 61, 62, 63, 64)
ELEMENT 83 [General_Image ](69, 70, 71, 72 )
ELEMENT 144 [Rectangle_Image](140, 141, 142, 143, 150)

```

```

ELEMENT 39 [Identification](NUM)
ELEMENT 42 [Height](NUM)
ELEMENT 43 [Width](NUM)
ELEMENT 45 [Y_coordinate](NUM)
ELEMENT 46 [X_coordinate](NUM)
ELEMENT 47 [Width](NUM)
ELEMENT 49 [Font_size](NUM)
ELEMENT 50 [Color](CHAR)
ELEMENT 57 [Weight](NUM)
ELEMENT 58 [Style](NUM)
ELEMENT 59 [Font_family](NUM)
ELEMENT 60 [Variant](NUM)

```

ELEMENT 61 [Decoration](NUM)  
 ELEMENT 62 [Transform](NUM)  
 ELEMENT 63 [Vertical\_align](NUM)  
 ELEMENT 64 [String](CHAR)  
 ELEMENT 69 [Y\_coordinate](NUM)  
 ELEMENT 70 [X\_coordinate](NUM)  
 ELEMENT 71 [Width](NUM)  
 ELEMENT 72 [Height](NUM)  
 ELEMENT 140 [Y\_coordinate](NUM)  
 ELEMENT 141 [X\_coordinate](NUM)  
 ELEMENT 142 [Height](NUM)  
 ELEMENT 143 [Width](NUM)  
 ELEMENT 150 [Color](CHAR)

Appendix E shows the entire logical view, with additional comments regarding the value ranges.

### Step 3.2: Saving images

The BCL conversion from PDF to HTML produces a JPEG file for each image. These files are archived as is.

### Step 3.3: Saving the schema information

This step requires that we generate a bit stream containing an internal representation of the logical schema for the PDF data. In order to implement the schema for the PDF application, we rely on the general purpose tool described in Chapter 4.

The tool accepts a description of the logical view of the data as input data. The form of the input is precisely what we have used above when we described the logical view. The program parses the view definition, checks for mistakes, packages the information in memory and outputs it as a bit stream. In the process, it displays a pop-up window allowing the user to see the hierarchical structure at a high level or to go deeper and deeper in its details.

### Step 4: Writing the UVC program to decode the data

This is one of the most interesting aspects of the study. The UVC program for decoding the data has two parts. The first one decodes the textual information, the bookmark information as well as the descriptive data; the second one decodes JPEG images. This is important because the images found on a page are archived as JPEG images. (Remember that, at this point, the page image itself is kept as an uncompressed raster bit map, and is not integrated with the rest of the information (see Chapter 8).

As explained earlier, the first program needs to decode data that has been packaged into a bit stream during the archiving process (see section 3.3, option 2). Actually, the corresponding UVC program is rather straightforward since the information is stored in the order in which the elements are returned (a streamed process). The number of fields that have to be processed is substantial, however, and the program is about 2000 instructions long.

On the other hand, the decoding of a JPEG image is a different story. Here, we are dealing with option 1: we do not control the way the information is organized internally. JPEG essentially utilizes a very powerful compression method; it is also quite complex since it involves a series of mathematical algorithms designed for efficient storage and speedy execution.

The JPEG standard defines a suite of data encoding specifications for full-color continuous-tone raster images. It includes different modes of operation. However one of these, known as the Sequential Discrete Cosine transformation, is used much more

frequently than the others. We restricted our effort to that case since it handles a broad range of images on the Web as well as the images produced by the BCL process.

The decoding program works on the data file (loaded in memory by the Restore program), applies mathematical algorithms to all 8 by 8-pixel blocks, and generates a new image data structure which contains the decoded information. The decoding program contains about 2000 instructions as well.

#### **Step 5: Writing the UVC program to decode the image schema**

Since we created the schema information by using the general tool described in Chapter 4, decoding the bit stream is straightforward. Here as well, we wrote a general purpose program that reads the metadata bit stream and displays the information in a pop-up window style similar to the one used in step 3.3.

#### **Step 6: Constructing the bit stream**

After all of step 3 is done, the complete information is available, ready to be stored as a bit stream and archived.

## 5.3 Recovering the Information (in the Future)

Chapter 3.2 mentions the three steps required. Using the implemented UVC interpreter we were able to decode the portion of the bit stream that contained the schema information and the portion that contained the textual information. The restore program that we developed not only restores the data but uses it in a rather fancy way. It displays the page image and lets the user specify a query such as "Find the occurrences of a certain text in the displayed document where a certain text occurs in italics". The result displays the pages that satisfy the criterion and highlights the matching occurrences of a certain text. It illustrates how a future user will be able to retrieve the image, the text (with all its presentation attributes) and the correlation between the text and the image.

We separately recreated JPEG images that were extracted from the pages. However, we did not use them in the Restore application since we display the full raster of the page anyway (see Chapter 8 regarding future work).



# 6/

## 01 a note on 010000010 the UVC Convention

We define the UVC Convention as the set of information items that must be recognized today and preserved indefinitely. The convention includes:

- ∄ The UVC architecture document.
- ∄ The interface to the UVC emulator.

A future client must know how to invoke the UVC interpreter with the requisite information. First of all, the restore program invokes the UVC interpreter with the requisite information, first for 'Open', then for multiple 'Get Next's. The call passes a certain number of pointers to the interpreter, which are the addresses of some bit strings or slots. The interpreter makes sure that these pointers become the actual pointers to memory locations for <space 0, address 0>, <space1, address 0>, and so on for spaces 2, 3, and 4. The only other convention relates to the code in <space 0, address 0>: The first 8 bits at address 0 in space 0 is 00000000 for the open and 10000000 for subsequent calls for 'Get Next'. The content of the other arguments is left up to the designer of the UVC program (obviously, that information must be recorded in the metadata).

- ∄ The schema for reading the schemata (the definition of the logical view of a schema).

Anything else is an individual choice that is relevant at a specific time but does not need to be permanent.

*A future client must know how to invoke the UVC interpreter with the requisite information. First of all, the restore program invokes the UVC interpreter with the requisite information, first for 'Open', then for multiple 'Get Next's.*



## 7.1 The UVC

The PDF archiving application is the first real test of the UVC-based approach for long-term preservation. It has been an excellent test bed for the UVC itself, the emulator, the assembler and the program for defining logical views. The decoding of a JPEG file, in particular, posed serious challenges for the UVC and the feedback has been quite positive.

As far as the UVC architecture is concerned, we did not need to modify the current specifications; we did not need to add any instructions or constructs. The memory model (bit-addressable) and the general load/store/move instructions to/from any bit address proved to be ideal for the kind of bit manipulation needed to analyze and construct bit streams.

One construct appeared somewhat too low level: the manipulation of the sign associated with a numeric value. Loading a signed value from memory into a register currently requires five instructions; two new instructions would reduce that number to 1; we plan to incorporate that change.

During the development of the JPEG decoder, it quickly became clear that defining variables in the assembler could be cumbersome. The addition of a shared segment containing shared variables (and particularly constants) resolved this problem quite efficiently.

## 7.2 The Application: Archiving PDF Documents

We successfully demonstrated that the concepts are quite sound, can be applied to archiving PDF documents, and can be implemented.

We archive an image for each page of a document. We did not try to compress the image at this time because we want to build and evaluate some specific compression methods in a subsequent phase of the project. We also extracted all textual elements from the document including their presentation characteristics and positions as a page. For some environments, archiving the page layout may be sufficient. However, archives are of greater value if they can be accessed based on content. Without making any decision on how the information should or should not be indexed, we show how the information could

at least be made available. To demonstrate this feature, we prepared a demo that illustrates how a future application may want to use the restored data.

The application interface accepts a query to find a specific word or group of words that appears in a document with certain presentation characteristics. The application finds the page, displays it, and highlights the words that match the keywords in the query. This fully tests the correlation between image and text.

Extracting the information from the PDF file was more complicated than we anticipated, even if we rely on available tools. We successfully extracted some data from the PDF file directly, but used the BCL tools for the bulk of the information. In general this worked well, but we encountered some small problems: a single text element was not decoded correctly into ASCII, and some special characters were not returned correctly. We anticipate that BCL will progressively eliminate these minor bugs.

*We successfully demonstrated that the concepts are quite sound, can be applied to archiving PDF documents, and can be implemented.*

# 8/ future work



We look at future work in two ways: 1. continuing our research into various facets of the technology, including the development of an enhanced prototype, and 2. applying the results of the research to the specific problems of a deposit library.

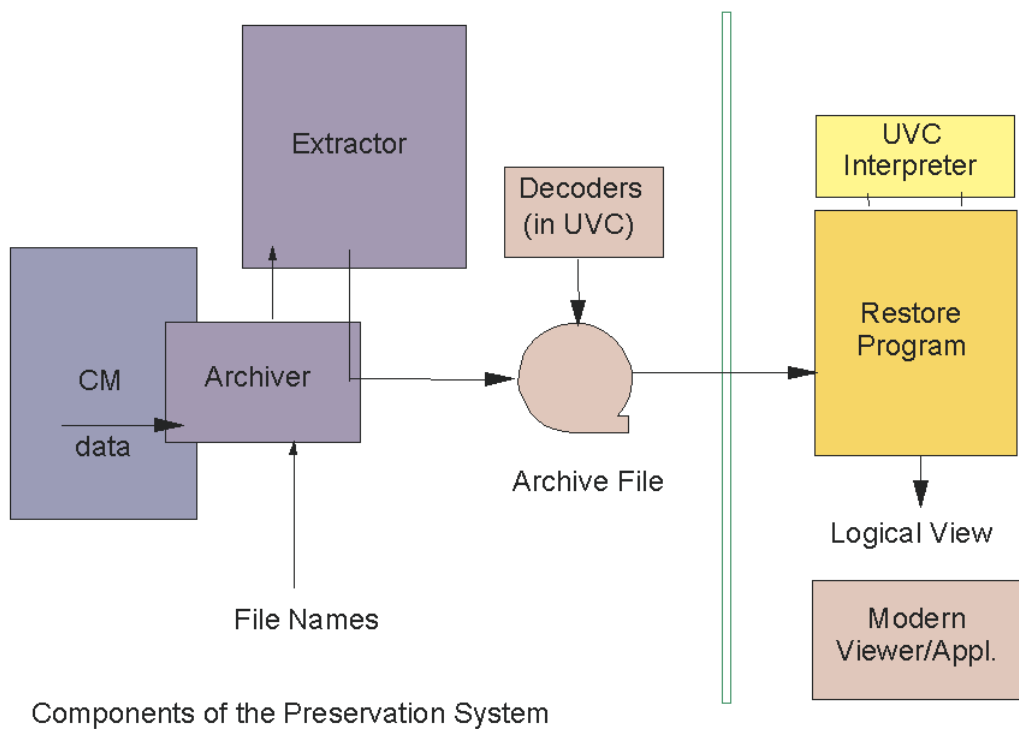


Figure 8.1 / The components of the system

## 1. Continued research effort

One of the goals is to construct an end-to-end prototype. It would act as an extension of a digital library system such as the IBM Content Manager (CM), as shown in Figure 8.1.

The CM contains the documents and all associated metadata; this includes multimedia data that will generally be archived as is (images, sound tracks, videos), and data accessed by complex - and sometimes proprietary - editors or viewers (Word, PDF, etc.).

As input the archiver receives a list of document identifiers specifying which documents are to be archived for the long term. That list may be drawn up using the existing CM facilities (for example, by asking a query). The archiver handles each document sequentially.

For a document, the archiver retrieves all the data needed from the CM and stores it in the bit stream, together with the relevant schema information and the UVC programs. Sometimes, the information can be saved as is; this is the case for multimedia files such as pictures or sound. Sometimes, the information must be extracted from the file, and repackaged before being stored in the bit stream as is the case for PDF documents. The archiver invokes the extractor to perform that task.

Once the archiver has gathered all the needed information, it can finish up the bit stream. In the future, the bit stream will contain enough information to recreate the initial data.

One of the interesting problems is deciding on the functionality for the extractor. In our PDF experiment, we rely on tools for decoding the document. Although this may be the best alternative in certain cases, it is worthwhile to investigate cases where some explicit text is already available (such as HTML, XML, SGML) or where some information can be extracted by document analysis of the raster image.

Once we archive documents a batch at a time, it becomes possible to factor out some information required by all or a subset of the documents. But some careful design is needed so that documents can be restored selectively. We also need to address the compression of page images, which are currently kept as raster images.

## **2. On the application side**

The joint study focussed on archiving the content of a PDF document. We want to address the few problems that were identified. In particular, the tools for extracting the text from PDF documents have evolved and our next prototype should take advantage of the new capabilities. We also plan to look at other document types.

Our broad goal is to demonstrate that the UVC-based approach is a viable technology for preserving documents over the long term.





# 01 appendix a: 1000010 references

[Adobe 2000]

Adobe Systems Incorporated, *PDF Reference, Second Edition*, Adobe Portable Document Format, version 1.3, Addison-Wesley, 2000,  
<http://partners.adobe.com/asn/developer/acrosdk/docs/PDFRef.pdf>.

[Aladdin 2001]

Aladdin Enterprises, *AFPL Ghostscript*, February, 2001,  
<http://www.cs.wisc.edu/~ghost/index.htm>.

[BCL 2002]

BCL Computers, Inc., *Welcome to goBCL*, <http://www.gobcl.com>.

[Brown and Shepherd 1995]

C. W. Brown and B. J. Shepher, *Graphics File Formats, reference and guide*, Manning Publication, 1995.

[Lorie 2001a]

Lorie, R.: *A Project on the Preservation of Digital Data*, RLG DigiNews, Volume 4, number 3, June 2001.

[Lorie 2001b]

Lorie, R., *Long-Term Preservation of Digital Information*, Joint Conference on Digital Libraries, ACM/IEEE, June 2001.

[Lorie 2001c]

Lorie, R., *Preserving Digital Information, an Alternative to Full Emulation*, Zeitschrift für Bibliothekswesen und Bibliographie, Frankfurt, Germany, 3-4/ 2001.

[Ockerbloom 2001]

Ockerbloom, J., *Archiving and Preserving PDF Files*, RLG DigiNews, Volume 5, Number 1, February 2001.

[Rothenberg 1995]

Rothenberg, J., *Ensuring the Longevity of Digital Documents*, Scientific American, (272)1, January 1995.

[Rothenberg 1999]

Rothenberg, J., *Avoiding Technological Quicksand: Finding a Viable Technical Foundation for Digital Preservation*, NFS Workshop on Data Archiving & Information Preservation, Washington, DC. March 1999.

[XML 2002]

World Wide Web Consortium, *XML Extensible Markup Language*, <http://www.w3.org/XML>.

# 01 appendix b: 01000010 glossary

**BMP:** BitMap format, Microsoft.

**Bookmark:** in PDF, a label that is associated with a specific page (or section) of a document. Bookmarks can be organized hierarchically.

**Content Manager (CM):** IBM Content Manager, a complete set of capabilities for capturing, storing and disseminating any kind of content.

**Conversion:** the process of modifying data and/or programs in order to make the data available on a new system.

**Data Preservation:** a process that ensures that data used today can be accessed in the future, even if the original system is no longer available.

**Digital Information Archiving System (DIAS)** is the core of the KB's electronic deposit system. Version 1 has been developed by IBM and was released in October 2002.

**DNEP:** In September 2000 the KB and IBM Netherlands signed the final contract which initiated the project "Depot voor Nederlandse Electronische Publicaties" (DNEP) [Deposit for Dutch Electronic Publications] to design and implement DIAS with a Long-Term Preservation Study as an integral part of the total effort.

**Emulation:** the process of executing a binary program on a new system by systematically simulating the behavior of every instruction.

**Hypertext Markup Language (HTML):** This is the set of markup symbols or codes inserted in a file intended for display on a World Wide Web browser. The markup tells the Web browser how to display a Web page's words and images to the user. Each individual markup code is referred to as an element (but many people also refer to it as a tag). Some elements come in pairs that indicate when a particular display effect is to begin and when it is to end.

**JPEG:** Joint Photographic Experts Group - a method for "Digital Compression and Coding of Continuous-Tone Still Images".

**KB:** The National Library of the Netherlands (Koninklijke Bibliotheek, KB).

**Logical view of the data:** a view of the data that is easily understandable because it follows the way the user normally thinks about the data, rather than the internal representation often designed for efficiency.

**PDF format:** format designed by Adobe, as part of the Acrobat products, and used to describe the appearance of a page when printed or displayed.

**Program Preservation:** process that ensures that the behavior of a program used today can be re-enacted on a different system in the future.

**Schema:** a formal definition of the logical view of the data.

**Schema for reading schemata:** a formal definition of the logical view of a schema.

**Universal Virtual Computer** or UVC is a virtual machine specially designed to develop programs today that will be able to run on a future machine by simply writing an emulator of the UVC in the future.

**XML:** Extensible Markup Language.

# 01 appendix c: 01000010 the UVC architecture

This document presents an informal view of the Universal Virtual Computer. It is not the document that will be passed on to future generations so that they may build an emulator from scratch; but it gives a good idea of what the virtual computer is. The UVC architecture relies on concepts that have existed since the beginning of the computer era: memory, registers, and a set of low-level instructions. The fact that the computer is virtual and that performance is of secondary importance, allows for a simpler, more logical, design.

## The memory model

The memory model is that of a segmented store. A segment contains a set of registers (the number is arbitrary) and a sequential bit-addressable memory. An address in the sequential memory is always specified through a register. Such a register "points" to a register that contains an integer or to the beginning of a slot in the bit-addressable memory. The register length is unlimited; an integer value occupies as many bits as necessary to the right of the register. The sign is a separate bit. Figure C.1 shows the addressing scheme for a store operation storing a 4-bit value '0110' from register <8, 3> to slot <9, 7>. A pair such as <8, 3> means segment 8, register 3. The value of <8, 4> is '100' or 4, indicating the number of bits involved in the operation.

In the remainder of this paper,  $R[s1]$  denotes the array of pointer registers for segment  $s1$ , and  $R[s1][r1]$  "points" to the actual register. In Figure C.1, the  $R[8][4]$ ->value is 4.

## The program unit

A program is made up of multiple units that call one another. A unit is a sequence of instructions; the first instruction (**start**) assigns a number to the unit. It is followed by a section of instructions that create constants and/or reserves space for variables associated with that section. The **ndc** (numeric define constant) instruction specifies a numeric register and the value that it should contain.

The following section (the executive section) manipulates data. For example, the instruction **nload** (numeric load) has two operands (two registers). It copies the content of the second register into the first one. **Add** adds the content of the second register to the first register. The instruction **ncmp** compares the first operand to the second one and

sets the machine **CC** (condition code) to minus, zero or plus if the content of the first register is less than, equal to, or greater than the content of the second one.

The flow of execution, normally sequential, can be altered by a branch instruction, based on the **CC**. A break instruction terminates the execution of a unit. There may be more than one break instruction in a unit.

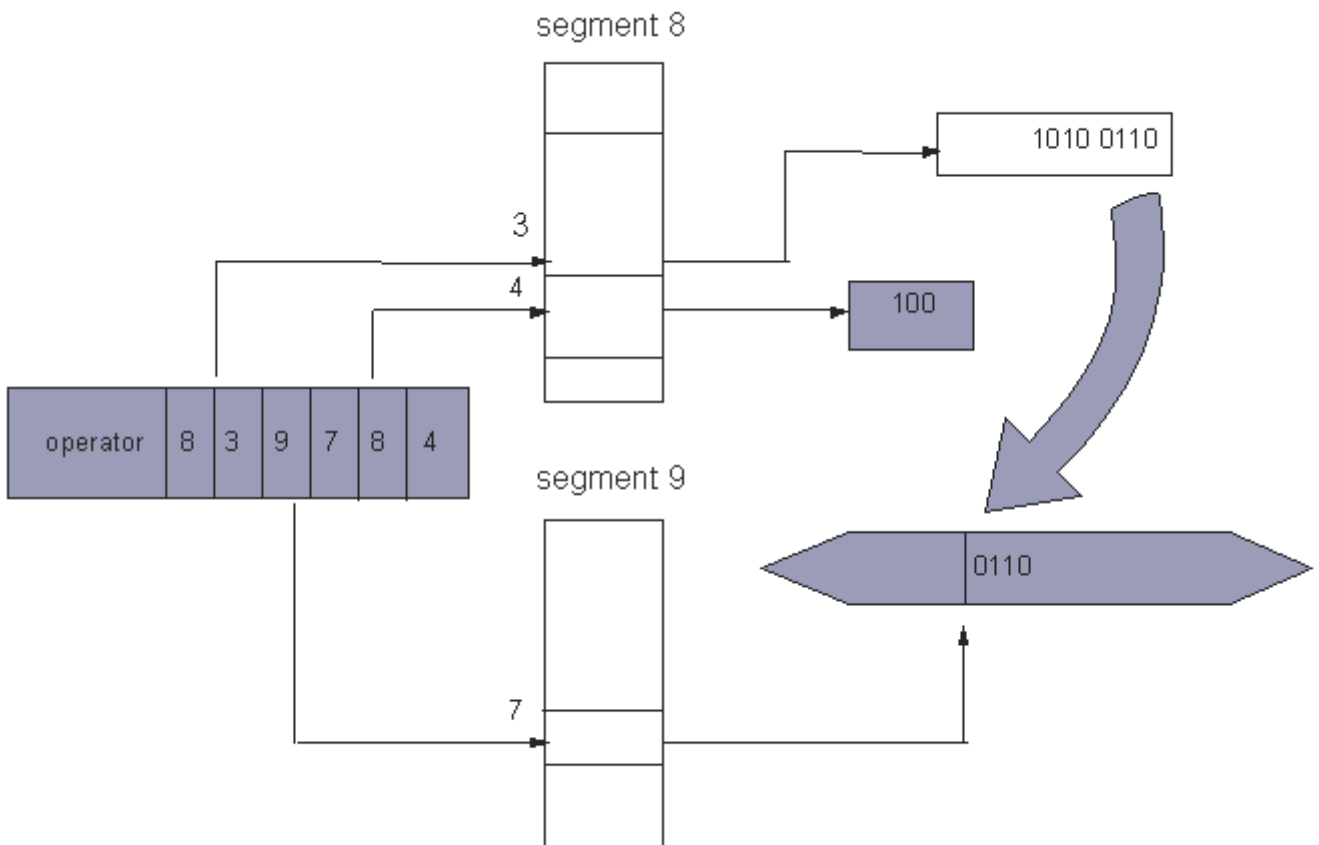


Figure C.1 / UVC memory model

### Dealing with numerical values (integers)

Following the conventions defined earlier, the following instructions deal with numerical values:

**ndc** s1, r1, n  
 Numeric define constant  
 Sets the R[s1][r1]->value to n.

**nds** s1, r1  
 Numeric define storage  
 Reserve a register identified by r1.

**add** s1, r1, s2, r2

Addition

$R[s1][r1] \rightarrow \text{value} = R[s1][r1] \rightarrow \text{value} + R[s2][r2] \rightarrow \text{value}.$

**subt** s1, r1, s2, r2

Subtraction

$R[s1][r1] \rightarrow \text{value} = R[s1][r1] \rightarrow \text{value} - R[s2][r2] \rightarrow \text{value}.$

**mult** s1, r1, s2, r2

Multiply

$R[s1][r1] \rightarrow \text{value} = R[s1][r1] \rightarrow \text{value} * R[s2][r2] \rightarrow \text{value}.$

**div** s1, r1, s2, r2, s3, r3

Divide

$R[s1][r1] \rightarrow \text{value} = R[s1][r1] \rightarrow \text{value} / R[s2][r2] \rightarrow \text{value}.$

The  $R[s3][r3] \rightarrow \text{value}$  is the remainder.

**ln** s1, r1, numeric literal

Load numeric

$R[s1][r1] \rightarrow \text{value} = \text{numeric literal}.$

**psign** s1, r1

Set sign to positive

$R[s1][r1] \rightarrow \text{value} = |R[s1][r1] \rightarrow \text{value}|.$

**nsign** s1, r1

Set sign to negative

$R[s1][r1] \rightarrow \text{value} = -|R[s1][r1] \rightarrow \text{value}|.$

**reset** s1, r1

Reset to zero

$R[s1][r1] \rightarrow \text{value} = 0.$

**nload** s1, r1, s2, r2

Numeric load

$R[s1][r1] \rightarrow \text{value} = R[s2][r2] \rightarrow \text{value}.$

**ncmp** s1, r1, s2, r2

Numeric compare

Compares ( $R[s1][r1] \rightarrow \text{value}$ ) with  $R[s2][r2] \rightarrow \text{value}$ . Sets up the condition code; this is then used in a branch instruction.

The full specification of the branch instruction is as follows:

branch code, label

where code is:

0 unconditional.

1 if the value returned by ncmp is 0.

2 if the value returned by ncmp is less than 0.

3 if the value returned by ncmp is greater than 0.

4 if the value returned by ncmp is less than/equal to 0.

5 if the value returned by ncmp is greater than/equal to 0.

6 if the value returned by ncmp is different from 0.

and label is the instruction number of the branch target address.

The only storage entities covered above are numeric registers. A common implementation of the UVC should support hundreds of registers. A register is of variable length but the significant bits are at the right of the register. A register expands to the left as needed.

### Dealing with the bit-addressable memory

As mentioned above, a segment also contains a sequential bit-addressable memory. (The size is arbitrarily). Instructions that access data in the memory refer to the data by a number  $p1$ .  $R[s1][p1]$  contains the address where the data starts; it is actually an integer representing a displacement from the initial memory address. In the store operation illustrated in Figure C.1,  $R[9][7]$  points to the area that is receiving the value '0110'.

The list of instructions used to manipulate data in the sequential memory is as follows:

**cds**  $s1, r1, k$   
Character define storage;  
Sets  $R[s1][r1]$  = address of a slot of memory of length  $k$ .

**cdc**  $s1, r1, 's'$   
Character define constant; the constant is the bit string  $s$   
Sets  $R[s1][r1]$  = address of a slot of memory of length equal to the length of  $s$   
 $R[s1][r1] \rightarrow \text{value} = s$ .

**load**  $s1, r1, s2, p2, s3, r3$   
Load from memory to register  
Sets  $R[s1][r1] \rightarrow \text{value}$  to the binary value represented by  $k$  bits, starting at displacement  $R[s2][p2]$  in memory; the length  $k$  is  $R[s3][r3] \rightarrow \text{value}$ .

**store**  $s1, r1, s2, p2, s3, r3$   
Store in memory  
Stores the rightmost  $R[s3][r3] \rightarrow \text{value}$  bits from  $R[s1][r1] \rightarrow \text{value}$  in memory at address  $R[s2][p2]$ . This is the reverse of load.

**ccomp**  $s1, p1, s2, p2, s3, r3$   
Compare characters  
Compares  $R[s3][r3] \rightarrow \text{value}$  bits from address  $R[s1][p1]$  to the same number of bits from address  $R[s2][p2]$ . Sets up the condition code as specified in `ncmp`.

**move**  $s1, p1, s2, p2, s3, r3$   
Move data from one address in memory to another.  
Moves  $R[s3][r3] \rightarrow \text{value}$  bits from address  $R[s2][p2]$  to address  $R[s1][p1]$ .

### Dealing with addresses

The instruction set also contains instructions that manipulate addresses such as  $R[s][p]$  rather than data such as  $R[s][p] \rightarrow \text{value}$ . The following two instructions are similar to load and save but they load and save an address in  $R[p1]$ .

**la**  $s1, r1, s2, p2, s3, r3$   
Load address  
Loads the value represented by the binary sequence starting at address  $R[s2][p2]$  in  $R[s1][r1]$ ; the length of the binary sequence is  $R[s3][r3] \rightarrow \text{value}$ . Thus,  $R[s1][r1] = R[s2][p2] \rightarrow \text{value}$ .

**sta** s1, r1, s2, p2, s3, r3  
 Store address; this is the reverse of load address (la).  
 Stores the rightmost (R[s3][r3]->value) bits in R[s1][r1] at addressR[s2][p2].

**incr** s1, p1, s2, r2  
 Increment address.  
 $R[s1][p1] = R[s1][p1] + R[s2][r2]$ ->value.

**copya** s1, p1, s2, p2  
 Copy address  
 Sets R[s1][p1] equal to R[s2][p2].

### Multiple units and invocation of a unit by another unit

A unit can invoke another unit using a **call** instruction. The **call** instruction uses two operands: the "address" of the unit being called and the number of a segment that contains the argument(s). For example, a **call** instruction invoking a unit starting at instruction number 123, with arguments in segment 9 would be written as:

**call** 123, 9

There is no explicit specification of what the arguments are or how many there are. Both the caller and the unit itself know which registers or memory slot contain the arguments. The called unit always sees the arguments in space number 1. The layout of the bit memory must be re-specified in each unit, using **ndc**, **cds**, or **cdc**, and must be the same as the one defined in the calling unit.

### Miscellaneous

The following instructions are only used for debugging:

**nprt** s1, r1  
 Prints R[s1][r1]->value.  
**cpri** s1, p1  
 Prints the first 80 bits of string starting at P[s1][p1].  
**apri** s1, p1  
 Prints P[s1][p1].

### Interface

A future client must know how to invoke the UVC interpreter with the requisite information. The restore program calls the interpreter in a loop, starting with an "Open", and followed by multiple "Get Next" instructions. The call passes a certain number of pointers to the interpreter, which are the addresses of some bit strings or slots. The interpreter then makes sure that these pointers become the actual pointers to the memory locations for <space 0, address 0>, <space1, address 0>, and so on for spaces 2, 3, and 4. The only other convention relates to the code in <space 0, address 0>: the first 8 bits at address 0 in space 0 is 00000000 for the open instruction and 10000000 for subsequent Get Next calls. The content of the other arguments is up to the person who designs the UVC program (obviously that information must be recorded in the metadata).



# 01 appendix d: 1000010 UVC assembler

The assembler is designed to facilitate writing a UVC program. It is important to note that it is not part of the UVC Convention. Actually, in a full-fledged implementation of a preservation methodology, there will be compilers that will take a high-level language code and compile it into a UVC machine code, probably without going through an assembly phase.

The implemented assembler supports a symbolic form of all instructions. It also permits different units to be written separately; names only have to be unique within a unit. All units are assembled at the same time making a link phase unnecessary.

One of the new features of the assembler is that it supports a shared segment. Variables (registers and memory slots) in that segment are automatically available to all units. This helped substantially reduce the number of instructions in a non-trivial program (such as the JPEG decoding program).

The assembler also supports two very useful macros:

## 1. A For loop statement:

It resembles a 'for' statement in the C language:

```
for (x = x1; x < x2; x + x3) {  
... code  
}
```

The code delimited by the curly braces is executed for  $x = x1$ ,  $x = x1+x3$ ,  $x = x1 + 2*x3$ , etc., while  $x$  is smaller than  $x2$ .

## 2. An if... else construct:

This is similar to the same construct in C.

```
if (expression) {... code1}  
else {... code2}
```

Code1 is executed if the expression is true, otherwise code2 is executed. The syntax of the expression is restricted.



# 01 appendix e : 000010 complete logical view of a PDF document

DOCTYPE PDF\_document  
[

ELEMENT Document(10?, 29?, 38+)  
ELEMENT 10 [Document\_information] (11?, 13?, 15?, 17?, 19?, 21?, 23?, 25?)  
ELEMENT 29 [Bookmark\_hierarchy] (30+)  
ELEMENT 30 [Bookmark] (31, 32, 33, 35, 36, 37)  
ELEMENT 38 [Page] (39, 42, 43, 44+, 83\*, 144\*)  
ELEMENT 44 [Line ] (45, 46, 47, 48+)  
ELEMENT 48 [Segment] (49, 50, 57, 58, 59, 60, 61, 62, 63, 64)  
ELEMENT 83 [General\_Image ](69, 70, 71, 72 )  
ELEMENT 144 [Rectangle\_Image](140, 141, 142, 143, 150)

ELEMENT 11 [Creator] (CHAR)  
ELEMENT 13 [Creation\_date](CHAR)  
ELEMENT 15 [Producer] (CHAR)  
ELEMENT 17 [Keywords] (CHAR)  
ELEMENT 19 [Subject] (CHAR)  
ELEMENT 21 [Title](CHAR)  
ELEMENT 23 [Author] (CHAR)  
ELEMENT 25 [Modification\_date](CHAR)

ELEMENT 31 [Bookmark\_level](NUM)  
ELEMENT 32 [Bookmark\_identification](NUM)  
ELEMENT 33 [Bookmark\_name] (CHAR)  
ELEMENT 35 [Page\_identification] (NUM)  
ELEMENT 36 [Parent\_level](NUM)  
ELEMENT 37 [Parent\_identification](NUM)

ELEMENT 39 [Identification](NUM)  
ELEMENT 42 [Height](NUM)  
ELEMENT 43 [Width](NUM)  
ELEMENT 45 [Y\_coordinate](NUM)

ELEMENT 46 [X\_coordinate](NUM)  
ELEMENT 47 [Width](NUM)  
ELEMENT 49 [Font\_size](NUM)  
ELEMENT 50 [Color](CHAR)  
ELEMENT 57 [Weight](NUM)  
ELEMENT 58 [Style](NUM)  
ELEMENT 59 [Font\_family](NUM)  
ELEMENT 60 [Variant](NUM)  
ELEMENT 61 [Decoration](NUM)  
ELEMENT 62 [Transform](NUM)  
ELEMENT 63 [Vertical\_align](NUM)  
ELEMENT 64 [String](CHAR)  
ELEMENT 69 [Y\_coordinate](NUM)  
ELEMENT 70 [X\_coordinate](NUM)  
ELEMENT 71 [Width](NUM)  
ELEMENT 72 [Height](NUM)  
ELEMENT 140 [Y\_coordinate](NUM)  
ELEMENT 141 [X\_coordinate](NUM)  
ELEMENT 142 [Height](NUM)  
ELEMENT 143 [Width](NUM)  
ELEMENT 150 [Color](CHAR)

]

The following table specifies the value ranges for some of the fields; it should be inserted as comments into the definition shown above, but we keep it separate for the sake of clarity.

Attribute	Value: meaning
Font size	Value (unit is pixel)
Color	Color is three integers R, G, B.
Weight	0: normal 1: bold
Style	0: normal 1: italic 2: oblique
Font family	0: serif 1: sans-serif 2: cursive 3: fantasy 4: monospace
Variant	0: normal 1: small caps
Decoration	0: none (normal) 1: underline 2: overline 3: line-through 4: blink 5: underline, overline 6: underline, line-through 7: underline, blink 8: underline, overline, line-through 9: underline, overline, blink 10: underline, line-through, blink 11: overline, line-through 12: overline, blink 13: overline, overline, line-through 14: overline, overline, blink 15: overline, line-through, blink
Transform	0: none 1: all in lowercase 2: all in uppercase 3: capitalize (every word starts with capital letter)
Vertical align	0: normal 1: subscript 2: superscript



IBM  
long

KB  
term

preservation  
study

