Eindhoven University of Technology

MASTER

# The design of a microprocessor with an object oriented architecture

van Hamersveld, F.P.

*Award date:*
1992

Link to publication

**Disclaimer**
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

**General rights**
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

EB 367

# The Design of a Microprocessor with an Object Oriented Architecture

F.P. van Hamersveld

Faculty of Electrical Engineering
Digital Systems Group
Eindhoven University of Technology

Eindhoven, february 1992

Supervisor: Prof. Ir. M.P.J. Stevens
Coach: Ir. A.C. Verschueren

The Department of Electrical Engineering of the Eindhoven University of Technology does not accept any responsibility regarding the contents of student project- and graduation reports.

# Abstract

Object oriented programming is being used more often. A disadvantage however is that the implementation of such languages is far more complex than procedure oriented languages. If a system can be build that supports object oriented languages, then it will become even more powerful. Therefore a microprocessor will be designed with an object oriented architecture.

A frequently used object oriented language is Smalltalk-80. This language compiles its source code to bytecodes, that are executed by the interpreter.
In the literature a microprocessor is described that supports object oriented programming. This processor, called the Rekursiv, can be programmed to run Smalltalk. Hereto the Smalltalk-interpreter is implemented in microcode.

The Rekursiv consists of three processes. The Logik contains the sequencer for the microprogram and the stack addressing unit. The Numerik contains the data manipulator (32-bit alu, barrel shifter and multiplier). The Objekt contains the object oriented memory management unit.

The Rekursiv can be improved when a few mechanisms are implemented in hardware. Instead of an on-line garbage collection routine in the microprogram, a parallel garbage collector in hardware should be used. Furthermore the dynamic binding mechanism should be implemented in hardware.

To design a comparable microprocessor, the micro-instructions have to be defined, together with the hardware that implements them. Hereto a model is defined, using the modelling tool Promod. When this model is satisfactory, an architectural model can be set up using the design and simulation tool IDaSS.

The author of this master's thesis had the assignment to design a microprocessor with an object oriented architecture. Until now, the model Objekt and the model of the garbage collection have been defined. The garbage collector uses a generation scavenging algorithm that is adjusted to run in parallel with the main processor.
Due to the fact that Promod is not ideally suited to define a hardware model, there are a few errors in the Promod report that gives the model.

Before the hardware can be designed, the entire model of the processor, including the Logik, Numerik, and dynamic binding mechanism, has to be defined.

# Contents

# 1. Introduction

Object oriented programming is being used quite widely in the fields of both software engineering and artificial intelligence. Like structured programming, object oriented programming is mainly a technique of programming.

Although the object oriented point of view is not new, it is more frequently used lately. Major advantages over the more traditional high-level languages, such as Pascal, is that it reduces the complexity of large software systems, and makes these systems easier to maintain.

A great disadvantage however is that the implementation of such object oriented languages is far more complex than comparable procedure oriented languages. The high-level instructions can not be implemented as efficiently on conventional hardware, as for example Pascal-instructions can: the semantic gap between these languages and typical hardware is greater.

At the Digital Systems Group (EB) of the Faculty of Electrical Engineering of the Eindhoven University of Technology, the Interactive Design and Simulation System (IDaSS) has been developed. This design and simulation environment for digital circuits has been implemented using the object oriented language Smalltalk.

To be able to run object oriented systems efficiently in the future, research is done to architectures that support these systems better. As his master's project the author was given the assignment to design a microprocessor with an object oriented architecture.

This master thesis will start in chapter 2 by giving a short introduction into object oriented programming. In chapter 3 Smalltalk-80 and the Smalltalk-80 virtual machine are explained briefly. Chapter 4 shows the Rekursiv architecture, which forms the basis of the master project. In chapter 5 the Rekursiv is discussed and improvements made to the Rekursiv are explained. The model of the Rekursiv is given in chapter 6. Finally, conclusions and recommendations are given in chapter 7.

# 2. Object Oriented Programming

Object oriented programming systems are built on the model of communicating objects. Large applications are viewed in the same way as the fundamental units from which the system is built. The interaction between the most primitive objects is viewed in the same way as the high-level interaction between the computer and the user. Objects support modularity. The functioning of any object does not depend on the internal details of other objects.

## 2.1. Some Terminology

### 2.1.1. Objects

An object represents a component of the object oriented system. For example, objects can represent numbers, characters, strings, queues, rectangles, text editors, programs, compilers, etc. An object consists of some private memory and a set of operations. The nature of the operations depends on the type of data it represents.

A message is a request for an object to carry out one of its operations. A message specifies which operation is desired, but not how that operation should be carried out. The receiver, the object to which the message was sent, determines how to carry out the requested operation.

The set of messages to which an object can respond is called its interface with the rest of the system. The only way to interact with an object is through its interface. A crucial property of an object is that its private memory can only be manipulated by its own operations. A crucial property of messages is that they are the only way to invoke an object's operations. These properties ensure that the implementation of one object cannot depend on the internal details of other objects, only on the messages to which they respond.
This principle is called information hiding. Information hiding is important for ensuring reliability and modifiability of software systems by reducing interdependencies between software components.

### 2.1.2. Classes and Instances

A class describes the implementation of a set of objects that all represent the same kind of system component. The individual objects described by a class are called its instances. Each object, each instance has one class, a class may have multiple instances.

A class describes the form of its instances' private memories and how they carry out their operations. The object's private properties are a set of instance variables that make up its private memory, and a set of methods that describe how to carry out its operations. The instances of a class all use the same set of methods. Each instance has its own set of instance variables, but they generally all have the same number of instance variables.

Typically, a method sends messages to other objects, which invoke other methods, until you reach the point where a primitive method is invoked. The primitive methods are built into the virtual machine and cannot be changed by the programmer.
Each message-send eventually returns a result to the sender.

## 2.2. Features of Object Oriented Programming

To be object oriented, a programming language must provide four major facilities, namely:

**Information Hiding**
Information hiding is important for ensuring reliability and modifiability of software systems by reducing interdependencies between software components.

**Data Abstraction**
Data abstraction is a way of using information hiding to mask the precise way in which data is represented, so that it is accessible only through an abstract procedural interface and is easily maintainable.

**Dynamic Binding**
Dynamic binding encourages uniform programming conventions by providing polymorphism. Polymorphism allows equal naming of messages in different classes. Comparable methods of different kind of objects can have different implementations and still wear the same name. Many highly dynamic systems would be impracticable without this capability.

**Inheritance**
Inheritance enables programmers to create classes and, therefore, objects that are specializations of other objects. Creating a specialization of an existing class is called subclassing. The new class is a subclass of the existing class. The existing class is called the superclass of the new class.
The subclass inherits instance variables, class variables, and methods from its superclass. The subclass may add instance and class variables and methods that are appropriate to more specialized objects. In addition, a subclass may override or provide additional behaviour to methods of a superclass. A large amount of code can be reused this way.

These features make that object oriented programming systems provide major advantages in the production and maintenance of software: shorter development times, a high degree of code sharing, and flexibility. This makes object oriented programming an important tool for building complex software systems.

Of course, object oriented programming has a few drawbacks too. Probably the most restricting disadvantage is that the implementation of object oriented languages is more complex than comparable procedure oriented languages, since the semantic gap between these languages and typical hardware machines is greater. This is also the reason why object oriented programming systems, implemented on conventional hardware, are often insecure. Using for example a simple debugger, individual datawords can be changed, that are not to be changed in the original object oriented environment.
Another disadvantage is the run-time cost of the dynamic binding mechanism. A message-send takes more time than a straight function call in conventional machines.
If hardware can be built that supports object oriented programming better than the traditional architectures, it would result in a faster and more secure system.

This chapter only gives a very short overview of object oriented programming. For more information on this subject, the articles [Pas86], [Pok89], [RaK90], and [Rob81] are recommended.

# 3. The Smalltalk-80 Virtual Machine

Smalltalk-80 is a frequently used object oriented programming system. Two major components of the Smalltalk-80 system are the virtual image and the virtual machine.
The virtual image consists of all the objects in the system. The virtual machine consists of the hardware devices and machine language (or microcode) routines that give dynamics to the objects of the virtual image.
The system implementer's task is to create a virtual machine. A virtual image can then be loaded in the machine.
This chapter will give an overview of the Smalltalk-80 virtual machine.

## 3.1. The Syntax of Messages in Smalltalk-80

Messages are described by expressions. A message-sending expression describes the receiver , selector and arguments of the message. When an expression is evaluated, the message it describes is transmitted to its receiver. Here are several examples of message-describing expressions:

1. frame center
2. origin + offset
3. frame moveTo:newLocation
4. list at:index put:element

Each expression begins with a description of the receiver of the message. The receivers in the examples are described by variable names: frame, origin, frame, and list, respectively.
The selectors represent the actions that are desired. The selectors in the examples are: center, +, moveTo: and at:put: respectively.

Messages without arguments are called unary messages. A unary message consists of a single identifier called a unary selector. The first example is a unary message.
A binary message has a single argument and a selector that is one of a set of special single or double characters called binary selectors. The second example is a binary message.

A keyword message has one or more arguments and a selector that is made up of a series of keywords, one preceding each element. A keyword is an identifier with a trailing colon. The third and fourth example are keyword messages.

For more information on Smalltalk-80, see [Xer81] and [GoR83].

## 3.2. An overview of the virtual machine

The source methods, written by programmers, are translated by a compiler into sequences of eight-bit instructions called bytecodes. The bytecodes are instructions for an interpreter. Below the interpreter in the implementation is an object memory, that stores the objects that make up the virtual image.

### 3.2.1. The compiler

A programmer doesn't interact directly with the compiler. When a new source method is added to a class, the class asks the compiler to translate the source methods in a sequence of bytecodes. These bytecodes are held in an instance of the class CompiledMethod.
In addition to the bytecodes, a CompiledMethod contains a set of objects called its literal frame. The literal frame contains any object that could not be referred to directly by the bytecodes.

The categories of objects that can be referred to directly by bytecodes are:
- the receiver and arguments of the invoking message,
- the values of the receiver's instance variables,
- the values of any temporary variables required by the method,
- seven special constants (true, false, nil, -1, 0, 1, and 2),
- 32 special message selectors.

The objects ordinarily contained in a literal frame are:
- shared variables (global, class, and pool variables),
- most literal constants (numbers, characters, strings, arrays, and symbols),
- most message selectors (those that are not special).

### 3.2.2. The interpreter

The Smalltalk-80 interpreter executes the bytecode instructions found in CompiledMethods. The state of the interpreter consists of five pieces of information:
1. The CompiledMethod whose bytecodes are being executed,
2. The location of the next bytecode to be executed. This is the interpreter's instruction pointer,
3. The receiver and arguments of the message that invoked the CompiledMethod.
4. Any temporary variables needed by the CompiledMethod,

5. A stack.

The interpreter repeatedly performs a three-step cycle:

1. Fetch the bytecode from the CompiledMethod indicated by the instruction pointer,
2. Increment the instruction pointer,
3. Perform the function specified by the bytecode.

When a message is sent, all five parts of the interpreter's state may have to be changed in order to execute a different CompiledMethod in response to this new message. The interpreter saves its old state in objects called contexts. The context that represents the current state of the interpreter is called the active context. When a send-bytecode requires a new CompiledMethod to be executed, the active context becomes suspended, and a new context is created and made active. The suspended context is called the new context's sender. Contexts are represented by instances of class MethodContext.

Another type of context is represented by instances of class BlockContext. A BlockContext represents a block in a source method that is not part of an optimized control structure.
Blocks are sequences of instructions, which will be evaluated when they receive the message value. For example:

incrementBlock ← [index ← index + 1]

will only be executed when the message value is sent to incrementBlock:

incrementBlock value.

In this case index is incremented.
A BlockContext responds to the message value: by becoming the active context.

When a send-bytecode is encountered, the interpreter finds the CompiledMethod indicated by the message as follows.

1. Find the message receiver. The message receiver is found on the stack.
2. Access a message dictionary. The original message dictionary is found in the receiver's class.
3. Look up the message selector in the message dictionary. The selector is indicated in the send-bytecode.
4. If the selector is found, the associated CompiledMethod describes the response to the message.
5. If the selector is not found, a new message dictionary must be searched (returning to step 3). The new message dictionary will be found in the superclass of the last class whose message dictionary was searched.

An exception to the way the CompiledMethod is found, is used when a super-send is encountered. the original message dictionary for step 2 is now found in the superclass of the class in which the currently executing CompiledMethod was found.

The bytecodes used by Smalltalk-80 are enumerated in [GoR83].

### 3.2.3. The object memory

Each object is associated with a unique identifier which is called its object pointer. The object memory and the interpreter communicate about objects with object pointers. The object memory

associates each object pointer with a set of other object pointers. Every object pointer is associated with the object pointer of a class. If an object has instance variables, the object pointer is also associated with the object pointers of instance variable values.

In Smalltalk-80, instances of class SmallInteger represent the integers -16384 through 16383. Each of these instances is assigned a 1 in the low-order bit position of their object pointer, and the two's complement representation of their value in the high-order 15 bits. An instance of SmallInteger needs no instance storage since both its class and its value can be determined from its object pointer.

The object memory provides the following five fundamental functions to the interpreter.

1. Access the value of an object's instance variable.
2. Change the value of an object's instance variable.
3. Access an object's class.
4. Create a new object.
5. Find the number of instance variables a class has.

More information on Smalltalk-80 and its virtual machine can be found in [GoR83].

# 4. The Rekursiv Architecture

Ivor Tiefenbrun is the founder of Linn Products, a firm with a worldwide reputation in the audio marketplace.
In the early 1980's, Tiefenbrun was convinced that an object oriented programming system would allow Linn Products to integrate all the factory's functions with flexibility. Since only few of such systems were commercially available, Tiefenbrun decided to have such a system designed. As a result, an object oriented programming system called LINGO, with many features of Smalltalk, was written.
However, the performance of LINGO on Linn's VAX-minicomputers proved far from adequate for the task of automating the whole factory. Therefore, Tiefenbrun decided to finance the development of a new processor architecture optimized to run object oriented languages orders of magnitude faster than conventional hardware can. Thus was born the Rekursiv project.

## 4.1. The Principles behind Rekursiv

On conventional computers, the high-level primitive instructions of an object oriented programming system are implemented in machine-coded programs. Apart from the fact that these machines execute very slowly, they are insecure because of the semantic gap between the language and the machine. Objects generated by the object oriented system can easily be changed by a program written in some other language, like assembler. Therefore,what is needed is a machine which provides an object oriented store, full persistence for all types of data, and a high-level instruction set which eliminates the semantic gap entirely by directly implementing the primitive operations.

### 4.1.1. Object Representation

Objects should be stored in a form which directly reflects their structure, their type and their relationship to one another:

1. an object should be stored and moved as a unit,
2. objects should be created by system primitives only, so that they cannot be forged. The system should be the sole creator of references to objects,
3. objects should refer to one another in some non-positional manner, so that they can be moved about within memory and migrate to or from backing store without having to perform address translation,
4. the type of an object should be inseparable from it, so that it can always be identified, and made immutable so that it cannot be altered,
5. the operations available for each class of objects should be accessible from the type, so that operators can be checked when applied, to prevent data being misinterpreted.

### 4.1.2. Persistent Storage

To create a single level store, main memory should be integrated with backing store. This requires that both media employ the same storage format for objects so that objects don't have to be transformed in order to be moved to or from disk.
Such a system provides a natural form of object persistence. This property should be independent of the data type. Just as all data can be passed between processes, all data is treated equally.

### 4.1.3. Object Swapping

In an object-based system, data can be loaded into memory simply by referring to it. There is no need to read data from disk explicitly. Similarly, it will move back to disk automatically, should references to it cease to be frequent. Furthermore, there is no need to transform any data that need to be moved.
This leads to a paging strategy which swaps individual objects as and when necessary, rather than swapping in large pages of memory around the address of the desired location.

### 4.1.4. Access Checking

When highly dynamic systems are used, run-time activities must be provided, which less powerful systems don't need to support. Typically, this might involve searching symbolic namespaces during execution and dynamic type and range checking. There is a requirement for appropriate hardware support for these activities. The reasonable thing is to carry out such checks if necessary in parallel with the limitation of the access.

If object access is to be secure, it will be necessary to determine dynamically that the types of indices are appropriate to the types of the structures into which they are indexed.

### 4.1.5. High-Level Instructions

Object oriented programming systems are often, as is the case for Smalltalk, implemented by way of an interpreter which executes a machine code routine for every instruction.
It would seem only logical to try to make a machine which provides such operations as proper instructions, so that, at least the interpretational overheads can be eliminated from the implementation.
A large part of the interpreter is usually devoted to storage management. What must be designed therefore, is a machine architecture in which storage is managed below the level of the instruction set, or within the instructions, so that from the viewpoint of a running program it is automatic. This makes a hardware object store far more secure than one simulated on a conventional virtual memory system.

Since all access to the fields of an object is through a set of self-contained instructions, the individual high-level instructions can perform type and range checking to guarantee that the right type of indexing is performed and that the index only addresses valid components.
Another aspect to data security, is polymorphism. Taking an operator in addition to data, they work out dynamically what the operator means to that data. The system will detect and reject requests to perform semantically senseless operations.

## 4.2. The Architecture

The Rekursiv architecture consists of three processors, each designed for a specific task, and a couple of distinct banks of memory (see fig. 4.1).

The basic processor architecture, as outlined in fig. 4.1, is built around the socalled Objective chipset comprising three gate arrays.

1. Objekt: object oriented memory management unit,
2. Logik: microsequencer and stack addressing unit,
3. Numerik: data manipulator.

Furthermore, the Rekursiv architecture contains six memory blocks that are implemented in static RAM (SRAM) and one large bank of dynamic RAM (DRAM). This DRAM is the main object store memory. The six SRAM blocks are:

CSTK: control stack,
ESTK: evaluation stack,
CS: control store,
CSMAP: control store map,
NAM: new language abstract memory,
pager tables.

The microcode is stored in the control store which has its own 16-bits address-bus. This control store can contain up to 16,384 (in Rekursiv) control words of 128 bits each. The control store map

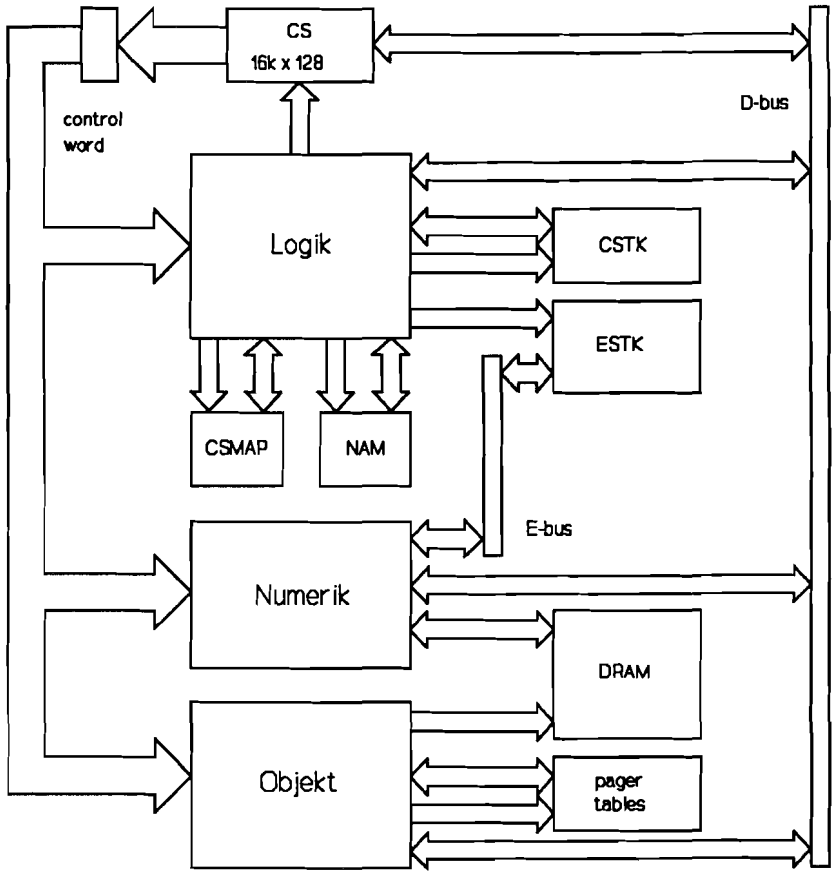



*Fig. 4.1: The Rekursiv architecture*

holds a table of 2048 microcode start addresses, and maps 10-bit opcodes onto the microcode that implements them. The control store and the control store map can be thought of as a Smalltalk-style bytecode interpreter implemented in hardware.
The NAM can store up to 524,288 words of 10-bit opcodes and their 30-bit arguments that form abstract or high-level instructions. The NAM behaves like a fast instruction cache. In a Smalltalk implementation, the abstract code in the NAM would be the methods of the most important system classes.

Usually, a major disadvantage of employing a tightly coupled cluster of processors is the overhead of communications between the processors. This disadvantage is overcome in the rekursiv by using a single control word to synchronise all the elements. This has resulted in a 128-bit wide control word, that permits many separate activities to be specified in parallel.

Such a wide control word is not effective, if contention can occur for some vital component in the system, which can only be allocated to one of the activities on a given clock cycle. If for example a variety of registers could be loaded at the same time except for the fact that all have to use the same data path to acquire their information, then that data path is a bottleneck which must be remedied if parallelism is to be realised.
This and the fact that a memory chip can be accessed only once a clock cycle, is the reason why the Rekursiv processor is heavily fragmented into many distinct banks of memory, for stacks, code and data. Each memory should be directly accessible and manipulable by the associated processing

elements, independently of the main system bus. In addition, each memory should have an appropriate system of control registers and sufficient local arithmetic logic to support the address calculation, thereby further reducing the bus traffic.
Each stack is therefore controlled by a separate addressing system. Each control register can be controlled individually, and offset addresses can be computed independently of the ALU.

In most general purpose machines, when a page fault occurs during the address translation phase of an instruction, it must back up the instruction to a point at which it can be re-entered, saving the microstate, and then invoke a system call to handle the fault. Then when the fault is fixed, the microstate must be restored and the instruction must be restarted from the middle. Clearly for an instruction which implements a high level algorithm during which memory is accessed frequently, the microstate may become very complex and the recovery process quite costly.

The Logik sequencer overcomes this limitation, because page faults are handled below the processor by the interface to the object oriented store. When a page fault (DPFLT) occurs, the Rekursiv is halted while the storage system fixes the fault by loading in the desired item from disk, updating the page tables and then restarting the Rekursiv.

Conventional microcodable machines so far have not been able to provide high-level instruction sets, simply because they don't provide nested or even recursive algorithms. To achieve this a different kind of architecture had to be developed.

The Rekursiv attaches a fixed size header to each object's storage image, thereby giving direct support to a very large number of types. This header contains an object's size and type, so they can be directly accessed.
Because backing store and main memory have to form a single object store, the storage format has to be the same in each case. The Rekursiv maps position-independent object numbers into the current physical address of objects, and will page them in if necessary.

In the memory management system are tables to hold an object's size, an object's type or class, an object's current address in memory, and the first word of the object. Each table element is indexed by the object number, modulo the table size. Each table is accessed at the same time, to fill individual output registers at the end of the paging cycle. These can be fed into range checking logic, so that checks can rapidly be carried out when the components of an object are accessed.

### 4.2.1. The Object Oriented Memory Management Unit (Objekt)

The major sub-units are shown in fig. 4.2.

The principal routes in Objekt are the VSB, VAB and VNB which are 24-bit data buses from the pager tables and represent an object's size, address, and number respectively. The 40-bit system D-bus forms a bidirectional path. PBA, a 40-bit quantity of which the low-order bits are stripped off to form an address, runs out to the pager tables, and a 24-bit address is sent out to main memory. Other inputs comprise the clock, reset signals and controlling opcodes. Other outputs are the condition code, an access validation signal, and a data page fault signal.

The object-identifier allocator is a 38-bit counter which increments to yield the object number of the next object to be allocated. The other two bits of an object number are flag-bits.

VR is a bank of eight 40-bit registers loadable from the system bus, able to source the object pager.
VNBC compares the identifier of the newly-paged object with the contents of the VNB slot addressed by that identifier. If these two quantities differ, it means that the desired object is not in memory, so that it must be fetched from backing store.
The index generator incorporates special hardware which can be microcoded to scan efficiently over the elements of objects.
The object space allocator, upon being presented with the desired size on the system bus, produces the base address that each newly created object will have.

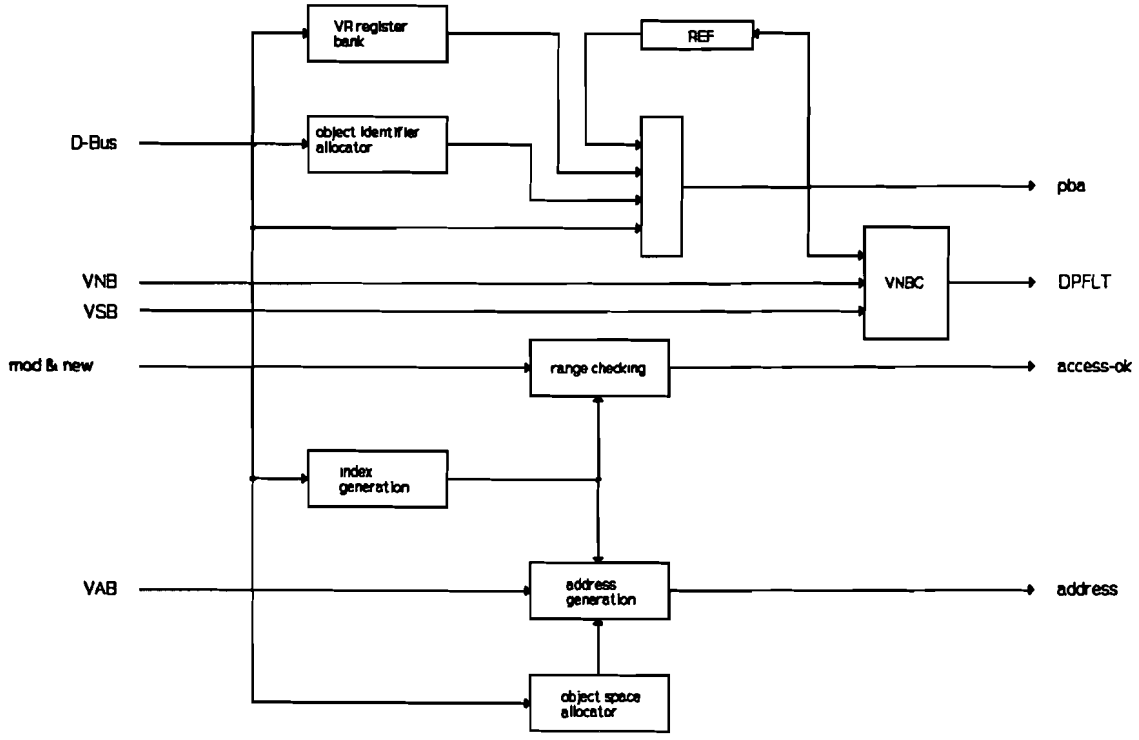



*Fig. 4.2: The internal architecture of Objekt*

Rekursiv has been designed with a brutal, but fast garbage collection strategy. The DRAM object store is divided into two halves, only one of which is used at any time. When objects have to be created that exceed the last address of the active half, more memory is required and Objekt invokes hardware garbage collection.
The garbage collector then identifies all the objects to be kept, and copies them into the free semispace, updating the entries in the address tables in the paging system. If this process still doesn't free enough memory, a second level of garbage collection is used to squeeze out infrequently accessed objects to disk. If even this fails then all persistent objects are squeezed back to disk.
These fallback strategies ensure that garbage collector performance degrades gracefully rather than failing with a bang. The sweeping of unwanted objects from disk is a software housekeeping problem that should take place off-line.

## 4.2.2. The Microsequencer (Logik)

The major sub-units are shown in fig. 4.3.

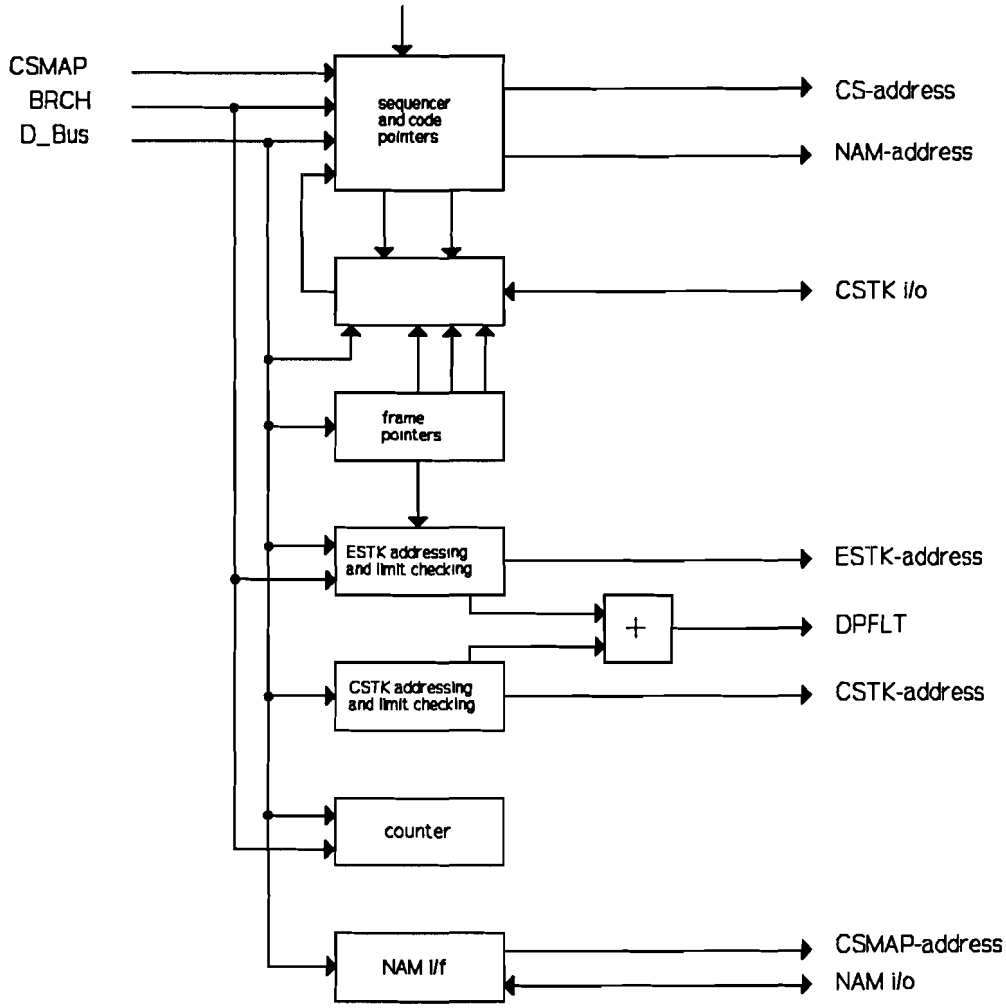



*Fig. 4.3: The internal architecture of Logik*

The microsequencer issues successive 128-bit control words.
Various addresses are produced by the Logik, including addresses for both stacks, the NAM kernel program code store and the microcode control store. The sequencer may select the next microaddress from one of its inputs or from an internal source. External sources are the control store map's microcode address register, the BRCH (a 16-bit field in the control word), the D-bus, or the top of the control stack.

A large multiplexer can be enabled to present data to the bidirectional control stack data lines, or these lines can carry the top of the control stack into the sequencer. Data that can be carried onto the control stack includes the code pointer of the microcode and the abstract codestream, three frame pointers (base of current microframe, argument pointer of the current abstract frame, or the current stackpointer), or the system bus.

More information on the operations that can be performed using the fields of the Rekursiv's control word, can be found in [Har88].

### 4.2.3. The Data Manipulator (Numerik)

As can be seen from figure 4.4, in addition to the main memory interface, the Numerik holds the 32-bit ALU, multiplier and barrel shifter. The results from the elements can be pushed out onto the evaluation stack either in 32-bit format or first be combined with an 8-bit extension specifying the type to form a compact object.

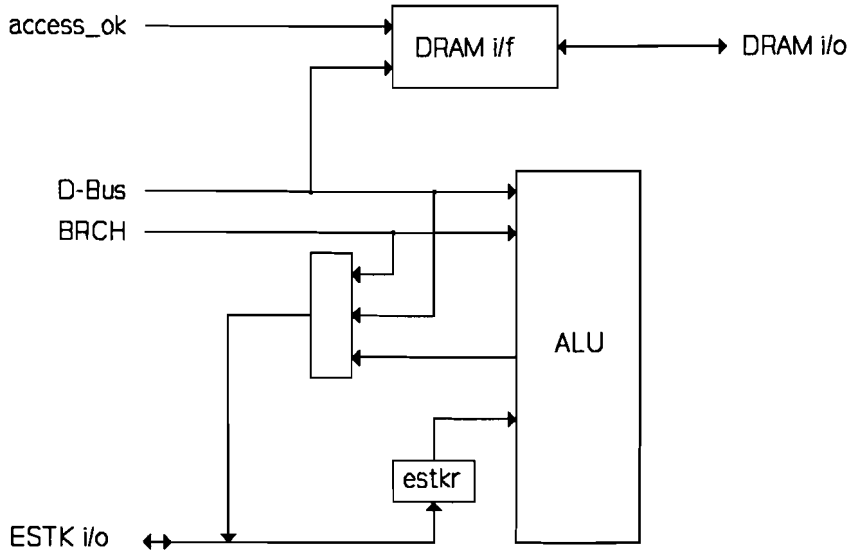



*Fig. 4.4: The internal architecture of the Numerik*

### 4.2.4. The Pipeline

An instruction takes four cycles in the pipeline, two of which are overlapped with the previous instruction. The basic sequence is:

| | |
|---|---|
| *inc apc* : | increment the abstract program counter (apc), |
| *read nam[apc]* → *mapar* : | load the opcode in the abstract code stream (nam) at the point addressed by the apc into the map address register (mapar), |
| *read csmap[mapar]* → *ucar* : | read out the control store map into the microcode address register (ucar), |
| *jv* : | jump to the microaddress specified by ucar. |

The two and two overlap will give:

```
inc apc                                  inc apc
read nam[apc] → mapar                    read nam[apc] → mapar
read csmap[mapar] → ucar                 read csmap[mapar] → ucar
jv                                       jv
```

so the shortest instruction can be reduced to two cycles.

For more information on the Rekursiv architecture, see [Har88] and [Pou88]. [HaB86-1] Describes the Objekt shortly. [Har86] and [HaB86-2] give more information on microcoding the architecture.

# 5. Evaluating the Rekursiv

This chapter gives the author's opinion on the architecture of the Rekursiv and how it could be modified in order to implement the Smalltalk-80 virtual machine efficiently.

An important principle behind the Rekursiv's architecture is that it has a writable instruction set. This means that a user can write his own instruction set, one that is best suited for his applications. Then he can load his instruction set in the control store to run his applications.
The Rekursiv does not necessarily have to run object oriented languages. For example, a C-instruction-set can be implemented as well. A C-instruction-set however will not make use of all unique features of the Rekursiv.
Because the processor is developed to run object oriented languages, and because it is easier to build, it is advisable not to give the processor a writable instruction set. This means that the microcode must be known to be correct when the processor will be produced.

As is already stated, the microcode that implements the instruction set, is stored in the CS (see fig. 4.1). The CSMAP holds a table with the microcode start addresses and maps 10-bit opcodes onto the microcode that implements them. In a Smalltalk-80 implementation, the CSMAP can use the number of the bytecode as an index, returning the start address of the microcode routine that implements the bytecode. In this way the CSMAP together with the CS act as a hardware bytecode interpreter.
Frequently performed operations can use the NAM as a fast instruction cache. For example the NAM could contain the methods of the most important system classes. Methods for user-defined classes would be stored in the main DRAM memory.

Because the microcode can use its own control stack (the CSTK), it can implement very high-level instructions, since it is now possible to use recursion and subroutine calling. This makes it possible to implement complex algorithms in microcode.
What still has to be done is to define the different fields of the control word and to provide the logic that implements these instructions.

In [Har88] nothing is said about any I/O-function. If one looks at the microcode instructions, no instructions are provided for basic I/O-operations. Since these instructions might be needed they have to be implemented.

This can be done by providing the Objekt with an outputline *IO/MEM. This line indicates whether the source or the target of a read- or write-operation is an I/O-register or the main memory. When the I/O-registers are accessed, the low-order bits of the address-bus indicate which I/O-register is accessed.
The instruction set has to be extended with two instructions, ioput and ioget. Ioput puts the value of the D-bus in the I/O-register indicated by the address. Ioget reads the desired register and puts its value on the D-bus.

## 5.1. Garbage Collection

The Rekursiv's main memory is divided into two semispaces. Only one of the semispaces is used at a time. In the active half, space is allocated to objects sequentially. When the active half overflows, garbage collection is invoked. The garbage collection process, as it is implemented in the Rekursiv, traces all objects that are reachable from the processor. These objects are copied to the inactive half, leaving behind all the dead ones. Finally it interchanges the two semispaces.

This garbage collection has the advantage that it is very simple. However the algorithm implies that the memory usage of the Rekursiv is very inefficient. Only 50 % of the total DRAM memory is used at a time.
Furthermore, when garbage collection is invoked, the process that was running is suspended. It is restarted after the garbage collection process has finished. These pauses are not always acceptable. When, for example, the microprocessor is a part of the controller for a complex chemical process, and garbage collection forces the processor to suspend its controlling function, strange things might happen.

Because the garbage collection process, as it is implemented in the Rekursiv, causes unwanted pauses in the execution of the application program, the decision was made to develop a garbage collector that performs its function in parallel with the main processes. It takes care of the removal of dead objects, and of the compaction of the main memory.

## 5.2. The Disk Processor

The disk processor (DP) is the interface between the Rekursiv and the background memory. Together with the DRAM memory and the disk it forms the one level memory.
When an instruction is encountered to page an object that is not yet in main memory, a datapage-fault (DPFLT) occurs. The disk processor now has to make sure that all the proper actions are performed to load the required object into main memory. If the pager tables entry of the wanted object is already occupied by another object, the disk processor must first store the associated object on background memory, if it is modified or new. This process is performed without the user explicitly having to instruct it.
The disk processor takes care of all stores on and fetches from disk. Furthermore, it has to perform the housekeeping of the disk. Dead objects must not be allowed to fill up the disk. These objects must be removed, and thus compaction of the disk is one of the tasks too.

The disk processor has an interface to the Objekt and an interface to the garbage collector for reasons that will be explained in the next chapter. This means that there is a lot of communication between the disk processor and the Rekursiv. This communication is made more difficult by the fact that a harddisk is an asynchronous resource.

## 5.3. Dynamic Binding

One of the time consuming mechanisms of an object oriented environment is the dynamic binding mechanism. When a non-primitive message is sent, the dynamic binding algorithm is used to find the corresponding method that handles the message.
Usually, the dynamic binding mechanism starts by looking in the so-called method dictionary of the class of the object to which the message was sent. When the method is not found here, the search continues in the superclass of the last-searched class until the method is found. In large systems, it may take a long time before the method is found. Because a message-send is one of the most frequent operations in an object oriented programming system, a lot of time is spent looking for methods. When an algorithm can be found that implements the dynamic binding mechanism as fast as possible, it will save a lot of time. Unfortunately, the dynamic binding mechanism is beyond the scope of this thesis.

## 5.4. The Strategy

This paragraph gives the strategy that is used to come to the result of this master thesis.
Now the decision is made to design an object oriented microprocessor with a microcoded instruction set, it can be divided in distinct parts. The microprocessor is divided into four parts:

- the memory management unit Objekt,
- the garbage collector,
- the sequencer and stack-addressing unit Logik,
- the data manipulator Numerik.

The memory management unit and the garbage collector heavily depend on eachother since they both employ the same memory. The structure of the memory is defined by these processes. The Logik and the Numerik can be developed independently.

The choice is made to design the memory management unit and the garbage collector first, because it is vital for the speed and efficiency of the total processor.

In accordance with the book in which the Rekursiv is described, [Har88], first the instruction set is defined, then the hardware that implements the instruction set is developed. The function of the hardware is defined in a model. Finally the model can be implemented using IDaSS.

# 6. Garbage Collection

Whenever the processor needs an object that is not already paged in, it instructs the disk processor to fetch the object from disk, and to store it in main memory. This works well, but when a large application is in progress, the memory will fill up, until it can not contain more objects. At this point the memory has to be re-arranged. All the space that is occupied by objects that are not needed any longer, can be freed. This process is called garbage collection.
Garbage collection usually takes place on-line. When memory overflows, the program that is currently being executed is suspended, and garbage collection is invoked. When this process terminates, the suspended program is started again.

As is already said in chapter 5, the object oriented processor is given a parallel garbage collector. This process frees the space occupied by 'dead' objects in parallel with the execution of the application program. The purpose is to perform garbage collection on all objects in the DRAM memory. The application should at all times have enough space to perform its task. It should be interrupted as little as possible. The garbage collection process should put as little overhead on the processor's execution as possible.

## 6.1. Garbage Collection Algorithms

The garbage collector must decide which objects can be removed and which objects must stay in main memory. Objects that are dead can be removed in any case. An object is said to be dead when no other object in the system contains an object pointer that points to it. It is dead because it can not be reached through any of the objects. Objects that can be reached from the processor's sources without going through an object on background memory, are said to be reachable. A strategy that can be used is to dispose of all objects that are not reachable. These objects are not necessarily dead, because they might be reached via an object on background memory.

Various algorithms for garbage collection have been developed through the years. Some of them will be explained in this paragraph. After this, an algorithm will be chosen that can be implemented as a parallel process.

### 6.1.1. Reference Counting

The reference counting algorithm ([GoR83], [Ung84], [Ung86]) keeps track of the number of objects that refer to a certain object, by associating a reference count with this object. When the reference count drops to zero, the object is no longer needed, and the memory it occupied can be freed.
Whenever an object pointer is stored in an object, the reference count of the former object to which was pointed must be decremented, and the reference count of the object to which the new pointer points must be incremented. When an object's reference count becomes zero, all the reference counts of the objects to which it refers must be decremented. This can initiate a chain of objects that must be deleted. It is apparent that an object is deleted as soon as it becomes obsolete.

The reference counting algorithm usually does not free all space occupied by non-reachable objects. Objects that refer to themselves or a group of objects that forms a cyclic structure are not deleted by this algorithm.The different kind of object chains are shown in fig. 6.1.

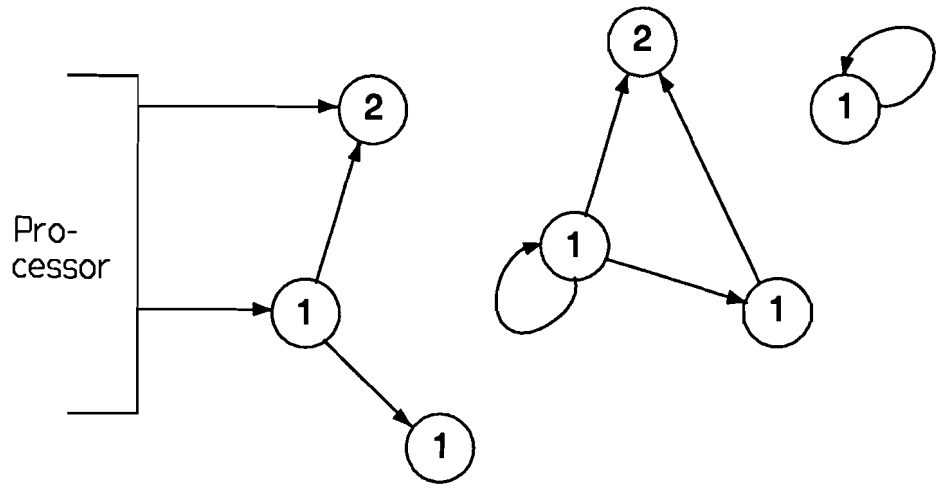



*Fig. 6.1: Object structures*

Objects are represented by circles. The numbers in the circles is the reference count. The three objects on the left are still reachable. Their reference count is greater than zero, and hence they are not removed. The next three objects form a cyclic structure. The respective reference counts of the objects is not equal to zero, but still they can not be reached. The object on the right has a reference to itself and is not removed.
When circular structures are not removed, it is just a question of time before memory overflows. This is the reason why this algorithm needs another garbage collection mechanism too at fixed times.

Another disadvantage of the reference counting algorithm is the overhead the algorithm puts on each store instruction. Furthermore pauses in the execution of the user program will occur when one or more objects are de-allocated. Each page table entry needs a field to count the references.

### 6.1.2. Mark and Sweep

The mark and sweep algorithm ([Ung84], [Ung86]) consists of two phases. The first phase, the mark phase, searches all objects that are alive (reachable), and marks these. The second phase, called the sweep phase, scans the memory for dead (not reachable) objects (which are not marked), and frees the space they occupy.
The algorithm needs only one bit extra per page table entry. The largest drawback is the time that is needed to perform garbage collection. This causes disturbing pauses in the execution of the user program.

The algorithm is split in two phases. When a parallel garbage collector employs the mark and sweep algorithm, the complete object structure can be changed in the time needed for the mark-phase. When the sweep-phase starts, many objects are kept that are actually dead.

### 6.1.3. Baker's Semispace Algorithm

Baker's semispace algorithm ([LiH83]) divides memory into two spaces, tospace and fromspace. New objects are created in tospace. When garbage collecting, live objects are copied from fromspace to tospace (called scavenging). When this process is finished the memory in fromspace can be reused. The algorithm finishes by interchanging the two semispaces, called flipping. Baker does not use an object table. Instead he uses forwarding pointers in the location where the copied object's object pointer was. An object that accesses this copied object follows the forwarding pointer to the new location, and simultaneously updates its pointers to the copied object.
A disadvantage is the inefficient memory usage, because only one semispace is used at a time.

The Rekursiv as described by Harland uses a slightly different version of Baker's algorithm. The Rekursiv does use an object table. This makes the idea of forwarding pointers redundant. Once an object is copied to tospace, the physical address of the object in the object table is changed. This makes the algorithm faster, but the inefficient use of the DRAM memory stays.

### 6.1.4. Generation Garbage Collection

An improvement to Baker's algorithm is described in [LiH83] (and [WiM89]). They use the fact that most young objects die young, and that old objects usually continue to live. They realised that a lot of time is wasted by repeatedly copying old objects.
With generation garbage collection, memory is divided into regions. These regions are organized into generations. The current generation number is periodically incremented. In this way newer generations (which contain a high percentage of garbage) can be garbage collected more often, than older generations (which contain a low percentage of garbage).
The process of garbage collecting a particular region, is initiated by condemning the region. When a region is condemned, new regions are created to hold the objects evacuated out of a condemned region. Each of the evacuation regions will inherit the same generation number as the condemned

region, but is assigned a version number one higher. The version number of a region counts how many times that generation has been condemned.
There is a chance that fragmentation of the memory occurs, as is the case in paged memory systems. Fragmentation can be diminished by making the regions larger.
To make the scavenging-process faster, pointers that point forward in time are restricted. Pointers that point backward in time are directed through an entry table. This means that when a region is condemned, only the regions of that generation and of younger generations (plus the entry table) have to be scavenged.

The disadvantage of the generation garbage collection algorithm is its complexity. The DRAM memory is divided into regions. Each region has its own generation and version number. This memory structure implies extra paging-tables. Furthermore the algorithm is difficult to implement.

### 6.1.5. Generation Scavenging

Generation scavenging ([Ung84], [Ung86], [Jac90]) also uses the fact that young objects tend to die young, and older objects are likely to remain.

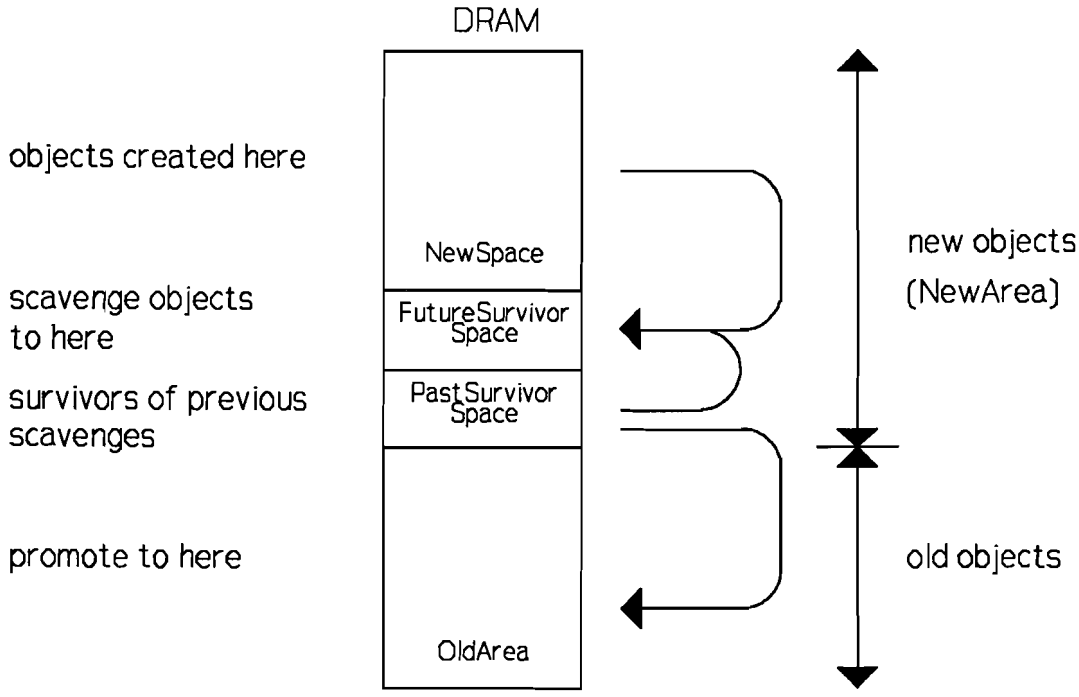



*Fig. 6.2: The memory structure as used with generation scavenging*

With generation scavenging memory is divided into two areas, OldArea and NewArea (see fig. 6.2). OldArea contains objects that have survived a certain number of scavenges, and hence are likely to stay. NewArea consists of three spaces, NewSpace, FutureSurvivorSpace, and PastSurvivorSpace. New objects are created in NewSpace. When this space fills up, all the live objects from NewSpace and PastSurvivorSpace are copied to FutureSurvivorSpace. When this has finished, Future- and PastSurvivorSpace are interchanged.

If an object survives a certain number of scavenges, it is promoted to the OldArea. This is called tenuring. The tenuring threshold can be determined dynamically. Once in a while the OldArea should be subjected to garbage collection too. This can be done by an off-line mark and sweep algorithm.
The search for live objects starts at the remembered set. This set consists of all "old "objects that contain pointers to 'new' objects. Objects are added to this set as a side effect of store-instructions.

The main idea behind the division in an OldArea and a NewArea, is that objects in OldArea are not copied over and over anymore. The only overhead that is put on the processor-instructions is related to the remembered set.

### 6.1.6. The choice of an algorithm

Which strategy is to be used depends on a couple of factors. First, the processor must be interrupted as little as possible. Second, the garbage collector should work correctly. Third the garbage collector should not need too much extra hardware (the processor is large enough as it is).

A major disadvantage of reference counting is that it does not free circular structures that are not reachable. The individual objects in such a structure all have a reference count that is not equal to zero, but still they can not be reached. Furthermore the reference counting algorithm puts additional overhead on the store-instructions.

Because the mark and sweep algorithm works in two phases it is unreliable. When the mark-phase is in progress, the application program can alter the object structure in memory. When the sweep phase starts, the object structure might not match the way it is marked. This does not lead to errors, because when an object dies after it is marked, it will be deleted in the following garbage collection cycle. The amount of marked dead objects should be as little as possible, since the purpose of garbage collection is to remove space occupied by dead objects for objects that are alive.
The reference counting algorithm and the mark and sweep algorithm will not be used for apparent reasons.

It must not take too long to perform an entire garbage collection cycle. The processor should be given as little time as possible to alter the object structure. To make the time needed for garbage collection shorter, the older objects that contain a low percentage of garbage, should not be subject to garbage collection as often as the newer objects. This eliminates Baker's algorithm as a possibility, which does not make a distinction between new and old objects. Only two algorithms are left, generation garbage collection, and generation scavenging.

A disadvantage of generation garbage collection is the use of regions. This makes the algorithm far more complex then generation scavenging, and it needs a lot more hardware (generation number, version number, and an entry table per object containing region, plus the indirection table).
The extra hardware needed by generation scavenging is a field in the pager tables that reflects the number of scavenges (NOS) an object has survived. This field does not need any additional memory. It can be put in the VNB slot of a non-compact object. The lower 16 bits of this slot are not used (they are given by the index in the pager tables), and hence can be used to store the NOS-field.

Because compared to generation garbage collection, generation scavenging needs less extra hardware, and is easier to implement, generation scavenging will be the algorithm used for garbage collection.

## 6.2. Parallel Generation Scavenging

This chapter gives a couple of observations on generation scavenging. A few adjustments are made in order to be able to implement the algorithm in parallel.

### 6.2.1. Memory Usage

As is said earlier, generation scavenging uses a memory configuration in which memory is divided into two parts, called NewArea and OldArea. NewArea is subdivided into three spaces, NewSpace, FutureSurvivorSpace, and PastSurvivorSpace. New objects are created in NewSpace. When NewSpace fills up, the scavenging process is activated. All live objects from NewSpace and PastSurvivorSpace are copied to FutureSurvivorSpace, leaving behind the dead objects. When the copying is done, Future- and PastSurvivorSpace are interchanged ('flipped'). NewSpace now can be reused. When an object has survived a certain number of scavenges, it is promoted to OldArea. This process is called tenuring. The tenured object is no longer subject to on-line garbage collection.

The OldArea can probably not be loaded into the DRAM entirely. The main part of the OldArea will be on background memory. When an object is requested that must be fetched from background memory by the DP, it is loaded into the OldArea directly. Space in the NewSpace is only allocated to newly created objects. In summary the NewArea is entirely in the DRAM, a part of the OldArea is spread over a part of the DRAM and the background memory.

There are two possible solutions to realise the copying of objects from one space to another space. When the objects are copied physically, memory is divided into four distinct banks of memory. The process actually copies an object from one part of the memory to another part.
The second solution does not really copy the objects. To each page tables entry a couple of bits are added, at least two, that reflect the virtual memory space in which an object resides. The virtual copying process consists of changing the space-bits of that object. The two methods are shown in fig. 6.3.

The benefit of virtual copying is the time is saves. Copying objects physically takes time. All virtual copying needs is two bits extra per pager tables entry.
However, the author thinks the whole idea of generation scavenging is abandoned, when virtual copying is applied. Dividing memory into different parts has become useless. The OldArea was added for older objets that ware likely to stay. These objects are not copied anymore. When virtual copying is applied, copying does nearly take any time, so why use an OldArea.
An advantage of generation scavenging is that it has a built-in compaction mechanism. When a scavenge starts the objects are copied to the FutureSurvivorSpace, starting at word zero. FutureSurvivorSpace is filled sequentially. When a scavenge ends (at least in the on-line case) NewSpace is empty and can be reused. When virtual copying is applied, the memory is not divided into distinct banks. This means that when an object is dead, it leaves an empty space and the

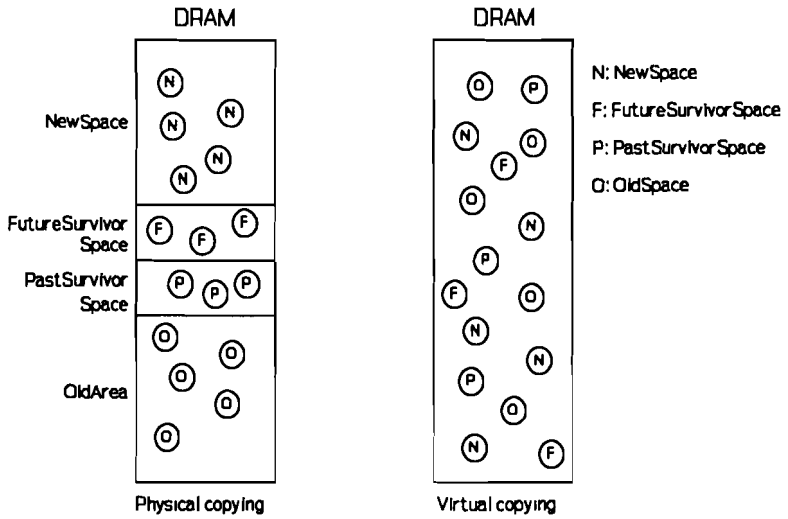



*Fig. 6.3: Two possible copying strategies*

memory gets fragmented. When the fragmentation becomes too heavy, a compaction mechanism must be applied. Furthermore the allocation of memory is more complex in a fragmented memory, since the space must be large enough to accommodate an object of specific size. The allocation of space is done by the main processor, so the execution speed decreases.
Virtual copying may decrease the duration of a garbage collection cycle, but this is only relative. The execution speed of the processor is decreased, and every now and then a compaction mechanism must be invoked. This is the reason why the physical copying is used.

## 6.2.2. Scanning the Objects

The purpose of garbage collection is to remove all dead objects from memory, so that memory space is obtained for live objects. There are different ways to find all live objects.

Generation scavenging as described in [Ung84] and [Ung86] starts searching for live objects at the remembered set. The remembered set consists of all the objects in OldArea that contain pointers to objects in NewArea. Objects are added and removed from the remembered set as a side-effect of store-instructions. This puts an extra overhead on the store-instructions, which of course, is not wanted.
This overhead can be eliminated, if one uses another point of view. Instead of starting scavenging at the remembered set, scavenging is started from the processor sources that can contain object pointers. When the search continues through all objects in the system, all live objects are reached.

The sources in the Rekursiv that can contain object pointers are the evaluation stack (ESTK), the eight VR-registers, and the REF-register. These sources have to be presented to the garbage collector.
In the Rekursiv the ESTK is implemented as a segmented stack, of which only one segment at a time is in the RAM-area. The other segments are stored on disk until they are needed. When a stack position must be accessed that currently is not in the ESTK-RAM, a page fault is issued and the DP loads the required stack-segment in the SRAM. Because each class in the object oriented

system has its own logical stack, the ESTK will be heavily fragmented into many distinct smaller stacks. Because these logical stacks are not necessarily ordered sequentially, the entire SRAM has to be searched for object pointers.
The problem that arises is that when we want to make a sequential scan of the ESTK in its configuration used in the Rekursiv, the DP will be called upon frequently, halting the processor accordingly. There are two possible solutions to this problem. The first solution is to give the garbage collector its own stack-segment. This however must be done very carefully to prevent that the processor alters a stack segment that is also loaded in the garbage collector. The second solution is to present the processor with the entire ESTK in SRAM (the Rekursiv uses a 256 kB ESTK). This also speeds up the processor because it does not have to store and load stack segments anymore on-line. Despite of the large SRAM the second solution is chosen.

### 6.2.3. Protection of Shared Resources

There are a couple of resources that are used by the main processor and the garbage collector. When these resources must be accessed by one of the processes, this must be done very carefully. When both processes want to write into a register simultaneously, the result can not be predicted. The processes that can be accessed by both processes are:

- every object in main memory,
- the register that points to the next location to be written in NewSpace (Objptr),
- the register that points to the next location to be written in OldArea (Old_Address),
- the ESTK,
- the VR-registers,
- the REF-register.

The last three resources only have to be read by the garbage collector. This can be done without errors when an extra read-port per resource is added. Then the main processor and the garbage collector can access the resources simultaneously.
The DRAM is provided with an extra input/output-port for the same reason.

When the main processor wants to modify a location of an object, it must first be sure that the garbage collector is not moving that object to another area in main memory. In the same way the garbage collector must be sure that the main processor is not writing into an object, before it starts moving that object. This can be done easily when one bit is added to each page table entry. Before a process wants to perform a critical action on an object, it first checks the lock-bit of that object. If the lock-bit is not set, the process sets the bit and performs its operation. It ends the operation by resetting the lock-bit. When the lock is set, the process has to wait until the lock-bit is reset by the other process, before it can perform the operation.

The two registers Objptr and Old_Address are provided with a lock-bit too. When a critical action has to be performed on one of these registers, it is done in the same way as described above.

## 6.2.4. The First Solution

The search for reachable objects can be done by traversing the objects in the order in which they are encountered. The search starts with an object pointer from a processor source. The object associated with the pointer is scanned for object pointers. When a pointer is encountered its object is accessed. This sequence is continued until all objects that can be reached from one processor source are found. Now the next processor source is read and the search is initiated again.
When an object pointer is encountered, the part of memory in which the object resides, determines where the object is copied to (if it needs to be copied).

The implementation of this search needs a stack-mechanism. When an object is reached, the algorithm decides whether the object is in the NewSpace, in the PastSurvivorSpace or in the OldArea. In the former two cases, the object has to be copied to FutureSurvivorSpace before scanning the object. In the latter case the object can be scanned immediately after setting the tag-bit. The tag-bit tells whether an object is already scanned or not. In this way the algorithm makes sure that each object is scanned only once.
The object is scanned by testing each object location for being a non-compact object pointer, whose object has not been copied and scanned before (the tag-bit is not set). When such a pointer is found, and the corresponding object must be moved to another part of the memory, it is copied. Then the object number of the object in which the pointer was found and the index of the location at which it was found, are pushed onto the stack, and the found object pointer is loaded as the new object number. Scanning continues with this object after it is copied (if necessary).
When the end of an object is reached, the object number and the index are popped from the stack and scanning continues from this location. When the stack is empty, the next processor-source is read.

This generation scavenging implementation stops the scanning of a chain of objects when an object pointer whose object is on background is encountered. In this way the objects are found that are reachable without having to activate the DP. However not all live objects are found.
This implementation works well for the NewArea. However dead objects from OldArea are not removed. The motivation for this was that the relative number of dead objects in the OldArea is smaller than the relative number of dead objects in the NewArea. The OldArea should be subjected to off-line garbage collection. This does not have to be done very often. This could for example be done at night, when no application program is running on the processor.

The reason why the choice is not fallen on this implementation of the generation scavenging algorithm is that the entire DRAM should be cleaned in parallel with the execution of the main processor. When for example the microprocessor is controlling a industrial process, there may not be time to perform off-line garbage collection of the OldArea. When the main memory is cleaned by the build-in garbage collector, the background memory can be kept clean by the disk processor, if necessary in co-operation with the garbage collector.

## 6.2.5. The Second Solution

This implementation of the garbage collection algorithm takes advantage of the fact that not all objects can contain object pointers. In order to find all live objects, only the objects that can contain object pointers have to be scanned. For example instances of the class String can not

contain object pointers, they only contain numerical values. This selective search saves a considerable amount of time. To implement the selective search, one bit of the object pointer has to be dedicated to indicate whether the object can contain object pointers or not.

The generation scavenging algorithm consists of two main parts. The first part searches for live objects from the processor sources that can contain object pointers.
If a live object is found that can not contain object pointers, and if that object resides in NewSpace or PastSurvivorSpace, that object is copied to either FutureSurvivorSpace or OldArea (depending on the number of scavenges (NOS) it has survived).
When a live object is found that can contain object pointers, its object pointer is stored in a content addressable memory (CAM). A CAM is a combination of RAM and logic that is able to address a location on the basis of its content very fast (ideally one clock cycle).

The second part makes a sequential scan of the CAM, and performs the following operations on the associated objects. Note that all objects whose objects pointers are stored in the CAM, can contain object pointers and hence must be scanned.

The garbage collector reads in a word from the CAM. When the object is not in main memory, it has to be fetched from disk. Because it is not the purpose to increase the amount of objects in DRAM, the object may not be loaded. Every incoming word from the disk processor is checked to see if it is a non-compact object pointer of an object that can contain object pointers. If so it is stored in a fifo for later use. When all incoming words are checked, the garbage collector checks whether the words stored in the fifo, are in the CAM yet. If this is not the case the object pointer is added to the CAM. The intermediate fifo has to be used because the rate of incoming words is one per clock cycle.
When the object is in main memory, it is copied if necessary. Now the object is searched for other object pointers. Again object pointers of live objects that can contain other object pointers, and that are not already in the CAM are added to the CAM. Live objects that can not contain object pointers are copied immediately.
When all live objects are found and, if necessary, copied, the algorithm first cleans NewSpace, then PastSurvivorSpace, and finally it reorganizes OldArea.

At the end of a garbage collection cycle, all live objects are found. All dead objects are removed from main memory. The pager table-entries of all dead objects are re-initialised and the OldArea contains a contiguous chain of live objects.
What still has to be done is the garbage collection of the external (disk) memory. This part has to be done by the disk processor, which is beyond the scope of this thesis. A solution could be: The disk processor can administrate which objects are alive. This would only need a small extension to the generation scavenging algorithm. When a garbage collection cycle has finished, the disk processor knows which objects are alive and which are dead. It then can reorganize the disk.

## 6.3. The Implementation of Generation Scavenging

The way the implementation of the garbage collector is obtained is as follows. First the algorithm is defined using flowcharts. This algorithm must be converted to hardware. The author has chosen to do this the same way as it is done in [Har88]. In this way a uniform description is obtained for both processes described in this thesis. An instruction set is made that can implement the

flowcharts. Each field of the control word controls a special piece of hardware. The hardware is developed to run the generation scavenging algorithm efficiently.
Finally, the model of the garbage collector is developed. The model is described in chapter 7, together with the model of the object oriented memory manager.

## 6.3.1. The Flowcharts

The algorithm can be divided into three parts. First the processor sources are read. If a non-compact object pointer is encountered whose object can not contain object pointers, the object is copied if necessary. If a non-compact object pointer is found whose object can contain other pointers, then the object pointer is stored in the CAM. The second part reads the words stored in the CAM sequentially and searches all the live objects, which are copied if necessary. The last part cleans the pager tables and reorganizes the DRAM.
Appendix A contains the flowcharts of the generation scavenging algorithm.

### Reading in the processor sources

Two address registers can be used to feed the address input of the CAM. Read_Address is used when reading out the CAM. Write_Address is used when storing object pointers in the CAM. At the start of a generation scavenging cycle they both are set to the start of the CAM.
In this part a counter called Loop_Counter is used to give the location of the word in a processor source that must be read next. It can be loaded with three values, the last location of the ESTK or the pager table that contains the object pointers of an object's class (its type), or to the last register in the VR-bank.

When a word from a processor source is read, the algorithm tests the word for being a non-compact object pointer. If not, the next word can be read. If so the object is copied immediately (if necessary) if the corresponding object can contain object pointers, or the object pointer is stored in the CAM if the object can contain pointers. The CAM is filled sequentially. Consecutively these operations are performed on the words found in the ESTK, the VR-bank, the REF-registers, and the VTB-bank (type-pointers).

The copying sequence finds out where the object currently resides. If object resides in the OldArea its tag-bit in the pager tables is set, marking that the object is alive (it has already passed a test).

### Finding the live objects

In this part the words in the CAM are read sequentially until Read_Address equals Write_Address. The objects whose object pointers are read in this part all can contain object pointers, so they all have to be scanned. If necessary the object is moved to another area in main memory. Next the object must be scanned. There are two possibilities, either the object resides in main memory, or it is on background memory.

When the object is in main memory, the object is searched through for object pointers. The procedure is the same as in the first part.

When the object is not in main memory, it has to be read in from disk. Hereto the disk processor is activated. When designing the algorithm, the assumption was made that the disk processor would be fast enough to, once the location of the object on disk was found, present the object to the garbage collector at a rate of one word per clock-cycle. In reality this is not true. A hard disk is an asynchronous device that usually is rather slow. This means that per word a handshake mechanism is activated. In this implementation the word-rate is one word per clock-cycle. The insertion of the handshake should be rather simple.
When the handshake mechanism is inserted, the first-in-first-out memory (fifo) that is used in this implementation is not needed anymore. Because the rate of incoming words is one per clock-cycle, there is no time to test the word entirely. When the word is a non-compact object pointer whose object can contain object pointers, it is stored in the fifo. When all words are read, the words that are in the fifo are investigated further. When they are not already in the CAM, they are added to it.

### Cleaning the memory

When all live objects are found and moved to the right areas, the dead objects have to be removed, and the areas have to be reorganized. The actual removal of objects is done by re-initiating their pager tables slot.

The first area of memory that is cleaned is the NewSpace. During a garbage collection cycle, the main processor still can allocate space to newly created objects. These objects must be moved so that the processor can start allocating at the start of the NewSpace again. To know which objects have been allocated after the start of the scavenge, the Objptr-register, which keeps track of the next location to write to, is saved in a register called Save_Objptr.
The VAB-bank of the pager tables, that holds the start addresses of the objects in DRAM, are implemented as a CAM too. This allows the garbage collector to find objects that are stored in the NewSpace very quickly. Hereto the match- and the mask-register of the CAM are loaded with values so that the whole NewSpace is covered. When an object is found whose start address is greater than Save_Objptr, the object is moved to FutureSurvivorSpace. If the start address is smaller than Save_Objptr, the object is dead, and its pager tables slot is re-initialized. When there are no more objects in NewSpace the Objptr-register is reset to the start of the NewSpace.

The next area to be cleaned is PastSurvivorSpace. The objects that are found in this region are all dead, so the only thing that has to be done is re-initializing their pager tables slots.

The last area, OldArea, will be fragmented. Only those objects are alive whose pointers are stored in the CAM or whose tag-bits are set. All live objects have to be moved forward to form a contiguous chain of objects. To implement this, two extra registers are needed. Copy_To keeps track of the location where the next word of a live object must be written. The Search_For register holds the address where must be looked for. This will be the start address of a live or dead object. The address in Search_For is clocked into the match-register of the VAB-CAM. If the found start address belongs to a live object, the size of the object is added to both the Copy_To- and the Search_For-register. When a dead object is found, the size of that dead object is only added to the Search_For-register. The next live object that is found is automatically written over the dead object.
All tag-bits that were set are reset, and of course all pager tables slots of dead objects are re-initialized. This procedure stops when Search_For equals Old_Address. Old_Address contains the address of the next word of an object that is allocated in the OldArea by the processor. The procedure ends with loading Copy_To into Old_Address.

**Finally**

A garbage collection cycle stops with interchanging Future- and PastSurvivorSpace. This is simply done by swapping the start- and end-addresses of Future- and PastSurvivorSpace.

## 6.3.2. The Instruction Set

The instruction set has been developed to control the hardware of the garbage collector. This microcode, together with the hardware implements the generation scavenging algorithm. The instruction set has been derived by examining how the hardware can be used to implement the algorithm efficiently.
The different fields of the garbage collector's control word are shown in appendix B.

One microcode word consists of 22 fields that all control a part of the hardware. The different fields are explained below:

**Sequencer** : The sequencer controls the microcode. It indicates at which address the next instruction to execute can be found. Hereto the microcode-store is presented with the current microcode-address (cua). The cua is stored in the csa-register for further use. The microprogram-counter (upc) is loaded with the cua incremented with one. When a jump-instruction is encountered, the jump-address is found in the Jump_Address-field. The instructions are:

| | |
|---|---|
| *noseq* | : cua = cua |
| *gccont* | : cua = upc ; upc = upc + 1 |
| *jmp* | : cua = Jump_Address ; upc = cua + 1 |
| *jon* | : jmp if object pointer and not compact |
| *jonc* | : jmp if object pointer and not compact and can contain object pointers |
| *jcco* | : jmp if can contain object pointers |
| *jic* | : jmp if object is in core |
| *jin* | : jmp if object is in NewArea |
| *jts* | : jmp if tag-bit is set |
| *jmf* | : jmp if a match is found |
| *jmot* | : jmp if a match is found or tag-bit set |
| *jamf* | : jmp if address-match is found |
| *jrw* | : jmp if Read_Address = Write_Address |
| *jlcz* | : jmp if Loop_Counter = 0 |
| *jieo* | : jmp if Index = Obj_Size |
| *jnos* | : jmp if NOS ≥ Tenure_Threshold |
| *jfe* | : jmp if fifo empty |
| *jol* | : jmp if Objptr locked |
| *jags* | : jmp if found address > Save_Objptr |
| *jces* | : jmp if Copy_To = Search_For |
| *jseo* | : jmp if Search_For = Old_Address |
| *joal* | : jmp if Old_Address is locked |
| *jif* | : jmp if object is in FutureSurvivorSpace (Start_Of_Future ≤ VAB ≤ End_Of_Future) |

*jamf* : jmp if address-match found

**SourceInstrx** : This field tells the main processor which source should be read. To do this in parallel with the normal processor execution, the ESTK and the VR-bank should have an extra addressing mechanism. Which location in the ESTK or which VR-register is required, is indicated by the Loop_Counter-register.
*nosourceinstrx*
*readvr* : read the VR-register indicated by Loop_Counter
*readestk* : read the ESTK-location indicated by Loop_Counter
*readref* : read the REF-register

**ObjptrInstrx** : This field controls the main processor's Objptr-register. This register holds the address of the next word to be written in the NewSpace.
*noobjptrinstrx*
*readObjptr* : load Save_Objptr with Objptr
*resetObjptr* : lock Objptr; reset Objptr; unlock Objptr

**DPInstrx** : This field instructs the disk processor, if necessary, to fetch an object from disk.
*nodpinstrx*
*DPfetch* : instruct the DP to fetch an object from disk and present it to the garbage collector

**OldAddrx** : This field gives the instructions from the garbage collector to the main processor's Old_Address-register. The Old_Address-register holds the address in the OldArea of the next word to be written there.
*nooldaddrx*
*lockoldaddr* : lock Old_Address for GC-use
*unlockoldaddr* : unlock Old_Address
*loadoldaddr* : load Old_Address with the content of Copy_To
*incoldaddr* : increment Old_Address

*The former four instruction-fields of the garbage collector's control word all interact with the main processor. The Rekursiv must be provided with the hardware that performs these operations.*

**LoopCounterx** : This fields controls the Loop_Counter-register. The Loop_Counter is used when reading in the processor sources to indicate which location should be read of the ESTK, the VTB, or the VR-bank.
*nolc*
*declc* : decrement Loop_Counter
*loadlcESTK* : load Loop_Counter with (ESTK-size - 1)
*loadlcVR* : load Loop_Counter with (#VR-registers - 1)
*loadlcVTB* : load Loop_Counter with (PT-size - 1)

**Indexx** : This field controls the Index-register that is used when copying objects to indicate the position in an object.
*noindexx*
*resetindex* : equal the value of the Index-register to zero
*incindex* : increment Index

**ReadAddrx** : This field controls the Read_Address-register. This register points to the location in the CAM that must be read next.
*noreadaddrx*
*resetreadaddr* : set Read_Address to point to the start of the CAM
*increadaddr* : increment Read_Address

**WriteAddrx** : This filed controls the Write_Address-register. This register is used to indicate the next location in the CAM where to write to.
*nowriteaddrx*
*resetwriteaddr* : set Write_Address to point to the start of the CAM
*incwriteaddr* : increment Write_Address

**FutAddrx** : This field controls the Future_Address-register. It gives the location in the FutureSurvivorSpace where the next word must be written to.
*nofutaddrx*
*resetfutaddr* : this is the flipping-operation. Load Future_Address with Start_Of_Past. Swap Start_Of_Past and Start_Of_Fut. Swap the End_Of_Past- and the End_Of_Fut-register
*incfutaddr* : increment Future_Address

**SearchForx** : This field controls the Search_For-register. This register is used when cleaning the OldArea. It indicates the location in the OldArea whose address must be looked for in the VAB.
*nosearchforx*
*resetsearchfor* : set Search_For to point to the start of the OldArea
*loadsearchfor* : load Search_For with the content of Copy_To
*addsearchfor* : add the current object's size to Search_For
*incsearchfor* : increment Search_For

**CopyTox** : This field controls the Copy_To-register. It is used when cleaning the OldArea to indicate where the next word of a live object is to be written to.
*nocopytox*
*resetcopyto* : set Copy_To to point to the start of the OldArea
*inccopyto* : increment Copy_To
*addcopyto* : add the current object's size to Copy_To

**NewStartAddrx** : This field controls the New_Start_Address-register. This register is used when copying objects to store the address of the first location where the object is copied to. This address is loaded into the VAB-bank at the end of the copy-sequence.
*nonewstartaddrx*
*loadfuture* : load New_Start_Address with the content of Future_Address
*loadold* : load New_Start_Address with the content of Old_Address
*loadcopyto* : load New_Start_Address with the content of Copy_To

**NOSx** : This field controls the NOS-register. This register holds the number of scavenges an object has survived. It is stored in the 16 least significant bits of the object's entry in the pager tables.

*nonosx*

*incnos* : increment the number of scavenges of the currently paged object and store it in the VNB-field of the PT

**Fifox** : This field controls the first-in-first-out-memory, that is used when scanning an object from disk.

*nofifox*

*push* : add an object number to the tail of the fifo

*pop* : get an object number from the head of the fifo

**DRAMx** : This field controls the interface between the garbage collector and the system's main memory.

*nodramx*

*readdram* : MemOut is loaded with DRAM[Phys_Address] (Phys_Address = Start_Address + Index)

*writetofut* : load DRAM[Future_Address] with MemOut

*writetoold* : load DRAM[Old_Address] with MemOut

*writetocopyto* : load DRAM[Copy_To] with MemOut

**CAMx** : This field controls the content addressable memory. The CAM is used to store all non-compact object pointers of objects that can contain object pointers. It is used to prevent objects to be scanned more than once.

*nocamx*

*resetcam* : load every CAM-location with the null-value

*readcam* : page the object pointer at the location pointed to by Read_Address

*writecam* : store the output of the CAMMUX at the CAM-location pointed to by Write_Address

*loadmatch* : load the Match_Reg with the output of the CAMMUX

*match* : issue the match-instruction

**VABCAMx** : This field controls the garbage collector's interface with the VAB-bank of the pager tables. This bank is implemented as a CAM or CAM-like structure. This is done to decide very fast whether a certain location of the DRAM is the start-address of an object.

*novabcamx*

*findnewspace* : load the Addr_Match_Reg and the Addr_Mask_Reg so that every VAB-address in the NewSpace is found

*findpast* : load the Addr_Match_Reg and the Addr_Mask_Reg so that every VAB-address in the PastSurvivorSpace is found

*loadaddrmatch* : load the Addr_Match_Reg with the content of Search_For

*addrmatch* : issue the match-instruction to the VABCAM

**CAMMUX** : This field selects which source of the CAMMUX-multiplexer is fed to the CAM. This can be either to be loaded into the CAM's match-register,

or to be loaded into the CAM itself. These are the input of the CAM-multiplexer:
*Obj_Number*
*Obj_Type*
*Fifo*
*Proc_Sources* (ESTK, VR, or REF)
*MemOut*

**PTAddrIn_MUX** : This field selects which source of this multiplexer is used as the index into the pager tables. The inputs of the multiplexer are:
*Obj_Number*
*Obj_Ptr_Out*
*Addr_Out*

**GCPagerx** : This field controls the interface of the garbage collector with the pager tables. To increase overall speed, the garbage collector has its own interface.
*nopagerx*
*settag* : set the Tag-bit
*resettag* : reset the Tag-bit
*setlock* : set the Lock-bit
*resetlock* : reset the Lock-bit
*readVTB* : load the Obj_Type-register with VTB[Loop_Counter]
*loadVAB* : load VAB with New_Start_Address
*resetentry* : VAB ← 0
VNB ← 0
VSB ← 0
VTB ← 0
VRB ← 0
reset the Mod-bit
reset the New-bit
reset the Cond-bit
reset the Tag-bit

**Jump_Address** : This field gives the next address in case of a (conditional) jump-instruction. It is an absolute address of the garbage collector's control store.

# 7. The Model of the Microprocessor

This chapter gives the model of the object oriented memory management unit Objekt and the garbage collector according to the author of this thesis. The model will utilise most of the features of the Rekursiv as described in chapter 4. Some characteristics of the Rekursiv however are changed. Other features are added. These changes and additions are mentioned in chapter 5.
First this chapter will give a brief summery of the used development method.

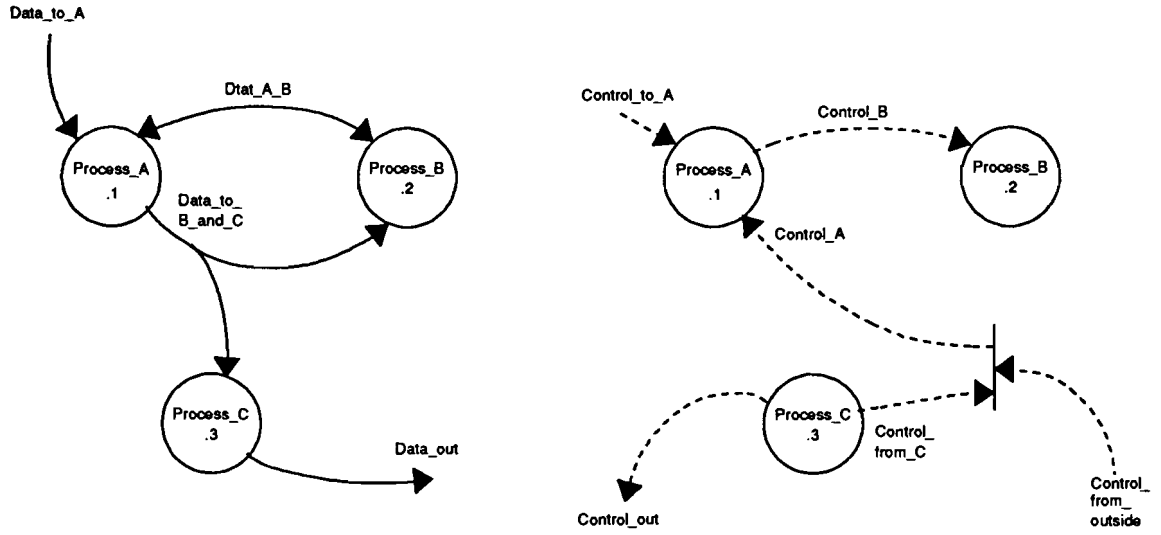
## 7.1. Structured Analysis According to Hatley and Pirbhai

In [HaP87] Hatley and Pirbhai describe a tool for describing real-time system specifications and general systems. This tool is very helpful in describing complex systems, which may be implemented in software or in hardware. The method can be divided in two parts. The first part can be used to describe the requirements model of the system. The second part has been developed to describe the architecture model.
This thesis only uses the requirements model of the method. The hardware model will be developed with another tool, called IDaSS. Both methods complement each other. Hatley and Pirbhai use a top-down approach to come to the requirements model. This model can be implemented using the bottom-up method of IDaSS.

The requirements model is used to define the system that is to be developed. It tells the user and developer what the system is, before describing how to produce it. The model shows the internal subsystems with their data and control flows.
To produce the requirements model a top-down strategy is used. First the context of the system to be developed is specified, and how the system communicates with it. Then the system itself will be modelled. Therefore the system is split up into processes. A diagram is made showing the communication between these processes. All the processes are refined into subprocesses until it can be defined in a simple way.

The context diagram or the model of a process comprises a data flow diagram (DFD) and a control flow diagram (CFD) (see fig. 7.1). They differ in the kind of behaviour they describe. In DFD's

the functional behaviour is described. These diagrams show the data flows between the processes (nodes). CFD's describe the control behaviour. These diagrams show the control flows between the nodes.
The context diagram usually contains objects called terminators. These represent the processes in the environment of the system that is to be developed. The system communicates with the terminators in a predefined way. The implementation of a terminator is not important for the designer and therefore not specified.
CFD's can contain socalled control specifications (CSPEC's). They transform the different control flows into each other and can activate the processes depending on the internal state and the control flows. CSPEC's can be made of state transition diagrams.




Data flow diagram

Control flow diagram

*Fig. 7.1: A data and control flow diagram*

As is shown in the fig. 7.1, the processes are represented by bubbles, and the flows by arrowed lines. A flow can be either unidirectional or bidirectional. CSPEC's are represented by vertical bars. The control flows that are used in the specification either originate from or start at the bar. Not shown in fig. 7.1 are stores. A store is used to store data or control information. It is represented by two parallel horizontal bars with the name in between.
A process is defined by a mini specification (MSPEC). Each process can have its own MSPEC. When a process is primitive, that means it cannot be refined, it must have a MSPEC, that gives the definition of the process. This can be stated in some structured form.
An important part of the model is the data dictionary (DD). This gives the definition of the data flows and the control flows that are used in the diagrams.

To model the microprocessor, a program called Promod is used. This program supports the method described by Hatley and Pirbhai for a very large part. Unfortunately ProMod has a few restrictions.

It is for example not possible to split up a bidirectional flow. This is a drawback when a system uses a large system bus that shares many sources.

The reason why the author chose to use Promod, is that it is a good tool to describe complex systems. The microprocessor uses many communicating processes that are activated by different fields of the control word. ProMod provides a way to model these processes, to define the data and control flows, and to state what the primitive processes should do. When the model is satisfactory, it can easily be translated to an architectural model using IDaSS.
However Promod is not entirely used according to the method described by Hatley and Pirbhai. The requirements model should not contain architectural details according to the Hatley and Pirbhai. However sometimes it was not possible to exclude architectural information. For example when in the implementation of a microprocessor stores are used, then an address must be generated to indicate which location of that store must be accessed. In Promod it is not possible to provide an address to a store, while the address generation process is a distinct process in the model. That's is why Promod is not always used as it is meant to be. However Promod is available, a good tool to define all processes, data and control flows, and contains tests to check for consistency and unambiguity.

## 7.2. The Model of the Processor

The processes in the processor are controlled by the control word. This control word consists of a large number of fields, all controlling some part of the processor. The model has been made by splitting up the control word into smaller portions that all control a part of the processor. When the number of fields that a portion has cannot be reduced further, a primitive process has been found, which has to be defined. In this way the processor has been modelled.
The control word is not the same as in the Rekursiv. This mainly due to the addition of a parallel garbage collector. The fields of the processor's control word as it is known now (the Logik and the Numerik have not been put in the model yet) can be found in appendix C.
Appendix D contains the Promod-report. It contains the diagrams (DFD's and CFD's), the process-hierarchy, the MSPEC's, the CSPEC's, and the DD.

### 7.2.1. The Context Diagram

The processor should be able to communicate with the environment. This is done via a databus and control signals. The parts the processor needs to operate are the disk processor (DP), the DRAM memory, and the I/O-registers. The DP forms the interface between the processor and the background memory. Together with the DRAM it forms a one level store.
Other parts or processors that should communicate with the microprocessor are represented by the terminator called Host. These parts communicate via the system D-bus. A Bus-sharer could be a host-computer in which the object oriented processor is placed, or a coprocessor. The interaction between the processor and the host goes via some control lines . These control lines are at this moment still to be defined, but probably will comprise a bus-request and a bus-grant line, and some interrupt-lines (for instance three maskable and three non-maskable prioritized interrupts).

The DP-control lines consist of a couple of instruction lines and a ready signal from the DP to the processor. The DP has two interfaces, one to the main processor, and one to the garbage collector. The I/O-registers and the DRAM are controlled by the main processor via the IOandDRAM_Control-lines. The garbage collector controls the DRAM via DRAM_Control. The address of the wanted I/O-register or DRAM-word is 24 bits wide and flows out of the processor. From the diagram it is clear that the DRAM has two interfaces, one to the main processor and one to the garbage collector.
As can be seen in the diagram, the DRAM's control- and address-lines end at a bar. This means that there is an associated CSPEC. This is done because they are not allowed to end at the store itself. The store can only contain data- and control-signal, but cannot process them. This means that there is not a clean way to give an address to a store and instruct it to perform a read- or write-operation. The CSPEC is used to tell what these signals are meant to. However it is not a clean way according to Hatley and Pirbhai.

## 7.2.2. The Processor

The main processor is divided in the same way as the Rekursiv. It gives a good overview of the main processes in the processor. The garbage collection process GC is added to the features of the Rekursiv. The four main processes are modelled in the Proc-diagrams.

The processor contains the following stores:

- CS, the control store. This contains the microprogram. The CS is a read-only store.
- CSMAP, the control store map. This store contains the starting addresses of the micro-routines in the CS. It maps opcodes onto the routines that implement them. It uses these opcodes as an index into the map.
- NAM, the new abstract memory. The NAM can contain the opcodes for the CSMAP and their arguments. It acts like a fast instruction cache. In a Smalltalk-environment, the NAM would contain the methods of the most important system classes.
- CSTK, the control stack. This stack is used by the microprogram. The CSTK provides a way to implement a very high-level instruction set.
- ESTK, the evaluation stack. The ESTK can be used by the program that is running on the processor.
- the pager tables. They hold frequently used information and status-information about objects. This information consists of the object's number, type, address in physical memory, size, and representation. The status-information consists of the bits Mod, New, Cond, and Tag. The Tag-bit is only used by the garbage collector. When an object is not compact the lower 16 bits of the object number-slot are used to hold the number of scavenges an object has survived.

Again all these stores have their own part of a CSPEC.
The communication with the stores goes (besides the CS) via a bidirectional data bus, and is controlled by some control flows and an address.

The major processes are:

- The Logik. This contains the microprogram sequencer and the stack addressing units. This is the heart of the processor, it controls the sequence of the microprogram. The reason why there is one large control word is that it gives control over the heavy communication that takes place between the different processes.
  The condition codes of the other processes are gathered here.

- The Numerik. The Numerik contains the data manipulator. This consists of an alu, a barrel shifter and a multiplier.
- The Objekt. This is the object oriented memory management unit. It controls the I/O registers, the DRAM, and the pager tables. It is called object oriented because it manipulates objects rather than single words, unlike conventional hardware.
- The GC. This is the garbage collector. It performs a parallel generation scavenging algorithm. It is designed to interrupt the main processor as little as possible.

So far only the model of the Objekt and the GC have been set up.

## 7.3. The Objekt

The Objekt-process contains the object oriented memory management unit. It makes sure that only valid locations of objects are accessed. The user does not have to worry about any paging strategies at all. This process takes over many operating system tasks. In fact the Rekursiv is designed to run without an operating system.
The Objekt produces control signals for the DRAM and the pager tables. Together with the DP it forms the one level store. it provides condition codes to the sequencer, and interacts with the GC.

The model of the Objekt consists of two main processes.
One process, called Pager, takes care of the pager tables. It determines which object should be paged. Then it actually pages the object. When the required object is not resident, it issues a DPFLT. The sequencer sees the DPFLT, and instructs the DP to fetch the object. When the object is paged in, normal execution continues.
There are three possible sources from which the pager address can be taken, the D-bus, one of the eight VR-registers, and the allocator. The VR-registers can be loaded from the D-bus to hold an object number. The allocator is a 37 bit counter that gives the object pointer for a newly created object. Bit 39, 38, and 37 are indicators for the type of the object it represents (see fig. 7.2).

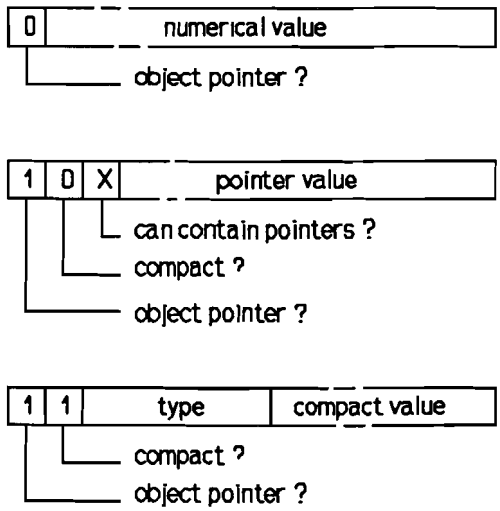



*Fig. 7.2: The different shapes of a 40-bit word*

When an object is already paged, the REF-register holds the complete object number as a pointer into the pager tables. The different pager table-slots are:

- VNB : 40-bits object number,
- VSB : 24-bits object's size,
- VTB : 40-bits object's type,
- VAB : 24-bits object's start address in DRAM,
- VRB : 40-bits object's representation. This is the first word of a non-compact object, or the lower 32 bits of a compact object.

Each slot has a corresponding output register that is loaded when the object is paged. These registers are respectively VNR, VSR, VTR, VAR, and VRR.

The pager tables also contain four status-bits per entry. These are:

- Mod : tells whether an object has been modified since it was last saved on background memory,
- New : tells whether an object is new,
- Cond : this bit can be used by the processor to indicate whether an object conforms to a certain condition,
- Tag : this bit is only used by the GC to indicate whether an object in OldArea has already been scanned.

The process called Address_Gen_and_Range_Check generates the address of the required DRAM-location and performs range checking. If this check works out positively the specified access is made.
A characteristic of an object is that it has a known size. This size is used to tell whether an access into an object is allowed. The wanted element of an object is specified by giving an index into the object. If this index is greater then zero and smaller or equal to the object's size, then the access is allowed.
The address is generated, either directly via the D-bus or the Objptr (this points to the location where a newly created object will be placed), or indirectly by adding the index to the base address of the object. Direct addressing should be used carefully, for example in communicating with the DP.

The fields of the control word that control the Objekt are:

- Addropx
- Memopx
- Idxldx
- Regx
- Pagerx
- Flagx
- Pgrfetchx

The function of the different fields is shown in appendix C.

### 7.3.1. The Address Generation and Range Checking Process

The address generation and range checking bubble consists of four processes:

- Index_Generation: this process performs instructions that manipulate the two index-registers. These registers can be programmed to scan over the elements of objects efficiently. This process takes care of the operations of Idxldx and Regx.

- Range_Checking: this process performs range checking on the index. It compares the index with VSB and sets Access_Ok if the index falls within the bounds of the object.
- Address_Generation. This process generates the address of the required element of an object. It performs the operations of Addropx.
- Object_Space_Allocator. This process gives the base address of objects to which memory space must be allocated (Objptr). The required object size is presented to the process on the D-bus. When the memory space for new objects overflows, Object_Space_Allocator sets NewSpace_Oflo as a trap to the sequencer.

As can be seen in the diagrams, there is a CSPEC associated with this process. It has some status- and control-signals as an input. It decides what DRAM- or I/O-operation is requested and sets the control signals accordingly, if the access is valid.

**Index_Generation**

This process contains two processes, one for each of the registers. The most important register is Idx. This is the register that is really used as an index into an object. The Idx-register is controlled by the process Idxldx_Operations. The other register, Idxreg, can be used, in combination with Idx to perform scanning operations on objects. It is controlled by Regx_Operations.

Idxldx_Operations performs the following operations:

- noidxldx : no operation,
- clridx : Idx ← 0,
- ldidx : load Idx with the value on the D-bus,
- incidx : increment Idx,
- decidx : decrement Idx,
- idxstep : add the value on the D-bus to Idx,
- idxnext : if the access is still allowed then increment Idx, else Idx ← 1. This instruction is added for scanning over the elements,
- newidx : load Idx with the value of Idxreg.

The last Idxldx-operation is performed by the process called Regx_Operations. This instruction (newidxreg) loads the Idxreg with the value of Idx. The Regx-operations are:

- noregx : no operation,
- ldreg : load Idxreg with the value on the D-bus,
- increg : increment Idxreg,
- decreg : decrement Idxreg.

There are a couple of condition codes generated in Index_generation:

- IdxOflo : true if index > VSB,
- IdxDone : true if index equals Idxreg,
- Idx_Eq_0 : true if index equals zero,
- Idx_Eq_1 : true if index equals one,
- IdxOk : true if index falls within the bounds of the paged object. IdxOk is equal to Access_Ok.

**Range_Checking**

This process carries out range checking whenever it receives an index. It checks whether the index is within the bounds of the paged object (index > 0 and index <= VSB). When the index is found to be correct then Access_Ok is set, else it is reset.

**Address_Generation**

This process generates the addresses for the DRAM- and I/O-operations. The DRAM is presented with a 24-bit address. The I/O- register that will be used is specified by a few of the lower address-bits. How many bits are used depends on the number of registers there are.
The address can be provided either directly or indirectly by adding an index to the base address. Address_Generation performs the following instructions of Addropx:

- noaddropx : do nothing,
- busaddr : load the address from the D-bus,
- incaddr : increment the address,
- decaddr : decrement the address,
- stepaddr : add the value on the D-bus to the address,
- loadaddr : equal the address to the addition of VAB,
- allocatenew : load the address with New_VAB provided by NewSpace_Allocator,
- allocateold : load the address with New_VAB provided by OldArea_Allocator.

When the generated address gets greater then the size of the DRAM then the condition code AddrOflo is set. When the result of the calculation of stepaddr becomes greater or equal to $2^{24}$ then MemAluOflo is set.

**Object_Space_Allocator**

When a new objects needs to be allocated, the start address of the object in physical memory is given by Objptr augmented by one. An object from disk that is loaded in the OldArea is given the start address of Old_Address. These start addresses is presented to Address_Generation via the process New_VAB_Selector. This process represents a multiplexer.

The Objptr-register and the Old_Address-register are represented by NewSpace_Allocator and OldArea_Allocator respectively. When a new object is allocated, the size of the object to allocate, that is given on the D-bus, is added to Objptr to form the base address of the next object to be allocated. When after this, the objptr is greater then the size of the NewSpace, the trap NewSpace_Oflo is set.
When the process receives the instruction from the GC to reset the Objptr-register, this is done. Because the Objptr-register is shared by two processes, accesses are protected.

OldArea_Allocator performs the same allocation-actions when an object must be loaded into the OldArea. This process performs a couple of instructions that are received from the GC, and hence a lock-bit is added to the register.

## 7.3.2. The Pager

The process Pager controls the pager tables and performs the operations on it. Furthermore it generates the condition codes associated with the paged object and the status of the processes. The Pager consists of five subprocesses:

- VR_Bank : this process controls a bank of eight 40-bit registers that can be used to provide an object number as an index into the pager tables. The VR-registers are loadable from the D-bus.
- Number_Allocator : this process assigns object numbers to new objects. It is a loadable 37-bits wide counter.
- Select_pba_Source : this process decides which of its inputs will be used to address the pager tables.
- Generate_CC : this process generates the condition codes of the pager.
- Select_Pager_Source : this process decides in case of a write-action in the pager tables what the source of the write-action will be.

The CSPEC of Pager decides what operation should be performed on the pager tables, which slot(s) will be accessed, and sets the control signals accordingly, if the operation is allowed.

### VR_Bank

The VR-bank consists of eight 40-bit registers that can hold object numbers. These registers can be loaded from the D-bus. The object numbers can be used as an index into the pager tables. They are presented to the pba-source selector. The number of the register that is to be used, is specified by the Flagx-field of the control word.
The instructions that can be executed by this process are of Pagerx:

- ldvr : load the VR-register specified by Flagx from the D-bus,
- page_vr : present the object number in the VR-register specified by Flagx to the pba-source selector.

Because the VR-registers are meant to contain object numbers, they must be presented to the GC. When the instruction is received (Read_VR is true), the register indicated by Loop_Counter is put on the bus Proc_Sources.

### Number_Allocator

When a new object is created, an object number is assigned to it. These object numbers are generated by the Number_allocator, which is a 37-bits counter. If needed the counter can be loaded from the D-bus. The latest number that was allocated is saved in alloc_d. Note that this is not the same function as is provided by Object_Space_Allocator. That process allocates the space needed by the object.
The instructions of Pagerx that can be executed are:

- ldallocator : load the allocator from the D-bus,
- page_allocator : present the value of the counter to the pba-source selector. Increment the counter. If the value of the counter becomes greater then $2^{37}$ the condition code AllocOflo is set.

**Select_pba_Source**

This process determines the source from which pba will be loaded. There are three different sources, the D-bus, the object number allocator, and one of the VR-registers. When none of these sources are selected, the last used number will be put on pba. This number is saved in the REF-register.
The following Pagerx-instructions are executed by this process:

- page_vr : use one of the VR-registers,
- page_bus : use the D_bus as the source,
- page_allocator : use the object number allocator.

Because the REF-register is meant to hold object pointers, it must be provided to the GC. When the GC gives this instruction by setting Read_REF the value of REF is put on the Proc_Sources-bus.

**Generate_CC**

This process generates all the condition codes that are associated with the currently paged object. Furthermore it transforms a data flow into a control flow that is needed in the CSPEC (not very nice, but it had to be done). Because many of these checks can be carried out in parallel, this process consists of eight subprocesses.
The eight processes are:

- Generate_DPFLT.
  This process generates the not-in-core signal (NIC) that is used in another process. If NIC is true (this is when the pba doesn't equal the corresponding VNB) then a DPFLT is issued if the Pgrfetchx field of the control word is set to fetch (DPFLT is allowed). If the DPFLT is inhibited (Pgrfetchx = nopgrfetch) then DPFLT is reset.
- Generate_Obj_Compact.
  This process tells whether the word that is currently on the D-bus is an object number, and if so, whether it is compact or not. If the word is an object number, then IsObj is set. If it is also a compact object, then IsCompact is set too.
- Is_pba_Compact.
  When the object number on the pba is compact, the pba_Compact is set. This signal is used in the generation of other condition codes.
- Generate_CondMatch.
  If the paged object is not compact and the corresponding Cond-bit in the pager tables is set, the CondMatch-condition code is set.
- Generate_IsMod.
  If the paged object is not compact and the Mod-bit of the paged object is set then IsMod is set.
- Generate_IsNew.
  When the paged object is not compact and the New-bit is set, then IsNew is made true.
- Generate_IsSqueeze.
  This process generates the condition code that specifies whether an object should be written to disk if another object needs its slots in the pager tables. It sets IsSqueeze when NIC is true, and either IsMod or IsNew is true.
- Transform_Idx.
  This process is needed because the condition code that specifies whether the index equals one has to be transformed into a control flow, that must be used in the CSPEC.

**Select_Pager_source**

Some of the slots of the pager tables can be loaded from other sources then the D-bus. This process determines from the control word, which source should be written into a pager table slot. If the Addropx-instruction allocatenew (allocateold) is issued, then the source for VAB is the Objptr- (Old_Address-) register.
The other different source is specified by the Pagerx-instruction ldnb. VNB must then be loaded with the value on the pba-bus.

## 7.4. The GC

The GC-process represents the garbage collection process. it is the model of the hardware that implements the generation scavenging algorithm as it is described in chapter 5.
The flowcharts that define the generation scavenging algorithm (see appendix A) can easily be translated to a microprogram. The instructions that are needed to make the translation are summarized in appendix B. The instructions that manipulate a certain resource (for instance a piece of memory or a register) form a field of the control word of the GC.
The model of the garbage collector is derived by grouping the fields of the GC's control word. The first division resulted in six groups. Five of these groups are represented by nodes, and one group is represented by a CSPEC. The CSPEC converts the control word into signals that flow out to other processes of the processor. These signals form instructions that concern the DP, the object pointers containing system sources, the Objptr-, and the Old_Address-register.

The five subprocesses are:

- GC_Sequencer.
  This is the sequencer for the garbage collector's microprogram. In accordance with the Rekursiv, the sequencer is given two microprogram-pointers. The current address is given by GC_CUA. GC_UPC is the microprogram-counter. It is loaded every cycle with the current address augmented by one. In most cases this will be the address of the next control word.
  When an unconditional jump instruction is encountered the next address is given by the control word's field Jump_Address. The microprogram will not be very large, so the overhead that will come in by this extra field is bearable.
  In case of a conditional jump instruction, the associated condition is evaluated. If the condition is true then a jump is made to Jump_Address. Else the next instruction is GC_UPC.

- Loop_Counter.
  Loop_Counter represents a loadable counter. This counter is used when reading in the processor sources or the VTB-slots. It gives the location in the ESTK, or the VTB, or it selects the VR-register. It can be loaded with the size of the ESTK, the VTB-bank, or the VR-bank.

- PT_and_DRAM.
  This process holds the processes that control the DRAM- and the PT-interface of the garbage collector.

- CAMMUX_and_Cond.
  This process represents a multiplexer. The output of the multiplexer goes to the CAM. It can be either loaded in the match-register of the CAM or be loaded directly in the CAM.
  This process also generates the condition codes that are associated with the object pointers.

- CAM_Processes.
  This process consists of all the processes that correspond to the hardware that is close to the CAM, and the CAM itself. It holds the Read_Address- and the Write_Address-register whose values are addresses of locations in the CAM that are accessed directly (without matching). Furthermore it holds a fifo for intermediate storage of object pointers that come from objects on disk.

## 7.4.1. The PT_and_DRAM Process

This process contains two subprocesses. The process called PT_Processes represents the pager tables controlling hardware. The other process, DRAM_Processes contains the interface of the garbage collector with the DRAM. They are put together in one process because they heavily communicate.

### PT_Processes

This process forms the interface between the GC and the pager tables. The real interface is formed by three processes, VABCAM, PT_Memory, and NOS_Process. The other two processes are Addr_In_MUX, and Save_Objptr.

The interface is split in three parts because they are all implemented differently.
PT_Memory controls all banks of the pager tables, except the VAB-bank. Furthermore it controls the status-bits. The resetentry-instruction re-initialises the pager tables entry indicated by PT_Addr_In. It equals VNB, VSB, VRB, and VTB to zero, and resets all four status-bits.
The VTB-bank has its own address-input. When the VTB-words are read, Loop_Counter is used as the address.

The VABCAM-process contains the content addressable part of the VAB-bank. The VABCAM's match- and mask-registers can be set to find all objects that reside in the NewSpace or in the PastSurvivorSpace. Furthermore specific addresses can be searched by loading them in the match-register and issuing the match-opcode.
Of course the normal VAB-operations (loading and resetting a VAB-location) are performed by this process too.

NOS_Process controls the lower 16 bits of the VNB-word of non-compact object pointers. These are not used to hold the object number, since that is given by the main processor's pba. They hold the number of scavenges an object has survived. Whenever an object is copied to FutureSurvivorSpace, this process receives the instruction to increment the NOS.

The Addr_In_MUX-process represents the multiplexer that selects the source to address the pager tables with.

The Save_Objptr-process controls the Save_Objptr-register. This register is loaded at the start of a garbage collection cycle with the content of the Objekt's Objptr register. All objects that are placed after Save_Objptr in NewSpace are placed there during the current cycle. They must be moved at the end of the cycle in order to be able to start with an empty NewSpace.

**DRAM_Processes**

DRAM_Processes contains all processes that are needed to generate addresses for the DRAM. Furthermore it models the interface of the garbage collector with the DRAM.
DRAM_Processes is divided into six subprocesses.

The process DRAM_Address is a multiplexer. It selects the address for write-actions in the DRAM. The inputs of the multiplexer are generated by the processes Future_Addr_Reg, Search_For_Reg, and Copy_To_Reg. Furthermore this process generates the control signals for the DRAM.

The address that is needed when a read-action on the DRAM is performed is generated by the process Index_Reg. This process contains the garbage collector's Index-register. The Index-register is used when copying or scanning objects. It points to a location in the object that is currently paged.
The read-address is given by Phys_Address. It is formed by adding the value of Index to the content of the Start_Address-register (this register contains the start-address of the currently paged object).

This process represents the New_Start_Address-register. It is used by the garbage collector when copying objects in the DRAM to hold the new start-address of the object. It is loaded at the start of a copying sequence, and is written in the VAB-slot of the copied object at the end of the sequence.

The process Future_Addr_Reg contains (as the name obviously shows) the Future_Address-register. This register is used to indicate to which location in FutureSurvivorSpace the next word of an object must be written.

The Copy_To-register is represented by the process Copy_To_Reg. It is used together with the Search_For-register when cleaning the OldArea. The Search_For-register is represented by the process Search_For_Reg. Copy_To contains the address of the location in OldArea where the next word of a live object must be written to. Before a live object can be moved toward another live object, its start-address must be found. Search_for is used to hold the value that might be the start-address of the next live object in the OldArea.

## 7.4.2. CAM_Processes

As is already said, this process contains the processes that use the CAM or that are used in co-operation with the CAM. The CAM is used to hold object pointers of objects that can control other

object pointers. It is added to the garbage collector to prevent objects to be scanned more than once per garbage collection cycle. CAM_Processes consists of three subprocesses.

### CAM_Memory

This process simulates a content addressable memory with an extra address-port for direct read- and write-actions. The match-register can be loaded with the output of the CAMMUX-multiplexer (Obj_Ptr_In). This input can also be loaded into the CAM.
When a CAM turns out to be too expensive, another solution can replace the CAM. For instance, a hash table can perform the same operations, but is slower. The CAM is merely put in the model for clarity.

### Fifo_Memory

This process simulates a first-in-first-out memory. It is used when reading in an object from disk. The incoming words are tested for being non-compact object pointers whose objects can contain other pointers. If so they are pushed onto the fifo. When all words are received from the DP, the words are popped from the fifo and tested further. The fifo is only meant for intermediate storage. For this application the memory does not necessarily have to be a fifo. It may be a lifo too (last-in-first-out).

### Read_and_Write_Address

This process holds the sources that can form the address for an immediate access in the CAM. It consists of three subprocesses. Two of the processes represent an address-register, and the third process (CAM_Addr_MUX) contains a multiplexer that selects the source for the CAM's address-input (Addr_In).
Write_Address_Reg holds the address-register Write_Address for write-actions in the CAM.
Read_Address_Reg contains the Read_Address-register that holds the location that is the source of a read-action.

## 7.5. Review

This chapter contains a few observation in connection with the above model. Of course, between the setting up the model of the processor and the writing of this thesis, a few improvements were noticed. Unfortunately time did not allow to carry through these improvements in the model.

The model assumes that the DP presents words of an object on disk to the garbage collector with a rate of one word per clock cycle. Usually however, a hard disk is a much slower device and. Besides a hard disk is an asynchronous device. When a required sector on disk is found it usually is stored in a buffer first.
Communication with the hard disk should take place on handshake basis.
This implies for the model of the garbage collector that reading in an object from disk gets easier. A request is sent out to the DP to give a word of an object. Then all operations can be performed

on that object before requesting the next word of that object. This means that the fifo becomes redundant. The implementations has become less complex, and cheaper.

When the interface of the disk processor is known, these details can be added to the model of the microprocessor. To define the interface all operations that must be performed by the DP must be known.

When the microprogram is written, a routine must be written too that initialises the garbage collector when the system is powered up. All pointer-registers must be set to the proper values etc. The garbage collector's microprogram must take care of initialising the Objptr- and the Old_Address-register.

A problem could arise when the counter that generates the object numbers for new objects rolls over. To prevent an object number to be assigned more than once, a way must be found to 'garbage collect' the object numbers (all object numbers of live objects must be changed to form a contiguous chain of object numbers), or to allocate only those object numbers whose former objects already died.
This model assumes that $2^{38}$ (= $2.748779069 \cdot 10^{11}$) non-compact object numbers (either pointer-containing or not) is enough to keep the system running for a long time.

The sizes of the different memories, registers, and buses can be adjusted. For instance investigations should point out the most efficient sizes of the different parts of the systems main memory collector in co-operation with the used garbage collector. During a garbage collection cycle, the main processor must not be given the opportunity to fill up the NewSpace completely. If this should happen, a trap is set, and the main processor is interrupted to give the garbage collector the chance to finish its current cycle.

As is already set, the CAM-memories are put in the model to show how the algorithm works. In reality, a CAM is an expensive device, and when a large memory space is required, a CAM takes far more chip-space than an ordinary RAM-device of the same size.
When the size of the CAM does not allow a CAM, another memory-structure should be used that has the same functionality, for instance a hash table. Of course the speed of such a device is lower.

# 8. Conclusions and Recommendations

## 8.1. Conclusions

- Object oriented programming systems can not be implemented efficiently on conventional hardware machines. The primitive operations of an object oriented language differ too much from the standard instruction sets of conventional microprocessors. When a simple assembler is used, the object oriented environment can be changed easily.
  A faster and more secure system can be obtained when a microprocessor is built that supports object oriented principles better. The primitive operations should be implemented in the microcode, and a few features of object oriented programming systems should be implemented in hardware.

- The Rekursiv as it is described by Harland can be improved when a couple of extra features are added. To make the processor faster, a parallel garbage collector and the dynamic binding mechanism must be implemented in hardware.

- The object oriented memory management unit Objekt and the garbage collector have been put in the model of the microprocessor. The sequencer and the stack addressing unit Logik, and the data manipulator Numerik still have to be modelled.
  The garbage collector uses a generation scavenging algorithm that is adjusted to run in parallel with normal program execution.
  A few improvements to the model are explained in chapter 7.5.

- The modelling tool Promod that supports the method described by Hatley and Pirbhai is not ideal for modelling hardware. Promod is designed to make a requirements model that should not contain details about the implementation. When such implementation details can not be avoided, Promod can not be used as it is supposed to be used.
  However the model has been made using Promod. Promod provides a way to define all processes, data- and control flows, and has built-in tests that check for consistency and description errors. When the model is derived with the top-down method of Promod, the hardware can be built using the bottom-up method of IDaSS.

A few errors resulted in the Promod-report as a consequence of the way Promod is used to build the model of the microprocessor. However the meaning stays evident.

## 8.2. Recommendations

- The improvements to the model of the microprocessor as they are stated in chapter 7.5. should be added to the model. These are simple modifications of the model.
  When the interface of the disk processor is known these details can be added to the model too. Therefore, all operations that the disk processor should perform have to be known.

- Before designing the hardware of the memory management unit Objekt and the garbage collector, research has to be done to the most efficient dimensions of memories, registers and buses. The model of the rest of the microprocessor (Logik and Numerik) has to be known. Only when the entire control word of the microprocessor is known and the hardware that implements the control word have been added to the model, a start can be made to design the hardware.

- The dimensions of the different parts of the garbage collector should be found out. When these dimensions are known, one can decide what implementation of the content addressable memory to use. When only a small amount of this memory is to be used, a real CAM can be used. However when a larger amount is needed, a real CAM is too expensive and consumes too much chip-space. Then another solution has to be found, for example an implementation with a hash table.
  The sizes of the different parts of DRAM depends on the duration of a garbage collection cycle. The main program must not be given the opportunity to fill the NewSpace during a garbage collection cycle.

# References

[GoR83] Goldberg, A. Robson, D.
**"Smalltalk-80: The Language and its Implementation"**, Amsterdam: Addison-Wesley, 1983, XX

[HaB86-1] Harland, D.M. Beloff, B.
**"OBJEKT: A Persistent Object Store with an Integrated Garbage Collector"**, ACM Sigplan Not., vol. 22, no. 4 (Apr. 1987), p. 70-79

[Har86] Harland, D.M.
**"A Recursively Microcodable Tagged Architecture"**, ACM Sigarch, vol. 14, no. 3 (Jun. 1986), p. 34-40

[HaB86-2] Harland, D.M. Beloff, B.
**"Microcoding an Object-Oriented Instruction Set"**, ACM Sigarch, vol. 14, Oct. 1986, p. 3-12

[Har88] Harland, D.M.
**"REKURSIV: Object-Oriented Computer Architecture"**, Chichester: Ellis Horwood, 1988

[HaP87] Hatley, D.J. Pirbhai, I.A.
**"Strategies for Real-Time System Specification"**, New York: Dorset House Publishing, 1987

[Jac90] Jackson, F.
**"Generation Scavenging"**, Dr. Dobb's Journal, May 1990, p. 16-28

[LiH83] Lieberman, H. Hewitt, C.
**"A Real-Time Garbage collection Based on the Lifetimes of Objects"**, Communications of the ACM, vol. 26, no. 6 (June 1983), p. 419-429

[Pas86] Pascoe, G.A.
**"Elements of Object-Oriented Programming"**, BYTE, vol. 11, no. 8 (Aug. 1986), p. 139-144

[Pok89] Pokkurini, B.P.
**"Object-Oriented Programming"**, ACM Sigplan Not., vol. 24, no. 11 (Nov. 1989), p. 96-102

[Pou88] Pountain, D.
**"REKURSIV: An Object-Oriented CPU"**, BYTE, vol. 13, no. 12 (Nov. 1988), p. 340-349

[RaK90] Ramackers, G.J. Kuil, W.J.J. van der
**"Wat is Object-Oriented Programmeren?"**, Informatie, vol. 32, 1990, p. 1024-1029

[Rob81] Robson, D.
**"Object-Oriented Software Systems"**, BYTE, vol. 6, no. 8 (Aug. 1981), p. 74-86

[Ung84] Ungar, D.
**"Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm"**, Proceedings of the ACM Symposium on Practical Software Development Environments, Pittsburgh, Penn., USA, April 1984, p. 157-167

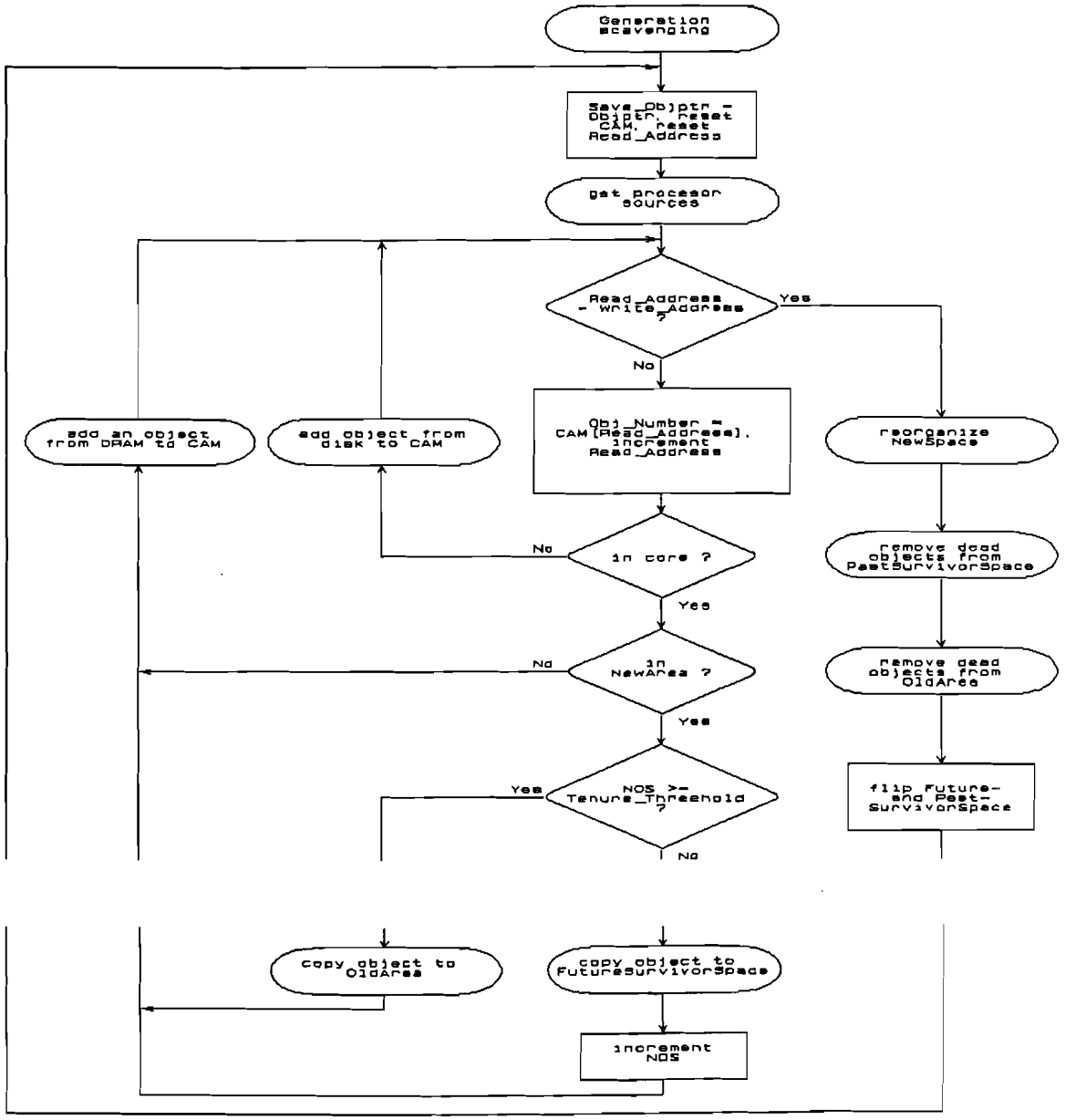
[Ung86] Ungar, D.M.
**"The Design and Evaluation of A High Performance Smalltalk System"**, An ACM Distinguished Dissertation, London: The MIT Press, 1986

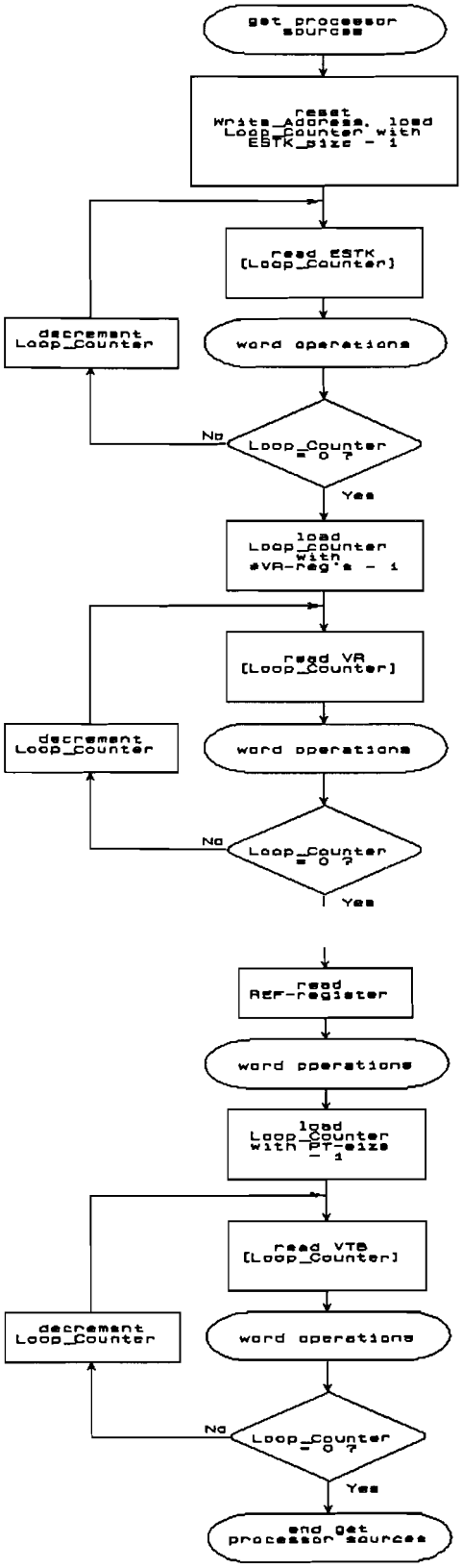
[WiM89] Wilson, P.R. Moher, T.G.
**"Design of the Opportunistic Garbage Collector"**, OOPSLA'89 Conference Proceedings, ACM, October 1989, p. 23-35

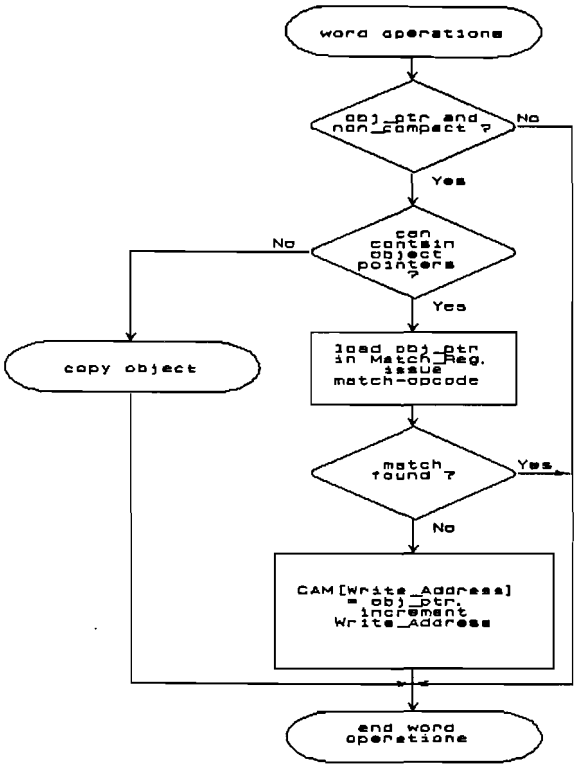
[Xer81] **"The Smalltalk-80 System"**, BYTE, vol. 6, no. 8 (Aug. 1981), p. 36-48

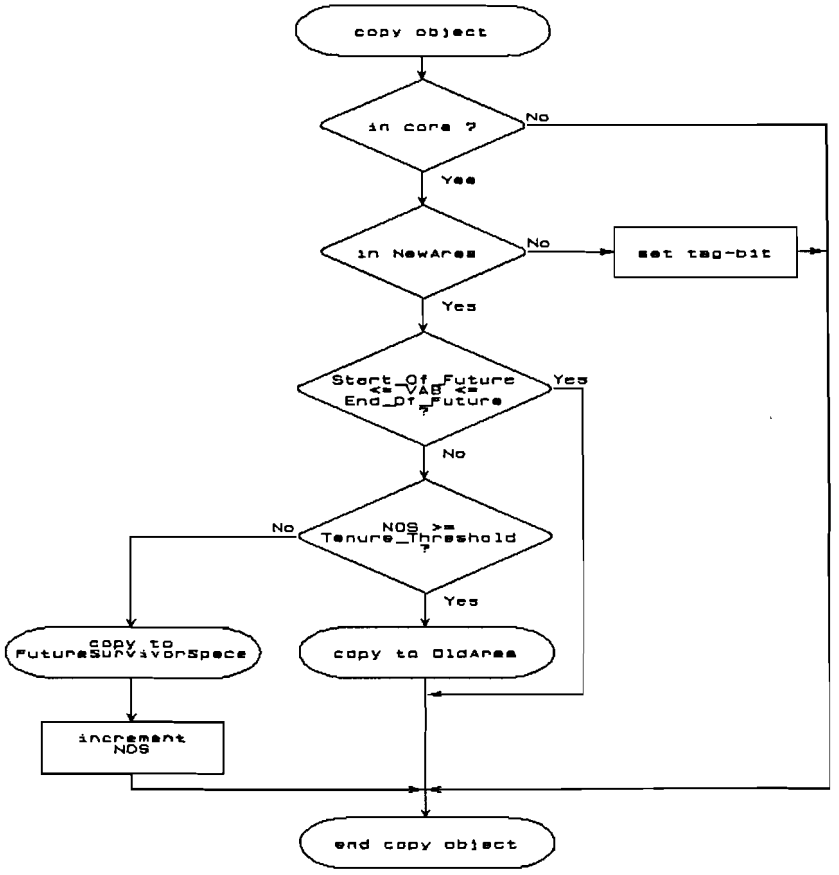
# List of Figures

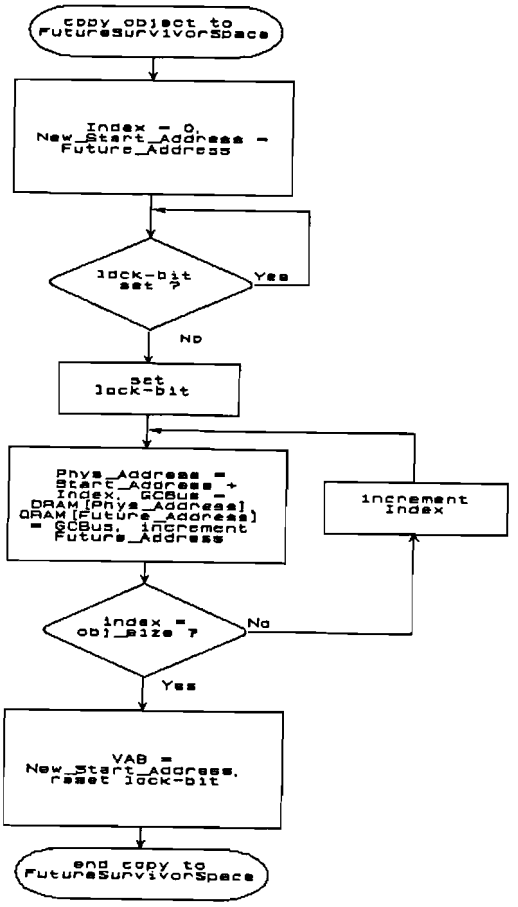
# Appendix A. Flowcharts of the Generation Scavenging Algorithm

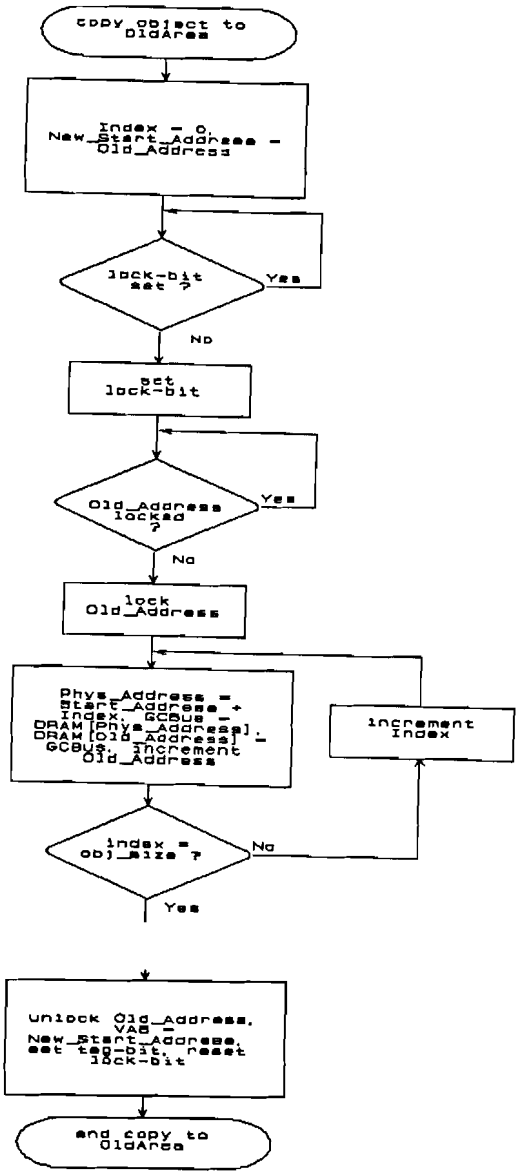


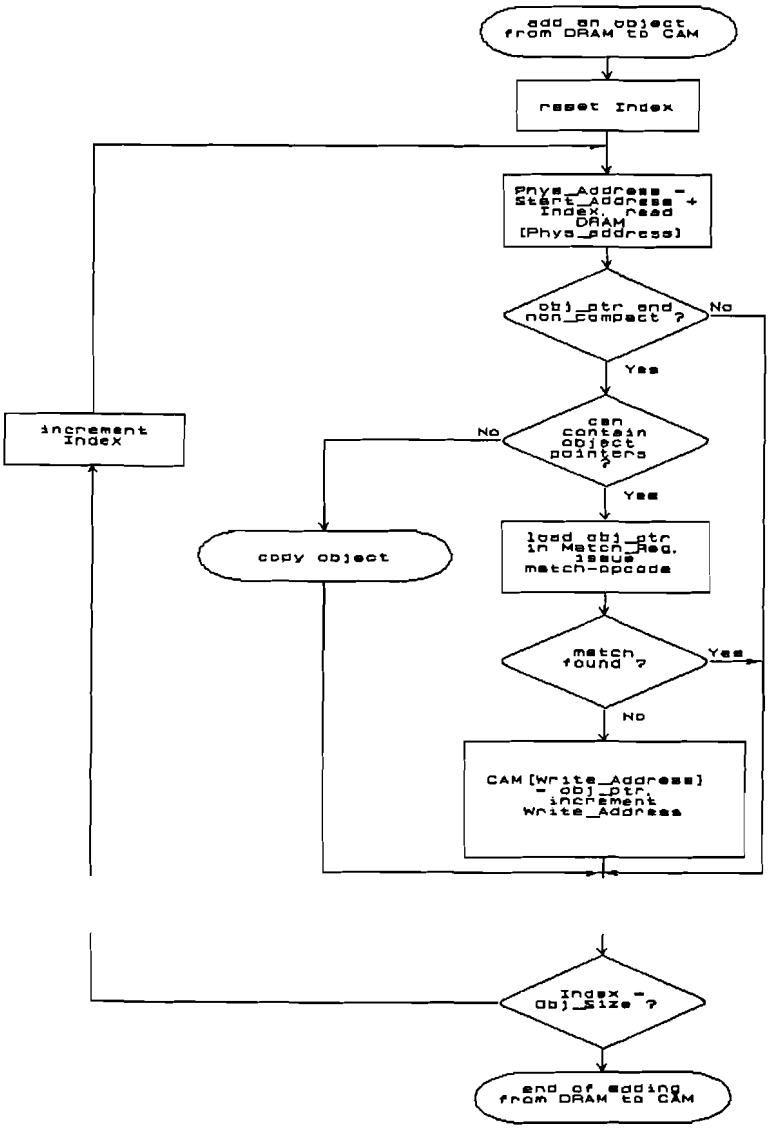

get processor sources
reset Write_Address, load Loop_Counter with ESTK_size - 1
read ESTK [Loop_Counter]
decrement Loop_Counter
word operations
No
Loop_Counter = 0 ?
Yes
load Loop_Counter with #VR-reg's - 1
read VR [Loop_Counter]
decrement Loop_Counter
word operations
No
Loop_Counter = 0 ?
Yes
read REF-register
word operations
load Loop_Counter with PT-size - 1
read VTB [Loop_Counter]
decrement Loop_Counter
word operations
No
Loop_Counter = 0 ?
Yes
end get processor sources

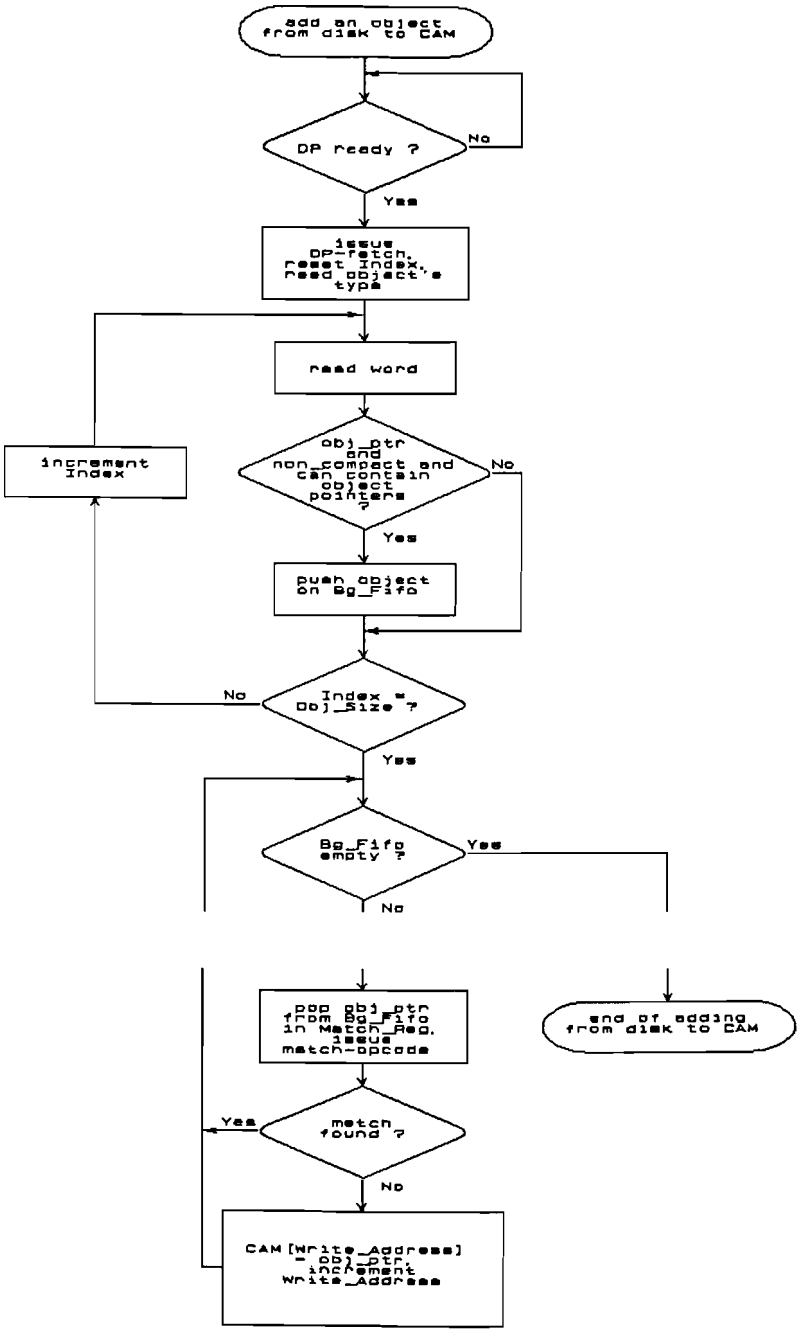

word operations
obj_ptr and non_compact ?
No
Yes
can contain object pointers ?
No
Yes
copy object
load obj_ptr in Match_Reg. issue match-opcode
match found ?
Yes
No
CAM[Write_Address] = obj_ptr. increment Write_Address
end word operations

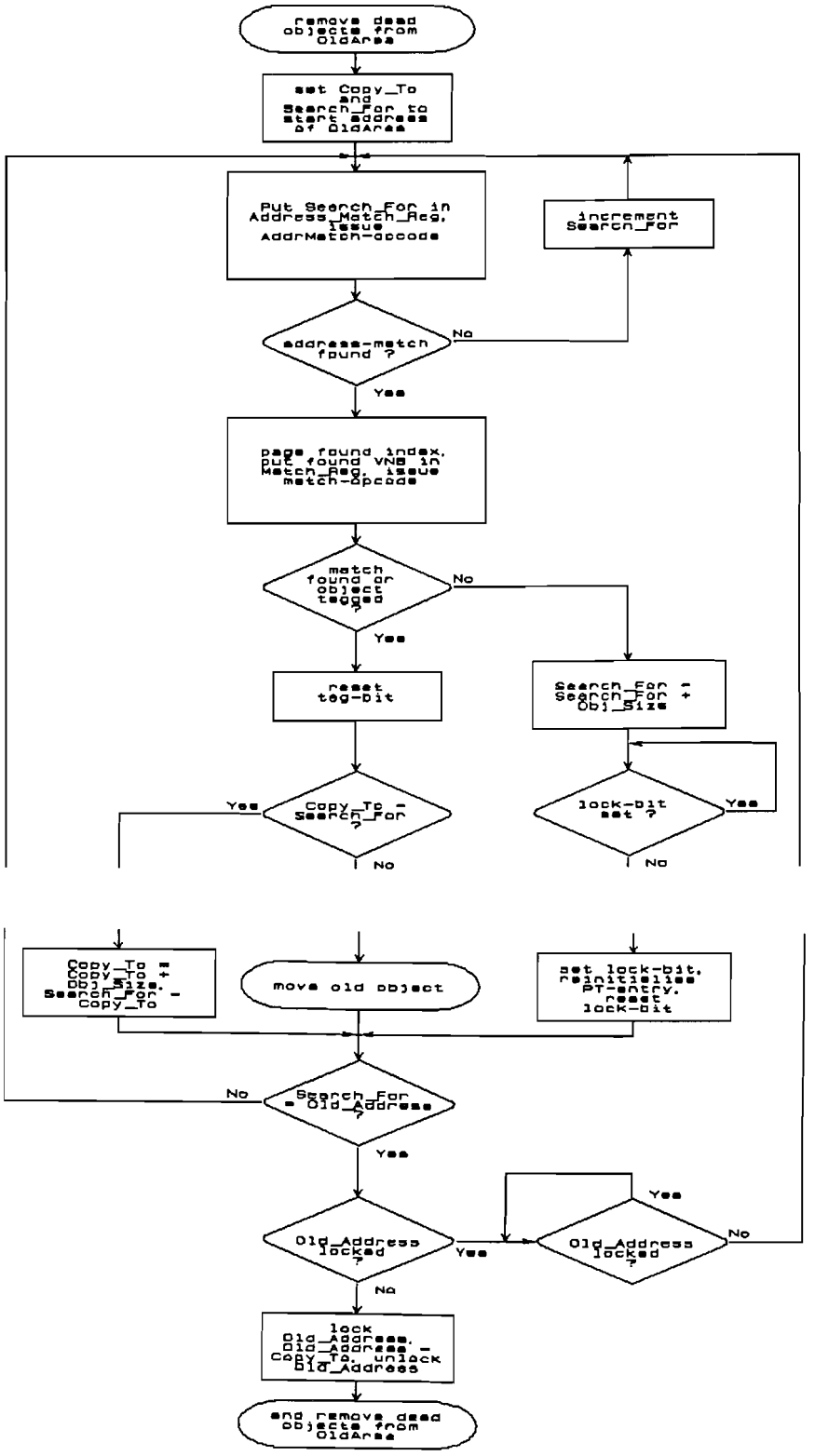

copy object
in core ?
No
Yes
in NewArea
No
set tag-bit
Yes
Start_Of_Future <= VAR <= End_Of_Future ?
Yes
No
NOS >= Tenure_Threshold ?
No
Yes
copy to FutureSurvivorSpace
copy to OldArea
increment NOS
end copy object

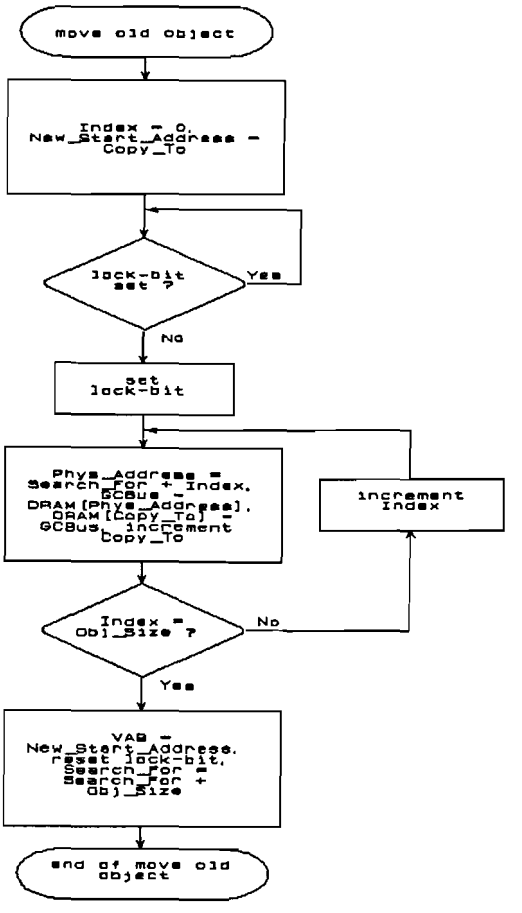

copy object to FutureSurvivorSpace
Index = 0, New_Start_Address = Future_Address
lock-bit set ?
Yes
No
set lock-bit
Phys_Address = Start_Address + Index, GCBus = DRAM[Phys_Address] DRAM[Future_Address] = GCBus, increment Future_Address
increment index
index = obj_size ?
No
Yes
VAB = New_Start_Address, reset lock-bit
end copy to FutureSurvivorSpace

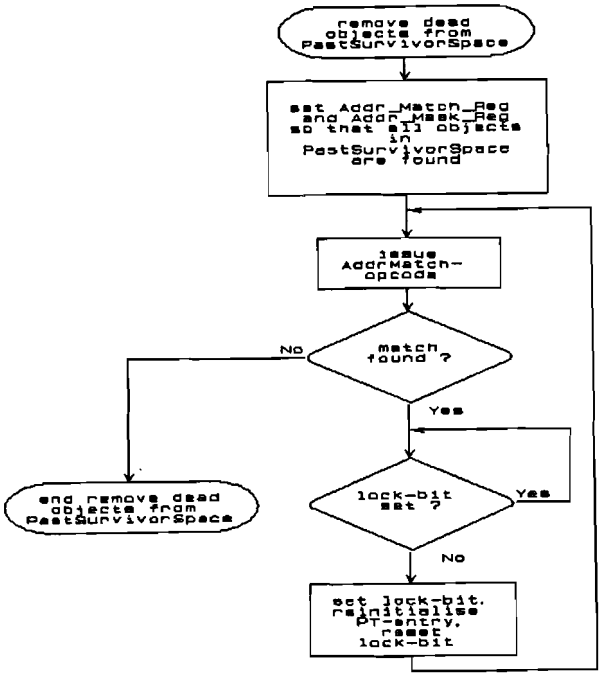

copy object to OldArea
Index = 0, New_Start_Address = Old_Address
lock-bit set ?
Yes
No
set lock-bit
Old_Address locked ?
Yes
No
lock Old_Address
Phys_Address = Start_Address + Index, GCBus = DRAM [Phys_Address], DRAM [Old_Address], GCBus, increment Old_Address
increment Index
Index = obj_size ?
No
Yes
unlock Old_Address, VAB = New_Start_Address, set tag-bit, reset lock-bit
end copy to OldArea

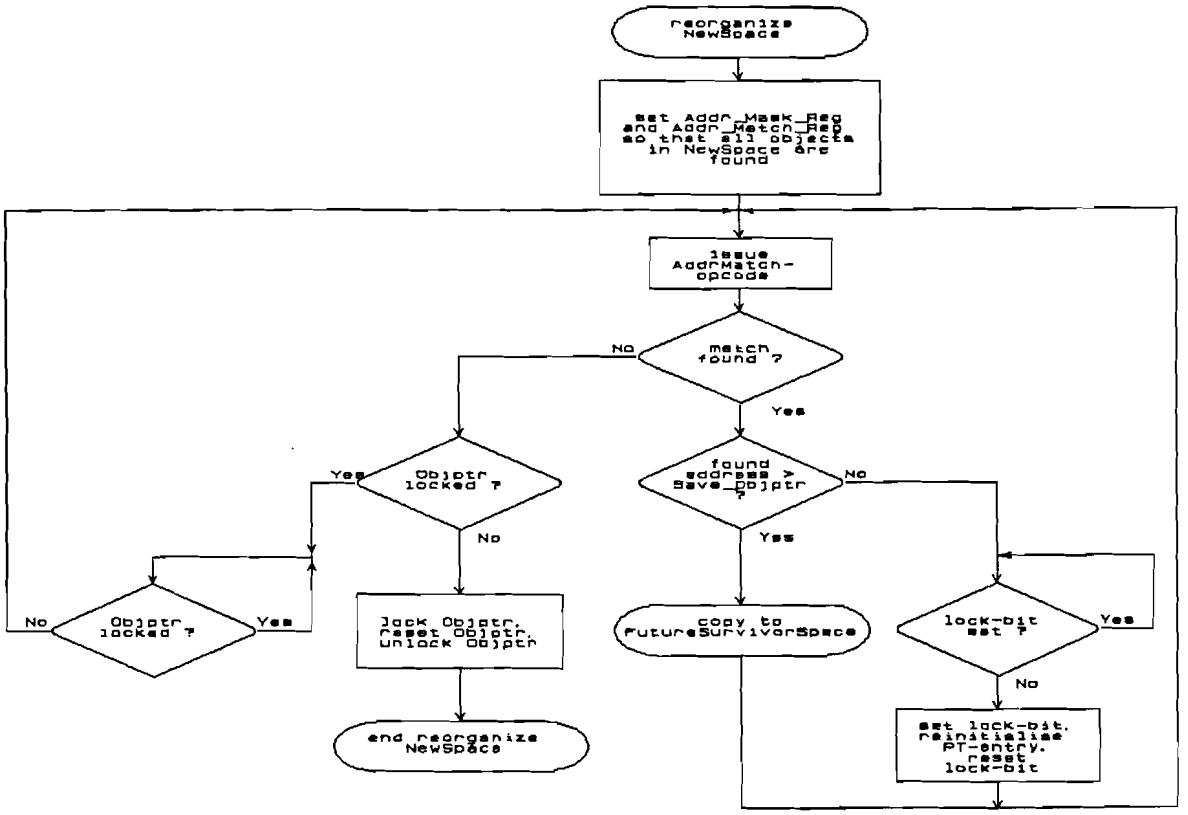

add an object from DRAM to CAM
reset Index
Phys_Address = Start_Address + Index; read DRAM [Phys_address]
obj_ptr and non_Compact ?
No
Yes
can contain object pointers ?
No
Yes
increment Index
copy object
load obj_ptr in Match_Reg. issue match-opcode
match found ?
Yes
No
CAM [Write_Address] = obj_ptr; increment Write_Address
Index = obj_size ?
end of adding from DRAM to CAM


add an object from disk to CAM
DP ready ?
No
Yes
issue DP-fetch, reset Index, read object's type
read word
obj_ptr and non_compact and can contain object pointers ?
No
Yes
increment Index
push object on Bg_Fifo
Index = Obj_Size ?
No
Yes
Bg_Fifo empty ?
Yes
No
pop obj_ptr from Bg_Fifo in Match_Reg, issue match-opcode
end of adding from disk to CAM
match found ?
Yes
No
CAM[Write_Address] = obj_ptr, increment Write_Address


remove dead objects from OldArea
set Copy_To and Search_For to start address of OldArea
Put Search_For in Address_Match_Reg, issue AddrMatch-opcode
increment Search_For
address-match found ?
No
Yes
page found index, put found VNB in Match_Reg, issue match-opcode
match found or object tagged ?
No
Yes
reset tag-bit
Search_For = Search_For + Obj_Size
Copy_To = Search_For ?
Yes
No
lock-bit set ?
Yes
No
Copy_To = Copy_To + Obj_Size, Search_For = Copy_To
move old object
set lock-bit, reinitialize PT-entry, reset lock-bit
Search_For = Old_Address ?
No
Yes
Old_Address locked ?
Yes
Yes
Old_Address locked ?
No
No
lock Old_Address, Old_Address = Copy_To, unlock Old_Address
end remove dead objects from OldArea


move old object
Index = 0,
New_Start_Address =
Copy_To
lock-bit
set ?
Yes
No
set
lock-bit
Phys_Address =
Search_For + Index,
SCBus =
DRAM[Phys_Address],
DRAM[Copy_To] =
SCBus, increment
Copy_To
increment
Index
Index =
Obj_Size ?
No
Yes
VAB =
New_Start_Address,
reset lock-bit,
Search_For =
Search_For +
Obj_Size
end of move old
object


remove dead objects from PastSurvivorSpace
set Addr_Match_Reg and Addr_MASK_Reg so that all objects in PastSurvivorSpace are found
issue AddrMatch-opcode
match found ?
No
Yes
end remove dead objects from PastSurvivorSpace
lock-bit set ?
Yes
No
set lock-bit, reinitialise PT-entry, reset lock-bit


reorganize NewSpace
set Addr_Mask_Reg and Addr_Match_Reg so that all objects in NewSpace are found
issue AddrMatch-opcode
match found ?
No
Yes
found address > Save_Objptr ?
No
Yes
Objptr locked ?
Yes
No
lock Objptr, reset Objptr, unlock Objptr
end reorganize NewSpace
copy to FutureSurvivorSpace
lock-bit set ?
Yes
No
set lock-bit, reinitialize PT-entry, reset lock-bit
Objptr locked ?
No
Yes

# Appendix B. The GC's Control Word

| | | | |
|---|---|---|---|
| **Sequencer** | : | *noseq* | : cua = cua |
| | | *gccont* | : cua = upc ; upc = upc + 1 |
| | | *jmp* | : cua = Jump_Address ; upc = cua + 1 |
| | | *jon* | : jmp if object pointer and not compact |
| | | *jonc* | : jmp if object pointer and not compact and can contain object pointers |
| | | *jcco* | : jmp if can contain object pointers |
| | | *jic* | : jmp if object is in core |
| | | *jin* | : jmp if object is in NewArea |
| | | *jts* | : jmp if tag-bit is set |
| | | *jmf* | : jmp if a match is found |
| | | *jmot* | : jmp if a match is found or tag-bit set |
| | | *jamf* | : jmp if address-match is found |
| | | *jrw* | : jmp if Read_Address = Write_Address |
| | | *jlcz* | : jmp if Loop_Counter = 0 |
| | | *jieo* | : jmp if Index = Obj_Size |
| | | *jnos* | : jmp if NOS ≥ Tenure_Threshold |
| | | *jfe* | : jmp if fifo empty |
| | | *jol* | : jmp if Objptr locked |
| | | *jags* | : jmp if found address > Save_Objptr |
| | | *jces* | : jmp if Copy_To = Search_For |
| | | *jseo* | : jmp if Search_For = Old_Address |
| | | *joal* | : jmp if Old_Address is locked |
| | | *jif* | : jmp if object is in FutureSurvivorSpace (Start_Of_Future ≤ VAB ≤ End_Of_Future) |
| | | *jamf* | : jmp if address-match found |
| **SourceInstrx** | : | *nosourceinstrx* | |
| | | *readvr* | : read the VR-register indicated by Loop_Counter |
| | | *readestk* | : read the ESTK-location indicated by Loop_Counter |
| | | *readref* | : read the REF-register |
| **ObjptrInstrx** | : | *noobjptrinstrx* | |
| | | *readObjptr* | : load Save_Objptr with Objptr |
| | | *resetObjptr* | : lock Objptr; reset Objptr; unlock Objptr |
| **DPInstrx** | : | *nodpinstrx* | |
| | | *DPfetch* | : instruct the DP to fetch an object from disk and present it to the garbage collector |
| **OldAddrx** | : | *nooldaddrx* | |
| | | *lockoldaddr* | : lock Old_Address for GC-use |
| | | *unlockoldaddr* | : unlock Old_Address |
| | | *loadoldaddr* | : load Old_Address with the content of Copy_To |
| | | *incoldaddr* | : increment Old_Address |

*The former four instruction-fields of the garbage collector's control word all interact with the main processor. The Rekursiv must be provided with the hardware that performs these operations.*

**LoopCounterx** : *nolc*
- *declc* : decrement Loop_Counter
- *loadlcESTK* : load Loop_Counter with (ESTK-size - 1)
- *loadlcVR* : load Loop_Counter with (#VR-registers - 1)
- *loadlcVTB* : load Loop_Counter with (PT-size - 1)

**Indexx** : *noindexx*
- *resetindex* : equal the value of the Index-register to zero
- *incindex* : increment Index

**ReadAddrx** : *noreadaddrx*
- *resetreadaddr* : set Read_Address to point to the start of the CAM
- *increadaddr* : increment Read_Address

**WriteAddrx** : *nowriteaddrx*
- *resetwriteaddr* : set Write_Address to point to the start of the CAM
- *incwriteaddr* : increment Write_Address

**FutAddrx** : *nofutaddrx*
- *resetfutaddr* : this is the flipping-operation. Load Future_Address with Start_Of_Past. Swap Start_Of_Past and Start_Of_Fut. Swap the End_Of_Past- and the End_Of_Fut-register
- *incfutaddr* : increment Future_Address

**SearchForx** : *nosearchforx*
- *resetsearchfor* : set Search_For to point to the start of the OldArea
- *loadsearchfor* : load Search_For with the content of Copy_To
- *addsearchfor* : add the current object's size to Search_For
- *incsearchfor* : increment Search_For

**CopyTox** : *nocopytox*
- *resetcopyto* : set Copy_To to point to the start of the OldArea
- *inccopyto* : increment Copy_To
- *addcopyto* : add the current object's size to Copy_To

**NewStartAddrx** : *nonewstartaddrx*
- *loadfuture* : load New_Start_Address with the content of Future_Address
- *loadold* : load New_Start_Address with the content of Old_Address
- *loadcopyto* : load New_Start_Address with the content of Copy_To

**NOSx** : *nonosx*
- *incnos* : increment the number of scavenges of the currently paged object and store it in the VNB-field of the PT

**Fifox** : *nofifox*

*push* : add an object number to the tail of the fifo
*pop* : get an object number from the head of the fifo

**DRAMx** : *nodramx*
*readdram* : MemOut is loaded with DRAM[Phys_Address] (Phys_Address = Start_Address + Index)
*writetofut* : load DRAM[Future_Address] with MemOut
*writetoold* : load DRAM[Old_Address] with MemOut
*writetocopyto* : load DRAM[Copy_To] with MemOut

**CAMx** : *nocamx*
*resetcam* : load every CAM-location with the null-value
*readcam* : page the object pointer at the location pointed to by Read_Address
*writecam* : store the output of the CAMMUX at the CAM-location pointed to by Write_Address
*loadmatch* : load the Match_Reg with the output of the CAMMUX
*match* : issue the match-instruction

**VABCAMx** : *novabcamx*
*findnewspace* : load the Addr_Match_Reg and the Addr_Mask_Reg so that every VAB-address in the NewSpace is found
*findpast* : load the Addr_Match_Reg and the Addr_Mask_Reg so that every VAB-address in the PastSurvivorSpace is found
*loadaddrmatch* : load the Addr_Match_Reg with the content of Search_For
*addrmatch* : issue the match-instruction to the VABCAM

**CAMMUX** : *Obj_Number*
*Obj_Type*
*Fifo*
*Proc_Sources* (ESTK, VR, or REF)
*MemOut*

**PTAddrIn_MUX** : *Obj_Number*
*Obj_Ptr_Out*
*Addr_Out*

**GCPagerx** : *nopagerx*
*settag* : set the Tag-bit
*resettag* : reset the Tag-bit
*setlock* : set the Lock-bit
*resetlock* : reset the Lock-bit
*readVTB* : load the Obj_Type-register with VTB[Loop_Counter]
*loadVAB* : load VAB with New_Start_Address
*resetentry* : VAB ← 0
VNB ← 0
VSB ← 0
VTB ← 0
VRB ← 0

reset the Mod-bit
reset the New-bit
reset the Cond-bit
reset the Tag-bit

**Jump_Address** : This field gives the next address in case of a (conditional) jump-instruction. It is an absolute address of the garbage collector's control store.

# Appendix C. The Processor's Control Word

| | | | |
|---|---|---|---|
| **Addropx** | : | *noaddropx* | |
| | | *busaddr* | memaddr ← D |
| | | *incaddr* | memaddr ++ |
| | | *decaddr* | memaddr -- |
| | | *stepaddr* | memaddr ← memalu(memaddr+D) |
| | | *loadaddr* | memaddr ← balu |
| | | *allocatenew* | memaddr ← Objptr + 1 |
| | | | lock Objptr |
| | | | Objptr ← Objptr + D |
| | | | lock PT[ref] |
| | | | var ← vab[ref] ← memaddr |
| | | | Mod[ref] ← false |
| | | | New[ref] ← true |
| | | | Cond[ref] ← false |
| | | | vnr ← vnb[ref] ← ref |
| | | | unlock PT[ref] |
| | | *allocateold* | memaddr ← Old_Address + 1 |
| | | | lock Old_Address |
| | | | Old_Address ← Old_Address + D |
| | | | lock PT[ref] |
| | | | var ← vab[ref] ← memaddr |
| | | | Mod[ref] ← false |
| | | | New[ref] ← false |
| | | | Cond[ref] ← false |
| | | | vnr ← vnb[ref] ← ref |
| | | | unlock PT[ref] |
| **Flagx** | : | | |
| **Idxldx** | : | *noidxldx* | |
| | | *clridx* | idx ← idxalu ← 0 |
| | | *newidx* | idx ← idxalu ← idxreg |
| | | *incidx* | idx ← idxalu ← idx + 1 |
| | | *decidx* | idx ← idxalu ← idx - 1 |
| | | *ldidx* | idx ← idxalu ← D |
| | | *idxstep* | idx ← idxalu ← idx + D |
| | | *idxnext* | idx ← idxalu ← ?(idx >= vsr) 1 / idx+1 |
| | | *newidxreg* | idxreg ← idxalu ← idx |
| **Memopx** | : | *nomemopx* | |
| | | *memget* | memout ← mem[memaddr] |
| | | *memput* | memout ← mem[memaddr] ← D |
| | | *idxget* | memout ← ?(access_ok) mem[memaddr] / undefined |
| | | *idxput* | ?(access_ok) |
| | | | lock PT-entry |

| | | |
|---|---|---|
| | | memout ← mem[memaddr] ← D |
| | | Mod[ref] ← true |
| | | unlock PT-entry |
| | | / memout ← undefined |
| | *ioget* | D ← io[memaddr] |
| | *ioput* | io[memaddr] ← D |
| **Pagerx** : | *nopagerx* | |
| | *ldvr* | vr[flagx] ← D |
| | *ldallocator* | allocator ← D |
| | | alloc-d ← D-1 |
| | *setmod* | ?(!compact(pba)) |
| | | Mod[ref] ← true / nothing |
| | *init_pager* | ?(!compact(pba)) |
| | | var ← vab[ref] ← D |
| | | vnr ← vnb[ref] ← D |
| | | vsr ← vsb[ref] ← D |
| | | vtr ← vtb[ref] ← D |
| | | vrr ← vrb[ref] ← D |
| | | Mod[ref] ← false |
| | | New[ref] ← false |
| | | Cond[ref] ← false |
| | | / nothing |
| | *clrflags* | ?(!compact(pba)) |
| | | Mod[ref] ← false |
| | | New[ref] ← false |
| | | Cond[ref] ← false |
| | | / nothing |
| | *page_vr* | ?(!compact(pba)) |
| | | var ← vab[vr[flagx]] |
| | | vnr ← vnb[vr[flagx]] |
| | | vsr ← vsb[vr[flagx]] |
| | | vtr ← vtb[vr[flagx]] |
| | | vrr ← vrb[vr[flagx]] |
| | | / strip up compact to set registers |
| | *page_bus* | ?(!compact(pba)) |
| | | var ← vab[D] |
| | | vnr ← vnb[D] |
| | | vsr ← vsb[D] |
| | | vtr ← vtb[D] |
| | | vrr ← vrb[D] |
| | | / strip up compact to set registers |
| | *page_allocator* | var ← vab[alloc] |
| | | vnr ← vnb[alloc] |
| | | vsr ← vsb[alloc] |
| | | vtr ← vtb[alloc] |
| | | vrr ← vrb[alloc] |
| | | increment alloc |
| | *ldnb* | ?(!compact(pba)) vnr ← vnb[ref] ← ref / nothing |
| | *ldsb* | ?(!compact(pba)) vsr ← vsb[ref] ← D / nothing |
| | *ldtb* | ?(!compact(pba)) vtr ← vtb[ref] ← D / nothing |

| | | | |
|---|---|---|---|
| | | *ldab* | ?(!compact(pba)) var ← vab[ref] ← D / nothing |
| | | *ldrb* | ?(!compact(pba)) vrr ← vrb[ref] ← D / nothing |
| | | *cldrb* | ?(idx=1 & access_ok) vrr ← vrb[ref] ← D / nothing |
| | | *csetcond* | ?(!compact(pba)) Cond[ref] ← cc / nothing |
| **Pgrfetchx** | : | *nopagrfetch* | inhibit the fetch DPFLT |
| | | *fetch* | enable the raising of the fetch DPFLT |
| **Regx** | : | *noregx* | |
| | | *ldreg* | idxreg ← D |
| | | *increg* | increment idxreg |
| | | *decreg* | decrement idxreg |