NEWCASTLE UPON TYNE
UNIVERSITY LIBRARY
ACCESSION No.
83-10990
LOCATION
Thesis
L2772

GENERAL PURPOSE DECENTRALISED COMPUTER ARCHITECTURE

Richard P. Hopkins

Computing Laboratory, University of Newcastle upon Tyne

Ph.D. Thesis 1983

ABSTRACT

This thesis is concerned with decentralised highly concurrent computer architectures which may eventually provide alternatives to the centralised sequential architectures of conventional general-purpose computers. There is currently considerable research into such alternatives, for which the principal motivations are the use of concurrency to improve performance, the support of various novel, very high level programming languages, and the exploitation of very large scale circuit integration (VLSI). The different proposed alternative architectures are surveyed and analysed, and architectures synthesising their underlying concepts are proposed.

The thesis consists of three main parts. The first part is an analysis and survey of proposed general-purpose decentralised architectures. Three classes of architectures are identified, namely control flow, data flow and reduction. The analysis shows that each class has particular, complementary, strengths and weaknesses.

The second and third parts cover the development of two architectures which combine the different concepts underlying control flow, data flow and reduction in order to overcome the individual weaknesses in each. Thus the second part presents a "data/control flow" architecture which is a synthesis of data flow and control flow. There is an experimental implementation of this architecture in which a number of standard microcomputers cooperate in the execution of a program.

In contrast the third part presents a "recursive control flow" (RCF) architecture which is a synthesis of control flow, data flow and reduction. This architecture is based on a set of general principles of recursive structuring which are intended to provide a common basis for decentralised system organisation at various architectural levels, ranging from VLSI design to geographically distributed networks. The RCF work is thus not only an investigation into the possibility of incorporating control flow, data flow and reduction concepts in a single parallel computer but also an initial investigation of the application of the recursive structuring principles. These two aspects of the work are closely related in that recursive structuring facilitates the modularity which is required for the synthesis of control flow, data flow and reduction into a coherent overall system.

An implementation of the RCF architecture, using a number of identical microcomputers, is proposed. The detailed design of a special-purpose LSI microcomputer chip for this implementation is currently being produced.

ACKNOWLEDGEMENTS

I gratefully acknowledge the diligence and interest taken in this research by my colleagues in the Computer Architecture Research Group, principally Philip Treleaven and Paul Rautenbach, and my thesis supervisor Brian Randell. Particularly I thank Philip Treleaven and Brian Randell for having patiently read various drafts of this thesis and having made some improvement in my ability to write.

This research would have been impossible but for financial support from the Science and Engineering Research Council of Great Britain both for myself as a research student and research associate, and for the computing facilities that I have used.

CONTENTS

INDEX TO FIGURES

1. INTRODUCTION

1.1. Background

Conventional von Neumann architectures can be characterised by a number of "von Neumann principles":

1. linear organisation of fixed size memory cells

2. one level address space, where each address is globally unique

3. low level machine code in which instructions are elementary operations performed on elementary operands

4. sequential, centralised control of execution

5. centralised machine organisation of a single computer incorporating processor, communications and memory

Although these principles have for over 30 years provided an adequate basis for general-purpose computers, there has recently been much interest in possible alternative general-purpose architectures. There are currently at least 20 research groups working on experimental, non-von Neumann architectures in which one or more of the above principles are modified.

There are three main, related, motivations for this interest in novel architectures. Firstly there is the continuing demand for increased computing power, particularly in applications such as weather forecasting and wind tunnel simulation. The technologies available (and ultimately the natural laws of physics) limit the possible performance

attainable for a single processor, and thus the ability of conventionally organised high-speed computers to meet these demands[1]. The utilisation of a large number of processors cooperating on a single task offers the potential of overcoming the technological performance limitations.

Secondly there is the interest in new classes of very high level programming languages, particularly functional or applicative languages[2,3], and logic languages[4], which are claimed to have greater expressive power than conventional languages and are intended to provide easier means of producing reliable programs. Whereas existing sequential assignment-based languages are well matched to the von Neumann architecture, these new classes of language are based on radically different principles, and implementations on conventional architectures tend to be relatively inefficient. An essential characteristic of these languages is that a program does not specify a sequence in which statements are to be executed and this naturally leads to highly concurrent implementations.

Thirdly there is the need to exploit very large scale integration (VLSI) in the design of general-purpose computers. There are a number of considerations in VLSI design which make it desirable to find an alternative to conventional architectures[5,6]. In particular there is the need to implement storage and processing functions close together on the same chip in order to minimise communications. These considerations lead to architectures such as [7] in which a large powerful computer is constructed from a multiplicity of simple single-chip microcomputers each incorporating a general-purpose processor and local memory.

Research projects motivated by one or more of the above considerations have resulted in a number of experimental special-purpose and general-purpose architectures which are able to utilise many processors operating concurrently on a single task under decentralised control. In the special-purpose decentralised architectures the hardware organisation is closely matched to the concurrency structure of a particular class of tasks, as for example in Systolic Arrays[8]. The concern of this thesis however is with general-purpose decentralised architectures which can be programmed to perform effectively, and exploit implicit concurrency in, a large range of tasks.

Three classes of general-purpose computer architectures can be identified as control flow, data flow and reduction. These classes differ fundamentally in their operational models, that is in the way instruction execution is initiated and the way data is communicated between instructions. In control flow the execution of an instruction is triggered by the flow of control through the program, and data is communicated between instructions by being stored in shared memory cells. (Conventional von Neumann architectures have a single flow of control whereas control flow architectures providing concurrency have multiple flows of control.) In data flow the execution of an instruction is triggered by the availability of the input data that it uses and when executed an instruction stores its result directly into those instructions which use that result. In reduction the execution of an instruction is triggered by the requirement for the output data which it generates and when executed an instruction is replaced by (is reduced to) its result. The single most important characteristic of the novel data flow and reduction models is that they are implicitly concurrent. If

several data flow instructions have all their inputs available then they can all be executed concurrently. If one reduction instruction requires the outputs from several other instructions then those instructions can be executed concurrently. In contrast, the concurrency provided by multiple flows of control is explicitly controlled by the programmer.

Parallel computers form one of two major fields of decentralised computing systems research, the other being that of geographically distributed computer networks. There is some degree of relationship between these two fields. The design of individual computers may be affected by the likelihood of their being connected into computer networks. Similar functions may be found in both types of decentralised system, for example to organise the cooperation between concurrently executing program units, and the decentralised allocation of system resources. There is the possibility of the same microcomputer being used as one of the processors for a general-purpose parallel computer, as the main processor of a node in a local area network, or as a component of an embedded system such as an on-board flight control system. This latter class of system could be considered as either a special-purpose parallel computer or a small-scale computer network. Thus the distinction between parallel computer architecture and computer network architecture is becoming more a matter of degree than fundamental difference. These considerations have contributed towards developments such as the "building sized computer" of [9], the Intel 432[10] and transputers[11]. In terms of the three classes of architectures identified above, these developments are all based on the conventional control flow model (rather than one of the novel, implicitly concurrent models) and carry forward most of the von Neuman principles.

1.2. Thesis Outline

The principal work reported here consists of an analysis of control flow, data flow and reduction computers and the synthesis of their underlying concepts, together with the application of principles of recursive structuring, in the design of general-purpose decentralised computer architectures. The application of recursive structuring principles in the context of geographically distributed computer networks is also considered. This research constitutes a major part of the work of the Computer Architecture Research Group at the University of Newcastle upon Tyne. Much of the work covered by this thesis, together with related work of other members of the group, has been previously reported elsewhere[12-18].

The first part of the thesis, Chapter Two, is an analysis and survey of existing control flow, data flow and reduction architectures. The basis for this analysis is the classification first developed by Treleaven and myself in [14] and subsequently expanded by ourselves and Brownbridge in a broad survey of data flow and reduction computers[16]. This Chapter repeats and further develops those parts of [16] to which I made a major contribution. It covers for each class of architecture the operational concepts on which it is based, the ways in which machine code programs are organised, and the ways in which machine resources are organised. The analysis shows that a fundamental characteristic of an architecture, largely determining its particular strengths and weaknesses, is the choice of mechanisms provided for propagating control and data through a program. The strengths and weaknesses of control flow, data flow and reduction are largely complementary which suggests

experimenting with combinations of the concepts and mechanisms found separately in each.

All the novel computer architectures discussed in this Thesis, data flow and reduction covered in Chapter 1 and the experimental architectures developed subsequently, provide for computing machines that are Universal, in the Turing Machine sense. Thus the only ultimate criteria for their evaluation is their "efficiency" in terms of hardware cost, program execution performance, and production costs of software to provide an effective user interface and to organise the effective exploitation of machine resources. Quantitative evaluation of an architecture in those terms requires extensive experimentation on "realistic" implementations. However at the time of writing, development of novel data flow and reduction architectures has only progressed to the point of "demonstration" rather than realistic implementations. Thus they can as yet only be evaluated in qualitative terms of particular conceptual strengths and weaknesses which may be expected to affect their quantitative characteristics. These comments apply even more to the highly experimental architectures covered in the subsequent Chapters which build on the concepts of data flow and reduction and are in an even earlier stage of development.

The second part of the thesis, Chapter Three, presents an experimental architecture which is a synthesis of data flow and (concurrent) control flow[13,15]. The original idea of such a synthesis was first suggested by Treleaven and developed into this architecture principally by Rautenbach and myself. This Chapter is a summary of the full report on the architecture[12] covering its operational model, program organi-

sation and machine organisation. Implementations of the architecture, based on several cooperating microcomputers, were subsequently developed by other members of the Computer Architecture Group and are reported on in detail elsewhere[19,20].

The third part, Chapters Four and Five and Appendix A, presents a second experimental architecture, the recursive control flow architecture, which is a synthesis of control flow, data flow and reduction obtained by generalising the conventional von Neuman architecture. This generalisation is based on the use of recursive decentralised structures, and involves the replacement of all the von Neuman principles by corresponding "recursive principles":

1. hierarchy of variable length memory cells

2. contextual address space, where an address is a variable length sequence of selectors like a telephone number

3. recursive machine language in which instructions may contain elementary or complex operations or operands

4. parallel, decentralised control of execution

5. network of computing elements with each element incorporating processor-communications-memory and a group of computing elements being functionally equivalent to a single computer

Chapter Four covers the architecture's operational model, its program organisation and machine organisation. Also, towards the end of Chapter Four there is a discussion of the applicability of the recursive control flow operation model and machine organisation in the context of

geographically distributed networks. Chapter Five covers a proposed hardware implementation of the recursive control flow architecture using large numbers of a special-purpose LSI chip. Appendix A defines a machine code instruction set for this implementation. Chapter Five also covers a number of other computing system implementations which incorporate similar recursive structuring concepts, including both a general-purpose parallel computer and a computer network system.

The material contained in these two Chapters and the Appendix is original to this thesis and has not been published elsewhere, although it is based on outline ideas proposed by Treleaven and myself in [17]. The proposed implementation is now being developed by another member of the Computer Architecture Group in order to produce a detailed chip design.

Chapter 6 presents the main conclusions of the thesis. It is claimed that the recursive control flow architecture enables the different styles of control flow, data flow and reduction programs to be easily combined, so that each can be used where appropriate. Also that the architecture in general provides a promising basis for future general-purpose decentralised VLSI computing systems, and warrants further development. Further work to be done and possible lines of future development are then identified.

2. ANALYSIS AND SURVEY

This Chapter covers recent research into general-purpose decentralised architectures, particularly in the areas of data flow and reduction. Currently there are many research groups working in these areas, a number of whom have produced experimental machine designs[16]. The purpose of this Chapter is to identify the concepts and relationships within these areas of research and to discuss the advantages and disadvantages of different approaches. The Chapter starts by introducing the operational models underlying control flow, data flow and reduction architectures, then discusses ways in which the architectures are programmed and implemented, and finally evaluates their main advantages and disadvantages.

2.1. Operational Models

Control flow, data flow and reduction architectures are each based on different concepts of the way programs are represented and executed and the way control and data are passed from one part of a program to another. In order to illustrate and compare these basic concepts I will discuss the representation and execution of an example program fragment, the statement **a:= (b+1) * (b-c)**, in terms of simple operational models for each class of architecture. For the operational models an abstract machine code representation will be used in which an instruction is a sequence of arguments delimited by brackets. For example the **(b+1)** might be represented as the machine code instruction **(+ b 1 t1)** with four arguments, the last of which, **t1**, identifies the destination of the result. At a particular point in the execution of a machine code

program there will be (one or more) active instructions to be executed. For example in a conventional architecture the instruction pointed to by the program counter is the single active instruction. In discussing the operational models, the progress of execution through a program will be indicated by marking a currently active instruction with a ¦, referred to as an "activity". For example **(¦+ b 1 t1)** means that the + operator is about to be executed and the activity, ¦, can be thought of as representing an actual processor to execute that operator.

2.1.1. Control Flow

Figure 1 shows a conventional control flow representation for the example **a:= (b+1) * (b-c)**. There is a sequence of instructions, **i1, i2** and **i3**, and some shared memory cells, **b, c, t1** etc., for passing data between instructions. Each instruction (e.g. **i1**) consists of an operator (**+**), input arguments each of which may be a literal (**1**) or a reference (**b**) to a memory cell, and a reference (**t1**) to the memory cell for the result. The references to memory cells are also shown as solid arcs.

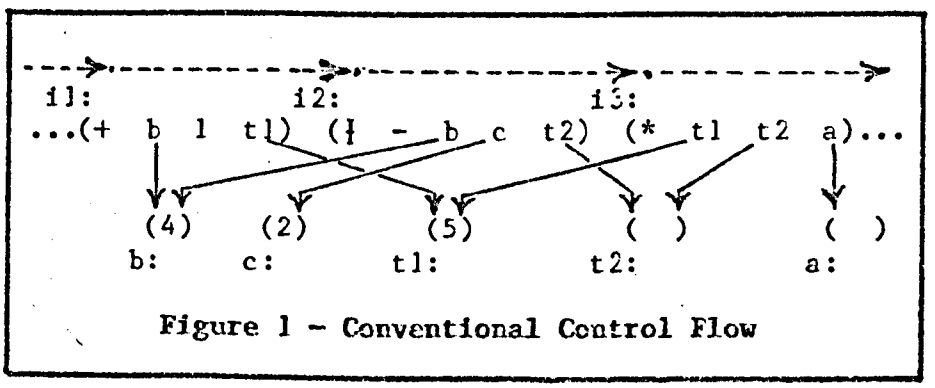


Figure 1 - Conventional Control Flow

There is a single activity, ¦, shown at **i2**, which passes from one

instruction to the next along the thread of control shown by dotted arcs. To execute an instruction, addressed operands are loaded from memory, the result is computed as defined by the operator, the result is stored back in memory, and the activity then implicitly passes on to the next instruction in sequence.

There are also forms of control flow which provide concurrent program execution. In the "multi-thread" form of control flow, conventional control flow is augmented by special control operators for creating and synchronising multiple threads of control, such as the **FORK** and **JOIN** shown in Figure 2. After the execution of the **FORK** there are two activities, one at its implicit successor, **i1**, and one at the addressed instruction, **i2**. Instructions **i1** and **i2** thus execute concurrently. Both activities will then reach the **JOIN** instruction, either via the explicit **GOTO** following **i1**, or as the implicit successor of **i2**. The **JOIN** synchronises the two activities and then execution continues with its implicit successor, the multiply instruction. Apart from the inclusion of such special control operators multi-thread control flow is similar to conventional control flow.

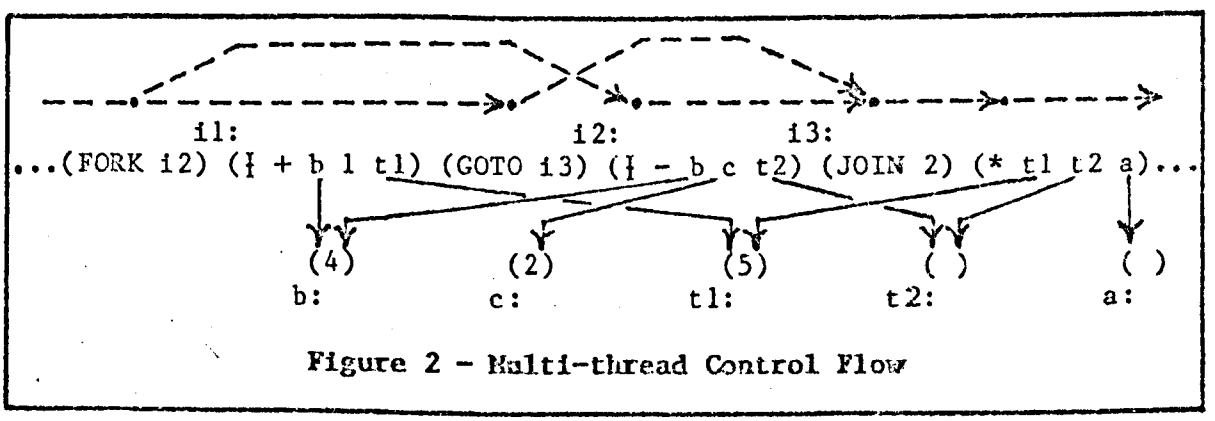


Figure 2 - Multi-thread Control Flow

Another form of concurrent control flow, shown in Figure 3, is "parallel" control flow in which instruction execution is controlled by explicit control signals or control tokens, rather than there being implicitly sequential threads of control. All instructions can potentially execute in parallel, but each instruction requires some number of control tokens from other instructions before it can actually execute. In Figure 3 each ? in an instruction is a control argument representing the requirement for a control token. When an instruction has executed it transmits control tokens to successor instructions identified by its final reference arguments (the flow of control tokens is represented by dotted arcs). A reference for a control token, such as **i3/2** in **i2**, specifies a particular argument position, **2**, in a particular instruction, **i3**. This form of control flow is logically equivalent to multi-thread control flow, with multiple successors (as in **i4**) corresponding to a **FORK**, and multiple control arguments (as in **i3**) corresponding to a **JOIN**.

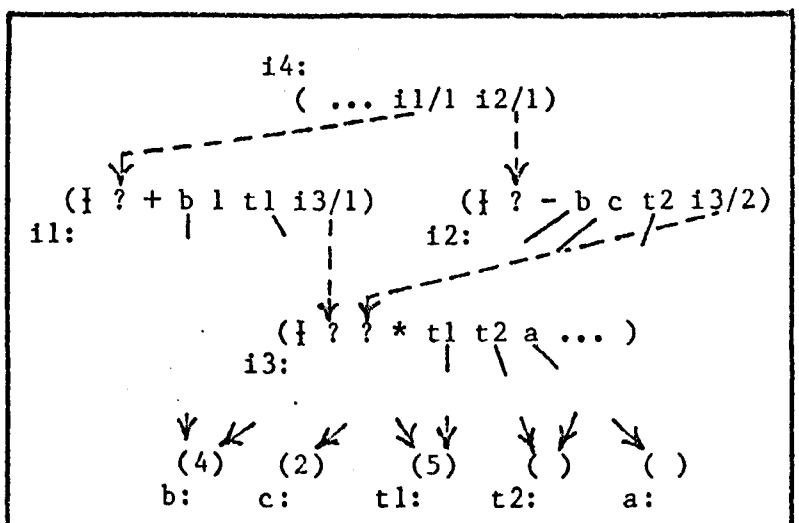


Figure 3 - Parallel Control Flow

2.1.2. Data Flow

Data flow architectures provide highly concurrent program execution without the programmer needing to organise the concurrency explicitly. This is achieved by basing execution on the availability of data so that concurrency is implicit - any instruction can execute if all its operands have been produced.

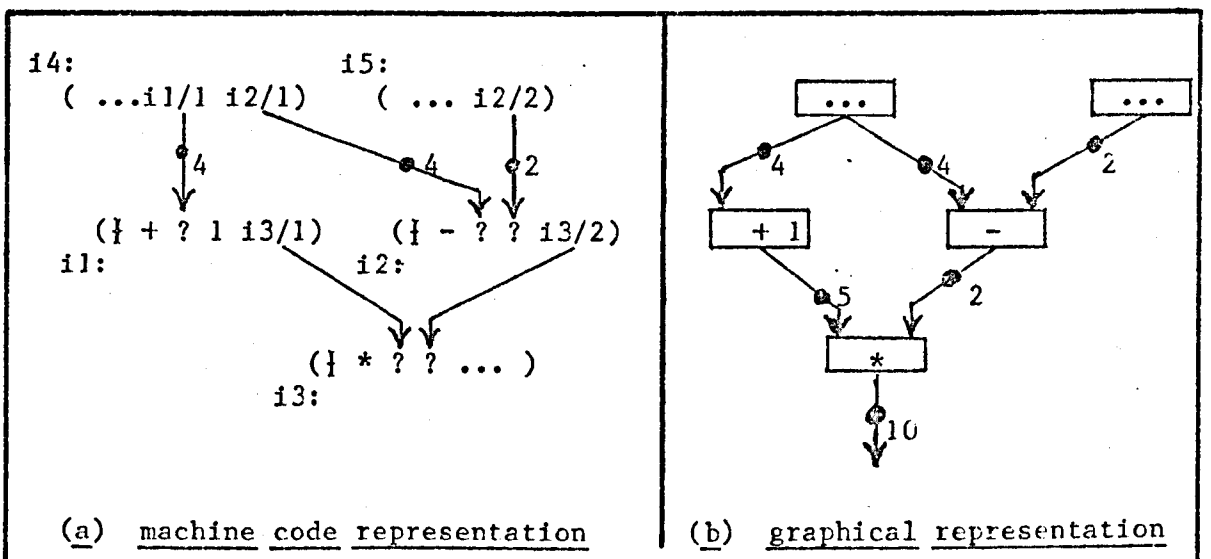


Figure 4 - Data Flow

Figure 4 shows a data flow program for the same example, using in (a) a machine code representation and in (b) a more usual graphical representation (data flow programs are often referred to as data flow graphs, with the instructions being referred to as nodes). An instruction (e.g. **i1**) consists of an operator (+), input arguments each of which may be a literal (1) or an unknown (?), and output arguments each of which is a reference (i3/1). An output reference identifies a destination instruction (i3) and a particular unknown argument position (1). (These references specify the flow of data through the program which is

also shown by solid arcs.)

All instructions are notionally active and can potentially execute in parallel, execution being constrained by the availability of data. When an instruction (e.g. **i4**) has executed it transmits its result (the value of **b**, i.e. 4) as data tokens to the other instructions using that result. (Data tokens are shown as black dots flowing down the arcs.) A data token overwrites an unknown in the destination instruction, thereby making its value available to that instruction. When all of an instruction's operands are available, i.e when all unknowns have been replaced by results, it can execute, computing its result which is then transmitted to further instructions. (In terms of the graphical representation, an instruction executes when a data token is present on each of its input arcs and places its result on all its output arcs.)

2.1.3. Reduction

In data flow and concurrent forms of control flow, the major divergence from conventional control flow is in providing for concurrent program execution. These architectures retain the conventional form of program representation in which a program is a collection of fixed size instructions whose arguments are primitive operators and operands. In contrast, the principal concern in reduction architectures is the direct support of very high level applicative programming, which motivates a different form of program representation. In reduction an "instruction" is an expression comprising a function or primitive operator and its arguments. Each argument may be a simple operand or a nested expression. A program is a set of named expressions each defining a value.

For example the program fragment -

$$a \equiv (b+1) * (b-c) \text{ where } b \equiv 7-3,\ c \equiv 2$$

gives a definition for **a** using subsidiary definitions for **b** and **c**. Unlike a named variable of control flow, a named definition has a fixed value (possibly determined by parameters). Thus a reference such as **b**, its defining expression, **7-3**, and its value, **4**, are all mathematically equivalent and can be freely interchanged without affecting the result of the program. The interchangeability of a reference and the referenced expression is an important property known as "referential transparency". This property is claimed to facilitate both informal understanding of a program and its formal manipulation in for example program verification.

In executing a reduction program instructions are progressively evaluated and replaced by their results until the program has been reduced to its simplest form. Reduction architectures differ both in the order in which instructions are selected for evaluation and in the way in which instructions are represented and manipulated. The order of evaluation is not explicitly controlled by the programmer but is determined by an implicit "computation rule". There are a number of possible computation rules[21], the main differences being between "innermost" and "outermost" rules. An innermost computation rule only selects instructions with literal (fully evaluated) operands (initially the instruction defining **b** in this example). The evaluation of these instructions will result in other instructions being selected. An outermost computation rule starts with the instruction for the required result (the multiplication in this example) and is recursively applied

to evaluate the required operands. Every computation rule produces the same result for a program, if the program terminates with that rule. However there are programs which terminate with an outermost computation rule but not with an innermost rule, and subsequently I will concentrate on reduction architectures using outermost computation.

There are two forms of expression representation and manipulation found in reduction architectures, referred to as string reduction and graph reduction. In string reduction, as illustrated in Figure 5(i), an expression is represented as a string of symbols comprising operators, operands and references to other definitions, with structure represented by delimiter symbols (and). As the expression is evaluated the referenced definitions are copied into the string. In graph reduction, as illustrated in Figure 6(i), the components of an expression are always separate definitions, referenced by the expression. Referenced definitions are executed in place rather than being copied into the referencing expression.

Figure 5 shows some stages in the execution of a string reduction program. At (i) the evaluation of **a** is required, indicated by an activity at its outermost instruction, the multiplication. The evaluation of an instruction, such as the multiplication, demands the evaluation of its operands, indicated in (ii) by activities at their operators, and then suspends until their evaluation is complete. Where an operand is a reference to a separate definition, such as the references **b** and **c** in (ii), a copy of the definition is taken, replacing the reference, and its evaluation is then demanded (iii). Where an instruction has purely literal operands, as in the first subtraction in (iii), it is

executed and replaced by its result so that the result is stored directly as a literal into the instruction using it (iv). When all an instruction's operands have become literals it is re-enabled and executed as shown in (iv) - (vii).

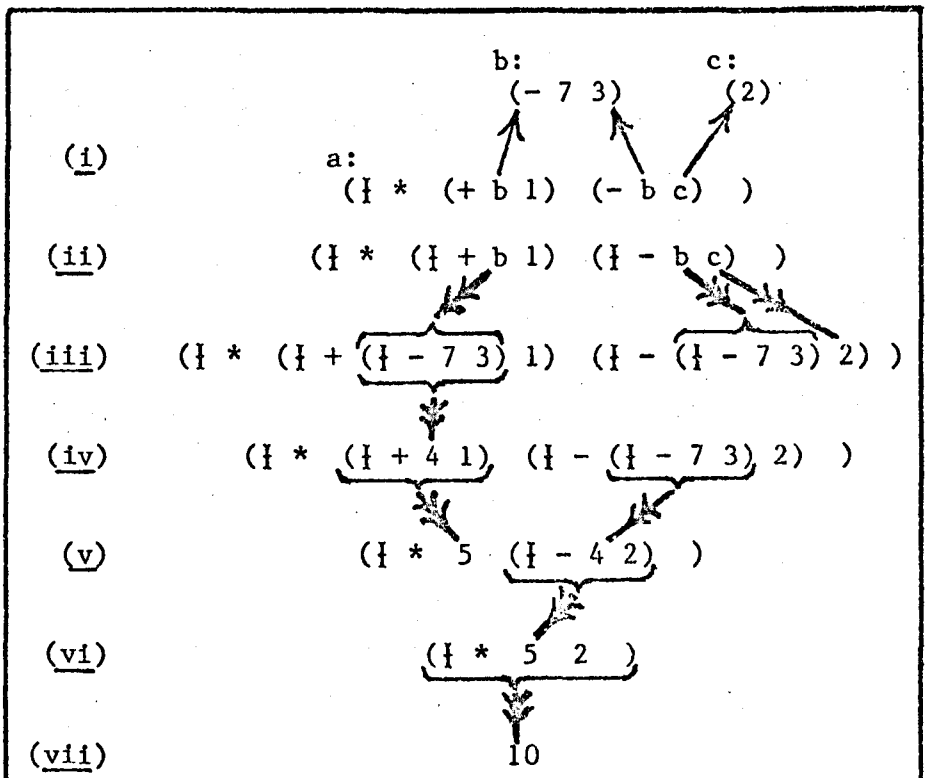


Figure 5 - String Reduction

Figure 6 shows some stages in the execution of a graph reduction program. At (i) the evaluation of **a** is required, indicated by an activity at its operator. The evaluation of an instruction demands the evaluation of instructions referenced by its input arguments, and then suspends. In (ii) such demands for evaluation have thus propagated throughout the program. Each activity at an instruction represents a demand for its result and the dotted arc represents the source of that demand which has to be re-enabled when the demanded instruction has been

evaluated. A demanded instruction with purely literal operands, such as **b** in (ii), is executed and replaced by its result so that the instructions, **t1** and **t2**, referencing it then reference the actual values for their operands (iii). Those instructions are re-enabled, their operands loaded, and the instructions replaced with the results, as shown in (iv). Any instructions which are no longer referenced, such as **b** and **c** in (iv) are deleted. In (v) the evaluation of **a** is complete, and all intermediate results in that evaluation have been deleted.

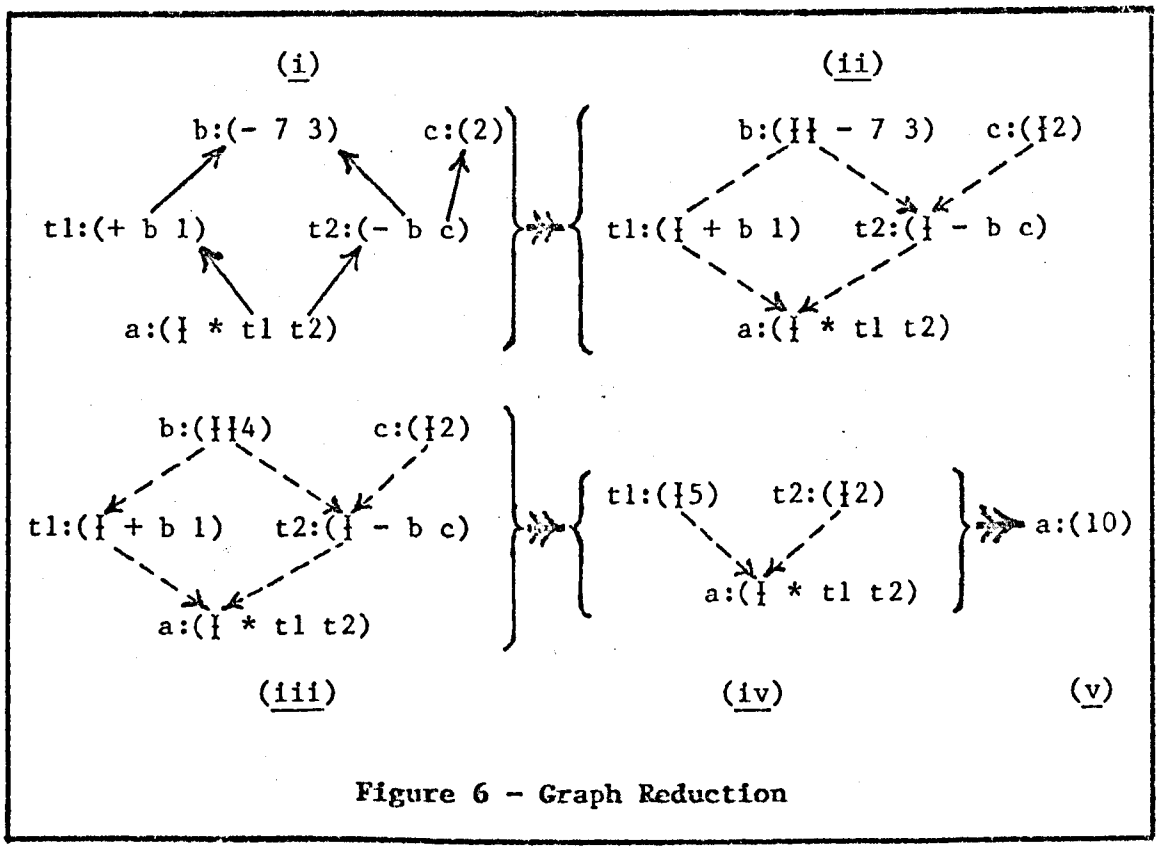


Figure 6 - Graph Reduction

These examples have illustrated concurrent evaluation of reduction programs, the concurrency being achieved by simultaneously demanding both operands of an operator. However reduction programs can just as well be executed without concurrency in which case the second operand is not demanded until the first has been evaluated.

2.1.4. Discussion

Each of the various operational models presented in this Section has particular advantages and disadvantages which are largely a result of the particular mechanisms used for organising the flows of data and control through a program[14]. A data mechanism defines how an argument can be accessed by a number of instructions and three such mechanisms can be identified:

1 by-literal (in all models) - an argument's value is known at compile time and is included in each accessing instruction

2 by-value (in data flow and string reduction) - an argument is evaluated at run-time when a separate copy of its value is stored in each accessing instruction

3 by-reference (in control flow and graph reduction) - an argument is evaluated at run-time and its value is shared by each accessing instruction having a reference to it

A control mechanism defines how one instruction causes the execution of other instructions, and again three mechanisms can be identified:

1 sequential (in conventional and multi-thread control flow) - a thread of control signals an instruction to execute and passes from one instruction to its implicit successor

2 parallel (in data flow and parallel control flow) - control signals the availability of (control or data) arguments with an instruction being executed when all its arguments are available

3 recursive (in reduction) - control signals the need for arguments with an instruction being executed when a result it generates is required by an invoking instruction

The relationship of these mechanisms to the operational models is summarised in Figure 7.

		Data Mechanisms	
		by-value (& -literal)	by-reference (& -literal)
Control Mechanisms	sequential		conventional control flow multi-thread control flow
	parallel	data flow	parallel control flow
	recursive	string reduction	graph reduction

Figure 7 - Control and Data Mechanisms

All models have the by-literal mechanism in some form since otherwise all constant data would have to be provided as run-time input. Each model also has one or other of the by-value and by-reference data mechanisms. The advantage of by-value compared with by-reference is that the data is directly available in the instruction using it and thus the extra step of loading data from a separate memory is avoided. The corresponding disadvantage is that there is a separate copy of a value (or its definition) for every instruction using it.

Each model has just one of the sequential, parallel and recursive control mechanisms. The sequential and parallel control mechanisms are in a sense opposites in that the former is best suited for programs that are mainly sequential whereas the latter is best suited for programs

that are highly concurrent. For sequential execution the "implicit successor" of the sequential control mechanism is more efficient than the explicit propagation of tokens required by the parallel control mechanism. Conversely, using separate instructions, such as **FORK** and **JOIN**, for organising concurrency is relatively inefficient when there is concurrency at the level of individual instructions.

The recursive control mechanism is not primarily concerned with concurrent execution. The major benefit of the recursive control mechanism is that instruction execution is only initiated when actually needed thus conserving machine resources and to some extent freeing the programmer from that consideration.

The particular combination of by-reference data mechanism and recursive control mechanism found in graph reduction allows it to support "Lazy Evaluation"[22,23] in which only the minimum necessary computation is actually performed. In both string and graph reduction the recursive control mechanism means that a computation is not performed unless its result is actually needed. In string reduction the by-value data mechanism means that a shared definition is evaluated more than once, as in the multiple evaluation of **(- 7 3)** in Figure 5 (iii)-(v). In graph reduction however the by-reference data mechanism allows such a shared definition to be evaluated only once, as in Figure 6(ii)-(iii). The operations discussed so far, such as addition, are "strict" which means that the result always depends on the value of all operands. For a program with only strict operations all instructions will always need to be executed and thus there is no real advantage to the recursive control mechanism and lazy evaluation. However these are at an advantage

in handling non-strict operations for which one or other of the operands will not be needed in a particular case. The most common example of non-strictness is in conditional instructions, to be discussed in the next Section.

An important difference between control flow, data flow and reduction is the extent to which an instruction interacts with other instructions during its execution. In the case of data flow, an instruction receives all its inputs before it starts executing, processes them completely and then outputs its results. This form of execution is referred to as "atomic" since to the external environment execution appears as a single indivisible operation. In control flow and reduction execution is non-atomic due to memory accesses in the former and acceptance of results demanded from other instructions in the latter.

The non-atomicity in control flow generally has undesirable consequences. For example consider a program which contains the three instructions - **i1: a:=1, i2: a:=2, i3: a:=a-a** - and has a program design constraint that the content of location **a** should always be non-negative. Each instruction individually preserves that property, but in executing them concurrently there is a possible ordering of the five separate accesses to **a** which would result in **a** having a final value of **-1** (the storing of **2** in **a** by **i2** occurring between the two loads from **a** by **i3**). Such undesirable consequences cannot however occur with reduction since changing a location's content only changes the representation of its fixed value, rather than changing its value. Furthermore, non-atomic execution can be beneficial in increasing concurrency and allowing the termination of programs that would not terminate with atomic execution.

To illustrate the potential benefits of non-atomic execution consider a logical **OR** operator in for example **C = A OR B.** If, say, the value **True** were received first for **B**, then the value **True** could be produced for **C** before the value for **A** had been calculated. This would result in additional concurrency between the evaluation of **A** and those parts of the program which depend on **C.** Also it might be possible for the program to produce its final result despite non-termination of the evaluation of **A.** The differences between atomic and non-atomic execution are important not only in individual instructions but also in groupings of instructions as functions and procedures, as will be discussed in the next Section.

2.2. Program Organisation

The term program organisation is here being used to cover the way programming requirements are supported by an architecture's machine code. This Section discusses how control flow, data flow and reduction architectures support data structures, conditionals, procedures, iteration, and non-determinacy.

2.2.1. Data Structures

Most general-purpose architectures allow the representation and manipulation of data structures. In control flow this is generally achieved by allowing explicit program control over storage allocation and the ability to apply ordinary arithmetic operators to addresses. Components of a structure can be stored in contiguous memory cells with the whole structure represented by a reference to the first component and any other component accessed by for example adding a displacement to that reference. In data flow and reduction architectures however storage allocation is not under explicit program control and special facilities are provided to support data structures.

In a data flow architecture supporting data structures (not all do so) there are special operators for structure manipulation, such as the **append** and **select** operators of Id[24] which respectively extend a data structure with a new component and select a specific component from within a data structure. The result of an **append** operator is a slightly modified copy of the input data structure and this result is communicated as a single (large) data token to those instructions (**selects** or

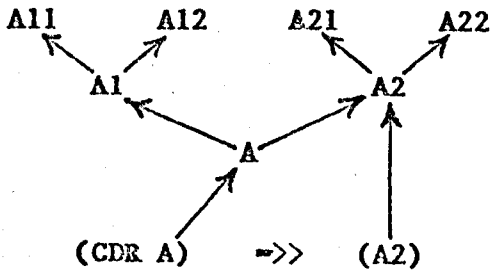
further appends) which use it.

A reduction architecture typically provides the **CONS**, **CAR** and **CDR** operators of LISP[25] which respectively form two (simple or structured) values into a two-component structure and select one or other of the components from such a structure. In string and graph reduction architectures data structures are represented in different ways, corresponding to the different ways expression structures are represented. In string reduction a data structure is represented as a string of data items with structure being represented by delimiters, as in for example **((A11, A12), (A21, A22))**. An operator such as **CDR** ("second component") operates on whole data structures -

... CDR ((A11, A12), (A21, A22)) ...

=>> ... (A21, A22) ...

In graph reduction a data structure is represented using references to its components and an operator such as **CDR** manipulates references -



The by-reference data mechanism of control flow and graph reduction means that a data structure can be efficiently communicated and manipulated using references whereas the by-value data mechanism of data flow and string reduction means that, at least in principle, the whole data structure has to be copied and manipulated. In practice some data flow

architectures use control flow concepts to support data structures with structures being stored in separate memory and references being passed as tokens[26].

A particular benefit of graph reduction is that (in conjunction with lazy evaluation) it can support unbounded data structures. These are data structures for which there is no explicit specification of actual or maximum size. The data structure is defined in terms of a notionally infinite number of components and is incrementally generated as components are actually needed in the computation. Consider for example a data structure **Primes**, defined as a (probably infinite) list of the prime numbers -

```
Primes ≡ P (2)
        where P (x) ≡ CONS (x, P (Next (x)))
                                  where Next (p) ≡ (...)
```

This definition uses a recursive function P (recursive functions will be discussed in more detail later). This function has a prime as its parameter, **x**, and notionally returns a list of all primes including and following that parameter. This is achieved by **CONS**tructing a list with **x** as the first component and the remainder of the components being the sub-list returned by the recursive call **P (Next (x))**. In this call of P the **Next (x)** returns, as the call's parameter, the next prime after **x**. The list is actually incrementally generated as its components are accessed, as shown in Figure 8. In (b) is shown two stages in the partial generation of the **Primes** structure, namely that before (b1) and after (b2) the evaluation of an accessing expression shown in (a).

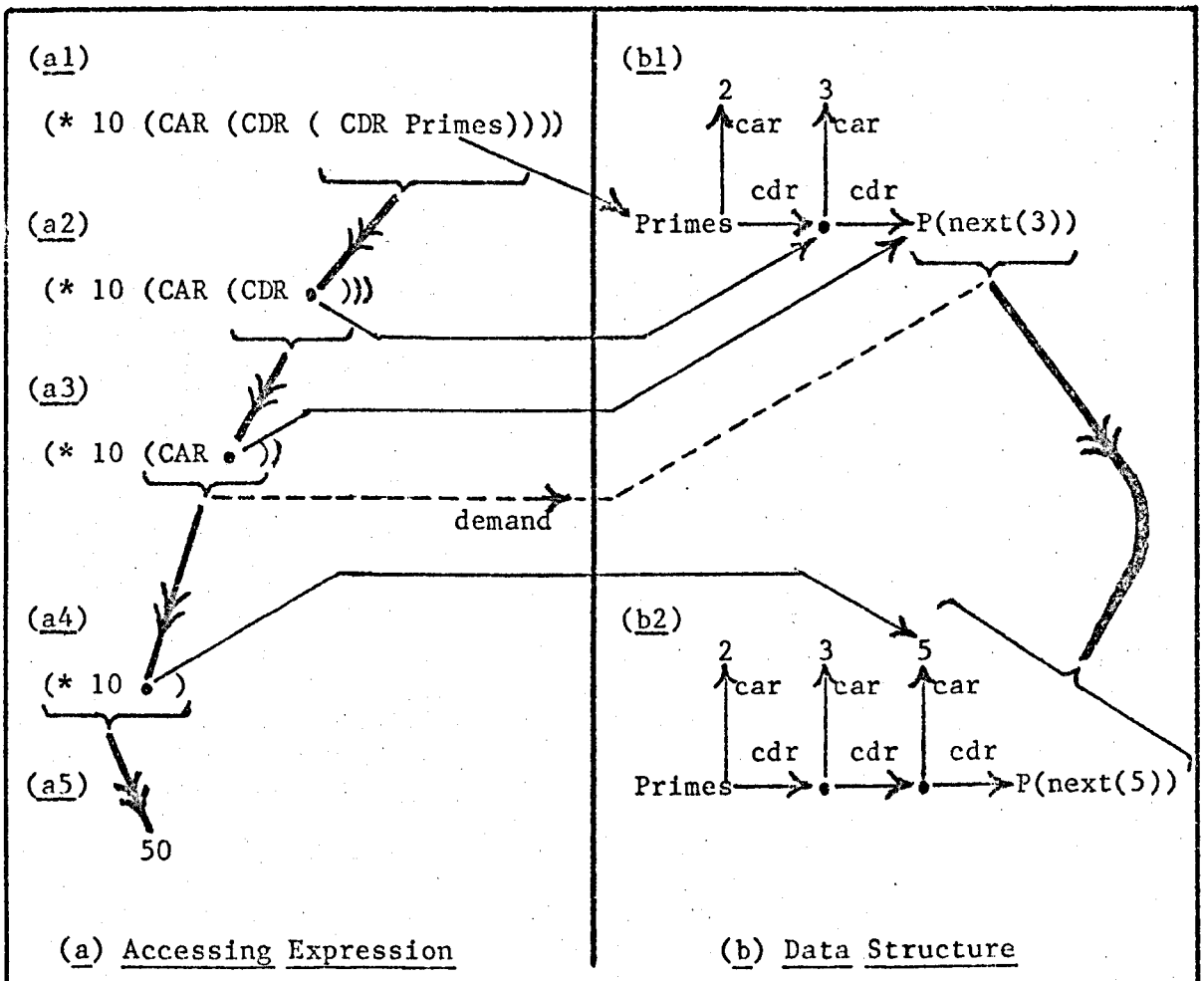


Figure 8 - Unbounded Data Structures

The expression in (a1) accesses a particular prime by selecting the appropriate node in the **Primes** structure using **CDR** and **CAR** operators. The initial stages of evaluation ((a1) - (a3)) select existing nodes of the structure in (b1) and so do not require any further generation of that structure. From stage (a3), further evaluation of the expression requires the actual value of **CAR(P(Next(3)))**. Thus the evaluation of the expression **P (Next(3))** is demanded in order for the first component **(CAR)** of its data structure result to be accessed. As shown in (b2) the execution of that data structure expression will cause it to be

replaced by a node for which the **car** arc points to **5** and the **cdr** arc points to the next recursive call. The required operand, 5, is now available and is used in the original expression ((a4) - (a5)) without requiring further generation of primes. The same technique could be used in string reduction, but the lack of lazy evaluation would mean that every time a particular prime was needed, all the primes up to that point would have to be re-generated.

2.2.2. Conditionals

Every general-purpose architecture provides some form of conditional instruction to select between alternatives. In a control flow architecture there will be a conditional branch instruction which has two alternative successor instructions. When executed it selects one or other of the alternatives thus, so to speak, switching an incoming flow of control onto one of two output paths.

In data flow architectures there are two possible forms of conditionals - one with conditional selection of values to be used and the other with conditional selection of instructions to be executed. Both are shown in in Figure 9 using as an example **z = - (if c then a else b)** where **a, b** and **c** are computed from **i** by program fragments **A, B** and **C.** The first form (value selection) uses an instruction with a conditional operator, referred to as a "switch-in" instruction. This is illustrated in Figure 9(i) with the switch-in represented as a hexagonal node. Like all pure data flow instructions, the switch-in executes when all its inputs (**a, b** and **c**) are available. It then outputs for use by **Z** either the **a** or the **b** depending on whether **c** is True or False.

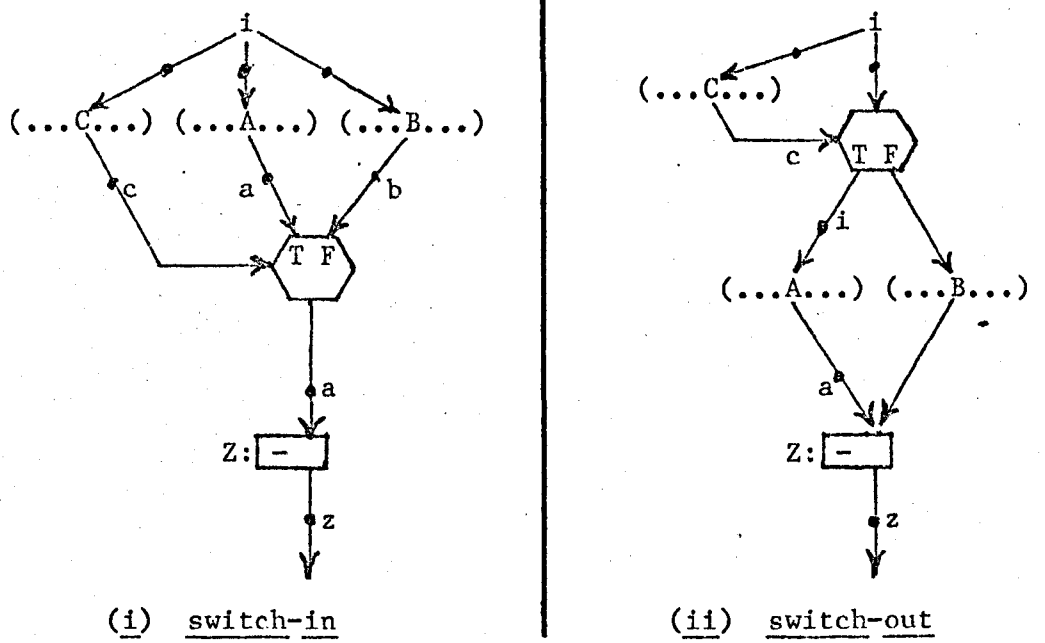


Figure 9 - Two Forms of Conditional in Data Flow

The second (instruction selection) form of data flow conditional, illustrated in Figure 9(ii), uses a "switch-out" instruction which controls the flow of data through the program. Here the hexagonal node is a switch-out instruction which switches the **i** input onto one or other of its output arcs, depending on the value of the **c** input. The instruction causes either **A** or **B** to be activated and thus determines whether **Z** receives **a** or **b**. In (ii), taking **c** as True, **B** is not executed even though its only input, **i**, is available. This departure from pure data availability is to avoid unnecessary computation and is essential if recursion is to be supported. (If this example were part of a recursion, with **c** the termination condition and **B** the recursive call, then the pure data availability of (i) would result in non-termination which is avoided in (ii).)

In reduction there is a single conditional construct, namely a conditional operator which selects between alternative values and also, due to the recursive control mechanism, selects between alternative instructions. A conditional operator, **if**, will be part of a conditional instruction such as **(if C then A else B)**. This is executed by first demanding the evaluation of **C** and then, depending on the result **c**, demanding the evaluation of either **A** or **B** and replacing the whole expression with that result, either **a** or **b**.

An important consideration in the organisation of conditionals is the extent to which unnecessary computation may be performed. This can be illustrated by considering the following two alternative conditional structures for computing a value **r** -

```
(a)  a:= ( ... A ... ); b:= ( ... B ... );
     r:= (if c1 then a else b) * (if c2 then a else d);
```

and -

```
(b)  r:= (if c1 then ( ... A ... ) else ( ... B ... )) * (if c2
                                                  then ( ... A ... ) else d);
```

In both control flow and data flow either of these structures is possible. (In data flow, (a) is conditional value selection as in Figure 9(i) and (b) is conditional instruction selection as in Figure 9(ii).) If **c1** and **c2** are both True then, using structure (a), **B** would be unnecessarily evaluated, whereas using structure (b) **A** would be executed twice. Thus for either choice of structure there is the possibility of unnecessary computation being performed. In string reduction the copying of definitions means that **B** might be executed twice. Graph reduction, using lazy evaluation, is the only model for which unnecessary computation can always be avoided for this type of example.

2.2.3. Procedures and Iteration

An essential requirement for any general-purpose computer is to allow one program fragment (e.g. procedure) to be executed many times using different sets of data values (e.g. parameters) each time. The term "environment" will be used for the data values and other information which are different for different executions of the same program fragment. There are two ways of obtaining such multiple execution in conventional control flow, namely procedure calls and iterations. (The term procedure will be used to cover both procedures with side effects and functions which return their results.) With (recursive) procedure calls several nested environments for the same procedure will coexist. Consequently there must be mechanisms for keeping the environments separate and allowing a single reference in the shared procedure code to identify data within different environments. This is achieved by using a stack to store environment information (parameters, local variables and return links) and indirect addressing via a register which references the current environment within the stack. In iteration each execution of the repeated code is completely finished before the next execution starts. Consequently the successive environments can share storage and there is no need for any special storage and addressing techniques.

Data flow architectures can also support both procedures and iteration (although not all do so). With procedures (even without recursion) there may be several concurrent executions of the same procedure. The environment for each execution consists of data tokens for the parameters, internal partial results (corresponding to local variables in

control flow) and the return links. Whereas in control flow the by-reference data mechanism allows data for each environment to be stored in a separate area of memory accessed via a reference, this is not possible with the by-value data mechanism. Instead a scheme of labelling tokens is used to separate environments. Each active environment has a distinct identifier which is used to label all tokens associated with that environment. An individual instruction in the procedure may execute many times in different environments. Each execution takes a set of input tokens with the same environment identifier and uses that identifier to label its output tokens.

Figure 10(i) shows the operation of this scheme using as an example the procedure **F** in the following definition for factorial (which is somewhat unusual as it will also be used to illustrate iteration) -

Factorial (n) ≡ F (n,1)

F (n, m) ≡ if n=1 then m else F(n-1, n*m)

The inputs to the procedure graph are the parameters **n** and **m** and the return link **r**. These inputs are tokens labelled with an environment identifier (the Q in e.g. Q:n). The B represents the graph to compute the partial results **n'** and **m'** as parameters for the next call. (If the termination condition **n=1** holds then B will instead pass **m** as the final result **f** to the **RETURN** instruction - the actual construction of B is shown in (ii).) The tokens output from B are labelled with the same environment identifier, Q, as the input tokens from which they are produced. The **CALL** instruction allocates an environment identifier R to label a new set of input tokens for the nested procedure call and sends those tokens back to the top of the graph. Eventually the nested call

produces a result token **Q:res** which becomes the output **Q:f** of the **CALL** instruction. The **RETURN** instruction requires two inputs, **f** which is the result value and **r** which indicates the **CALL** instruction to which the result token is sent and the environment identifier with which it is labelled. There are actually three tokens shown which are available to this instruction, the **Q:f** and **Q:r** for this call and the **P:r′** from some other call (such as an outer level of this recursion). The effect of the labelling scheme is to enable the correct pairing of the tokens available to a particular instruction - in this case **Q:f** being paired with the **Q:r** rather than with the **P:r′** which would have arrived earlier.

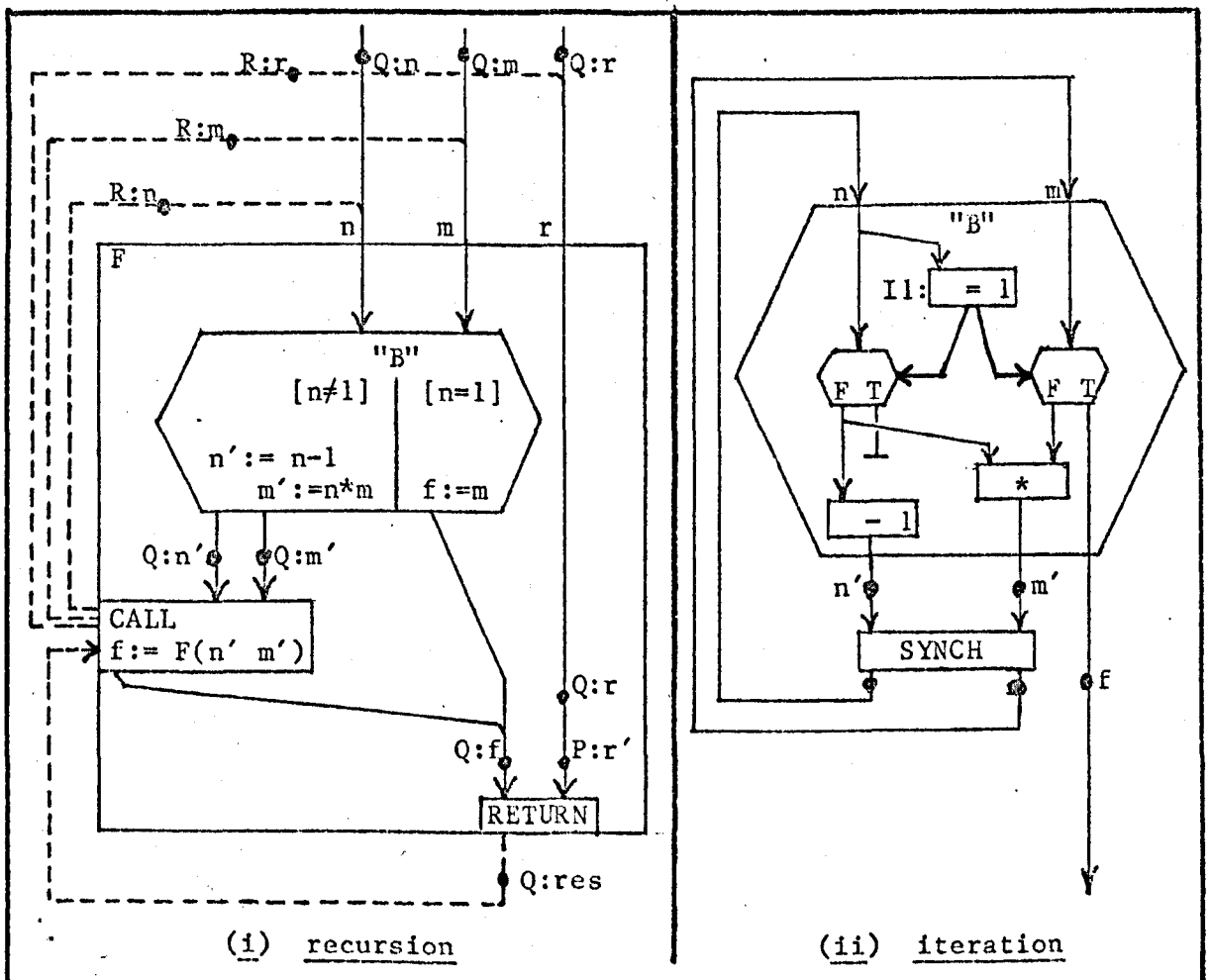


Figure 10 - Procedures and Iteration in Data Flow

Like an individual instruction, a data flow procedure is atomic. All its parameters must be produced before it starts executing and if it has more than one result all of them must be produced before any can be used. This can restrict the potential concurrency in a data flow graph. In this example the evaluation of **n'** does not depend on the value of the second parameter, **m**, and so in principle the procedure could start executing with just its first parameter, **n**, evaluating **n'** concurrently with the evaluation of the second parameter, **m**, by the caller.

There are two general approaches to supporting iteration in data flow. The simpler is that illustrated in Figure 10(ii) for the Factorial example. The body **B** of the iteration is the same as in the equivalent procedure. **I1** is a comparison instruction, testing for the termination condition and controlling two switches. If **I1** gives False then the switches send **n** and **m** to the subtraction and multiplication instructions which calculate **n'** and **m'** as the next values for the iteration. Alternatively if **I1** gives True then the switches send **m** out as the final value **f** and discard **n** (indicated by a $\perp$). Each iteration completely terminates before the next one starts, this being achieved by a synchroniser instruction (**SYNCH**) which requires both **n'** and **m'** as inputs but performs no computation on them, just passing them back for the next iteration. If the synchronisation were omitted then the subtraction instruction might generate successive values of **n** faster than the multiplication instruction could consume them and without additional mechanisms there is no way to pair the **m** with the correct **n**.

In the alternative scheme for supporting iteration the synchronisation is omitted and there is some additional mechanism to order the

tokens on an arc. In Irvine Dataflow[24] this ordering is achieved by labelling every token with a sequence number (similar to the environment identifier used for procedures). This is incremented at the end of each iteration. In the DDM1 data flow architecture[27] the ordering is obtained by associating a first-in-first-out queue or "stream" of tokens with each input of an instruction.

Whereas in control flow and data flow the basic operational models have to be extended to accommodate procedures this is not the case for reduction, which is based on function applications (corresponding to procedure calls). Figure 11 shows a string reduction sequence for calculating the factorial of **3** using the same definition for **F** as above.

```
                    F(n,m) ≡ if n=1 then m else F(n-1,n*m)
(i)   ... F(3,1) ...
(ii)  ... (if 3=1 then 1 else F(3-1,3*1) ) ...
(iii) ... F(3-1,3*1) ...
(iv)  ... (if (3-1)=1 then 3*1 else F((3-1)-1,(3-1)*(3*1)) ) ...
(v)   ... F((3-1)-1,(3-1)*(3*1)) ...
(vi)  ... (if ((3-1)-1)=1 then (3-1)*(3*1)
                 else F( ((3-1)-1)-1 , ((3-1)-1)*((3-1)*(3*1)) ) ...
(vii) ... ((3-1)*(3*1)) ...
(viii)... 2*3 ...
(ix)  ... 6 ...
```

Figure 11 - Procedures in String Reduction

The initial call in (i) is the instruction **F(3,1)** which is as usual executed by replacing the reference **F** by a copy of the referenced

definition, and in doing so formal parameters are replaced by actual parameters giving (ii). The conditional is evaluated and replaced by the selected alternative (iii) which is another call with different parameters giving (iv). This process continues until the final result is produced in (ix).

Any iteration can easily be transformed into an equivalent recursive procedure. Thus it is not necessary to support a separate iteration mechanism in addition to that for recursion, and reduction does not do so. In control flow and data flow, iteration is where possible used in preference to an equivalent recursion in order to avoid passing the final result back up through all the levels of recursion (compare Figure 10(i) and 10(ii)). However with the reduction model this advantage is automatically obtained in the execution of those procedures (known as "tail recursive") which are directly equivalent to iteration. This is illustrated in Figure 11 where the result (ix) calculated when the recursion terminates (vii) has been returned directly to that part of the program from which the original call was made in (i).

An important aspect of the reduction model is its ability to support "higher order functions" (procedures), that is procedures which have procedures rather than data values as parameters and/or results. This is illustrated in Figure 12 for a higher order procedure **Bind1(g)** which has procedures as its parameter and result. In (v) there is a call of **Bind1** for which the parameter is a two-parameter procedure **G** (just the addition operator), and the result is the one-parameter procedure **H**. In the definition of **Bind1** (iv) **H** is obtained by binding the first parameter of **G** to a particular value (**F1**), leaving **G**'s second

parameter free as H's only parameter. When procedure H is called in (vi) with parameter P2 the effect (viii) is as though procedure G had been called with parameters P1 and P2.

```
(i)          P1 ≡ 10

(ii)         P2 ≡ 20

(iii) G(p1,p2) ≡ (+)

(iv)  Bind1(g) ≡ (g (P1))

(v)      H(p2) ≡ Bind1(G)


(vi)    (H (P2) )
(vii)           ->> ((Bind1(G)) (P2))
(viii)                          ->> ((G (P1)) (P2))

(ix)            ->> (((+) (P1)) (P2))
(x)                             ->> ( +10 (P2))

(xi)    ->> 30
```

Figure 12 - Higher Order Procedures in Reduction

The "trick" to this technique is that all procedures and operators have at most one argument. Thus the expression **(P1) + (P2)** must be represented, as shown in (ix), as **(((+) (P1)) (P2))**. The operator + has one argument, P1 with value 10, and returns as its result (x) the operator +10. This operator has one argument, P2 with value 20 and adds 10 to that argument to produce the result 30 (xi).

The significance of higher order procedures lies in their ability to encapsulate a general pattern of program execution in a procedure (function) definition which need only be programmed once. This is illustrated in the example below where a procedure Incgen is used to

encapsulate the general pattern of incremental data structure generation used in Figure 8.

```
Incgen ( generator, initialvalue ) ≡
  CONS ( initialvalue,
         Incgen ( generator, generator ( initialvalue )))
P ( init ) ≡ Incgen ( Next, init )
Next ( val ) ≡ (....)
Primes ≡ P ( 2 )
```

2.2.4. Non-Determinacy

Any concurrent program is non-deterministic at least in the (operational) sense that the exact order of events in its execution is not pre-determined. I will use the term "non-determinate" in the more limited (functional) sense of the final output of a program (or part of a program) being not uniquely defined by just the program and the inputs it is processing. Although non-determinacy is usually an undesirable characteristic of a program there are situations in which the use of a non-determinate program is necessary. The most common example is in the management of access to shared resources, such as files, within a multi-processing operating system. To ensure the integrity of the shared resource the non-determinacy must be controlled, typically by grouping a number of related individual accesses by one process into one "transaction" for which the process has exclusive use of the resource.

To support non-determinate programming some primitive mechanisms for allowing and controlling non-determinacy must be provided. In concurrent control flow non-determinacy is implicit in the basic opera-

tional model where two concurrent instruction executions can simultaneously update the same memory cell with different values. There is usually some additional explicit mechanism, such as a "test and set" instruction, for controlling that non-determinacy.

In the data flow model there is a similar potential for implicit non-determinacy in that two instruction executions could emit tokens with different values for the same input of another instruction, with the result of the program depending on which arrived first. However the data flow code generated by a compiler is always constructed so as to avoid such implicit non-determinacy. For example in Figure 10(ii) the synchroniser instruction is included to prevent non-determinacy in the execution of the multiply instruction; in Figure 9(ii) the switch-out instruction ensures that only one of **A** and **B** sends a token to the single input of **Z**, even though both could do so.

There is only one data flow architecture[24] which provides explicit non-determinacy. This architecture uses the technique of a "resource manager" to control non-determinate access to a sharable resource, as shown in Figure 13. The resource manager **RM** is a form of procedure which has two inputs, the resource **R** (e.g. a file) and a complete transaction (**p**, **q**, **r** or **s**) to be processed. The result of one call is a modified version of the resource, **R'**, which is fed back as input to the next call. The arcs carry sequences or "streams" of tokens and each token is labelled with a "stream number", **1**, **2**, **3** etc. These numbers serve the same role as the environment identifiers in the procedure call example, ensuring that the resource manager itself is determinate. The non-determinacy and its control are combined in the expli-

cit MERGE operator which merges its two input streams into one output stream. On each execution it takes a token from either input, ignoring the input stream numbers, and outputs it labelled with the next output stream number.

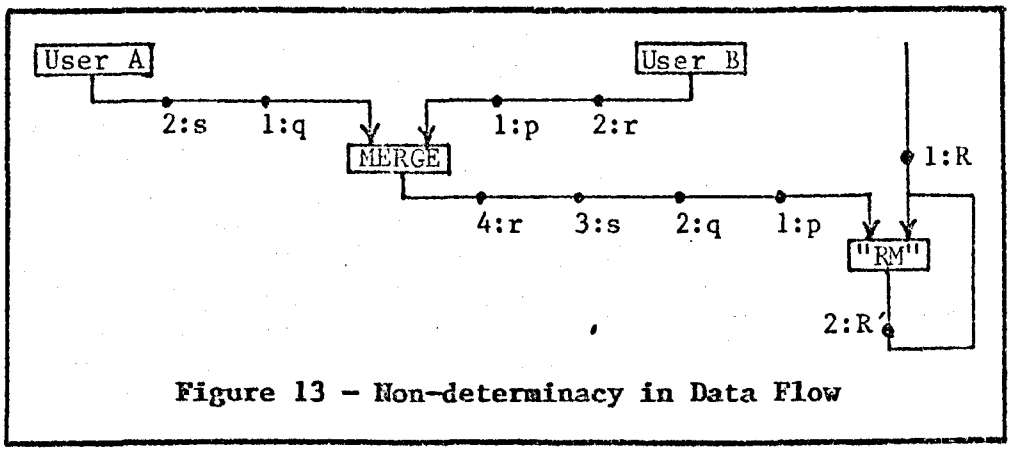


Figure 13 - Non-determinacy in Data Flow

In reduction the only form of update is to replace an expression with the result of evaluating it. This cannot lead to non-determinacy, as can the less constrained form of update found in control flow. If non-determinacy is required it must be introduced with some special operator. Non-determinate operators that have been proposed are the "AMBiguity operator"[28], and more recently FRONS[29]. An example of an expression using AMB is

$$\mathbf{r \equiv ((A)\ AMB\ (B))}$$

The result of the whole expression is the result of evaluating one or other of its operands, **(A)** or **(B)**. Operationally, demanding the value of **r** causes both **(A)** and **(B)** to be demanded and computed concurrently. When one computation, say **(A)**, has terminated its result **a** replaces the whole definition. The other computation is then no longer needed and thus forcibly terminated. AMB can be used to program a merge procedure

which then forms the basis for resource management in the same style as in Figure 13. Although necessary for resource management, the inclusion of non-determinacy destroys some of the desirable properties of determinate reduction programs, such as referential transparency. For example the definition **x ≡ r-r** would give **x** the value zero for any value of **r**, but replacing the reference **r** by the defining expression above gives

x ≡ ((A) AMB (B)) - ((A) AMB (B))

In this expression the two distinct **AMB** operators may give different results and thus a non-zero **x**.

There are uses for non-determinacy other than in the resource manager type of application. An example is when there are two algorithms **(A)** and **(B)** for computing a value **r** and one or other may be very inefficient or fail to terminate for particular cases which are difficult or impossible to predict. The expression **r ≡ ((A) AMB (B))** will always terminate with a result for **r** if either algorithm terminates. Furthermore (assuming machine resources are divided fairly between **(A)** and **(B)**) its execution time in a particular case will be proportional to that for the more efficient of the two algorithms for that particular case. In this example concurrency in evaluation of **r**'s operands is essential to the logic of the program. (The use of concurrency in this way is explored at some length in [30].) However for other examples of potential concurrency, such as in **r ≡ ((A) * (B))**, the logic of the program is not dependent on there being actual concurrent execution. For reduction architectures which do not support operators such as **AMB** the exploitation of potential concurrency is entirely an implementation

issue. As discussed in the next Section, some reduction implementations do exploit concurrency and some do not.

2.3. Machine Organisation and Implementations

The term machine organisation is being used to cover the way a machine's resources are configured and allocated. This Section starts by identifying the different classes of machine organisations used for control flow, data flow and reduction, and then outlines some particular implementations.

2.3.1. Classification

Three classes of machine organisation can be identified which are referred to here as the centralised, packet communication, and expression manipulation organisations. These organisations, illustrated in Figure 14, are principally distinguished by the organisation of communication(**C**) between processor(**P**) and memory(**M**) resources.

Centralised - This organisation consists of one or several processors and one or several memory units with a central communication system providing each processor with direct access to all memory. In this organisation a program (or part of a program) being executed by one processor has one active instruction, the execution of which typically requires several communications with the global memory.

Packet Communication - This organisation consists of a circular instruction execution pipeline of separate resources in which processors and memories are interspersed with "pools of work" through which they communicate. Parallelism can be obtained by having a number of identical resources between successive pools as shown; or by replicating and connecting the circular pipelines. In this organisation an executing

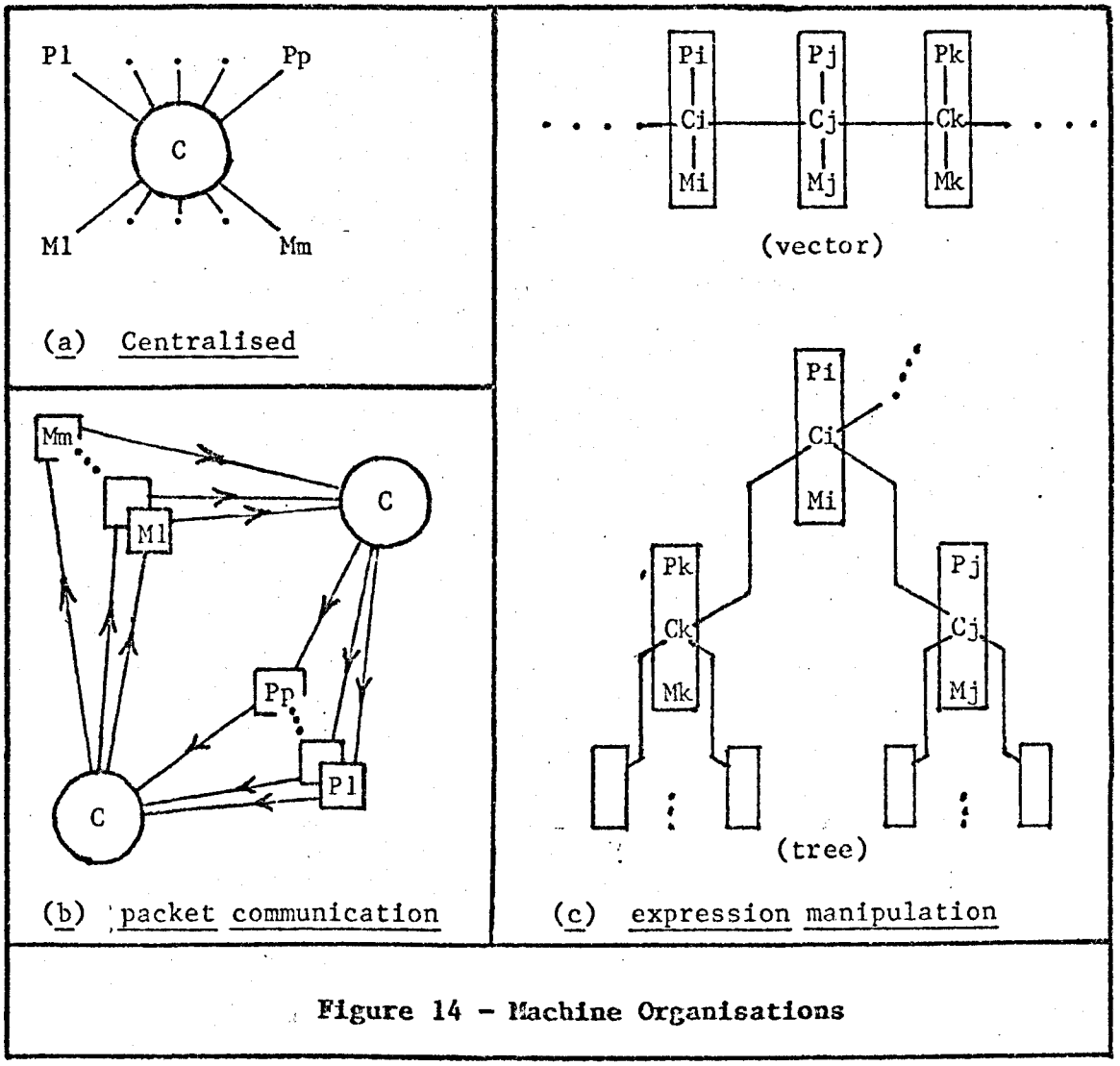


Figure 14 - Machine Organisations

program consists of a large number of independent self-contained items or work packets (e.g. individual instructions and tokens) which split and merge and move into different work pools for different stages of processing. The operation cycle for a resource is to take a packet from an input pool, process it in isolation from other resources, and produce a modified packet in an output pool.

Expression Manipulation - This organisation consists of a large number of resources, referred to as "computing elements", each being a complete microcomputer with local processor and memory and a capability for

communicating with a small number of other computing elements. The communications capabilities are used to connect the computing elements into some regular structure such as a vector or tree. In this organisation an executing program consists of one large program structure ("expression") with logically related parts of the program being allocated to physically related computing elements. Some components of this structure are active whilst some are inactive. Each computing element examines its part of the overall program structure looking for active items to execute. Executing an item will involve communications with the other items that it references. Programs generally exhibit considerable "locality of reference" which means that references tend to be between items closely related in the logical structure. Consequently most communications are internal to one computing element or between close neighbours.

A major motivation for the novel packet communication and expression manipulation organisations is to alleviate the communication problems that arise in a centralised organisation as the number of processors increases. In a centralised organisation the performance of a processor is dependent on the transit time of a memory access through the global communication system (including in "transit time" any queueing for access to the communications system). In the packet communication organisation however (provided there is sufficient concurrency) performance is not dependent on the transit time of a packet through the communication system but only on the total communication bandwidth being sufficient to match the total throughput of the processors. Whereas the transit delay must eventually increase as system size and load increases, there are communication systems organisations[31] where the

bandwidth per processor remains substantially constant as the system is extended to accommodate more processors and memories. In the expression manipulation organisation locality of reference is relied on to overcome the global communication problem.

Modern computer designs often incorporate adaptations to their basically centralised (single-processor) machine organisation which to an extent overcome the similar processor-memory communication problems that arise from increasing processor power. These adaptations are the pipelining of instruction execution, producing a similar effect to that of a packet communication organisation, and the introduction of memory caches, to exploit locality as in an expression manipulation organisation. The approach of pipelining conventional architectures (introducing some parallelism) is restricted however by the limited number of pipeline stages that can be introduced and the need to support a strictly sequential operational model. A multi-processor machine organisation incorporating a local cache with each processor would in effect be a form of expression manipulation organisation.

Most of the machine organisations described above are at least being investigated for the implementation of each different class of architecture, as shown in Figure 15. The rest of this Section discusses some of these proposed implementations.

	concurrent control flow	data flow	reduction string	reduction graph
Centralised	CM*		GMD	SKIM
Packet Circulation	GCF	MIT, MAN		Alice
Expression Manip.	RIMMS	DDM	SRM	AMPS

CM* - Modular Multiprocessor[32]
GCF - Generalised Control Flow[33]
RIMMS - Reduced Instruction Set Multiprocessor System[34]
MIT - MIT Data flow computer[35]
MAN - Manchester Dataflow Computer[36]
DDM - Utah Data Driven Machine[27]
GMD - GMD Reduction Machine[37]
SRM - Newcastle String Reduction Machine[38]
SKIM - Cambridge S-K-I Reduction Machine[39]
Alice - The Alice Reduction Machine[40]
AMPS - Applicative Multi-processing System[41]

Figure 15 - Some Novel Architectures

2.3.2. Control Flow

For control flow implementations the main issue is the way in which global memory access is provided. In the centralised and expression manipulation organisations an instruction being executed will send load and store messages to the appropriate memory units and wait for the replies before continuing. A more novel approach is that taken by a packet circulation organisation such as the Generalised Control Flow architecture where the instruction goes to the data rather than vice-versa. In this architecture there is a single pool of work accessible to all resources which takes the form of a slotted communication ring circulating each slot past every resource as shown in Figure 16. A slot may contain a work packet, such as an instruction requiring an operand

to be loaded from memory. A resource that can service a work packet, such as the memory unit containing an instruction's operand, removes it from the ring as it passes, leaving that slot empty. When the service has been performed the modified packet, such as an instruction with is loaded input operand, is placed back on the ring in a passing empty slot. The modified packet is then picked up by another resource, such as a processor to compute its result.

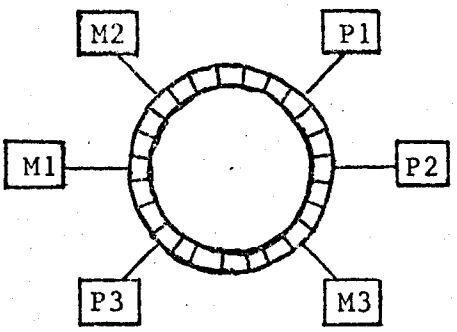


Figure 16 - Packet Circulation for Multi-thread Control Flow

2.3.3. Data Flow

A data flow implementation requires a scheme for handling the activation of concurrent instructions by data tokens. There are two such schemes[42] namely "token storage" in which a token is stored directly into an instruction and "token matching" in which tokens are kept separately until a complete set of tokens for one instruction execution have been matched together.

Examples of the token storage scheme include the MIT computer whose packet circulation organisation is shown in Figure 17(a). In this

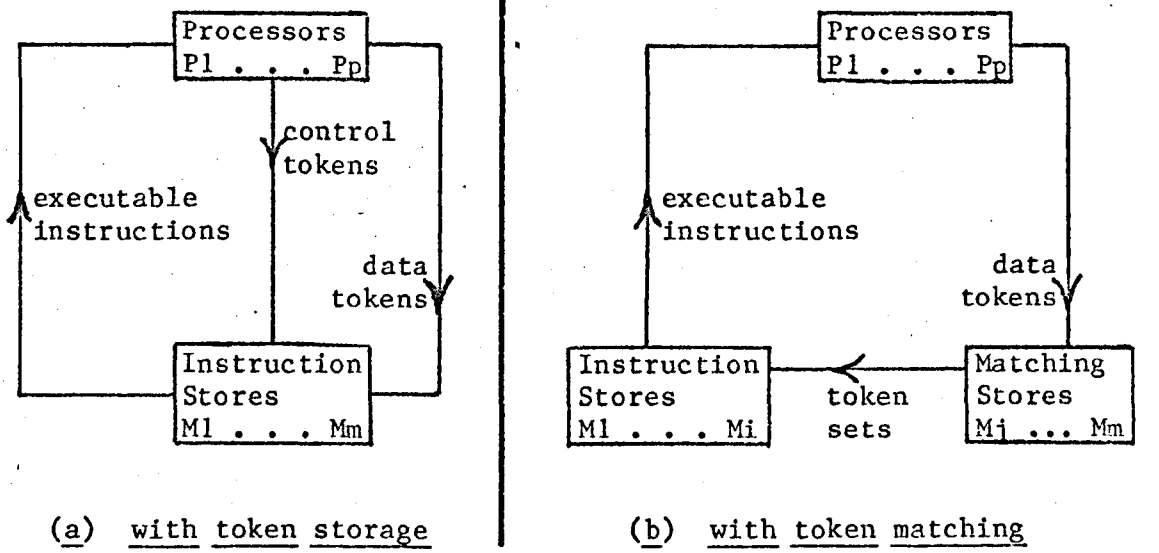


Figure 17 - Packet Circulation for Data Flow

organisation each path between the processors and the instruction store memories is a routing network which delivers packets to their destinations and also acts as a work pool temporarily storing packets. A data token packet emitted by a processor arrives at an instruction store and its value is stored directly in the appropriate argument position of its destination instruction. If it is the last token for that instruction a copy of the instruction with those argument values is sent as an executable instruction packet to the processors. This token storage scheme does not support the simultaneous existence of tokens for separate instances of an instruction since one could overwrite the other in the instruction store. Thus recursion is not supported and iteration requires some synchronisation scheme such as is shown in Figure 10(ii). In the MIT data flow model the synchronisation required for iteration is achieved by a "control token" being always implicitly sent from the receiver of a data token to its producing instruction indicating when it is safe for another instance of the token to be produced.

Examples of data flow computers with token matching include the Manchester machine whose packet circulation organisation is shown in Figure 17(b). Here there is a matching store, separate from the instruction store, which collects together into a token set those tokens for a particular execution of an instruction. When a complete set has been collected it is sent as a token set packet to an instruction store where it is combined with a copy of the instruction and sent to the processors for execution. For the token matching scheme the environment identifiers of Figures 10(i) can be used for (recursive) procedures. There may simultaneously be several partially complete token sets for the same destination instruction which are distinguished in the matching store by their different environment identifiers.

2.3.4. Reduction

Whereas supporting concurrency is the major motivation for the novel control flow and data flow architectures, that is not the case for reduction architectures. In three of the implementations included in Figure 15 (Alice, SRM and AMPS) there is concurrent instruction execution, and in two (SKIM and GMD) there is not. In either case the principal issues are organising the recursive control structure and the storage management needed for a program which expands and contracts as it is executed. In for example the centralised organisation of the GML string reduction machine there is a single point of execution in the program and stacks are used to deal with both issues. The whole program is a parenthesised expression which is initially stored on one stack. The processor repeatedly traverses this expression by copying it from one stack to another replacing any executable expression by its result

in the process. For simple expressions, such as (* (+ 4 1) (- 4 2)), the effect is that the processor traverses to the right until a) is encountered; then to the left until an operator is encountered which is executed; then to the right etc.

The Newcastle string reduction machine works in a similar way but with many processors and many connected stacks. This has an expression manipulation organisation with a vector of processors as shown in Figure 18. The expression being executed is stored in a sequence of double-ended queue (DEQ) memories. Each processor has access to one end of each of two DEQs which to a processor appear as stacks, allowing it to traverse part of the program expression.

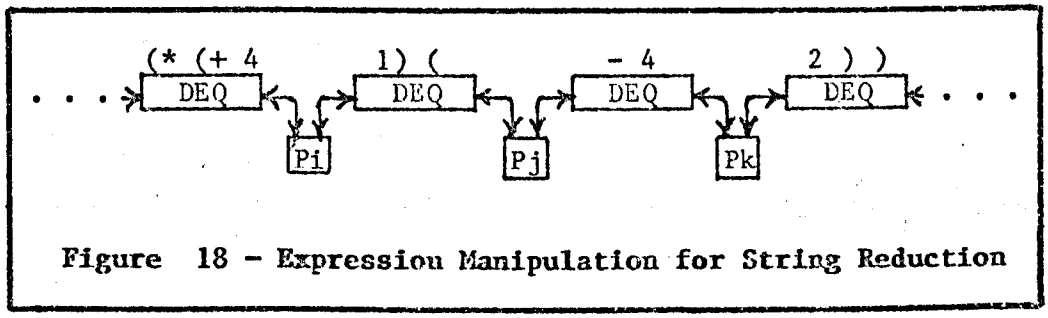


Figure 18 - Expression Manipulation for String Reduction

In graph reduction pointers are used to allow sharing of results in an executing program and storage management requires periodic garbage collection of unreferenced parts of the program. To organise the recursive control structure there must be a return pointer from a demanded instruction to each instruction demanding it, as was shown in Figure 6. If the operands of an instruction are evaluated in sequence then there will be a simple chain of these return pointers which can be stored on a separate stack as in conventional procedure calling. This is the scheme used in the SKIM reduction machine (which is based on the interesting

concept of "combinators as machine code"[43]). If however the operands of an instruction are evaluated concurrently then the return pointers will form a more general graph structure which can be implemented using the "pointer reversal" scheme shown in Figure 19. Here propagating demand from **i1** to **i2** and **i3** adds to those instructions return pointers and replaces the original pointers with unknowns. This part of the program now has the form of a data flow graph and can be executed in the same way. This is the scheme used for AMPS in which "demand tokens" carrying return pointers flow up the program graph and data tokens carrying computed values flow back down.

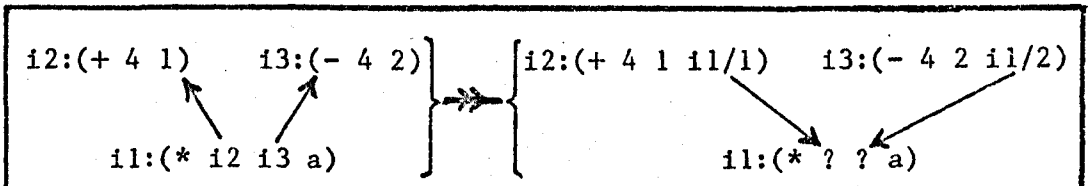


Figure 19 - Pointer Reversal for Implementing Graph Reduction

2.4. Evaluation

In the introduction three motivations were given for the development of novel general-purpose architectures, namely supporting languages with increased expressive power, increased performance through concurrency and suitability for VLSI implementation. This Section discusses the extent to which concurrent control flow, data flow and reduction architectures appear able to satisfy each of these particular motivations. It then considers the range of applications which these architectures can effectively support and thus the extent to which they can be considered general-purpose.

2.4.1. Expressive Power

The expressive power of a programming language is largely concerned with how much organisational detail has to be explicitly specified in the program. A benefit of the novel data flow and reduction architectures is that they directly support more expressive applicative languages by automatically handling some of the operational details. Control flow architectures are based on a low level operational model in which the programmer has to organise all the details of program execution - sequencing instructions to ensure that values are computed before being used, allocating storage for intermediate results, and explicitly manipulating, and even performing arithmetic on, references. Extending sequential control flow to concurrent control flow generally compounds rather than simplifies these problems with the programmer often having to deal with complex synchronisation requirements.

Data flow and reduction provide higher level models in which the programmer is not explicitly concerned with sequencing and storage allocation. Sequencing is implicitly controlled by the availability or need for values and storage is automatically allocated and released as values are created and used. Reduction is more powerful than data flow since computation is only performed when absolutely needed and so the programmer is not concerned with avoiding the initiation of unnecessary computation. This is particularly important in recursion where unnecessary computation could lead to non-termination. Also, (graph) reduction supports the use of unbounded data structures and higher order functions used in very high level applicative programming languages. It is claimed by proponents of such languages that the use of these techniques often leads to a simplification of program structure, compared with what would be required in a conventional language, and that they thus provide an easier means for producing reliable programs.

Greater power of course requires more sophisticated implementation for its efficient realisation. In a control flow implementation the major complex component is the processor for instruction execution. In a data flow implementation there is additionally the mechanism for token matching and in some cases sophisticated storage for data structures. In a reduction implementation there are also the mechanisms for propagating demand and for garbage collection.

2.4.2. Concurrency and Performance

Exploitation of very high levels of concurrency requires a simple scheme for activating and synchronising instructions and this is best

provided by data flow. In concurrent forms of control flow the communication of control is separate from the communication of data and is a considerable overhead when the grain of concurrency is at the level of individual instructions. In reduction there is the overhead and delay of propagating demand through the program. Although this ensures that only the necessary computation is performed it reduces the concurrency and better performance might be obtained for a large number of processors by initiating possibly unneeded computation as can happen in data flow.

Data flow's activation by data availability gives highly concurrent programs, although in some architectures pure data availability is abandoned. There are two particular potential difficulties in the performance of data flow architectures. Firstly a large amount of token storage may be required because the producer of a value generates separate copies for all its users. Secondly there is the fact that individual instructions and procedures are atomic, and may depend on the availability of data which is not actually needed. This means that resources that could otherwise be usefully employed may be wasted on computing that data, and that spare resources may be unused because computations that could use them are delayed awaiting the availability of that data.

2.4.3. Exploiting VLSI

One of the most important considerations in VLSI design is to minimise communication both between chips and between different areas of the same chip. This requires close association of processing and

storage functions (which is made possible in VLSI by the use of the same technology for storage and processing elements). An expression manipulation organisation, with each computing element being a single chip or even an area within a chip, is the only machine organisation which attempts to satisfy these requirements. As is shown in Figure 15, all classes of architecture can be implemented using the expression manipulation organisation. The success of such an implementation is dependent on programs exhibiting locality of reference. In data flow and string reduction there is very strong locality at least within a procedure since the only more global communication is for parameters and results at the start and end of the procedure. In control flow any instruction can in principle reference any item of memory but in fact most referencing is local to a procedure and this is particularly true when modern design methodologies such as data abstraction[44,45] are used. For graph reduction with lazy evaluation the actual execution tends to have a very convoluted structure in which there may be very little locality of reference. This is because the evaluation of one expression may cause the evaluation of another expression anywhere in the program structure.

2.4.4. Generality

Control flow architectures are recognised as being very general-purpose. Evidence for this is that they have been successfully employed in a very large range of applications, and for implementations of all classes of programming languages including applicative languages which require emulation of the reduction operation model. Their generality is largely due to the low level of the operational model which in giving

explicit control to the programmer/compiler allows any required behaviour to be produced. This is particularly important in real time applications where sequencing and timing may be crucial. Data flow is probably the least general-purpose. Most data flow architectures do not provide the mechanisms for controlled non-determinacy needed in operating systems applications and there are some data flow architectures which do not even support arrays or recursion. It is impossible to effectively emulate such features if they are not directly supported, or to emulate a different operational model, as can quite easily be done in control flow. The advocates of applicative programming see reduction architectures as truly general-purpose and there have been successful experiments in producing applicative file systems[46], graphics systems[47] and compilers and interpreters[48] (although not yet applicative process control systems which have been identified by a pioneer of applicative programming[49] as typifying the last remaining application area in which the adequacy of applicative programming is in doubt). The main difficulty is in the acceptability of applicative languages which are radically different to currently popular languages. Also the inclusion of non-determinate operators, necessary for operating system applications, is still somewhat questionable.

In summary, the main benefit of control flow is its lack of restriction and its operational nature which make it very general-purpose; the corresponding weakness is the need to exercise careful operational control which can be extremely difficult in programming a complex highly concurrent algorithm. The benefit of data flow is that it is simple and highly concurrent offering potentially high performance in suitable applications; its principal weakness is that it is not very general-

purpose, having difficulties with more sophisticated control and data structures. The main benefits of reduction are in its expressive power and the main disadvantages are that the applicative style of programming is not the most natural or efficient in many cases. Compared with control flow, data flow is primarily an improvement in performance, reduction is an improvement in expressive power and both are less general-purpose.

3. COMBINING DATA FLOW AND CONTROL FLOW

As shown in the last Chapter control flow and data flow architectures have complementary advantages and disadvantages. This Chapter describes the combination of control flow and data flow concepts in an experimental highly parallel architecture referred to as the Data/Control Flow (DCF) architecture. The DCF is based on an operational model which includes the (parallel) control flow and data flow models as subsets and so allows compilers to generate control flow or data flow style machine code. Thus the architecture supports both conventional programming languages used to program control flow machines, and the class of simple applicative languages, known as "single assignment languages"[50,51], which are used to program data flow machines. It would also be possible for these two styles to be usefully mixed. For example a compiler for a conventional language might generate data flow style code for evaluating expressions (which have an applicative structure) and control flow style code for the other elements of the program.

This Chapter follows the same structure as Chapter Two. First the basic concepts of the DCF operational model are covered. This is followed by a discussion of program organisation. Finally the architecture's machine organisation and two implementations are discussed. The DCF architecture is described at greater length in [12] and details of the two implementations are given in [19] and [20].

3.1. Operational Model

The complementary advantages and disadvantages of control flow and data flow are largely a result of the particular control and data mechanisms found in their operational models. Parallel control flow and data flow both have a parallel control mechanism, the difference being that in the former instruction activation is by control tokens whereas in the latter it is by data tokens. Control tokens give the programmer explicit control over instruction activation whereas data tokens provide implicit activation by data availability. The DCF model has a parallel control mechanism which is a synthesis of these two schemes. Instruction activation is controlled by generalised tokens, each token either carrying data for the activated instruction (a data token) or carrying "null" (a control token).

The control flow and data flow operational models each provide two of the three data mechanisms: by-reference and by-literal for control flow; by-value and by-literal for data flow. The DCF model includes all three mechanisms allowing data to be embedded in the instruction (by-literal), communicated as a data token (by-value), or communicated via a shared memory cell (by-reference). In control flow a memory cell can contain not only basic values but also references to instructions and other memory cells. In DCF there is the same generality in the kinds of data that can be stored in memory and carried by tokens.

Figure 20(a) shows a representation of DCF machine code for the example **a:= (b+1) * (b-c)** and (b) shows an abbreviated representation similar to that used for data flow graphs (a control token is shown as

#). This machine code representation combines elements from the parallel control flow and data flow representations used for the same example in Figures 3 and 4(a).

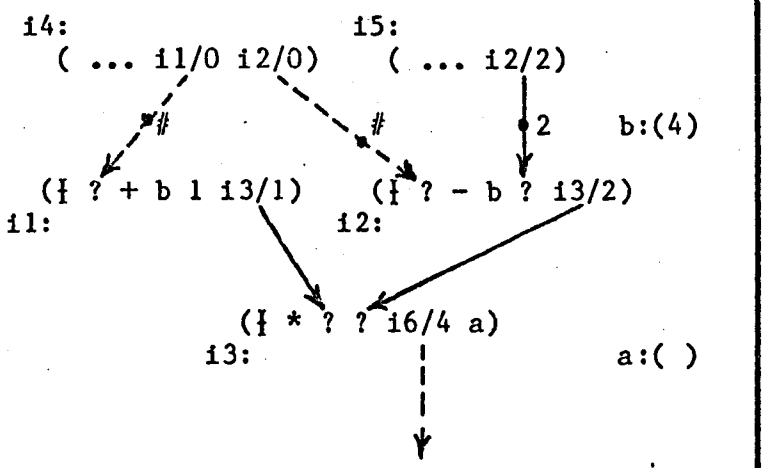


Figure 20 - Data/Control Flow

Each instruction consists of a sequence of arguments. The types of arguments are operators, literals, unknowns (represented by ?s, each specifying the requirement for some input - a control token or data token), memory references (for accessing operands and storing results), and instruction references (for communicating tokens to other instructions). First will be (zero or more) control arguments (?s) each specifying the requirement for a control token as in control flow. Next comes the operator (e.g. +) which is followed by input arguments. An input argument may be a literal (e.g. 1), an unknown (?) specifying the requirement for a data token as in data flow, or a memory reference (e.g. b) as in control flow. Finally there are output arguments each of which references either a memory cell to be updated with the

instruction's result (e.g. **a** in **i3**) or the destination for a token (e.g. **i3/1** in **i1** for a data token carrying the result, or **i6/4** in **i3** for a control token indicating that the result has been stored).

All instructions are notionally active and can potentially execute in parallel, execution being constrained by the arrival of tokens. The instruction execution cycle combines the control flow and data flow cycles, comprising the following stages.

(i) activation (as in parallel control flow and data flow) - the instruction is activated when tokens for all ? arguments have arrived

(ii) memory load (as in control flow) - any required data residing in memory is retrieved using memory cell references

(iii) operator execution (as in all models) - result data is computed from operand data as defined by the operator

(iv) memory update (as in control flow) - data is stored into memory cells identified by memory cell references

(v) token emission (as in parallel control flow and data flow) - output (control and data) tokens are emitted to argument positions in other instructions identified by instruction references and these tokens then contribute to the activation stage of those instructions.

In this execution cycle any information used (i.e. a data item, a memory cell reference or an instruction reference) in a particular stage may be provided as an embedded literal in the instruction itself or dynamically

provided at any preceding stage (as a data token received at stage (i), loaded from memory at stage (ii), computed as the operator's result at stage (iii)). This allows considerable flexibility in the low level organisation of programs and gives orthogonality between operators and data access.

A data item output by an instruction, as the value for a data token or memory cell update, may be not only its operator's result but also any of its input data. This allows the possibility of low level optimisation of the program graph. For example in a graph for **((b - c) + b)** the subtraction instruction could emit two tokens to the addition instruction, one providing its own result and the other passing on the value of **b**. This would mean that in the interval between the execution of the instruction generating **b** and the execution of the subtraction there is only one copy of **b**. (In contrast, in pure data flow there would be two copies, each occupying machine storage resources, one copy for use by the addition and one copy for use by the subtraction.)

The essential point however is the flexibility of combining the control flow and data flow models at the level of individual instructions. For instance **i1** uses control flow for input and data flow for output whereas **i3** uses data flow for input and control flow for output. In this particular example the data flow model is used for communicating an expression's partial results (which are only used once), whereas the control flow style is used for manipulating and controlling access to shared data such as **b** and **a**. However in the DCF model there is no particular constraint on the way instructions use the data tokens, memory cells and control tokens which support the instruction level combination

of control flow and data flow. The combination of control flow and data flow at higher levels of program organisation is discussed in the next Section.

3.2. Program Organisation

This Section discusses how the forms of program organisation discussed for data flow and control flow architectures in Chapter Two are combined in the DCF architecture.

3.2.1. Data Structures

In the DCF architecture data structures are manipulated in the same way as in control flow architectures, namely by communicating references and performing arithmetic on them. However a reference representing a data structure can be passed as a data token giving the effect of communicating the whole data structure as a single token, as in data flow architectures. In particular, the availability of the structure can activate the instructions that manipulate it.

3.2.2. Conditionals

In the DCF architecture the various forms of conditional found in control flow and data flow architectures are supported by a single conditional switch operator. Figure 21 shows three instructions which use the switch operator (represented as **Y**) to produce the effect of (a) the switch-in instruction of data flow which selects its output value from two alternative inputs; (b) the switch-out instruction of data flow which switches a data token to one of two alternative instructions; (c) the conditional branch instruction of control flow which switches a flow of control (control token) to one of two alternative instructions.

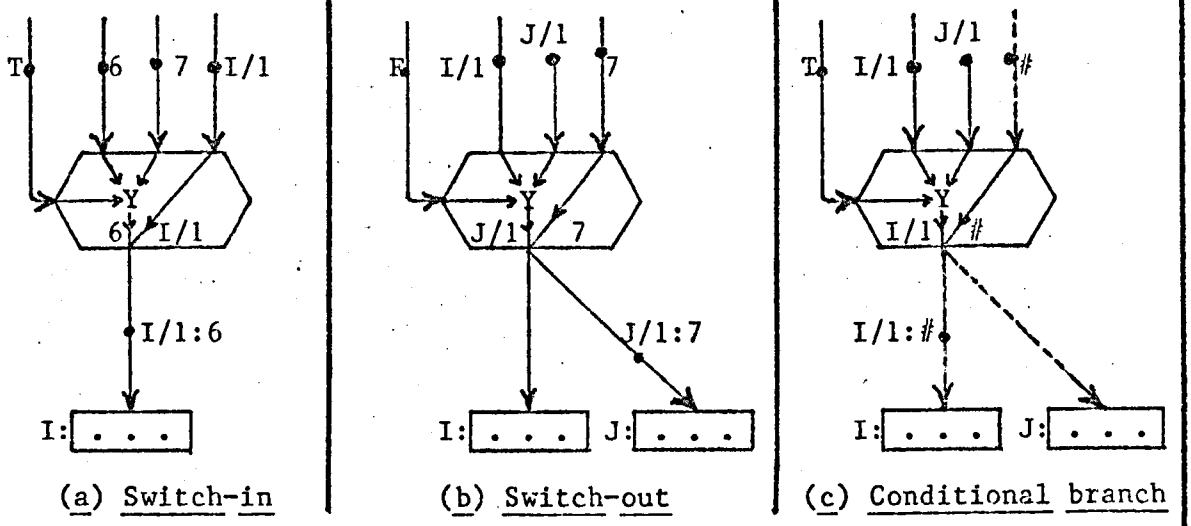


Figure 21 - Conditionals in DCF

The switch operator, **Y**, has three operands and one result. The first operand is a boolean which selects one of the other two operands as the result. Each instruction has four inputs. One of the inputs is the boolean (**True** or **False**) controlling the switch. Each of the other inputs is either a data token carrying a value (**6** or **7**), a data token carrying an instruction reference (**I/1** or **J/1** identifying the first argument position of instruction **I** or **J**), or a control token (represented as a **#**). Each instruction has one output token represented as "destination:value" (with **#** as the value for a control token). In the token emission stage of instruction execution this output token is constructed from the instruction's inputs and the operator's result. The different effects of the switch operator in these three instructions depend on whether its result is used as the value for the instruction's output (in (a)) or as its destination (in (b) and (c)). These examples particularly illustrate the orthogonality between operator and data access. The same switch operator could also be used in instructions to

store a conditionally selected value in memory cell or store a value in a conditionally selected memory cell.

3.2.3. Iteration and Procedures

The DCF architecture supports both iterations and procedures. Iteration is possible using either the scheme used in control flow or that used in data flow. As in control flow memory cells can be reused for successive iterations and an instruction, such as the conditional branch of Figure 21(c), can transfer control back to the start of the iterative code. In the data flow scheme for iteration, which was illustrated in Figure 10(ii), data tokens are synchronised at the end of each iteration and fed back to the start. This scheme requires a "synchroniser" instruction which would be a normal DCF instruction with a **NOOP** operator (just outputing its inputs, rather than producing a new result). The two schemes can also be combined with some data items being communicated to the next iteration via memory cells and some being passed as data tokens.

The support of procedures depends on the separation of coexisting environments accessed by shared code. In the DCF architecture an environment can consist of both information stored in memory cells (as in control flow) and tokens (as in data flow). Each active environment has a distinct identifier which provides both a token label (as used in data flow procedures) and a reference to a local memory area for the environment (as used in control flow procedures). An individual instruction in a procedure may execute many times in different environments. Each execution takes a set of input tokens with the same

environment identifier; all memory addresses in the instruction for loading and storing data are relative to the reference provided by that identifier; and all output tokens are labeled with that identifier.

In a data flow architecture procedure call and return are provided as single instructions (as illustrated in Figure 10(i)). The DCF architecture provides more primitive instructions for manipulating environments which can be used to construct procedure calls and returns. These primitives include: **NewE** which creates a new environment, allocating its local memory area and generating the unique environment identifier; **Kill(E)** which terminates the specified environment, deallocating its local memory area and deleting any remaining tokens; **SetE(E)** which passes information to a specified environment - instead of the environment identifier of its input tokens being used to identify the environment of its outputs (tokens and memory updates) implicitly, the explicit operand value (E) is used so that those outputs go to that environment.

Figure 22 illustrates the general structure of the way these primitives can be used for procedures. The procedure **F** has two parameters, **p1** and **p2** calculated by program fragments **P1** and **P2**, and two results, **r1** and **r2** calculated by program fragments **F1** and **F2**. The procedure call consists of five separate instructions: **N, A1, A2, C1, C2.** Instruction **N** creates the new environment for the called procedure and passes its identifier, E′, to the other instructions in the call. The creation of the new environment by **N** is activated by a control token from program fragment **P0** which determines whether or not the procedure call is actually made (e.g. depending on the termination condition in a recursion). The instructions **C1** and **C2** pass the parameters, **p1** and **p2**, into the new

environment. Parameter **p1** is communicated by data tokens, whereas parameter **p2** is communicated via memory cells, **m1** and **m2**, with associated control tokens. The instructions **A1** and **A2** pass into the new environment the information needed for the two results to be returned to the calling environment. The required information is the calling environment's identifier **E** and references to the instructions **X1** and **X2** in the calling environment.

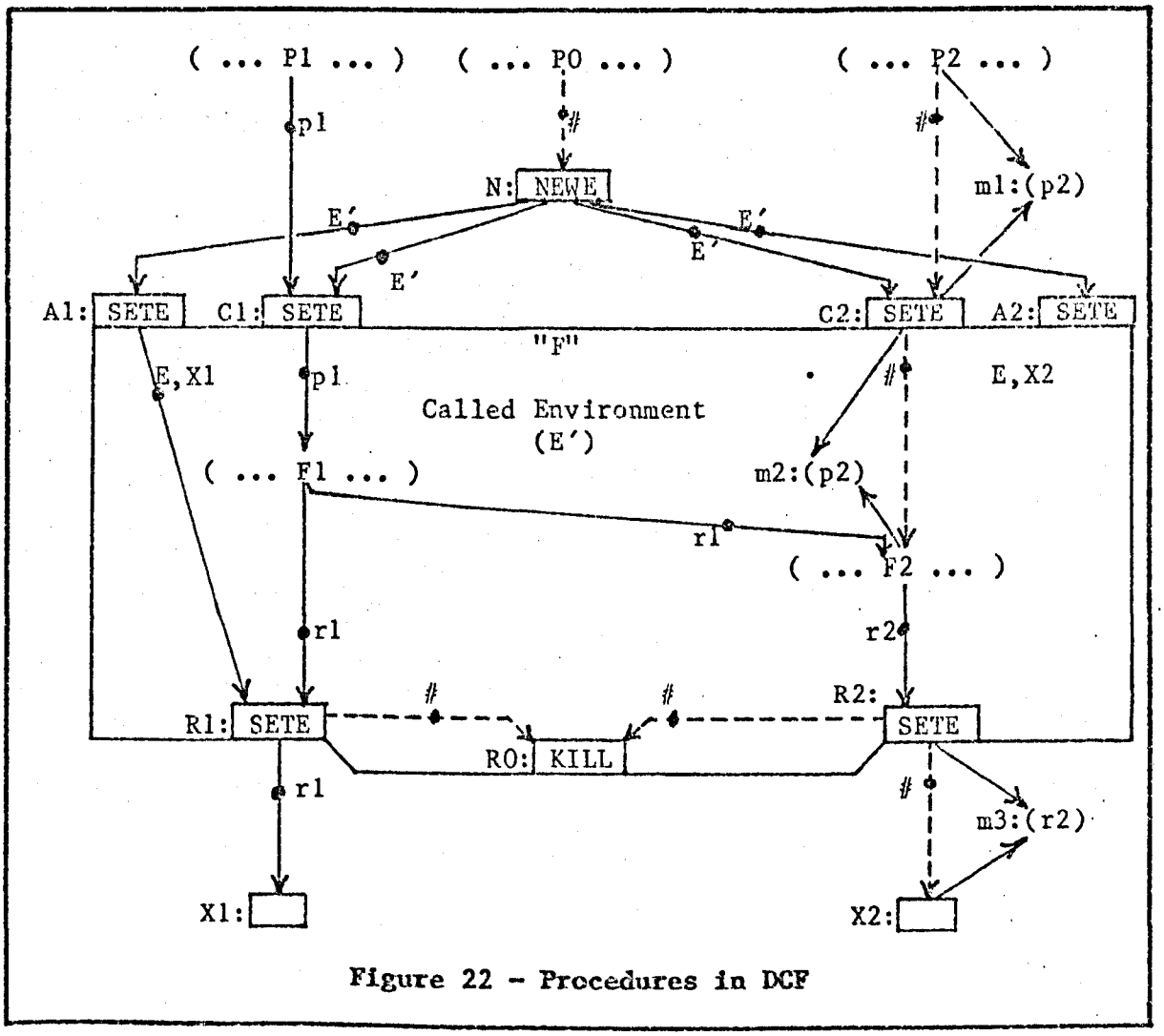


Figure 22 - Procedures in DCF

The procedure return consists of three separate instructions: **R0**, **R1** and **R2**. Instructions **R1** and **R2** pass the results back to the calling

environment and R0 terminates the called environment when both results have been returned. For **r1** all communication uses data tokens, whereas for **r2** both data token and memory cell communication is used.

In this example some of the parameters and results are communicated as data tokens (as in data flow procedures) and some are communicated via memory cells (as in control flow). The ability to combine control flow and data flow at this procedure level is a direct consequence of the combination of control flow and data flow provided by the basic operation model at the level of the individual (**SETE**) instructions used to communicate the parameters and results.

The procedure structure used here has separate instructions for communicating each parameter and each result, so that the procedure is non-atomic. Non-atomic procedures generally provide greater possibilities for concurrency. In this example instructions in **F1** which calculate result **r1**, and then **X1** which uses **r1**, are concurrent with instructions in **P2** which calculate the parameter **p2** (not itself used in the calculation of **r1**). Non-atomic procedures also tend to lead to a clearer structuring of programs in that the grouping together of instructions into procedures can be determined purely by the logical structure of the program without being influenced by the need to exploit potential concurrency. In order to exploit the potential concurrency between **X1** and **P2** using atomic procedures, such as are provided in data flow, it would be necessary to split **F** into two procedures, one containing the instructions of **F1** and the other containing those of **F2**.

A **Sete** instruction, as any other instruction, can have a number of inputs and outputs. All outputs would be labelled with the specified

environment identifier. Thus the primitives provided in DCF could be used to construct atomic procedures. For the example in Figure 22 this would be achieved by combining all the input **Sete** instructions (i.e. **A1, C1, C2** and **A2**) into a single **Sete** instruction having **E'**, **p1**, and **p2** as inputs. Similarly the output instructions (**R1** and **R2**) would be combined. With an atomic procedure structure there is the disadvantage of a possible loss of potential concurrency. The benefit is that the reduced level of concurrency can be organised with fewer instructions and with fewer tokens (one E' token instead of four, one control token for instruction R0 instead of two).

3.2.4. Non-Determinacy

In the DCF operational model there are two potential sources of non-determinacy, namely simultaneous updates of memory cells (as in control flow) and the simultaneous emission of two tokens for the same destination (as in data flow). The synchronisation of normal instruction activation can be used to control non-determinacy. This is illustrated in Figure 23 using the same resource manager structure as was used for data flow in Figure 13.

There are two concurrent users **A** and **B** of a resource which they access by sending "transaction" data tokens **(p, q)** to the resource manager. The resource manager is a critical region which can only process one transaction at a time and is protected by **P** and **V** instructions as in standard control flow techniques using semaphores. (These two instructions are here shown with **NOOP** operators although any operator could be used in such instructions). The two inputs needed for the

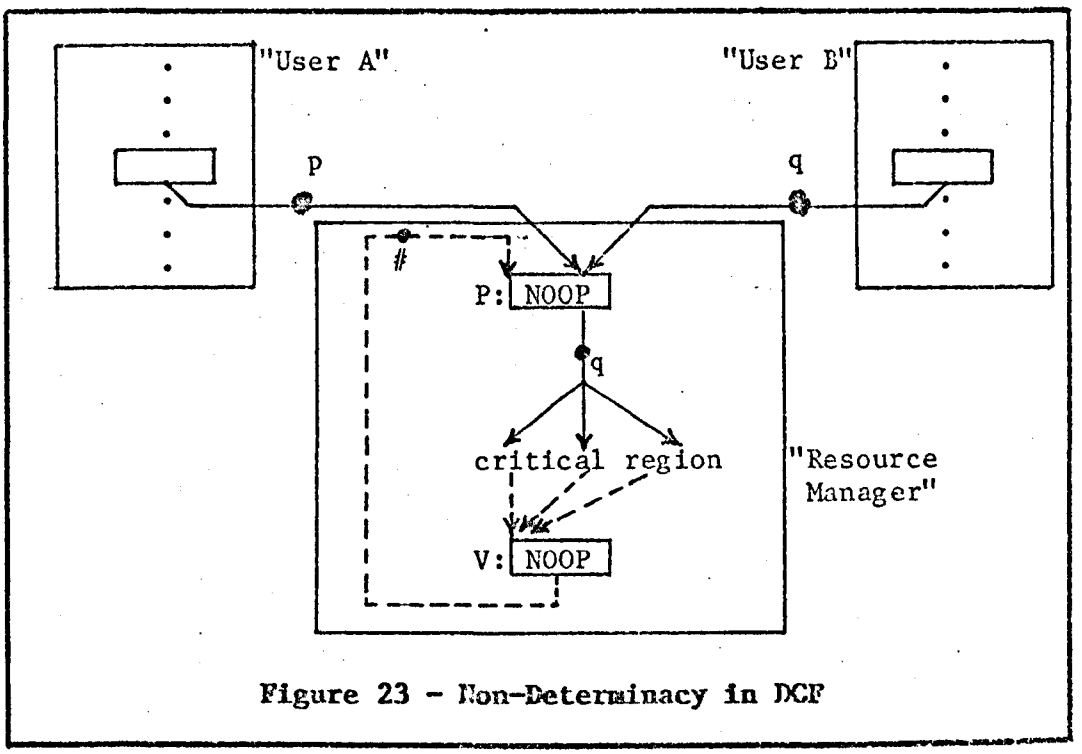


Figure 23 - Non-Determinacy in DCF

activation of P are a control input provided by a "semaphore" control token which signals that the critical region is free; and a data input provided by one of the transaction data tokens which are sent, possibly simultaneously, by users A and B. In the activation stage of executing instruction P there would be an arbitrary choice of one of the available data tokens to match with the control token, the other data token remaining until it could be matched with another semaphore token. The operator of instruction P is a NOOP and the instruction just outputs its data input, thus releasing the transaction into the critical region. When all the critical processing (updating the resource) has been completed instruction V is activated. This again is a NOOP to free the critical region by providing a control token for another activation of P and thus release another transaction into the critical region.

3.3. Machine Organisation and Implementation

There were two experimental DCF implementations, carried out as separate dissertation projects by M.Sc. students. Both implementations, one in software[20] and one in hardware[19], are based on the design shown in Figure 24. This design has a packet communication organisation with token matching, similar to the data flow machine organisation shown in Figure 17(b). The successive stages of the instruction execution cycle are split into separate machine resources which form a circular execution pipeline. Packets of information flow around the pipeline, each packet representing an intermediate state in the execution of an instruction or a communication from one instruction to another.

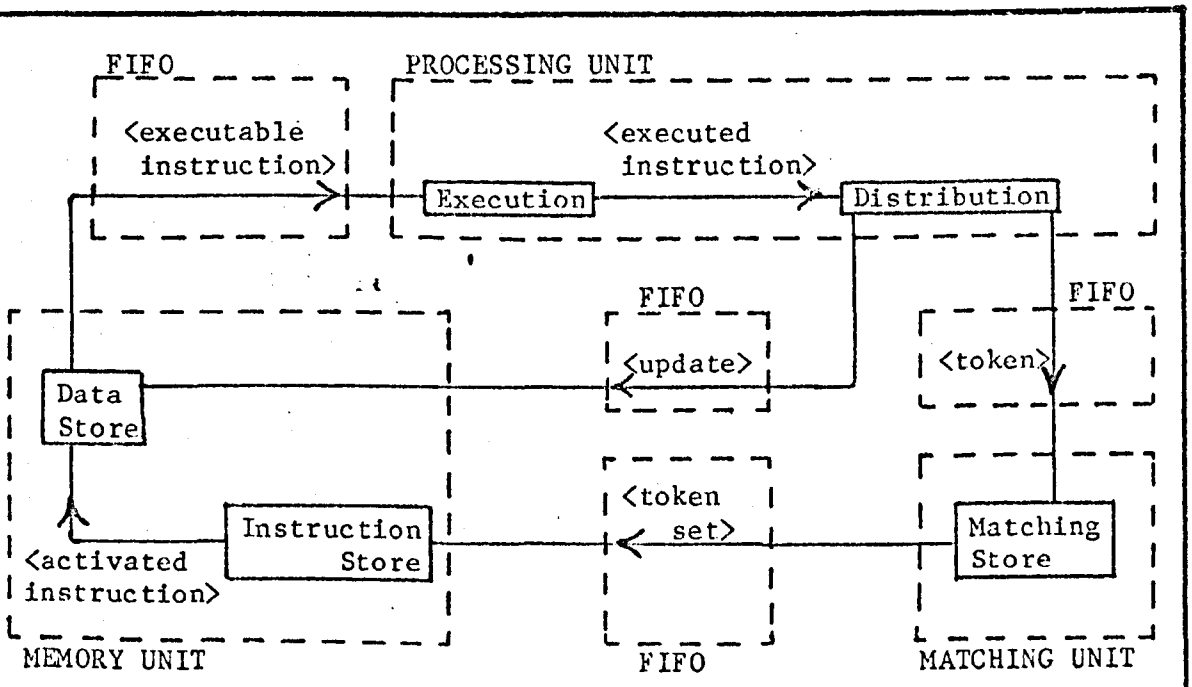


Figure 24 - DCF Machine Organisation and Implementation

The Matching Store receives tokens and collects them into token sets. Each token set contains tokens for the same destination

instruction and is sent to the Instruction Store when the set is complete. There the tokens combine with the destination instruction to form an activated instruction. Memory cell references are then dereferenced by loading data from the Data Store to produce an executable instruction which is sent for execution. The operator is executed to give an executed instruction which specifies some memory updates and output tokens for distribution to the Data Store and Matching Store.

The main components of the design are organised into three functional units. The Matching Unit implements the Matching Store. The Memory Unit implements the Instruction and Data Stores as a single address space (so that the machine can compile its own programs). The Processing Unit implements operator execution and output distribution. These Units communicate by sending packets which are temporarily stored in separate first-in-first-out (FIFO) queues between the Units. The design could be easily extended for greater parallelism by including several Processing Units.

In the hardware implementation each of the three functional units is a separate M6800-based microcomputer. The purpose of this implementation was as an exercise in building a novel multi-processor computer from "off the shelf" components. In the software implementation each functional unit is a process in a SIMULA program. The purpose of this implementation was both as a machine code interpreter to experiment with programming the DCF architecture and as a machine simulator to determine performance characteristics. Results of experiments performed using the simulator are reported in [20]. The main conclusion that can be drawn is that the performance of the Matching Unit is the limiting factor in

the design. On average each instruction execution generates two tokens and for each token it receives the Matching Unit has to search its Matching Store to find any previously received tokens with which the newly-received token can match. To balance the design it would be necessary for token matching to take half the time taken by operator execution. This would be difficult to achieve using conventional memory organisation for the Matching Store and it would probably be necessary to use a special-purpose associative memory or have several Matching Units per Processing Unit.

3.4. Summary and Discussion

The DCF architecture combines elements from control flow and data flow architectures. It includes both their separate data mechanisms (i.e. by-reference and by-value) and has their common parallel control mechanism and packet circulation machine organisation. The DCF operational model includes as subsets the operational models of control flow and data flow and integrates their various forms of program organisation so that both control and data flow styles of machine code can be easily combined.

Comparing the DCF architecture with the more conventional sequential and multi-thread forms of control flow architectures, the major difference is in the parallel control mechanism and by-value data mechanism. These mechanisms, supported by the packet circulation machine organisation, facilitate the execution of highly concurrent programs. Comparing the DCF architecture with data flow architectures, the real difference is in the more primitive level of machine code interface that it provides. As in the DCF architecture, data flow architectures can include separate memory accessed by "reference tokens" (for data structures), "control tokens" (both the acknowledge tokens in the MIT data flow machine and the outputs of switch-out conditional instructions act like control tokens), environment manipulation (for procedures) and non-determinate instruction activation (for resource managers). In data flow such features are incorporated into higher level constructs, for example **CALL** and **MERGE** instructions. In contrast, they are provided as primitive features in the DCF architecture. The more primitive interface of course puts a somewhat greater burden on the

programmer/compiler, but gives a more flexible general-purpose architecture. This flexibility is illustrated by the ability to have both atomic and non-atomic procedures and the ability to use a control token for any synchronisation requirement.

The main deficiency of the DCF is that although it includes all three of the data mechanisms it only includes one of the control mechanisms. The absence of the sequential control mechanism means that there would be poor performance for a purely sequential program since a control token would have to circulate completely around the machine between each instruction execution. The absence of the recursive control mechanism means of course that the DCF still has the disadvantages discussed in Chapter Two of data flow and control flow compared with reduction.

There are two particular weaknesses in this combination of control flow and data flow concepts. Both of these concern the way addressable memory has been incorporated. Firstly there is a potential implementation problem in supporting the conventional view of memory which requires that an instruction's memory updates are completed before its successor instructions are activated. In the actual implementation this view is easily supported because both memory updates and instruction activations go to the same single Memory Unit and the former can be given priority. However there might be a serious difficulty in a more distributed implementation.

Secondly there are the differences between "instructions" and "data". Both an instruction argument and a memory cell act as a container into which an instruction can store a value. However these two

types of containers have very different properties: an instruction argument can contain either an unknown or an actual value (as a literal) whereas a memory cell can only contain a value; a memory cell can be updated whereas an instruction argument cannot (an attempt to overwrite a literal with a data token results in an error); the arguments of an instruction form a structure with each component being identified by a two-part instruction/argument reference, whereas memory cells are unstructured. Essentially there are two separate, by-value and by-reference, data mechanisms, with their respective characteristics inherited from data flow and control flow. In its data mechanism the DCF architecture is thus only a combination of control flow and data flow, whereas in its control mechanism it is a genuine synthesis. The control mechanism provides a single integrated notion of activation by a complete set of tokens which includes, as special cases, activation by just control tokens (as in control flow) and activation by just data tokens (as in data flow).

4. GENERALISING CONTROL FLOW

Having described in the previous Chapter an architecture, the DCF, which supports both control flow and data flow, this Chapter describes an architecture, referred to as the Recursive Control Flow (RCF) architecture, which supports all three of control flow, data flow and reduction. Whereas the basis of the DCF was a direct combination of specific concepts from parallel control flow and data flow, the RCF is based on a set of general principles, the "recursive principles" mentioned in the introduction, which have been proposed[17] as the appropriate basis for future general-purpose decentralised computing systems using VLSI technology. These principles of recursive architecture are a generalisation, based on the use of recursion, of the von Neumann principles underlying conventional computing systems. The von Neumann principles not only form the basis for conventional general-purpose computers and languages used to program them, but also govern much of the design of computer networks since their components are von Neumann computers. The recursive architecture principles are here developed principally in the context of highly parallel computers. They are however also relevant in the context of geographically distributed computer networks[52] and that aspect is explored towards the end of the Chapter. The concept of recursive architecture came originally from the early work of Glushkov[53] and subsequent work of Barton[54] and Wilner[55] whilst its realisation in the RCF architecture is based on the investigations of control flow, data flow and reduction covered in Chapter Two.

This Chapter follows the same general structure as the preceding Chapters, starting with the RCF operational model, followed by a

discussion of program organisation and finally covering machine organisation. A proposed implementation of the architecture is covered in some detail in the subsequent Chapter, together with a comparative discussion of of several other decentralised computing systems (including a computer networking system) which incorporate similar recursive principles.

There are two major aspects to the RCF architecture. The first is the general principles of recursive structure. Sections One and Three concentrate on these principles and their realisation in, respectively, the RCF operational model and the machine organisation supporting it. The second aspect is the combination of control flow, data flow and reduction in a single architecture. Section 2 concentrates on this aspect, covering the various forms of program organisation discussed for control flow, data flow and reduction in Chapter Two.

4.1. Operational Model

The conventional, control flow, operational model is based on the von Neumann principles for the organisation of storage, addressing and program representation and execution. This Section discusses the alternative recursive principles and their realisation in the RCF operational model.

4.1.1. Storage Organisation

A conventional architecture provides static, linear storage structures of fixed size units, such as words of main memory or blocks of backing store. The elementary actions on such a structure are to copy,

replace or execute the contents of a single storage unit such as a memory cell containing a simple data item or machine code instruction. A recursive architecture provides a dynamic storage structure supporting a single hierarchy of variable length units, referred to as "objects", similar to the hierarchy of a block structured program or operating system filestore. Each object is either a primitive object, such as an integer or instruction, or recursively a compound object comprising a sequence of component objects. The elementary actions supported for such a structure are to copy an object, replace an object, insert a new component into an object, delete a component from an object, or execute an object.

In the RCF architecture the hierarchic object structure is represented as a string of symbols comprising data symbols for representing primitive objects and structuring symbols (and) for delimiting objects. For example, with integers as data symbols, a 3X3 multiplication table might be represented as -

((1 2 3) (2 4 6) (3 6 9))

At the machine code level the data symbols are just **0** and **1** and each integer in this table would actually be represented as a delimited sequence of 1s and 0s, such as -

(((1) (10) (11)) ((10) (100) (110)) ((11) (110) (1001)))

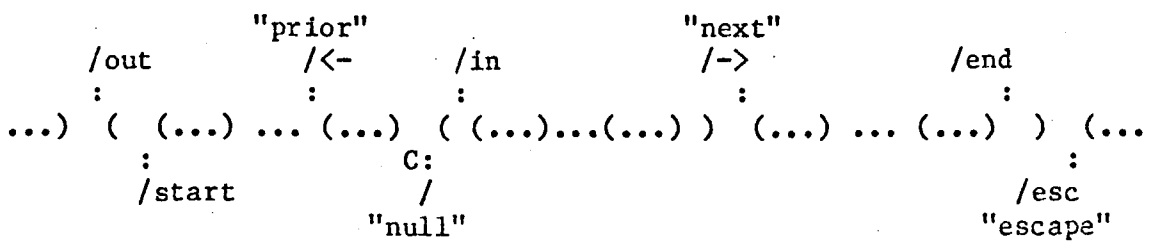
4.1.2. Addressing

A conventional architecture uses absolute addressing within a one level address space, an address being for example the ordinal number of a word in main memory. A recursive architecture uses contextual addressing within a hierarchic address space, an address being similar to a filename in the filestore of an operating system such as UNIX[56] or an international telephone number. This contextual addressing is based on a set of selectors which can be applied in the context of one object to select a related object. For example a selector **/2**, meaning "second component", applied in the context of the object **(3 6 9)** would select the object **6**. An address is a sequence of such selectors, with each selector being applied in the context selected by the preceding selector. Thus an address, such as **/6/3/2** meaning "2nd component of 3rd component of 6th component", specifies a path from an initial context to a particular object. (In this Chapter a **/** will be used at the start of all addresses and individual selectors, in order to distinguish an address from the object it addresses and to delimit the component selectors of an address.)

In the RCF architecture, where the object structure is represented by a string of symbols, a selector identifies a position in the string between two symbols, selecting both the existing object to the immediate right of that position and a space where a new object could be inserted. The selectors are illustrated in Figure 25(a) with each selector, **/<-** ("prior"), **/->** ("next") etc., being shown as labeling the position in the string which it would identify relative to the position labeled **C** (which is itself identified by the null selector, **/**). Excluding the

/start and /escape selectors, these are a minimum set of primitives from which an address for any relative position in the structure can be constructed. The various non-primitive selectors that will be used are also defined in Figure 25(b) as sequences of the primitive selectors.

(a) Simple positional selectors



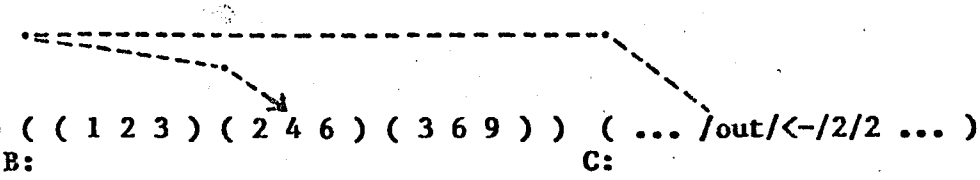
(b) Derived Selectors

"nth next"	"nth prior"	"nth component"	
+1 = /->	-1 = /<-	1 = /in	/start = /out/in
+2 = /->/->	-2 = /<-/<-	2 = /in/->	
+3 = /->/->/->	-3 = /<-/<-/<-	3 = /in/->/->	/esc = /out/->
.	.	.	
.	.	.	
.	.	.	

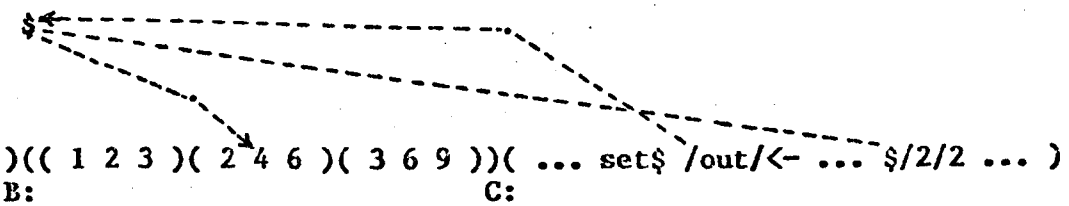
Figure 25 - Addressing Selectors in RCF

There are two forms of addresses, "self-relative" and "base-relative". The former is just a sequence of selectors, such as **/out/->/->**. This would itself be an object in the string, with the individual selectors as its component objects, **((/out) (/->) (/->))**. The initial context for such an address is the position in the string at which the address itself starts. Thus using the multiplication table above as an example data structure **B**, the self-relative address for the object **4**, from within a program fragment **C** adjacent to that structure

would be as in -



In base-relative addressing a previously selected position in the string is used as the initial context or "base" for a sequence of selectors -



The $ in the base-relative address **$/2/2** identifies the base position, **B**, previously selected by the **set$ /out/->**. The remaining selectors **/2/2** are relative to that base, identifying the same object **4** as before. This form of addressing corresponds to the use of "working directories" and "root directories" in the UNIX file naming system, the use of "currencies" in the data manipulation language of a network data base model [57], and, in conventional machine code, addressing relative to a base address stored in a register.

Compared with conventional storage and addressing there are a number of benefits in a dynamic recursive storage structure and contextual addressing such as are provided by the RCF architecture. There is direct support for the representation of commonly used storage structures (such as lists, stacks and queues) and the required addressing (e.g. "next", "first component" and "end") and manipulation (e.g. "insert") of their components. There is no architectural limit on a

machine's address space, as would be imposed by a fixed address size. The total address space of a machine is contained in one (outermost) object and for example the machine's address space can be extended by the insertion of a new object at the end of that outermost object. Also the address spaces of several machines can in principle be combined (thus extending each) by embedding each of their outermost objects as components within a larger containing object. Such address space extensions do not affect the validity of previously used addresses and the resulting address space is completely homogeneous.

Contextual addressing does however have two particular potential drawbacks. Firstly, the representation and interpretation of an address is relatively inefficient if the address comprises a long sequence of selectors. (References will however tend to be between objects which are relatively close in the structure and thus most addresses can be expected to be fairly short.) Secondly there is not necessarily a fixed correspondence between an address and the object which it will select when used. In the situation shown above the self-relative address **/out/<-/2/2** selects the object **4**. However that address would identify a different object if say an additional component were inserted between **B** and **C** or if the address were used in the context of **A** rather than **C**. This can clearly cause problems, particularly when one part of the program needs to communicate an address for use in another part of the program.

The second form of addressing, using a pre-selected base as the initial context, alleviates both of these problems. Once a base has been set to a particular position it directly identifies that position

and can be used many times without the need to re-specify and re-interpret the full selector sequence. Also it continues to identify that same position regardless of where it is used or of changes to the object structure. (A base in the object structure is analogous to a bookmark in a loose-leaf folder - it identifies a position between two objects (leaves) and retains significance as objects in its vicinity are inserted and removed, but is no longer usable if the surrounding object (whole folder) is destroyed.)

In subsequent examples "absolute addresses" will be used, such as the absolute address /B to select the 4 from two different contexts in -

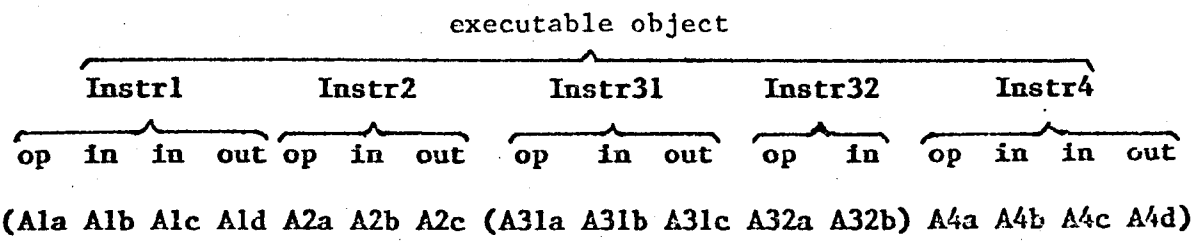
```
( ... /B/2/2 ... ) ( ( 1 2 3 ) ( 2 4 6 ) ( 3 6 9 ) ) ( ... /B/2/2 ... )
A:                 B:                                 C:
```

This is just a notational convenience and the /B in these addresses is intended to represent the appropriate sequence of selectors (/out/-> from within A, or /out/<- from within C).

4.1.3. Program Representation

The operational model of a conventional architecture is embodied in a low level machine language in which instructions are elementary operations performed on elementary operands. A recursive architecture provides a recursive machine language supporting nested program structures, such as is found in the string reduction model. In such a language an instruction has an "operation object", which may be a simple operator or say a procedure, operating on "operand objects", which may be simple data items or complex structures.

The operational model of the RCF architecture is a generalisation of the conventional sequential control flow model which includes the control flow, data flow and reduction models as subsets. In this model a program is structured into executable objects, instructions and arguments, as for example in the structure -



The executable object contains a (sub-structured) sequence of instructions (**Instrs**), each of which comprises a sequence of arguments (As), themselves executable objects. Execution of such a object, invoked by some instruction in the program, would normally proceed sequentially from instruction **Instr1** through to instruction **Instr4** and then terminate. In a subordinate object such as **(Instr31 Instr32)** the contained instructions are executed in sequence. Of the arguments forming an instruction such as **Instr1**, the first is the operation (**op**) which determines the interpretation of the following arguments. The operation is followed by input arguments (**ins**) providing its operands. These arguments are subordinate executable objects, e.g. expressions, which are recursively invoked in parallel by the operation and return the operands. The simplest form of operand is a data item such as an integer which when invoked returns itself. Following the input arguments are the output arguments (**outs**) specifying where the results of the operation are to be stored. For an instruction at the end of its containing object an output argument may be absent, as for instruction

Instr32, in which case the result is returned to the invoking instruction. (An executable object may comprise a number of such return instructions and will thus in the general case return a sequence of results rather than a single result). An address may be given in an argument position, in which case the addressed object is used as though it occurred as the actual argument.

Figure 26 shows a representation of the example from Chapter Two -

a = (b+1) * (b-c) where b=7-3, c=2

The object **A** comprises two instructions, **i1** and **i2**, which are executed in sequence. The instruction **i1, - 7 3 /B**, computes the value **4** and stores it at **B** for use by subsequent instructions, as in the conventional control flow model. Instruction **i2, * (i3) /I4**, is structured as in a reduction model: it has no output argument so that its result is returned to whatever operation invoked **A**; its operands are provided by recursively invoked instructions **i3** and **i4**. The first of these, **i3**, is embedded directly as an actual argument whereas the other, **i4**, is invoked via its address **/I4**. Both **i3** and **i4** have the address of **B** as an input argument to access the value computed by **i1**, and have no output arguments so that they return their results to **i2**.

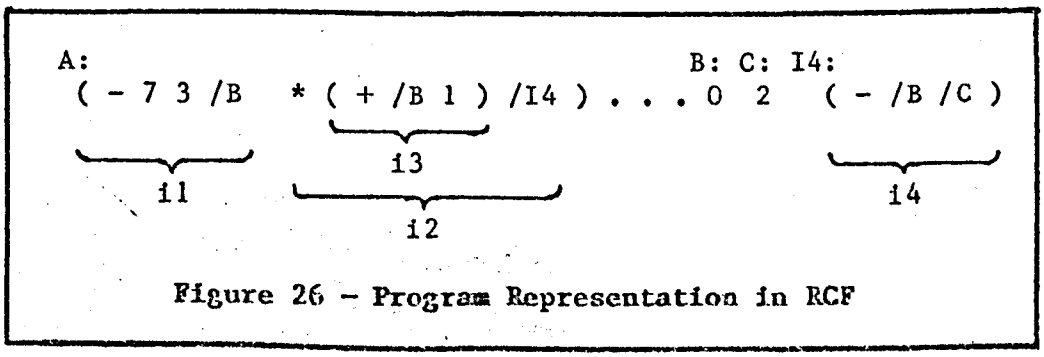


Figure 26 - Program Representation in RCF

This example has illustrated the combined use of concepts from control flow and reduction styles of program representation. The Section on Program Organisation will consider in more detail the way in which RCF programs can be organised as in control flow, data flow and reduction. In addition to the general concepts of program representation presented here, this will require the provision of specific types of primitive arguments, such as the ? ("unknown") arguments used in data flow and parallel control flow.

4.1.4. Program Execution

The conventional operational model has a sequential control mechanism with a single locus of control or "activity" proceeding serially through the program. The interface between an operator and its arguments reflects the basic processor/memory interface of load and store operations on single memory cells. A recursive program representation requires some form of recursive control mechanism such as that of the reduction models presented in Chapter Two. In a reduction model there is a tree of concurrent activities each evaluating some part of the program structure and returning its result to the activity which invoked it. The basic operator/argument interface is the demand for an argument to execute and the return of the result produced by its execution.

The RCF model incorporates the sequential, parallel and recursive control mechanisms. An executable object (generally comprising a sequence of instructions) is executed by an activity which moves sequentially through the instructions executing each operation in turn. There may be several independent activities executing instruction sequences in

parallel. The execution of an operation by an activity will involve subordinate activities which recursively execute the objects forming the operation's arguments. To accommodate various operation/argument interfaces, such as demanding and returning operand values, the model incorporates general communication of "message" objects between superior and subordinate activities as they cooperate in the concurrent execution of a program structure.

This model of execution is illustrated in more detail in Figure 27 which shows successive stages in the execution of the example in Figure 26. The position of activities in the program string are, as previously, indicated by ↑ symbols. To illustrate the progress of execution this Figure also shows the tree structured relationships between superior and subordinate activities (identified as **P**, **Q**, **R**, etc.) and the messages communicated between them. These messages are primitive objects representing integers or specifying particular actions to be performed at the destination. For example in (v) the **replace-with** message followed by the message **4**, sent to the activity **S**, causes the object **4** to replace the object **0**, at which **S** is positioned.

Execution of an operator consists of the following five steps, exemplified by activity **P** executing (in (i) - (vi)) the **-** operator of the control flow style instruction **- 7 3 /B**, and the same activity then executing (in (vii) - (xi)) the ***** operator of the succeeding reduction style instruction *** (+ /B 1) /I4.**

1. As shown in (ii) and (viii) two subordinate activities, **Q** and **R**, are positioned at the input arguments and a subordinate activity, **S**, is positioned at the output argument (if there is one).

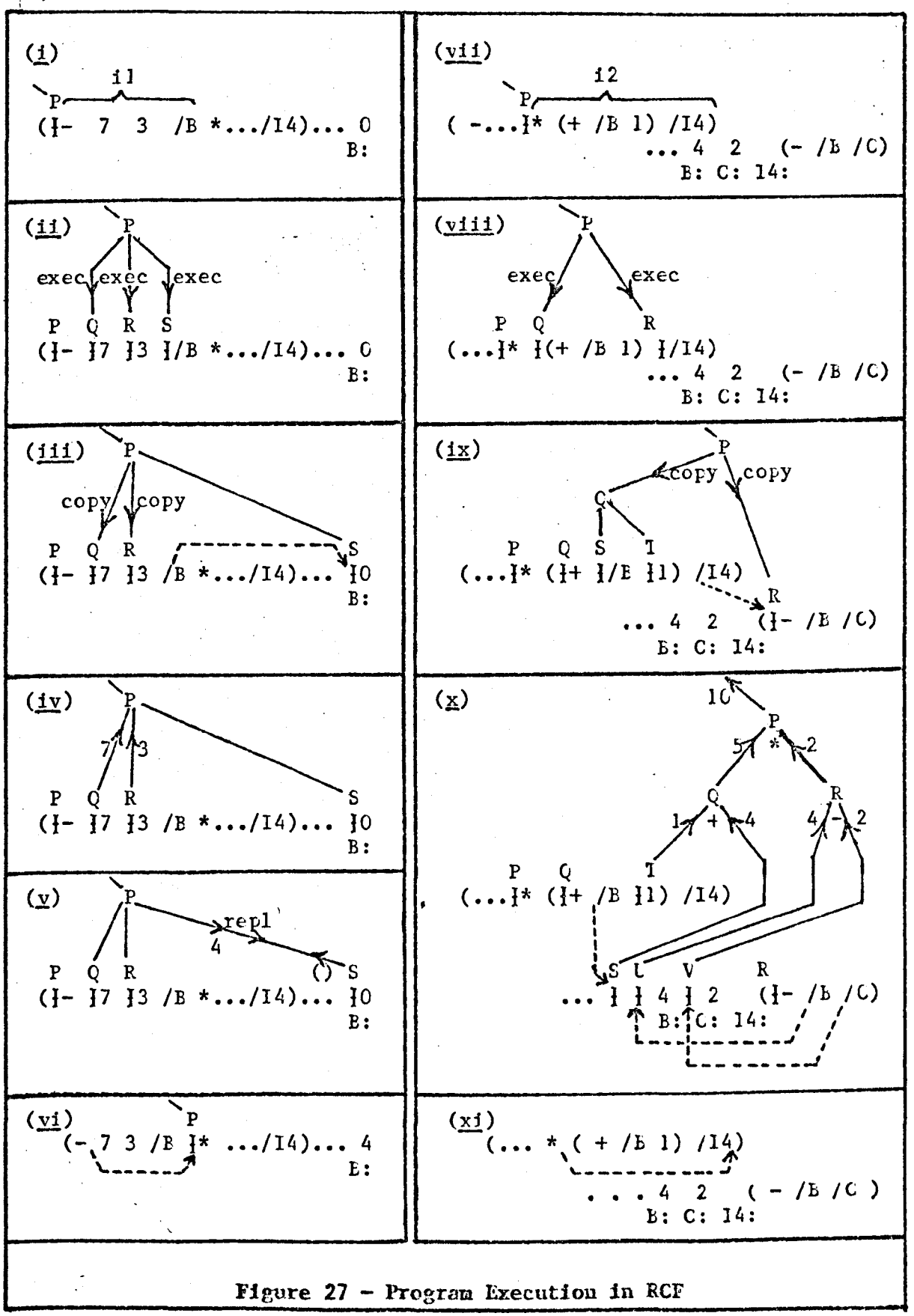


Figure 27 - Program Execution in RCF

2. These subordinate activities are sent **execute** messages to initiate the execution of those arguments. There are three main types of arguments that may be executed.

2(a). In the case of an address, such as the **/B** executed by **S** ((ii) - (iii)) or the **/I4** executed by **R** ((viii) - (ix)), the executing activity is re-positioned at the addressed object. It continues executing there as though that object were the actual argument.

2(b). In the case of a data item, such as the **7** executed by **Q** ((ii) - (iii)) or the **0** executed by **S** ((iii) - (iv)), no further evaluation is possible. The executing activity just remains positioned at that object until a message is received specifying some action on it, such as **copy** (iii) or **replace-with** (v).

2(c). In the case of a subordinate instruction, such as the **(+ /B 1)** executed by **Q** ((viii) - (x)), the same steps are recursively applied in the execution of its operator, creating in (ix) the activities **S** and **T** as subordinates of **Q**. The complete tree structure of activities which is created by this recursive execution, and the positioning of those activities by the various addresses, is shown in (x). The operators being executed by **P**, **Q** and **R** are shown at the nodes of the activity tree. The activity tree structure corresponds to the infix structure **((B+1) * (B-C))** of the expression being evaluated.

3. After **P** has initiated the (concurrent) execution of the input arguments by subordinate activities, **Q** and **R**, their results are accessed by sending **copy** messages ((iii) and (ix)). Where the input argument is a data item, a copy of that item is returned

((iii) - (iv)). Where an argument is a subordinate instruction, **(+ /B 1)** or **(- /B /C)**, the result, **5** or **2**, of that instruction is returned ((ix) - (x)).

4. The returned values are then used to compute a result which is handled in one of two ways depending on whether an output argument was used. If there was an output argument then, as in (v) - (vi), the result, **4**, is sent with a **replace-with** message to the activity, **S**, which executed that argument, causing the result to replace the object at which that activity is positioned. A subsequent instruction may need to access the new value of the replaced object (in this example **i2** does so). To ensure correctness in such a circumstance **P** will not continue until the **replace-with** has taken effect, as indicated by an acknowledgement message from **S** (the empty object **()** is here used, arbitrarily, for the acknowledgement). If there was no output argument then, as in (x), the result, **10**, is returned. In (x) is shown the complete flow of returned values up the tree of activities recursively created by the execution of the nested instructions.

5. Finally subordinate activities terminate and **P** moves past the operator's arguments to execute the next operator in sequence (vi) or terminate if it is at the end of the sequence (xi).

In this model the organisation of an activity, **P**, is analogous to the organisation of a conventional processor. The activity's position in the program string corresponds to a processor's instruction counter. The other activities with which it can communicate correspond to registers each of which makes available a value or a location in addressable

storage. In this example the only activity "registers" used were those for an operator's operands (e.g. P's subordinates Q and R) and for the result (P's S subordinate or superior). The actual machine code (defined in Appendix A) provides for a larger number of subordinate activities which act as "general-purpose registers". Apart from accessing operands, a subordinate activity, $, can be used as the base for an address such as the **$/2/2** discussed earlier, with the remainder of the address being relative to the activity's position. In fact all addressing is relative to the position of an activity, the usual form of address, e.g. **/out/<-/2/2**, being relative to the initial position (the address itself) of the activity executing the address.

4.1.5. Further Examples

The concepts of program representation and execution in the RCF model were introduced above by an example based on the concepts of control flow and reduction introduced in Chapter Two. In that example an instruction's operation object is a conventional operator; each argument is (the address of) a simple data item or an instruction returning a simple data item; the communications between activities consist of storage access requests **copy** and **replace-with** (supporting load and store of the conventional operator/argument interface), **execute** requests (supporting demands of reduction's operator/argument interface), and simple data items returned in response to those requests. More generally, an operation or argument is a program fragment which organises the creation of subordinate activities, the sending of messages to subordinate and superior activities and the processing of messages received from them. A message communicated to an activity can be any structured or primitive

object, including a data item, a selector to re-position the activity and a specification of one of the elementary actions on an object, namely: **execute**, **replace-with**, **copy** (these three have been illustrated in Figure 27), **insert** (followed by an object which is inserted at the activity's position), and **take** (which is a destructive **copy**, deleting the object). The use of the model for different possible operator/argument interfaces will be discussed using two examples shown in Figures 28 and 29. The first of these uses the storage actions **take** and **insert** applied to streams of values as an alternative pair to **copy** and **replace** applied to simple memory cells. The second illustrates the use of actions other than the elementary ones directly supported by the architecture.

An alternative to the use of addressable memory cells for communicating data is the use of addressable streams of values. Whereas for a memory cell the data's producer replaces the value of the cell and the consumer copies the latest value, for a stream the producer inserts a value at the end of the stream and the consumer takes the first unused value from the front of the stream.

The example shown in Figure 28 illustrates the use of streams as operands and result of an addition instruction, **i1**. The addresses, **/A/in**, **/B/in** and **/C/in/end**, used in this instruction identify the first elements of the operand streams **A** and **B** and the end of the result stream **C**. The state of the streams prior to the execution of the instruction's operator by activity **P** is shown in (i). The operator stream+ behaves exactly as the usual addition operator + except that instead of using **copy** and **replace-with** to access arguments, **take** is used for inputs and

insert is used for output. Subordinate activities **Q**, **R** and **S** for the input and output arguments are created and positioned by the addresses as shown in (ii). In (ii) is also shown all the communication between **P** and its subordinates, namely **take** messages to access operands, the returned values **3** and **6**, and an **insert** message with the result **9** of the addition. (An **insert** does not generally require the acknowledgement used for **replace-with** in Figure 27(v).) In (iii) the first elements of the operand streams have been deleted by the **take** actions of **Q** and **R**, and the **9** has been inserted at the end of the result stream.

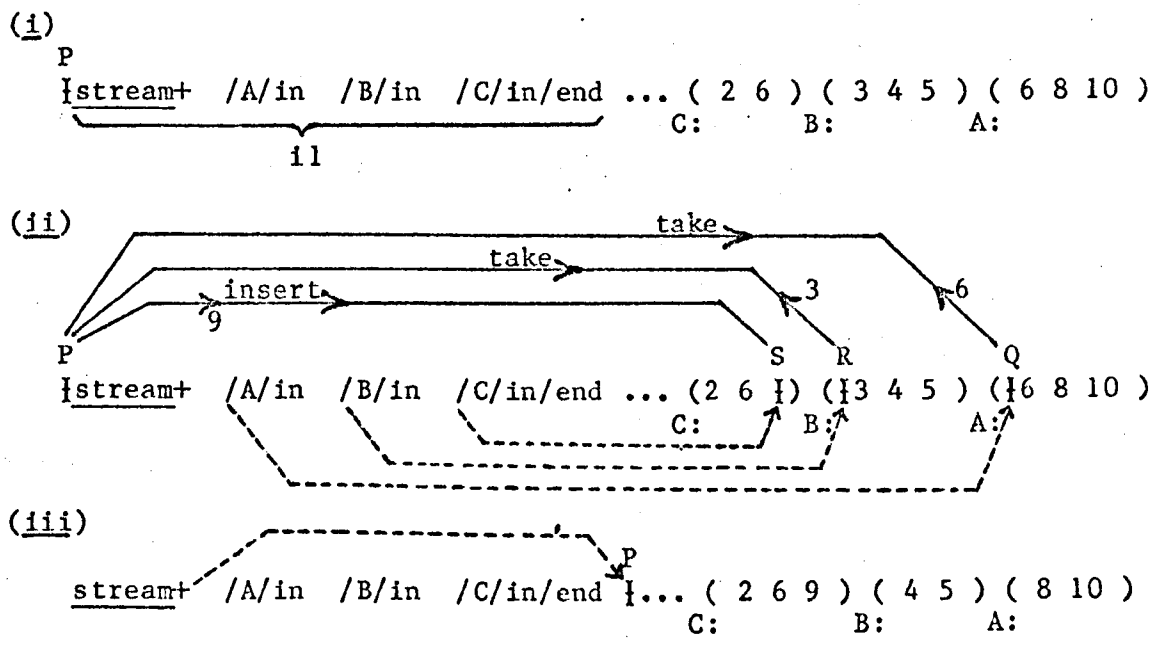


Figure 28 - An RCF Instruction which Operates on Streams

In the organisation of concurrent systems communication via streams, as in **i1** of Figure 28, and communication by updating shared memory, as in **i1** of Figure 27, can be considered as complementary mechanisms of equal importance. The UNIX operating system[56] supports both as primitives, in the form of files and "pipes", and there are

programming languages [58,59,18] in which both have equal status. In the next Section there will be particular examples of the use of the general concept of "stream operands" in the organisation of iteration and resource management.

The second example, in Figure 29, illustrates the possibility of instructions with complex operators and operands.

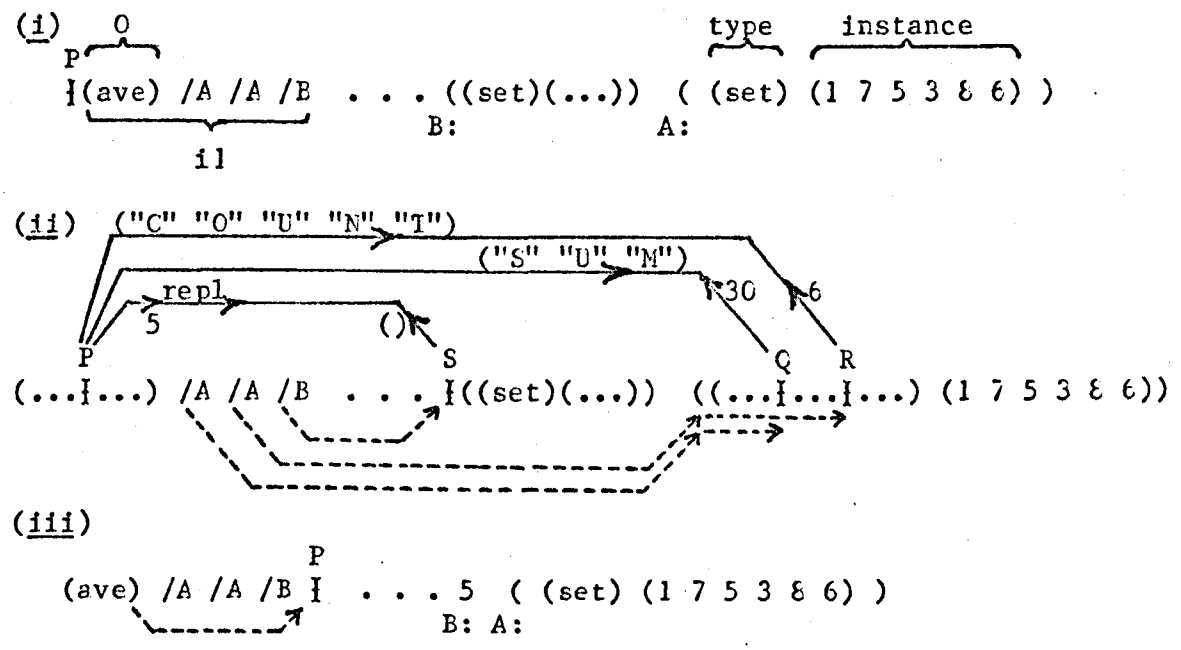


Figure 29 - An RCF Instruction with Programmed Operator and Arguments

The instruction **i1** has two input arguments and an output argument. Rather than being an elementary type of data item, such as an integer, an operand object, **A**, is of a program defined type, namely a set of numbers **(set)** for which the defined access actions include **sum** (returning the sum of all members), **count** (returning the number of members) and **replace-with** (which as usual replaces the whole object with a specified value). Rather than being an elementary operator, such as +, the opera-

tion **O** is a program defined operator **ave** which takes **set** operands, dividing the **sum** of the first operand by the **count** of the second so that if the two operands are the same set (as in this example) the result is its average.

In (i) activity **P** is about to execute the operator of the instruction **i1**. This operator is a program fragment comprising instructions to achieve the effects shown in (ii). As for an elementary operator, subordinate activities **Q** and **R** are created to execute the operands and are sent messages specifying actions on them. Here the messages are compound objects, with characters **"S"**, **"U"** etc. as components, specifying the actions **sum** and **count**, rather than primitive objects specifying the elementary **copy** or **take** actions used previously. The first component of the object **A** represents the **A**'s type, **set**, as a program fragment, executed by accessing activities **Q** and **R**. This type object comprises instructions to recognise messages such as the character string **("S" "U" "M")**, and implement the specified actions. (The second component of **A** contains the data specific to this particular instance of the **set** type, namely the actual set members, manipulated by the "type" program fragment.) The value, **30**, returned by **Q** is divided by the value, **6**, returned by **R** to produce the result, **5**. As for a standard operator, this result is sent to the subordinate activity **S** with a **replace-with** message which is acknowledged by a **()** message. The **set** program fragment executed by **S** responds to this message by replacing the whole of its containing object with the **5**, as shown in (iii).

The program structuring approach used in this example is similar to that used in "object-oriented" programming[60]. This approach is

intended to support modularity and flexibility in the construction of programs from more or less independently conceived program fragments. The basis for this approach is that the code for performing actions such as **count** on a data structure is included as part of the data structure, rather than being part of the program fragment using the data structure; and that there is a common framework for interactions between all operators and operands, whether elementary or program-defined. Examples of the resulting modularity and flexibility are that the program containing **O** can be used without modification on instances of any data types (e.g. lists and matrices) which support the required actions; changes in the implementation of the **set** data type (e.g. the **count** and **sum** being maintained as members are inserted, rather than calculated on each access) can be implemented by changes local to the "type" program fragment; an instance of the **set** type supporting the standard **replace**-with action could be used as the result destination of even an elementary operator.

4.1.6. Discussion

In all aspects of the RCF model, as in any recursive system, there is a general framework which embodies the essential concepts of the system and a set of relatively arbitrary primitives which are a basis for constructing more complex structures within that framework. The essential concept of storage is the grouping together of an arbitrary number of (primitive or compound) objects as a compound object. The essential concept of addressing is a sequence of selectors each being relative to the position identified by its predecessors in the sequence, or, for the first selector, relative to the pre-established position of an activity in the object structure (either the activity executing the address and

thus the position of the address itself, or a distinct "base" activity). For program representation the essential concept is an "instruction" being the application of a (primitive or compound) operation to (primitive or compound) arguments. For program execution the essential concept is the execution of objects by dynamically created trees of concurrent activities with communication along the arcs of a tree.

For storage and addressing a fairly minimal set of primitives have been adopted which although inadequate for a practical system are sufficient to illustrate the concepts of recursive storage and contextual addressing (and to initially investigate the utility and implementation of those concepts). The storage structure primitives are the delimiters **(** and **)** for constructing sequences of objects and the primitive objects such as integers, ultimately represented as delimited sequences of **0**s and **1**s. The addressing primitives are selectors identifying simple positional relationships (such as "prior") between objects. For program representation and execution a similarly minimal set of primitives would be: (i) the elementary actions of **copy**, **take**, **replace-with**, **insert** and **execute** applied to an object; (ii) simple facilities for the creation of subordinate activities and communication between related activities; (iii) data operations on communicated values, just **NOR** on uninterpreted bit-strings being sufficient; (iv) the composition of these primitives by sequential execution. Programming in the general style of Figure 29 requires that primitive machine operations such as creating and communicating with subordinate activities be directly available at the machine code interface. The earlier examples were discussed in terms of slightly higher level constructs, such as the operators **+**, ***** and stream+, which use those primitive machine operations in a standard way

of sufficient general use to be provided as elementary operators in a machine code instruction set. The following Section describes a complete set of such machine code constructs for the representation of programs organised in the particular styles of control flow, data flow and reduction.

4.2. Program Organisation

One of the main purposes of the RCF architecture is to allow programs to be organised in the styles of control flow, data flow and reduction programs. This Section discusses the way in which the basic control flow, data flow and reduction models are supported and then, as in previous Chapters, discusses data structures, conditionals, procedures, etc. First however it is necessary to describe the (machine-code level) programming constructs used in this Section, elaborating the basic concepts of program representation and execution introduced in the last Section.

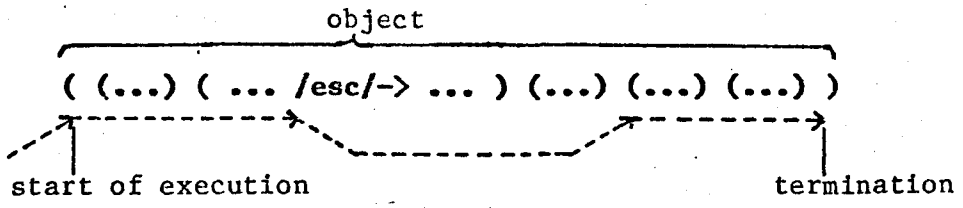
4.2.1. Notation

Structures -

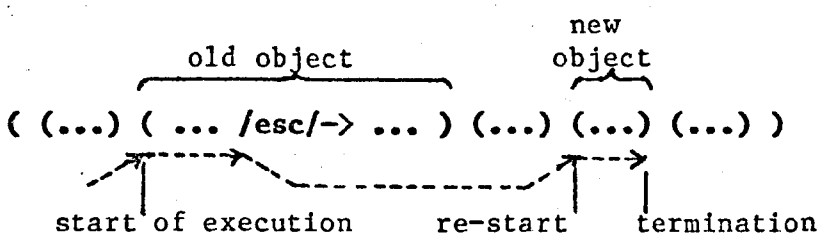
(Instr1 Instr2 (Instr31 Instr32) Instr4)

In executing a structured object its components are executed in sequence. Execution terminates at the final **)**. Internal delimiters (such as those containing **Instr31** and **Instr32**) are ignored and in particular the empty object **()** acts as a **NOOP**. Termination at the end of a structured object is a default that can be overriden by, for example, addresses which explicitly transfer control. This default requires a difference between transferring control within the object being executed and transferring control to a different object. Whilst the activity remains within its current object, reaching the end of that object will

terminate the activity -



If however an address takes the activity outside of its current object then it is the end of the addressed object which causes termination of the activity -



Data manipulation operators -

(i) **... operator input ... output ....**

(ii) **(... operator input ...)**

(iii) **... streamoperator input ... output ...**

The operator defines the number of input arguments (usually two) and output arguments (usually one), all of which are objects executed concurrently by subordinate activities. In the usual case (i) the operator uses the objects returned by the activities executing the input arguments or copies the objects identified by them, producing a result which replaces the object identified by the execution of the output argument. Execution then continues following the last argument used. In the case of an instruction with no output argument, a "return instruction" (ii), the result is instead returned to the superior activity. For a stream operator (iii), as in the stream+ of Figure 28, **take** and **insert** are used

instead of **copy** and **replace**. The data manipulation operators that will be used in this Section are: subtraction (-), addition (+), multiplication (*), identity (:=) for which the result is just the value of the single operand, comparison (=) for which the result is True if the two operands are identical and False otherwise, and conditional (**if**) which has three operands, the result being the second operand if the first operand is True, and the third operand is the first operand is False.

Data items -

integer

boolean

.

The executing activity stops at the data item, identifying it for subsequent use by the superior activity. If there is no superior activity then the activity executing the data item just terminates. The . (stop) represents an arbitrary data item included only for its effect of stopping the executing activity.

Addressing -

(i) **/sel/sel/.../sel**

(ii) **$/sel/sel.../sel**

(iii) **set$ argument**

As discussed earlier, a sequence of selectors (**/sels**) forming an address acts as a branch instruction, re-positioning the executing activity at the addressed object. An address is either "self-relative" (i), i.e. relative to its own position in the program, or "base-relative" (ii), i.e. relative to a previously selected base position. The $ prefix in a

base-relative address is a form of selector which identifies the context for the remainder of the address as being the position of a subordinates activity of the particular activity executing the address. (For the purposes of this Section only one base, identified by $, is needed, however in the machine code there may be several subordinates providing different bases identified as **$1**, **$2** etc.) The **set$** operator (iii) is used to select the base position for the second form of address. It has a single argument (an address) which positions a subordinate activity at an object for use as a base in subsequent addresses.

Concurrency control -

(i) ... par **Ob1** par **Ob2** par ...

(ii) ... (? ...) ...

(iii) ... (excl ...) ...

The parallel operator par (i) supports the parallel control mechanism in which program fragments are executed independently. An independent activity is created and positioned at the following object, **Ob1**, whilst the original activity continues at the object following that. If (as in (i)) pars are used to separate the program fragment components **Ob1**, **Ob2** etc., of a structure then those components are all executed in parallel). There are two constructs for synchronising concurrent activities. The first (ii) is an "unknown", represented as a **?**, similar to that used in the data flow model. An unknown within an object, say **Ob**, causes the executing activity to be suspended to the immediate left of **Ob**. It remains suspended until it is no longer adjacent to **Ob**, i.e. until **Ob** is deleted or replaced, or a new object is inserted before it. Execution then continues normally with the object following the deleted

object, or with the replacement or inserted object. The second construct (iii) is an "exclusion" argument, excl, which prevents simultaneous execution of an object by more than one activity. When first executed the excl turns into a ?. Thus the first activity to attempt to execute an object containing a excl will succeed, but subsequent activities will be suspended by the ? resulting from the first execution.

Evaluation control -

(i) **/sel/sel ... /sel"**

(ii) **"object**

(iii) **... /sel/seleval/sel ...**

(iv) **eval operand**

The default is that an argument is executed and similarly that the object finally identified by the last selector in an address argument is itself executed (although the objects identified by preceding selectors in the address are not executed). These defaults can be overridden as indicated by the use of (underlined) quote (") and eval constructs. An address with a quote suffix for its last selector (i) (which will be referred to as a quoted address) indicates that the finally addressed object is to be treated as though it were a data item - the activity executing the address is just positioned at the addressed object rather than executing it. Similarly an object with a quote prefix (ii) (referred to as a quoted object) is itself treated as a data item rather than being executed. The inhibition of evaluation produced by the use of quotes is complemented by an evaluate construct, eval, which forces extra evaluation. This can be included as a suffix to a selector within an address(iii). The object, say Ob, identified by the selector

sequence up to that point will be itself executed by that activity, say P, which is executing the address. Typically Ob will itself be an address to re-position P and that new position becomes the context for the next selector in the original address. The eval construct can also be used as an operator (iv). The single input argument is executed by a subordinate activity to produce the actual operand in the normal way. That operand (e.g. a computed address) is then itself executed, just as though it were in the place of the eval operator.

Structures, operators, data items and addressing have been discussed at some length in the preceding Section. However the newly introduced constructs for controlling concurrency and evaluation require some discussion before their uses are illustrated in the remainder of this Section.

The unknown and exclusion arguments for controlling concurrency each support a particular synchronisation structure in which one activity delays the execution of its program fragment until another activity completes the execution of its program fragment. In the case of the exclusion argument the two activities are executing the same program fragment - this is the synchronisation structure used with the sequential control mechanism (typically with multiple processes executing a shared program). In the case of the unknown argument the two activities are executing different program fragments (e.g. two data flow instructions) - this is the synchronisation structure used with the parallel control mechanism. The third synchronisation structure is the implicit synchronisation of an operator waiting for the return of an operand. In this case the two activities are executing nested program

fragments - this is the synchronisation structure used with the recursive control mechanism.

The evaluation control constructs (" and eval) support the important requirement that an executable program fragment can also be treated as data, as for example in editing a program, and vice-versa, as for example in executing the data produced by a compiler. In a control flow architecture the content of a memory cell is treated as data if used as an operand for an instruction, but is executed if that memory cell is encountered in the flow of control. Most data flow architectures maintain a strict separation between the program (graph) and the data (tokens), and consequently most data flow machines cannot be used to compile or edit their own programs. In reduction everything is normally executed and there are special operators such as the **QUOTE** and **EVAL** operators of LISP[25] for explicitly inhibiting and forcing execution. For the RCF architecture the only essential is that there be the ability to control precisely execution of addressed objects using the eval suffix to force execution and the " suffix to prevent execution. (A normal address without selector suffixes, **/s1/s2/s3**, is equivalent to **/s1"/s2"/s3**eval, i.e. the final addressed object is executed but intermediate objects selected are not.) The other two constructs, the eval operator and a " prefixing an object, correspond to the **EVAL** and **QUOTE** operators of reduction and are included for their convenience. The effect of using a quoted object could be achieved by using the quoted address of the object - for example executing **("Ob)** has the same effect as executing **((/esc") Ob)**, i.e. the executing activity is just positioned at **Ob**. The effect of the eval operator could be achieved by storing the operand returned by its argument and transferring control to

that operand (as is done in control flow).

4.2.2. Including Control Flow, Data Flow and Reduction

Figure 30 shows six different organisations for the example expression **a = (b+1)*(b-c)** the first five being organised as in one of the specific models covered in Chapter Two. The conventional control flow program (i) has essentially the same representation as in the standard control flow model (originally illustrated in Figure 1) and requires no further explanation, other than to observe that quoted addresses are used in order to reflect conventional semantics.

In parallel control flow and data flow (as originally illustrated in Figures 3 and 4) all instructions are executed concurrently and communicate by passing tokens. In (ii) and (iii) parallel operators (par) are used as separators between objects to create the required concurrency and the execution of each object is explicitly terminated by a final **.** The need for a "token" is represented by an unknown, **(?)**, to suspend execution until the token arrives. In (ii) each unknown represents the need for a "control token". Such unknowns precede the instruction's main operator, thus completely suspending execution of the rest of the object. Each control token is communicated by an instruction such as **1** which replaces a specific **(?)** in another object by a **()** (i.e. a NOOP) to remove one synchronisation constraint on the execution of that object. For this instruction, **:= "() /X**, the output argument is the address of the unknown; the single input argument is a quoted object so that the actual operand is that object, **()**; the operator is the identity operator, **:=**, so that the result with which the addressed object is

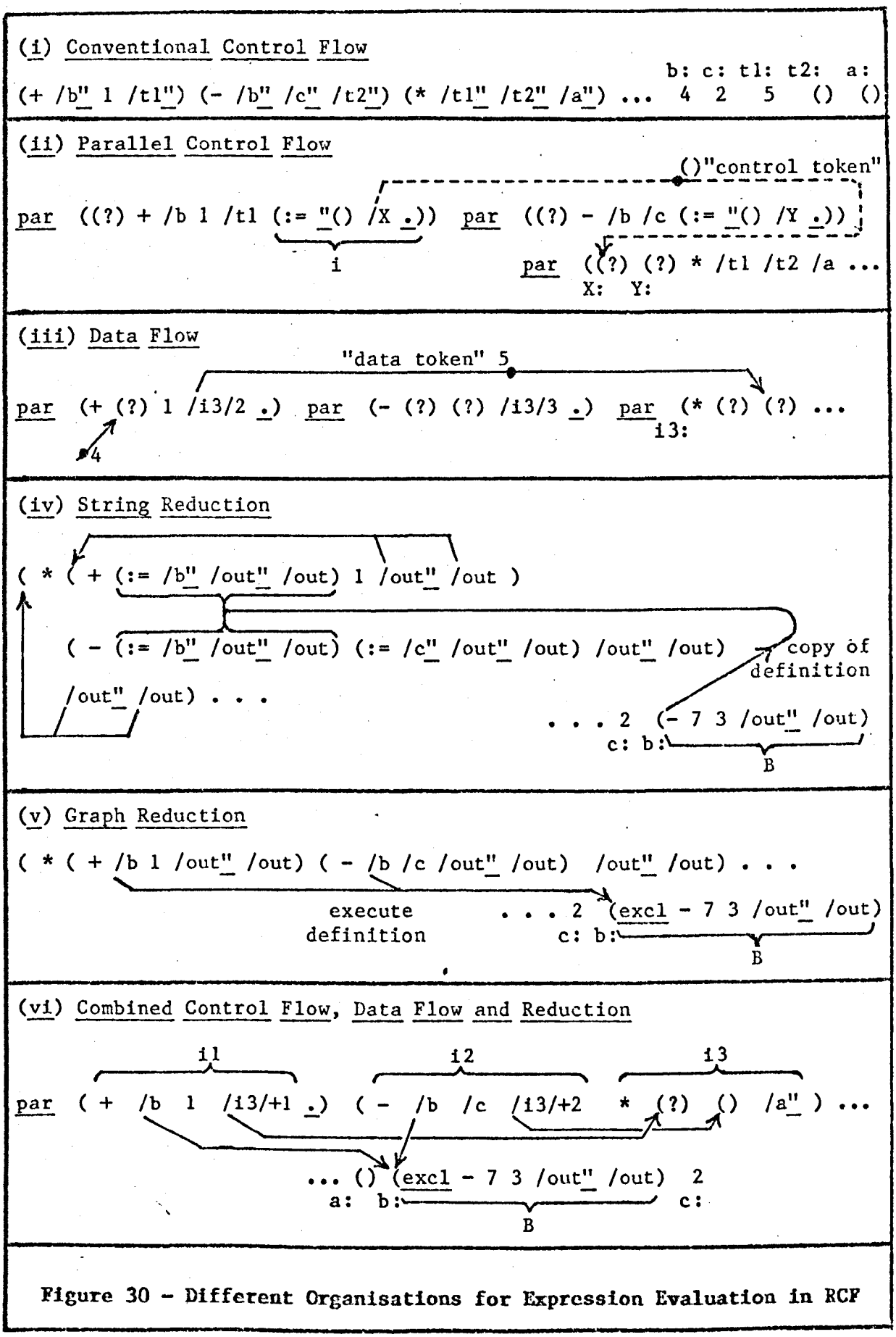


Figure 30 - Different Organisations for Expression Evaluation in RCF

replaced is just the operand. In (iii) each unknown represents the need for a "data token" and forms a particular argument of an instruction, thus suspending execution of just that argument. Each data token is an instruction's result stored at the appropriate argument position in the instruction using that result.

In string and graph reduction (as originally illustrated in Figures 5 and 6) an instruction is an expression which replaces itself with its result. That result is then executed (in general the result may be a further expression which again replaces itself with its result). In (iv) and (v) reduction expressions are represented by objects of the form **(operator operand operand /out" /out)**. This includes two single-selector, **/out**, addresses which address the object itself. The first such address is the output argument for the operator, causing the result to replace the whole object (this must be a quoted address to specify the object itself rather than cause its execution). The second such address is a simple branch instruction causing execution to continue with the result.

String and graph reduction treat referenced definitions such as B in different ways. In string reduction a copy of the definition replaces the reference and is then executed. This form of self-replacing reference is represented in (iv) by an explicit instruction **(:= /b" /out" /out)**. (This instruction has the form of reduction expression with an identity operator for which the input argument is the quoted address of the referenced object, B, so that the operand is B itself rather than the result of executing it.) In graph reduction the referenced definition is executed in place and so a simple address, **/b**,

is used to reference it in (v). Graph reduction is able to support lazy evaluation where a shared expression is only evaluated once. In (v) lazy evaluation of **B** is achieved by preceding its operator with an exclusion argument, excl, to prevent multiple execution. The first activity to execute **B** will change the excl to a **?** and continue executing the operator. Any subsequent activities will encounter the **?** and thus be suspended at **B** until it is eventually replaced by the result of the operator.

A possible combination of these different organisations is used in (vi). The independent instructions **i1** and **i2** are executed in parallel as a result of the initial par operator. In contrast, **i2** and **i3** (which is dependent on **i2**'s result) are executed sequentially. The shared sub-expression, **B**, is organised for lazy evaluation as in graph reduction. Both **i1** and **i2** store their results directly into the input arguments of **i3** which is the only user of those results. The argument in **i3** for the result of **i1** is an unknown to synchronise those two instructions. In contrast, the argument for the result of **i2** is just the empty object, acting as a place-holder, since synchronisation is unnecessary.

The RCF model is based on the conventional control flow model with implicit sequential execution and explicit transfers of control, and the ability to address and manipulate any storage location explicitly. The inclusion of data flow and reduction depends on two generalisations of the conventional model. Firstly there is the generality of information structure and addressing. This allows an instruction to address another instruction's input argument as a storage location for its result, as required for data flow. It also allows a large object such as a struc-

ture of nested expressions to be directly replaced with its result, as required for reduction. Secondly there is the generalisation of including execute as well as load and store for access to arguments, and the convention of using the more flexible execute for initial access to arguments. Consequently control over the interpretation of an argument resides in the argument rather than the operator using it, allowing an argument to be not only a literal or address, as required for control flow, but alternatively a subordinate instruction, as required for reduction, or a synchronisation primitive, as required for data flow. The approach of synthesising control flow, data flow and reduction as subsets of a more general model allows considerable flexibility in the way these three are combined, as illustrated in (vi) above. This approach also gives orthogonality between the various elements of program representation, as illustrated by all the above examples using the same multiply operator which is independent of the way its operands are organised. An alternative approach to combining these three models of, say, providing three different classes of instructions, would limit this flexibility and orthogonality and lead to a generally more complex architecture.

It is worth reflecting on the choice of control flow as the basis for a generalisation synthesising control flow, data flow and reduction. The advantage of control flow for this purpose is its low-level operational model and the separation between control and data which allow maximum choice in how a program is organised. Data flow and reduction are higher level models incorporating particular assumptions of program organisation, particularly that control of instruction activation is tied to the availability of input data or the need for output data. The

data flow notion of activation by data availability can be generalised to include the flow of control, (as was done in the DCF architecture) by including "control tokens" as "null" data items. It is however difficult to see how to generalise the data flow model to conveniently include activation by need, or to conveniently generalise the latter to include activation by the flow of control or data availability.

The motivation for including control flow, data flow and reduction within a more general model is to obtain the particular advantages of each, which were identified in Chapter Two. Each model in itself provides particular program organisation benefits, specifically in its particular way of controlling instruction activation. In some circumstances the explicit activation of instructions provided by control flow is an advantage, whereas in other circumstances the implicit activation of instructions by the availability of data, as in data flow, or by the need for data, as in reduction, is an advantage. The different organisations used in Figure 30 allow these different activation mechanisms to be used and thus the advantages of each to be obtained where needed. Also for graph reduction there are particular advantages of being able to use higher order functions and unbounded data structures as program organisation constructs. The use of these constructs within the RCF model will be discussed later in this Section.

Apart from program organisational advantages each model has particular advantages concerning the performance of particular types of programs on computers implementing the model. The RCF architecture, and its implementation discussed in the next Chapter, is principally aimed at exploiting concurrency. Thus it would not be very effective for a

program predominantly based on the conventional sequential control flow model. However, the basic sequential control mechanism of an activity moving from one instruction to its immediate successor is the basis of the architecture and the use of short control flow sequences of instructions is an effective way to program the architecture. The data flow and reduction models are implicitly concurrent and thus more suitable for the overall organisation of programs for the RCF architecture. Although the main motivation for the reduction model is its program organisational advantages, there are some performance advantages. Principally there is the advantage of lazy evaluation in graph reduction which prevents processing resources being used on unnecessary computation. The graph reduction form of program organisation used in Figure 30(v) achieves this effect. For a shared expression object of the form **(excl op (...A...) (...B...) /out" /out)** there may be several activities positioned at the object. However only one of those will actually evaluate the shared expression, which might be very large. The principal motivation for the data flow model is the exploitation of concurrency for improved performance. The execution of a data flow style program is discussed in some detail in the next Chapter on Implementation.

The examples in Figure 30 have illustrated the use of the various programming constructs, introduced at the start of this Section, for the organisation of control flow, data flow and reduction programs at the small scale of expressions on simple data items. The remainder of this Section illustrates their use in the larger scale organisation of programs.

4.2.3. Data Structures

One of the important principles of recursive architecture is that an operand can equally be a simple data item or a data structure such as an array. The representation of data structures must ensure that the execution of a data structure operand is compatible with the execution of a simple data item, that is the executing activity must be positioned at the data object. This is illustrated in the following example where two operands, the first of which is a data structure, are compared for equality by the = operator -

```
                a:
(¦ =  ¦ (/out" 1 2 3) ¦7)
 P    Q              R
```

Within the data structure **a**, the first component acts as type information declaring **a** to be a compound data item. This "type" is the quoted address of **a** itself, that is the **/out** selector quoted to prevent re-execution of **a**. When activity **Q** executes this quoted address it will be positioned at **a** which will thus be used in its entirety by the operator. (It is important that, as a result of contextual addressing, all data structures can have the same **/out"** first component, and thus there is correct comparison of structures without this detail of structure representation being incorporated into comparison operators.)

A data structure using this representation can be communicated between instructions in any way that a simple data item can. As in control flow and graph reduction, a data structure, or its components, can be addressed by the instructions sharing it. As in data flow, a data structure can be passed as a "data token" replacing a **(?)** as an

instruction's argument. As in string reduction, a data structure generated as a result can replace the instruction generating it as the operand for a containing instruction.

An important characteristic of graph reduction is that it allows the use of unbounded (notionally infinite) data structures which are incrementally generated as the components are needed. Figure 31 shows a possible RCF representation of such a data structure, **L**, and an accessing expression, **E**.

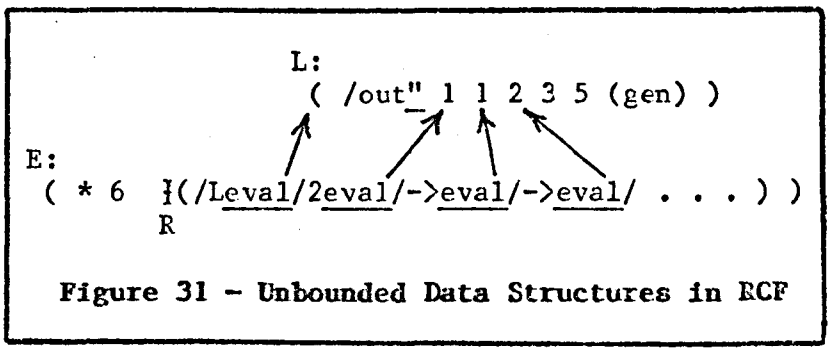


Figure 31 - Unbounded Data Structures in RCF

L is a partially generated list of the fibonacci numbers with the infinite tail of the list represented by a program fragment **gen**. On each execution of **gen** it will insert to its left the sum of the preceding two numbers and position the executing activity at that newly inserted object. The expression **E** contains as its second operand (executed by activity **R**) the address of an as yet un-generated element of the list. This address is a sequence of selectors to identify first **L** itself, then its second component (the first genuine element of the list), then the next component etc. Each of these selectors has the eval suffix so that the object which it identifies is executed by R before the next selector is applied. If the object is an actual data item (or data structure), as in the first few cases, then the effect is the same as if a normal

selector had been used (rather than an eval selector). That is, activity **R** will just be positioned at the object. If however the object is the **gen** program fragment then it will be executed by **R** with the result that the next component of the list is generated. **R** is positioned at that new component where it will either apply the next selector in the address (thus generating the next component) or be used by its superior to access that component.

This scheme generalises to any level of nested lists. For example if **L** were a list of the rows in Pascal's triangle then **gen** would not insert the actual data structure representing a row. Instead it would insert an object to incrementally generate the elements of the row on demand. The inserted object would have the form **(/out" subgen)** where **subgen** implements the algorithm for generating the next element in a row from the other elements of the triangle. The address of, say, the 2nd element of the fourth row would be -

```
/Leval/2eval/->eval/->eval/->eval/2eval/->eval
 \___/ \_____________________________/ \_____/
triangle          4th row            2nd element
```

Execution of this address causes generation of only those elements of the triangle needed to compute the addressed element.

4.2.4. Conditionals

In the RCF architecture the conditional operator can be used to support various forms of conditionals including those discussed in Chapter Two. This operator **if** has three operands all of which are evaluated concurrently. Its result is the second or third operand

depending on whether the first operand is True or False. Figure 32 shows three uses of this operator for the different control flow, data flow and reduction organisations of the conditional structure **x:= if c then ra else rb** where **ra** and **rb** are the result values of program fragments **(...A...)** and **(...B...)**.

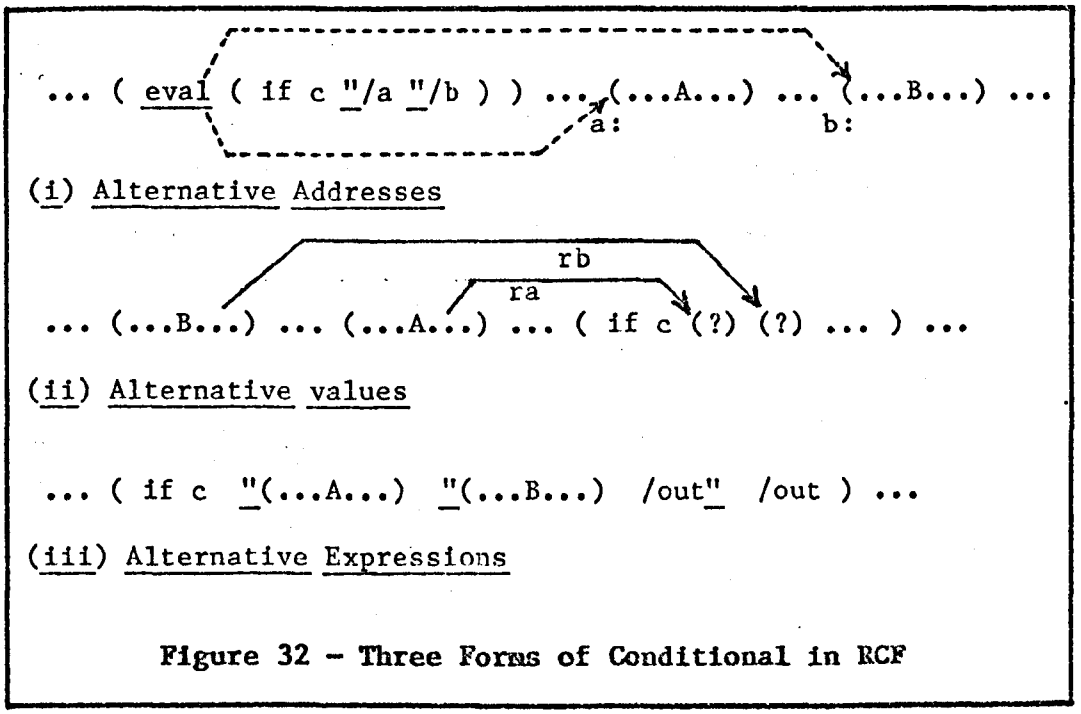


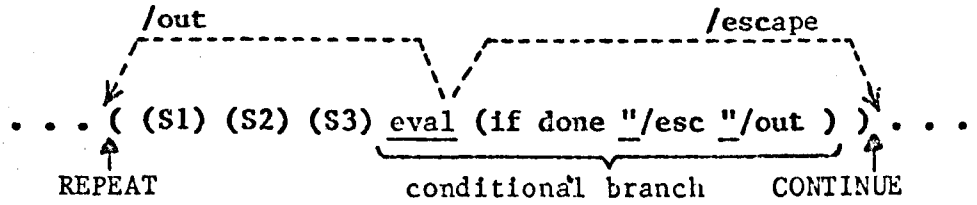
Figure 32 - Three Forms of Conditional in RCF

In (i) the control flow organisation is used. This illustrates the typical use of the eval operator to force execution, in conjunction with the inhibition of execution by "s. Here the conditional operator uses quoted objects ("/a and "/b) as its input arguments. Consequently the objects themselves (addresses **/a** and **/b**) are the actual operands and the result is one or other of those addresses. This address result is the operand of an eval operator, causing that conditionally selected address to be executed. The complete instruction thus has the effect of a control flow conditional branch instruction to execute either **(...A...)** or **(...B...)**. In (ii) the data flow organisation is used. Here the

conditional returns one of two alternative values, **ra** or **rb**, which are provided as "data tokens" from instructions in **(...A...)** and **(...B...)**, giving the effect of data flow "switch in" instruction. In (iii) the reduction organisation is used. Here the conditional returns one of two alternative expressions, **(...A...)** or **(...B...)**, which replaces the complete instruction, as in reduction. As in (i) the expressions forming the operands of the conditional operator are quoted objects so that the actual expressions are used rather than the results of evaluating them.

4.2.5. Iteration

Iterative execution is obtained by a **/out** single-selector address being executed at the end of any repeated object. For a control flow style iteration a whole sequential structure of instructions is repeated and there would be a conditional branch instruction (as in Figure 32(i)) controlling the iteration. For example **DO S1; S2; S3 UNTIL done** might be represented as -



Here the **/out** positions the executing activity at REPEAT whereas the **/escape** positions it at CONTINUE. For a data flow style iteration each instruction is separately repeated by terminating it with a **/out** -

. . . par (**operator operand operand result /out**) par . . .

One way of organising data flow iteration discussed in Chapter Two is that in which an operator's arguments are streams of tokens, as in the DDM1 data flow architecture[27]. Figure 33 shows this technique, using as an example **e:= (a+b)-6** which is repeatedly executed for sequences of values **a1, a2, a3,** etc., and **b1, b2, b3,** etc.

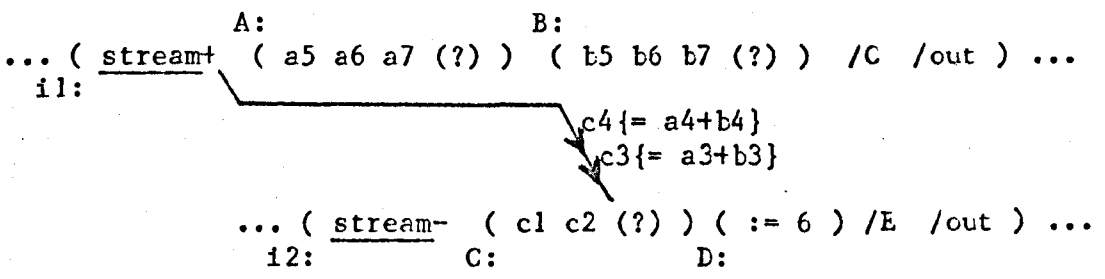


Figure 33 - Iteration Using FIFO Queues in RCF

There are two instructions, **i1** and **i2.** Instruction **i1** repeatedly performs additions to produce results for repeated subtractions performed by instruction **i2.** Each input argument of instruction **i1** is a stream of operand values terminated by an unknown. The subordinate activity executing an input argument will either provide the first item in the operand stream, e.g. **a5,** as the operand or (if the stream is empty) be suspended by the terminating **(?)** until an item is inserted. The operator stream+ is a stream operator, as previously used in Figure 28, with the input arguments being accessed by destructive **take** rather than **copy** operations, and the result being inserted at the position identified by the result argument rather than replacing the object there. Thus on each execution of **i1** a pair of operand values is taken from its operand streams and the result is added to the end of **i2's** operand stream. Instruction **i2** has the same structure except that one operand is an

embedded literal, 6, for which there is an instruction, (:= 6), to return that value on each execution.

In reduction there is no special organisation for iteration, this being achieved entirely by recursive procedures.

4.2.6. Procedures

Procedures can be of two different types in the RCF architecture. These are a replicated procedure where the procedure definition is copied by each call and executed in the context of the call, and a re-entrant procedure where the procedure definition is executed in place by possibly several independent concurrent calls. Procedures using data flow and reduction style instructions, which are modified during execution, would need to be copied, whereas those using control flow style instructions, which are re-entrant, need not be. For both replicated and re-entrant procedures it is generally necessary to support both "dynamic binding" where the procedure definition contains addresses relative to the context of the call (e.g. for accessing parameters) and "static binding" where the definition contains addresses relative to the context of the definition (e.g. for calling other procedures or modifying "own variables").

Figure 34 illustrates the two types of procedure for the same example, a call on procedure **F**. This call has the same effect as a + operator (which the called procedure uses). In both cases the procedure call instruction consists of an operation object, the **call**, followed by parameter arguments **p1**, **p2** and **p3**. As for a simple + operator, the **p1** and **p2** arguments are executed to provide the operands, the **p3** argument

is executed to identify the position at which the result is stored, and execution continues at S, immediately after those arguments. The **call** itself is constructed as an **apply** operator which is used as though it were an elementary operator with the procedure address **/F** as its single operand.

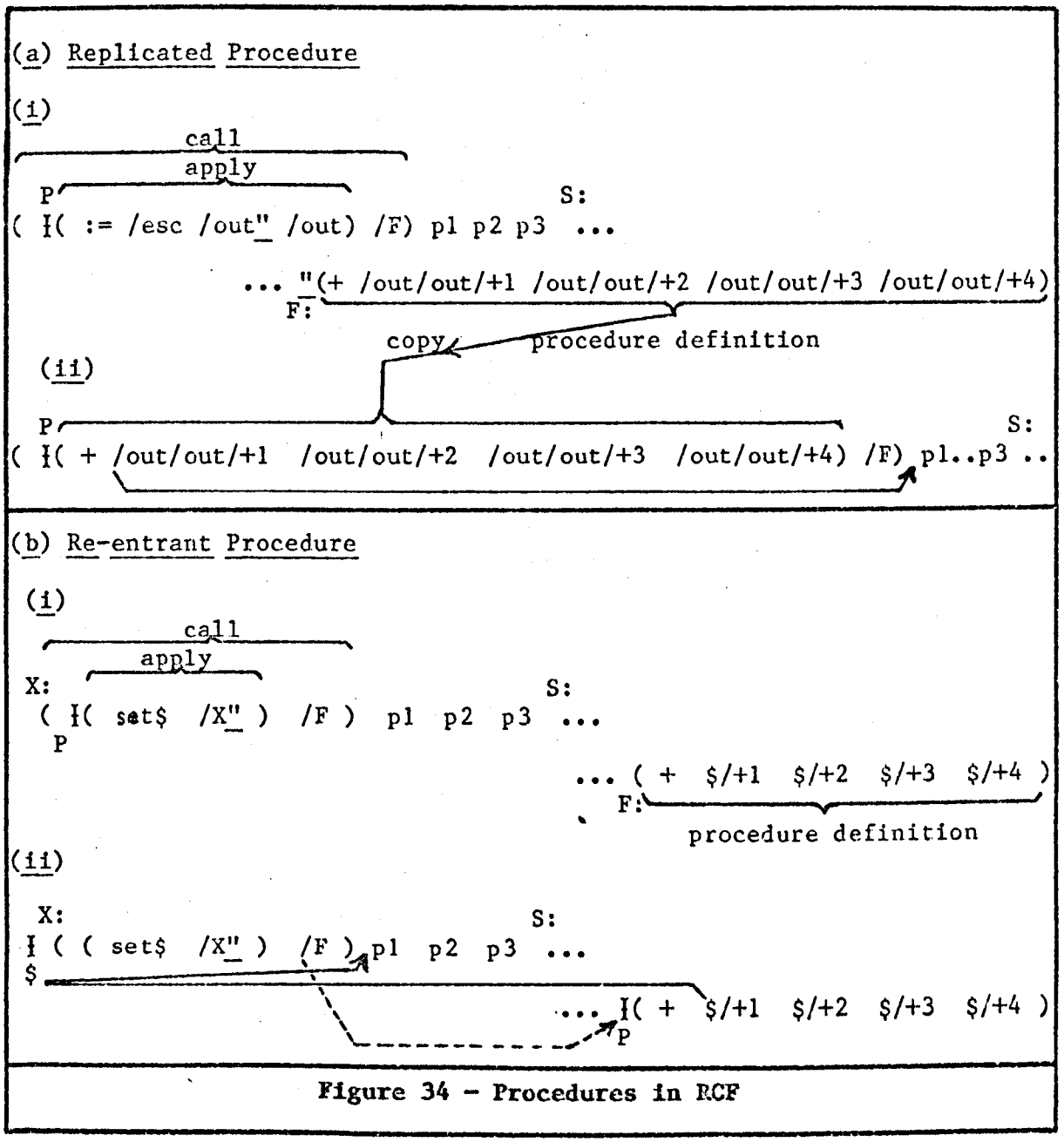


Figure 34 - Procedures in RCF

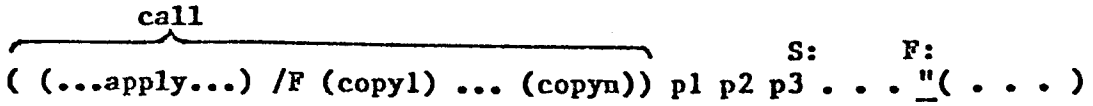
For the replicated procedure (a) the **apply** instruction replaces itself with and then executes a copy of the addressed procedure, as in string reduction. The procedure definition is a quoted object as otherwise it would be executed where it is rather than being copied to the call for execution there. The effect of an activity **P** executing the call (i) is shown in (ii). The replicated procedure is executed in the context of the call. Consequently normal addressing, such as the **/out/out/+1** to access parameter **p1**, gives dynamic binding. To support static binding (not actually needed in this case) the procedure address is retained in (ii) as an extra parameter which the procedure code could use for addressing objects in the context of its definition (as will be illustrated in the following discussion of higher-order procedures).

For the re-entrant procedure (b) the activity **P** executing the **call** (i) is re-positioned at the procedure definition and executes it in place (ii). Thus normal addresses in the procedure definition give static binding. The dynamic binding needed to access the parameters is provided by the base-relative addresses such as **$/+1** which address parameters relative to the position of the activity **$**. This activity is a subordinate of activity **P**, set up as a result of **P** executing the **apply** instruction. That instruction's operator is **set$** and its operand is the quoted address of the call, **X**. The activity **$** is positioned at the addressed object to provide the necessary context for subsequent addressing from within the called procedure. Thus in this case the **apply** has a function similar to that of an instruction to load the return address onto the stack in a conventional call sequence. After executing the **apply**, **P** executes the address **/F** re-positioning it at the procedure definition. In general the procedure will be a nested program

structure executed by a sub-tree of activities with P as its root. Nested objects in such a structure would need to address the procedure's parameters. Thus an activity executing one of those objects needs to inherit P's $ activity to provide the required context for parameter addresses. (In the machine code this inheritance is achieved by a default mechanism - if when an activity executes a **$/...** form of address it does not itself have a $ subordinate then that of its superior is used, or that of its superior's superior etc.)

The replicated procedure mechanism is simpler than the re-entrant procedure mechanism in that it does not require the relatively sophisticated base-relative type of addressing. The form of procedure call in (a) for a replicated procedure is self-modifying in a way that prevents it from being used in a program that is intended to be re-entrant. It is in fact possible to construct a call instruction for a replicated procedure which can be part of a re-entrant program fragment. This would be achieved by a change to the **apply** instruction in (a) to insert the procedure copy at the end of the **call** object, rather than overwriting the apply instruction. Thus if there are several concurrent executions of the **call** then there would be multiple copies of the definition

-



These copies are all hidden within the **call** object and have no effect on the surrounding program structure. For example the correct interpretation of a parameter address within a copy is not affected by the unknown number of other copies.

Similarly, a procedure executed in place as in (b) could contain self-modifying instructions, which would require that the procedure definition insert and execute a copy of itself, adjacent to itself. Thus whether or not procedures are copied to the context of the call does not necessarily depend on whether self-modifying or re-entrant instructions are used but can be determined by other criteria. For example if data accesses (to parameters) in the context of the call predominate over accesses (to "own variables") in the context of the definition then it would be appropriate to copy the procedure to the calling context.

4.2.7. Higher-Order Procedures

One of the major features of (graph) reduction is its ability to support higher-order procedures (functions). The principal benefit of these is that for a multi-parameter procedure, e.g. **H(p1, p2, p3)**, successive parameter expressions, **P1 - P3**, can be bound to the procedure definition at different points, giving a progression of more particularised procedures -

H(p1, p2, p3) ≡ some function of three parameters

G(p2, p3) ≡ **H(P1)**

F(p3) ≡ **G(P2)**

E() ≡ **F(p3)** ≡ **((H (P1)) (P2)) (P3))**

Figure 35 illustrates the way this procedure structure might be represented in RCF. It also shows a call, **X**, of the final, parameterless, procedure **E**. The base procedure **H** is an expression involving all three parameters. Each of the intermediate procedures **G**, **F** and **E**, contains

three components. The first components form a chain of addresses which leads to H and is used in executing the actual call, X. The second component is the particular parameter which that intermediate procedure provides for H. Each such parameter, e.g. p1 in G, is the address of the actual parameter expression, P1, which will generally use and be used by other expressions in the neighbourhood of G. The third component is the (quoted) address of the next more general procedure. These addresses form a chain which is used by H in accessing parameters.

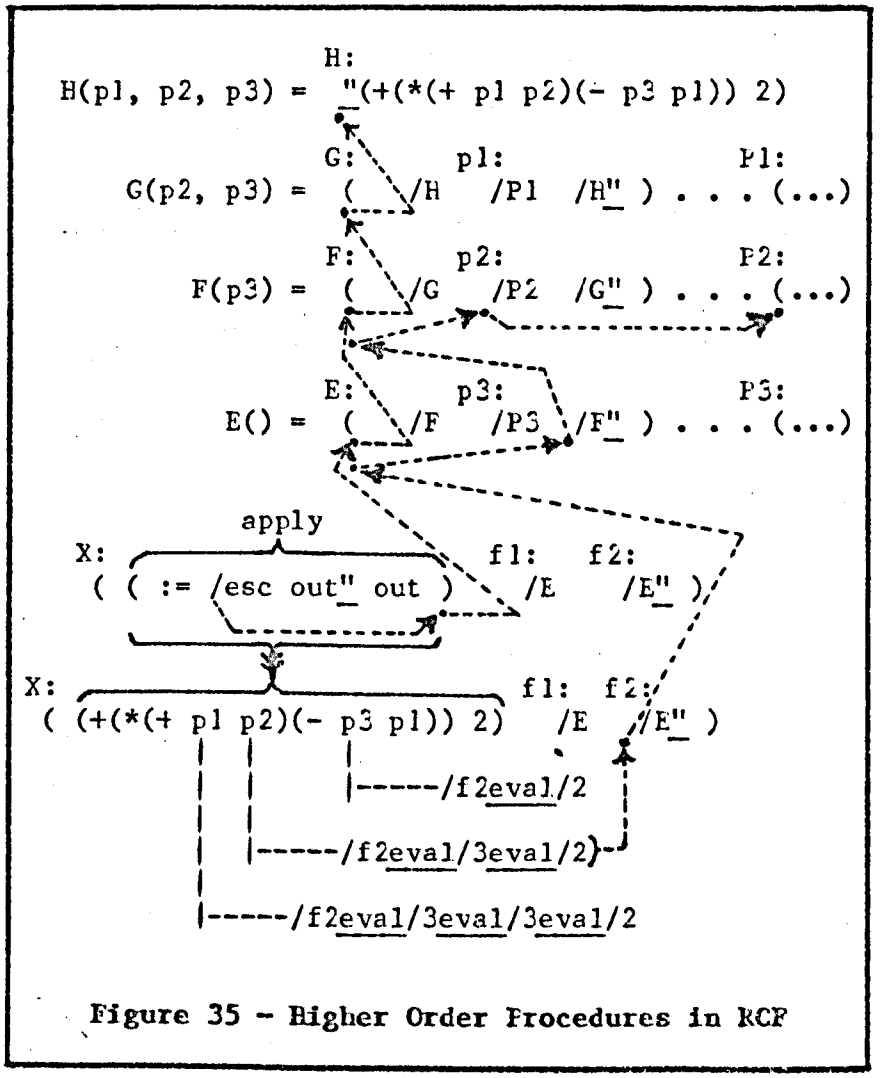


Figure 35 - Higher Order Procedures in RCP

The call, X, is very similar to the call used for a replicated pro-

cedure in Figure 34(a). It comprises the same **apply** instruction which when executed replaces itself with the procedure object identified by the execution of the following argument, namely the address **/E**. The activity executing **/E** will execute the addressed object, **E**, and thus its first component, namely the address **/F** of **F**. Similarly the first components of **F**, and then **G**, are then executed. Thus the activity follows the chain of addresses back to **H**, and so it is a copy of that procedure which replaces the apply instruction in **X** and is executed there.

As for each intermediate procedure, the last component of **X** is an address used by **H** to access its parameters. The address **p2** for the second parameter **P2**, executed from within the copy of **H**, is -

/f2eval**/3**eval**/2**

The first part of the address identifies **f2**, the address of **E**, which is executed (as a result of the eval suffix), thus identifying **E**. The second part identifies **E**'s third component, the address of **F**, which is also executed, thus identifying **F**. The final part identifies **F**'s second component, the address of the desired parameter, which is the final destination of the complete address. The activity executing the original address is thus positioned to execute the address, within **G**, of the parameter and thus executes the parameter in place. That parameter will be a graph reduction type of expression, replacing itself with and then returning its result. The addresses for the other two parameters are similar, each having a different number of /3eval parts depending on the distance along the addressing chain of the required parameter.

This representation uses two address chains, running through the

first and third components of the intermediate procedures. The first chain, used in the call, is structured in such a way that it is always implicitly followed to its end. Thus the program fragment forming the call, **X**, is independent of the length of the chain. For example **X**, and intermediate procedures **F** and **G**, would be unaffected if, say, **H** were replaced with a further intermediate call of another function with one more parameter. However, the chain for accessing parameters is constructed in such a way that a parameter address in **H** can go as far along the chain as is necessary to access the particular parameter required. Thus the program fragment forming **H** is dependent on the length of the chain, necessarily so since that length is the number of parameters expected by the procedure. The representation used for these chains also illustrates the use of the eval selector suffix for achieving quite sophisticated addressing structures. As used here, an eval selector corresponds to an indirect address where the addressed object is itself an address. This RCF notion of indirect addressing is actually more powerful than that in conventional architectures since the addressed object is executed and thus can be not only an actual address but also a program fragment to achieve the effect of an address. For example the object **F** could have been a (lazily evaluated) procedure call, say **Bind1(G)** where **Bind1(g) ≡ g(P2)**. When first executed as a result of a call such as **X**, this call would replace itself with the required structure.

The above scheme for procedures achieves most of the benefits of graph reduction for this type of example. Specifically, different parameter expressions can be bound to the procedure at different points in the program structure, and there can be lazy evaluation of those

expressions. For example if **P2** had been previously evaluated by some other construct in the neighbourhood of **F**, then that expression would have been replaced by its result, and thus not evaluated again by the call, **X**. The main shortcoming of this scheme for the RCF model, compared with one based on a purely graph reduction model[43], is the complete copying of the procedure **H** into the call **X**. One effect of this copying is that there may be multiple evaluation that would be avoided in a pure graph reduction model. One of the expressions within H, namely **(+ p1 p2)**, does not use the last parameter, **P3**. If there were several intermediate procedures, **E1** and **E2**, at the level of E, that particular expression would be a constant in those procedures. In pure graph reduction, there would only be one evaluation of that constant, say by a call of **E2**, and the resulting value would be used by calls of **E1**. In the RCF scheme used for this example, the copying of H means that both calls of **E1** and **E2** would cause evaluation of that constant.

Avoiding such multiple evaluation requires that all procedures be executed in place, as occurs in graph reduction. The use of higher-order procedures results in programs comprising very general procedures which are very frequently used. Although this is beneficial from the programming viewpoint, access to a frequently used procedure definition is a potential system bottleneck, particularly if each access is a complete execution of the definition in place rather than just the copying of the definition for execution elsewhere. For this reason execution in place is recognised by researchers in the graph reduction field[61] as being a possible major disadvantage of the pure graph reduction model for a decentralised computer architecture. For the RCF model to support execution in place with higher-order procedure structures a greater

sophistication in the base-relative addressing scheme would be required to handle the more convoluted addressing structures. In view of the potential access problem and the fact that the benefit of avoiding multiple evaluation is likely to be a minor effect, the incorporation of such additional sophistication is unlikely to be worthwhile.

4.2.8. Non-determinacy

In the RCF architecture there is the potential for non-determinacy in that two concurrently executing instructions can simultaneously modify the same object. There are two schemes for controlling non-determinacy which correspond to the techniques used in control flow and data flow. These are illustrated in Figure 36 for two different ways of organising a resource manager. In (a) the resource manager is a control flow style procedure executed re-entrantly by the activities, P1 and P2, which use the resource. The resource manager (i) contains a critical region protected by an exclusion argument, (excl). This acts in a similar way to the "test and set" instruction often used to implement critical regions in conventional architectures. When executed by one activity, P1, it becomes a (?) which prevents a subsequent activity, P2, entering the critical region (ii). On exit from the critical region (iii) instruction S is executed to set the (?) back to (excl) and thus allow another activity to enter the region.

In (b) the resource manager is a continuously executing object which on each execution takes one transaction from the transaction stream, in the same way as each execution of the addition operator in Figure 33 takes one token from its token stream. Each user of the

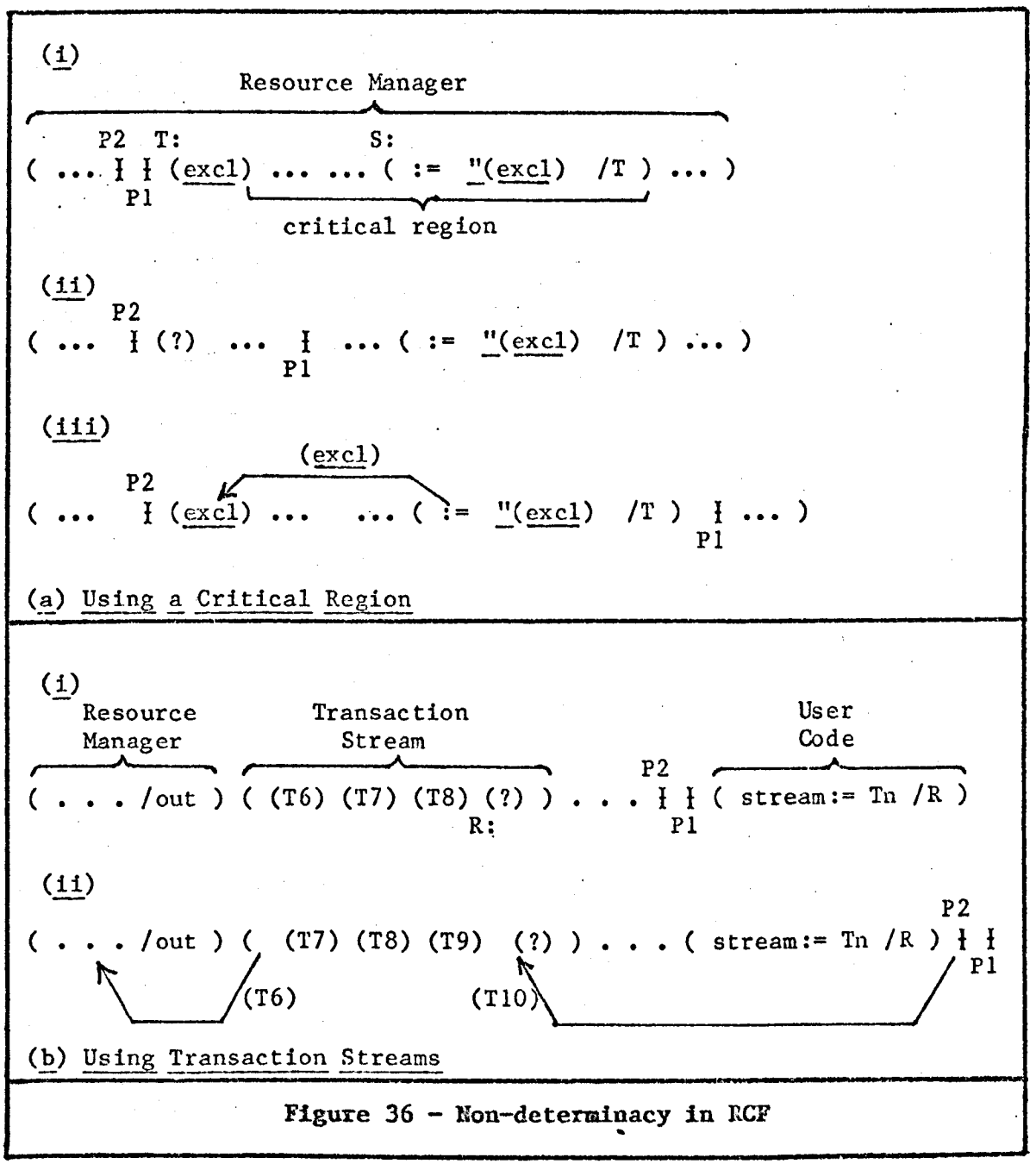


Figure 36 - Non-determinacy in RCF

resource (activity **P1** or **P2**, both executing re-entrant "user code") communicates with the resource manager by inserting a transaction, **Tn**, at the end of the transaction stream. This stream implicitly orders inputs inserted from concurrent sources into a single sequence and thus acts like the merge operator used in a data flow resource manager.

4.2.9. Discussion

The examples in this Section have attempted to illustrate two aspects of the RCF model. Firstly that it incorporates the basic models of control flow, data flow and reduction for simple expression evaluation and allows those models to be combined even within a single "instruction". Secondly that the RCF model can be used effectively for the representation of the program organisation constructs used to program architectures based on those other models. Of particular relevance to control flow programs is the use of repeated sequences of simple instructions and the use of re-entrant procedures. Of particular relevance to data flow programs is the use of stream operands for iteration and the ability to communicate whole data structures as single "data tokens". Of particular relevance to reduction is the use of unbounded data structures and higher order procedures. The examples at this higher level of program organisation did not illustrate any general combination of models as was illustrated for the lower level of simple expression organisation. Nevertheless it is hoped that the examples demonstrate that the RCF model would accommodate a programming language synthesising control flow, data flow and reduction constructs at all levels of program organisation. The actual achievement of such a synthesis would be a question of language and compiler design.

The set of program representation constructs used in this Section were principally developed for simple control flow, data flow and reduction instructions, as illustrated in Figure 30. These constructs are however of quite general utility in program organisation, as illustrated in the other examples discussed. Delimiter symbols (and) are used for

representing all levels of program and data structure, including expressions, arrays, loops and procedures. The explicit structure allows particular standard selector sequences to be used for particular addressing needs such as for access to procedure parameters and for loop repetition. One set of operators, such as +, are used in all instructions, whether control flow, data flow or reduction, in which single operands are accessed. A complementary set of operators, such as stream+, are used for all instructions in which operands are sequences of values. Normal addresses **/sel/.../sel** are used for all situations in which the addressed object is to be executed, such as for a control flow branch, for invoking a subordinate reduction expression, or for invoking a procedure. Quoted addresses, **/sel/.../sel"**, are used for all situations in which the addressed object is not executed, such as for the result location of an instruction, for a procedure definition to be copied, or for manipulating addresses as in conditionals. An unknown argument, ?, is used for all situations in which execution is dependent on the arrival of information, such as for the operand of a data flow instruction or at the end of a stream. An exclusion argument, excl, is used for all situations in which multiple execution is to be avoided, as for lazy evaluation in reduction and critical regions in control flow.

4.3. Machine Organisation and Implementation

The first Section of this Chapter discussed recursive structuring for storage organisation, addressing, and program representation and execution in the RCF operational model. The recursive model provides a general framework for constructing program structures from a (relatively arbitrary) set of "primitive" constructs. This Section starts by describing a similar general framework for the recursive organisation of machine resources in a decentralised computing system, and discusing the relationship between that recursive machine organisation and the RCF model that it supports. Three particular aspects are then discussed, namely the way in which the machine organisation and model accommodate special-purpose machine components within a general-purpose system, and their extensibility and locality properties.

There are two broad types of decentralised computing systems, namely parallel computer architectures and geographically distributed computer networks. Previous Chapters (and previous Sections of this Chapter) have been concerned solely with parallel computers, which is the main theme of the thesis. However the recursive machine organisation, and the concepts of the operational model it supports, are of relevance to computer networks and both types of decentralised systems are discussed here. Implementations of the recursive machine organisation, in both parallel computers and computer networks, will be discussed in the next Chapter.

4.3.1. General Structure

A conventional computer architecture has a centralised machine organisation comprising one computing element with a single processing unit connected to a single memory. A recursive architecture, as illustrated in Figure 37(a), has a recursive machine organisation comprising a structure of nested computing elements (CEs) and local communication (and control) systems (CSs). In the general case a computing element, such as CE0, functions as a complete general-purpose computer with memory, processing and communication capabilities and consists recursively of subordinate computing elements, CE1 - CE3, which provide its memory and processing capabilities and use its communication capability to cooperate in the concurrent execution of programs. The CS component may incorporate some relatively centralised control functions concerned with, for example "strategic" allocation of its subordinate computing elements' resources. This recursive structure will terminate in "primitive computing elements" such as CE1 and CE21 which do not continue the general structure of subordinate computing elements but have separate processor (PE) and memory (ME) elements.

This machine organisation accommodates heterogeneous components, as for CE0 where one component, CE1, is a primitive computing element and another, CE2, is a structure of computing elements. For a coherent system comprising heterogeneous components it is important that there be a common interface between components. The principal characteristic of a recursive machine organisation is that all computing elements at all levels in the structure support the same external interface. For the RCF architecture the common interface embodies the essential concepts of

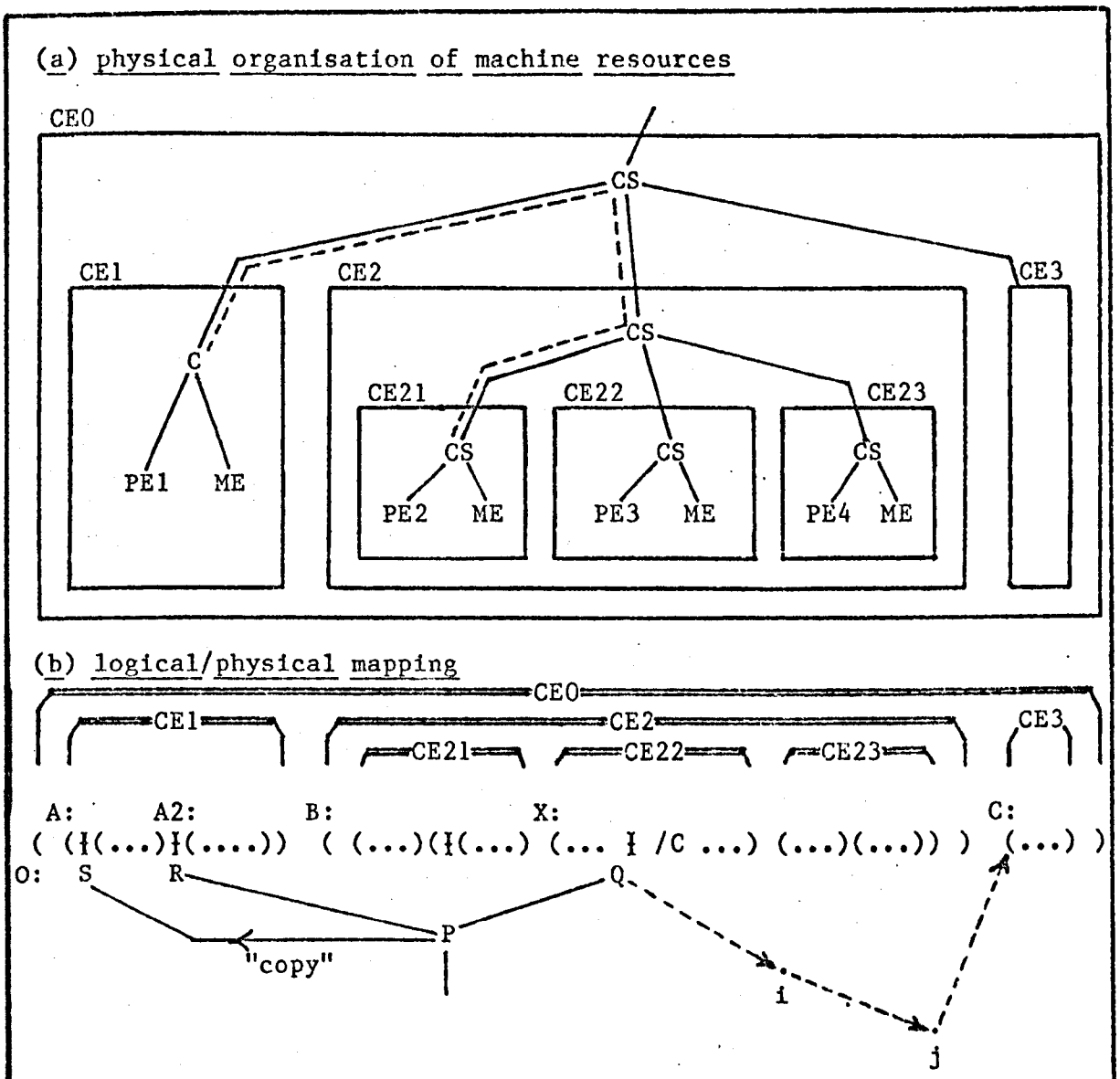


Figure 37 - Recursive Machine Organisation

the RCF model, namely nested variable length objects, contextual addresses and communication of messages within a dynamically created tree of concurrent activities.

In order to illustrate how the logical RCF model is supported by the physical machine organisation, Figure 37(b) shows an example of the former's object, address, and activity structure with a possible mapping onto the latter's computing element structure (the object structure is

not meant to be a particularly meaningful example). The computing element CE0 contains the object O, and its subordinate computing elements, CE1, CE2 and CE3, are each supporting one of that object's components A, B and C. At this level there is a strict (static) correspondence between logical objects and physical computing elements. However, as in the case of object B and computing element CE2, there need not be such a strict correspondence between the internal logical and physical structures. In this case each subordinate computing element will contain a relatively arbitrary (changing) sub-structure of the object, for example CE21 containing the entire first component of B and part of its second component. A primitive computing element will generally incorporate some internal memory management mechanism to support a changing structure of variable length objects. Similarly, a higher level computing element, such as CE2, may incorporate mechanisms for redistributing the object structure between its subordinates as different parts of the structure expand and contract.

The processing capability of a computing element will support activities positioned at the objects contained by the computing element. For example computing element CE0 is supporting the tree of activities comprising activity P and its subordinates Q, R and S, with processor PE2 supporting activity P, PE3 supporting activity Q and PE1 supporting both R and S. A primitive computing element will provide some (machine code) interface for contextual addressing, and creation of and communication between activities. Generally it will incorporate some multi-programming mechanism for sharing the capacity of its processing element between the activities it is supporting. (A processing element could possibly comprise a pool of processors - a primitive computing

element is not typified by there being a single processor, but rather by there being an internal separation of processing and memory resources such that it cannot be considered to recursively consist of subordinate computing elements.) Above the primitive computing element level, allocation of processing load to processing resources is, at least conceptually, integrated with the memory management mechanisms since an activity is allocated to the same computing element as the object which it is executing.

The communications capabilities together provide a multi-level communication system for transmission of messages between activities in different computing elements (as when activity **P** sends the **copy** message to activity **S**), and the migration of activities between computing elements (as occurs when activity **Q** executes the address **/C**). Logically, the **copy** message from **P** to **S** is transmitted along a direct channel from **CE21**, supporting **P**, to **CE1**, supporting **S**. This logical channel, shown by a dotted arc in Figure 37(a), passes through the communication systems of **CE2** and **CE0**, each of which provides a local logical channel ("virtual circuit") supporting a segment of the complete channel. Typically each communication system would have a pool of local channels multiplexed on its physical communications medium. Each local channel would have an identifier unique to that system which would be used to label a message traveling through that system in order to identify its logical channel and thus destination. (For example, using a "logical port" of the destination computing element to identify a channel, the **copy** message traveling within **CE2**'s system might be labeled **CE2/outport20**, causing it to be switched out to **CE0**'s system in which its label would be, say, **CE1/inport10.**)

It is the migration of an activity, a terminal point of a complete channel, which causes the allocation and deallocation of local channels, and establishes each connection between a pair of local channels in neighbouring systems which form adjacent segments of the same complete channel. The migration of Q, as a result of executing the address /C, requires a message from CE22 to CE3, carrying any state information that is associated with the activity (for example, information needed to implement the communication of messages between that activity and its related activities). For this type of message the destination is identified by the sequence of selectors forming the address, actually /esc/esc/esc. These selectors, identifying first position i, then j, then C, are interpreted incrementally by the successive computing elements on the path. The first selector, interpreted by CE22, causes Q to migrate to CE23 where the remainder of the selectors are interpreted and cause the further migration of Q to CE3 which contains the addressed object, C. When Q migrates to CE3 the channel connecting it to its superior, P, is extended into CE0's communication system. This extension requires the allocation of a local channel identifier there, which would be deallocated when Q terminates or, say, migrates back within CE2.

Although the concepts of recursive architecture have here been developed in the context of highly parallel computer architecture, they are equally relevant in the context of geographically distributed computer networks. Interpreting Figure 37 in that context, computing element CE0 would correspond to a local area network with its subordinate computing elements, CE1, CE2 and CE3, being the individual computers at the network nodes, and the objects A, B and C being their filestores.

The connections between activities such as between P and R and between P and S, model the multiple logical communication channels (ports) in an inter-node communication protocol. These connections would at this level be typically used for the transfer of files between nodes, as would result from the **copy** message sent to activity R positioned at "file" **A2**. The migration of an activity to execute an object in a different node, such as when Q migrates to **CE3**, corresponds to "remote execution". A subordinate activity executing in a different node than its superior corresponds to the concept of a "remote agent" used in the structuring of programs for geographically distributed systems.

In the implementation of various decentralised systems covered by the general recursive architecture there will be a variety of design choices reflecting, for example, differences between communication systems for occasional movement of large files between distant computers and those for frequent movement of small data items within a computer. There are a number of specific implementation issues which will be discussed in the next Chapter, particularly: the allocation of logical objects to physical computing elements; the memory management and address interpretation mechanism needed to support a dynamically changing structure of variable length objects; and mechanisms for organising communication between activities which can migrate between computing elements. First however some particular aspects of the general machine organisation will be discussed, namely the way in which it can accommodate special purpose computing elements and its extensibility and locality properties.

4.3.2. Special-Purpose Computing Elements

The recursive machine organisation and the RCF model which it supports are intended to accommodate special-purpose computing elements in much the same way as the memory organisation and operational model of conventional architectures support "special-purpose" memory cells for memory-mapped I/O, and the filestore organisation and system call interface of the UNIX operating system[56] accommodates for example line printers as special-purpose files. Incorporating such special-purpose entities into a general model requires that they must recognise the general interface, even though they may support it in a limited or specialised way. For example, a line printer, represented in the UNIX filestore as a file, supports standard file "write" operations and recognises but rejects standard file "read" operations. As an example of a special-purpose computing element, **CE3** might be a hardware implementation of the square root function. The functionality of this computing element is modeled as an object, **C**, in the information structure so that it can be integrated within the general RCF model. In this case there is a strict correspondence between logical object and physical computing element so that the address of **/C**, **/esc/esc/esc** from within **X**, is in effect the address of the corresponding computing element. That address would be part of a procedure call **X** (as in Figure 34(b)) to be executed by **Q**. When **Q** executes the address it will be positioned at and "execute" the "object" **C**, in fact invoking the square root operation provided by **CE3**. Provided that the same interface for accessing parameters, etc., is supported, the program fragment **X** is unaffected by whether the function of object **C** is implemented as a special-purpose

computing element or a normal procedure definition which can be executed by any general-purpose computing element.

The minimum capability required of any computing element supporting an object structure **S** is to recognise the arrival at **S** of an activity which has migrated from another computing element (to execute **S** or with the remainder of a partially interpreted address intended to position the activity within S); and the arrival of messages, such as the primitive **execute**, **replace-with** etc. messages, for an activity thus positioned in **S**. However a computing element can have a limited or specialised response to the arrival of activities and messages for them. In the case of the square root function, **CE3** would allow the execution of its object **C**, but would signal an exception in response, say, to an attempt to select a component of **C** or replace it with a different object.

The notion of special-purpose computing elements is also relevant in the context of networks as they often include specialised "server" nodes. For example if computing element **CE3** were a "print server", its corresponding object **C** would effectively be a print queue, supporting say the **insert** primitive for adding objects to the queue for printing, but rejecting any other attempted access. It would also be possible for the operational model of a special-purpose computing element to be a particular subset of the RCF model., For example **CE3** could be a "data flow server" with internally a data flow architecture optimised for and limited to the execution of data flow style programs. Such a computing element would accept the insertion of program fragments, as components of its object **C**, for subsequent remote execution. It would need to ver-

ify that a program fragment to be inserted used only the appropriate RCF subset (for example, verifying that it conformed to the structure used in Figure 30(iii) which is directly equivalent to that for a data flow computer, as in Figure 4). Thus a potential benefit of synthesising other models within the RCF model is that it allows a program fragment organised according to one of the other models to be executed either by a general-purpose computing element or by a computing element specialised for that model. Also it allows the specialised computing element to be easily integrated into the overall architecture.

It would in fact be possible for the machine organisation to consist entirely of special-purpose computing elements, for example a network of control flow, data flow and reduction systems with each system being a computing element with either control flow, data flow or reduction computers as its subordinates. In such an organisation the combination of the different models supported by the full RCF model would not occur at the level of individual instructions (as occurs in Figure 30(vi)). The combination would instead occur at a higher level of program organisation such as procedure calls between program modules written in languages based on different operational models and independently compiled for execution on different classes of computing elements.

4.3.3. Extensibility

A major motivation for a recursive structure is its potential extensibility. This is relevant in contexts of both geographically distributed networks and parallel computers, and particularly in the context of exploiting VLSI technology. Important extensibility charac-

teristics of the recursive architecture are (i) the common interface for all computing elements at all levels which means that a primitive computing element and any structure of computing elements are functionally equivalent and thus logically interchangeable; and (ii) the ability of the hierarchic object structure and contextual addressing to accommodate unlimited address space expansion. These characteristics allow major system extensions and re-configurations to be accommodated without correspondingly major re-design. In the context of computer networks such changes include: increasing the level of distribution by replacing a single computing element (e.g. multi-user computer) with a network of computing elements (e.g. personal computers and file servers); integrating previously independent computing elements (separate computers) as subordinates (nodes) in a new higher level computing element (network). In such changes it is possible to retain the original object (filestore) structures and the validity of previously used addresses (filenames).

Relevant extensions in the context of (VLSI) computer architecture are: to increase the processing power and storage capacity for a computer design by connecting a number of the computers together as computing elements within a larger computer of the same overall design; to accommodate the increasing amount of logic circuitry which can be integrated on a single chip as a result of continuing technological advances in miniaturisation of semi-conductor logic devices. In a more conventional, single processor, approach to computer architecture an increase in the logic circuitry available for a single "microcomputer" chip would be typically exploited by, for example, increasing the sophistication of the instruction set. This would entail major re-design of the chip itself and re-design or re-programming of the

hardware or software systems in which it is used. With a recursive architecture the design of a computing element as, say, a single chip can be scaled down to a fraction of a chip and replicated to give a multi-processor single chip computing element which is functionally equivalent (both in terms of instruction set supported and communications interface), but more powerful in storage capacity and processing power. Such a scaling down might entail some low-level re-design in that different design parameters vary in different ways as a function of the scale of the design, but the overall computing element organisation and its external interface would be stable.

4.3.4. Locality

An important consideration in decentralised systems, principally affecting resource allocation, is the need to support and exploit the locality properties of programs. Programs tend to be organised as logical hierarchies of "modules" such as procedures comprising instructions and local variables; sets of related procedures and the shared data structure on which they operate; and further groupings of such modules into higher level modules. Locality is the property that, at any level in the hierarchy, local references and interactions between logically "close" elements within the same module at that level will tend to be of greater frequency than global references and interactions between logically "distant" elements within different modules. The recursive architecture allows this logical hierarchy to be explicitly represented with each module being an object in the storage structure. The contextual addressing scheme, with variable-length addresses, means that the relatively frequent addresses between logically close elements will be

relatively short. The hierarchic communication structure of the machine organisation means that, in so far as logically close objects can be allocated to physically close computing elements (as occurs to a large extent in Figure 37), global communication will be minimised with local communication resources providing the necessary communication bandwidth for local interactions.

In computer architectures intended to utilise VLSI technology the need to localise communications will be an increasingly dominant factor. At the chip level this is because, with increasing miniaturisation, the costs in term of delay, power consumption and chip area associated with off-chip "global" communication increase dramatically relative to the costs associated with local processing and communication within a chip. Localising communication will also become an important factor in the internal organisation of a chip. Increasing miniaturisation of circuits on a chip produces corresponding increases in the time taken for "global" communication of data across a chip, relative to the time-scale of data processing by the logic circuitry. Eventually the discrepancy between system-wide communication and processing time-scales will mean that organising a complete chip as a single synchronous system becomes ineffective[62]. Thus it will become necessary for even a single chip to be organised as a decentralised system of asynchronous components, as provided for by the recursive machine organisation.

In summary, a general form of recursive architecture and machine organisation would provide a common model of extensible system organisation for heterogeneous decentralised systems, spanning both geographically distributed computer networks and multi-processor computers,

incorporating general-purpose and/or special-purpose computing elements. All computing elements at all levels would recognise a common interface and general-purpose operational model which might however be implemented in a limited or specialised way by a special-purpose computing element. The next Chapter will describe a simple implementation of the recursive architecture in the form of a parallel computer comprising identical general purpose components and will discuss some other recursive systems implementations, including a network of conventional computers.

4.4. Summary and Discussion

There are two related aspects to the RCF architecture covered by this Chapter, firstly the synthesis of control flow, data flow and reduction and secondly the general principles of recursive structure on which that synthesis is based. Section Two dealt principally with the ways in which control flow, data flow and reduction styles of program organisation are represented and executed in the RCF model whilst Sections One and Three dealt principally with the recursive principles and their application in the RCF operational model and machine organisation.

4.4.1. Combining Models

The initial motivation for the RCF architecture was to combine control flow, data flow and reduction program styles, in order to obtain the particular advantages of each, as discussed in Chapter Two. They can be combined by different arguments of an instruction being organised according to different models (which requires a programming language and compiler based on the RCF model). More modestly, they can be combined by a procedure organised according to one model calling a (separately compiled) procedure organised according to a different model, which might be supported by a different computing element specialised for that model. This ability to combine models is largely a consequence of the modularity and flexibility of the recursive architecture and the RCF operational model. Most important is that an operator's arguments are executed so that the organisation of an argument is largely independent of the particular operator and the organisation of other arguments in the same instruction. This approach requires that programming

primitives such as unknowns, addresses and literals be defined in terms of their effect when executed as independent arguments of an instruction, rather than in terms of the complete instruction. For example in the data flow model the unknown arguments in an instruction are used to indicate the number of data tokens required as operands for the instruction as a whole to be activated, whereas in the RCF model an unknown is an instruction to suspend the activity executing it. Thus the overall organisation of an instruction is independent of whether it contains any unknown arguments and an unknown argument can be combined with say a reduction expression argument in the same instruction.

The principal constructs identified for the representation of control flow, data flow and reduction program fragments are:

(i) an object containing instructions executed in sequence, providing the implicit sequentiality of conventional control flow

(ii) the parallel operator (par) which initiates the independent execution of its operand , explicitly providing the form of concurrency found in parallel control flow and data flow

(iii) the unknown (?) and exclusion (excl) arguments explicitly representing the synchronisations used in parallel control flow (?s indicating the requirement for control tokens), data flow (?s indicating the requirement for data tokens) and reduction's lazy evaluation (excl preventing multiple simultaneous executions of an object)

(iv) a data item which identifies (or returns) itself, providing the literals found in all models

(v) two kinds of address selectors, namely /seleval causing the selected object to be itself executed, as in reduction, and /sel" just identifying the selected object, as in control flow

Although these constructs were principally motivated by the need to explicitly represent control flow, data flow and reduction style instructions, they were shown to be of general utility in the organisation of programs.

Also important in the representation of reduction and data flow is the generality and flexibility of the storage, addressing and execution structures. The storage and addressing structure allows an instruction's operand to be a nested program structure as required for reduction, and allows a result argument to address an operand within another instruction as required for data flow, or to address the instruction itself as required for reduction. The execution structure of multiple trees of activities includes, as limiting cases, the single activity of control flow, the multiple independent activities of data flow, and the single activity tree of reduction.

An important consideration in control flow, data flow and reduction, discussed in Chapter Two, is their different control and data mechanisms. The RCF architecture has a sequential control mechanism in the sequential execution of an object's component instructions, a recursive control mechanism in the recursive evaluation of an operator's operands, and a parallel control mechanism in the independent execution of an operand of the special parallel operator. The architecture incorporates what is basically a by-reference data mechanism with an operator's operands and results being accessed from and stored into

explicitly addressed locations. However the storage and addressing structure allow any operand (even an array) to be adjacent to the operator, as in the by-literal mechanism, and a result to be stored directly into another instruction as in the by-value data mechanism.

4.4.2. Recursive Structuring

The second aspect of the RCF architecture is as one possible realisation of the recursive architecture principles for the organisation of decentralised systems. A recursively structured system provides a general recursive framework which embodies the essential principles of the system and a relatively arbitrary set of primitive constructs. The recursive structuring principles identified in Sections One and Three are:

(i) a recursive storage structure of nested variable length objects

(ii) a contextual addressing scheme in which an address is a sequence of selectors with the first selector being relative to the pre-established position of an activity and each subsequent selector being relative to the position identified by its predecessor in the sequence

(iii) a recursive form of program representation with an instruction consisting of (operation, input and output) argument objects any of which may be a nested structure of instructions

(iv) the recursive execution of objects by dynamically created trees of concurrent activities with communication along the arcs of a tree

(v) a recursive machine organisation of nested computing elements connected by local communication systems with all computing elements at all levels supporting the same basic interface for the addressing of objects it contains, the migration of activities between computing elements and the communication of objects between activities in different computing elements

In order to investigate the application of these principles, a particular set of primitives were identified for the RCF architecture and its implementation. These primitives included, for example, sequence delimiters, simple positional selectors, and **copy** and **take** actions.

The most important benefits of the recursive principles are the modularity and flexibility of the resulting architecture, and its locality and extensibility properties. Within the framework of the essential concepts summarised above, there is a considerable degree of logical independence between different entities in the recursive structure. In the object and addressing structure, each object **O** provides a local address space in which a selector **s** (such as "first component" or "next component") is independent of the possibly changing address spaces containing **O** and internal to **O**'s components, and independent of the other selectors in the address of which **s** is a part. The general framework of a particular operation/argument message interface, such as "**execute** and **copy**" for input arguments and "**execute** and **replace**" for output arguments, can accommodate interactions between a variety of types of operation and argument objects. These include primitive operators and data items, procedures and parameters, program defined operators and data types (where operation and argument objects may be complex program

structures) and even "objects" which correspond to special-purpose computing elements. Such different types of objects may be freely combined in a program, even in a single instruction, and the implementation of a particular object may change without affecting the program structure using it. In the recursive machine organisation, a computing element is concerned purely with the communication between its component computing elements, but not with their internal organisation. Thus there is a logical independence between component computing elements any of which may be primitive or structured and general-purpose or special-purpose; and a physical independence in that communications within a particular component are supported entirely by its local communications system. To the extent that programs exhibit locality of reference and logically "close" objects are allocated to physically close computing elements, there will be a minimisation of global communication, which is perhaps the single most important design goal for a decentralised system implementation whether at the level of asynchronous components within a single chip or at the level of a geographically distributed network.

The two kinds of extensibility that were discussed were "outward" extension of connecting together previously separate computing systems into a network, or connecting together replications of the same computer design to give a more powerful design; and "inward" extension of replacing a single computing element with a sub-tree of computing elements. The recursive system structure ensures that such extensions can take place without encountering address space limitations or requiring major re-design or re-programming. Most importantly the extended computing system has an homogeneous address space, existing addresses retain their validity, and there is the same mechanisms for communicating between

different parts of the program structure.

The main emphasis in this Chapter has been on the application of the recursive structuring principles to concurrent computer architecture. Those principles emphasise the relationship between entities (objects, selectors, instructions, activities and computing elements) rather than the characteristics of the entities themselves. As a consequence of this abstraction the general principles can be applied at all levels of computing systems organisation. For example an object can represent any level of information structure from an integer to a file or group of files. The same addressing scheme and accessing operations can be applied to selecting and manipulating an integer in an array, or a file in a directory. The communication between concurrent activities can model all communication within a computing system from the communication of a simple data item as an operator's operand to the communication of large data structures between concurrent processes executing in different computers. The recursive machine organisation can be applied at any level of implementation from that of a VLSI computer to that of a geographically distributed system. At the former level computing elements would be single chips or even parts of chips and a special-purpose computing element would provide for example floating point operations. At the network level computing elements would be separate computers and a special-purpose element would be for example a print server. The essential difference between the different levels of computing systems organisation lies in the degree of complexity of what are viewed as the primitive entities (size and sub-structure of objects, sophistication of addressing selectors, "power" of instructions and the logical activities and physical processors needed to execute them). As will be discussed

more fully in the next Chapter, these differences motivate corresponding implementation differences at different levels of recursive systems organisation. Despite such differences, the RCF architecture can provide a common programming model which allows multiple computing elements within a computer and multiple computers in a network to be programmed to cooperate in program execution in the same way, and facilitates changes in system organisation such as increasing geographic distribution or increasing integration of components onto a single chip.

5. RECURSIVE SYSTEMS IMPLEMENTATIONS

This Chapter covers a number of computing system implementations which, in varying degrees, incorporate the principles of recursive structuring discussed in the preceding Chapter. The first system described, referred to as LEGO, is a parallel computer designed as part of this thesis work specifically to support the RCF model. The remaining four systems are: a computer networking extension to the UNIX operating system (UNIX United); a reduced instruction set parallel computer (RIMMS); a recursive computer architecture (R.M.); and an interpreter for a programming language based on the RCF operational model (EASIX). These are more or less independently conceived systems included here to illustrate possible alternative realisations of the recursive architecture concepts and as a basis for discussing various implementation issues.

The UNIX United system is an illustration of the application of recursive structuring concepts in the context of conventional computer networks. RIMMS is closely related to the RCF work, representing an intermediate stage between conventional computer architectures and the full generality of the RCF architecture, achieved by minimal extensions to a conventional microcomputer design. The R.M. architecture is the one example, other than the RCF architecture, of principles of recursive structuring being used throughout a parallel computer design - the principal difference between the two is in their models of program execution. EASIX is a concurrent programming language rather than a parallel computer or computer networking design. It is included only because it is directly based on, and constitutes the first completed implementation

of, a recursive control flow model. Finally a number of general design issues in the implementation of a recursive machine organisation are discussed in the context of the UNIX United, RIMMS, R.M. and LEGO designs.

5.1. The LEGO Design

The LEGO recursive computer design is based on the general recursive machine organisation of Figure 37. It comprises a basic parallel computer design which can be extended to allow connection into higher levels of recursive machine organisation. The basic design, illustrated in Figure 38(a), is for a parallel computer comprising many (upto a few hundred) identical general-purpose computing elements (CE1 - CEn) connected into a ring. In addition there is a control element (CE0) performing special functions such as initialisation and external communication. Each element is intended to be implemented as a single LSI chip. The detailed design of these chips is currently being produced by another member of the Computer Architecture Group. The components of a computing element are a memory element (M), processing element (P) and communications unit (C). Each of these is connected to the corresponding components in the two adjacent computing elements. The control element CE0 is principally concerned with communication functions and does not have any general processing or memory element. The following description of a computing element's functional organisation (Figure 38(b)) represents the starting point for the the detailed design and chip layout work. Square brackets [...] are used to indicate estimated information about the detailed design.

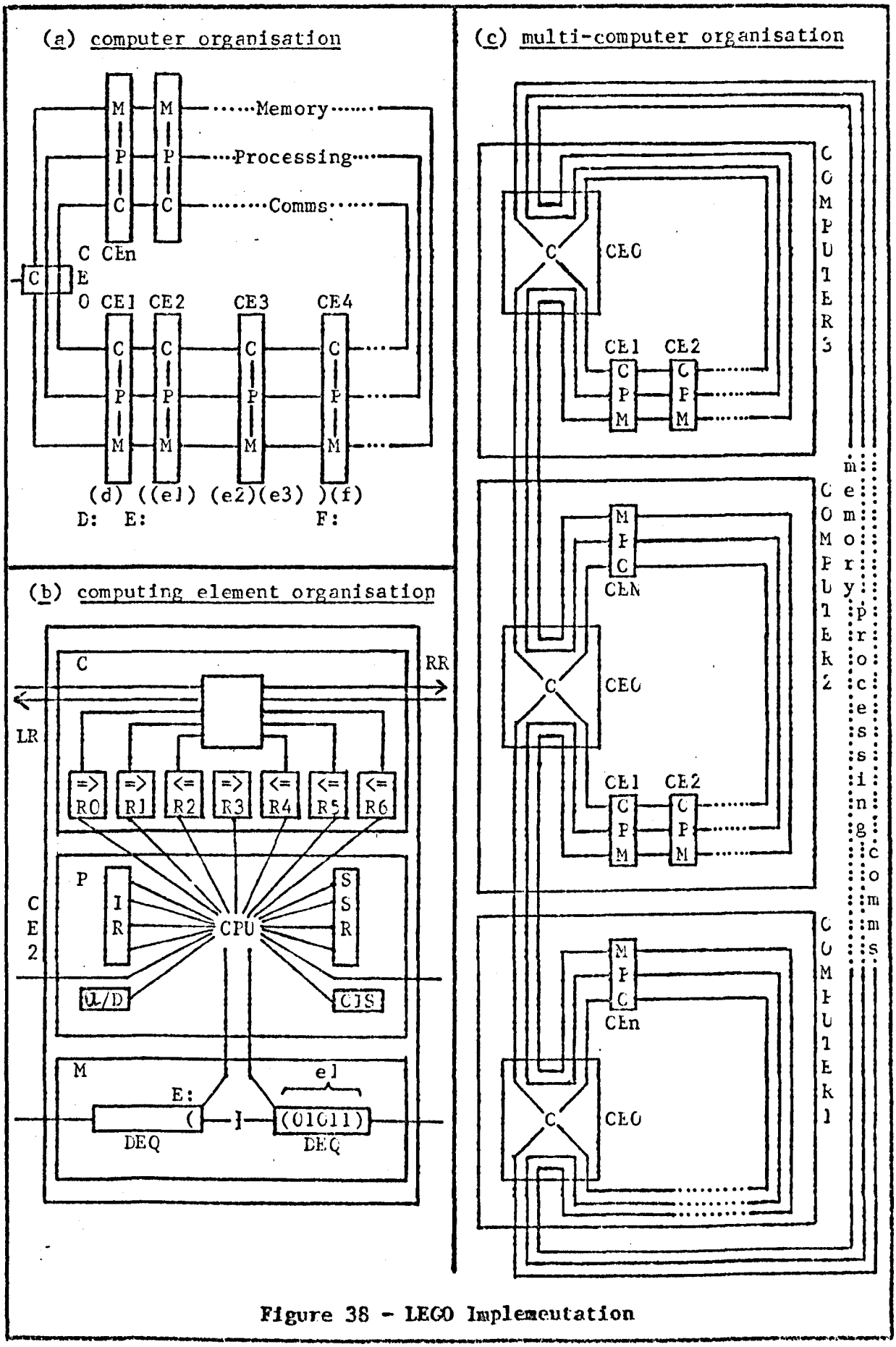


Figure 38 - LEGO Implementation

5.1.1. Memory

The total memory of the computer supports a hierarchic structure of variable-length objects, the program string, represented as a sequence of "quats". Each quat is one of the four symbols - (and) structure symbols for delimiting objects and 1 and 0 bit symbols for encoding primitive objects (such as integers and machine code instructions). The memory-memory connections between adjacent computing elements allow movement of the string around the ring as different parts of the total object structure expand and contract due to insertions and deletions. For example, in Figure 38(a), if new components were being inserted in object D and components were being (independently) deleted from F then there would be a general anti-clockwise shifting of the intervening section of the string. This shifting maintains the total sequencing of symbols whilst moving free storage capacity to the area where it is needed.

An individual computing element's memory, as shown in Figure 38(b), is physically organised as two double ended queues (DEQs), one on each side of the processing element. These provide variable length storage [for up to 100 symbols] which accommodate local string expansion and contraction and the shifting of the string between adjacent computing elements. Each DEQ provides the processing element with a stack interface (push and pop operations) which allows insertion of symbols into the string (push), deletion of symbols from the string (pop) and shifting of the string in either direction (pop and push on opposite sides). Each DEQ of one computing element is connected to the complementary DEQ of the adjacent computing element. This connection is used to shift

symbols from one DEQ to the other in order to balance the number of symbols held by each.

5.1.2. Processing

Each processing element can support (be occupied by) one activity, executing instructions contained in its associated memory element. The activity's position in the string, indicated by the ‡ in Figure 36(b) is between the two symbols at the internal ends of the two memory DEQs. Each activity can have a superior activity (identified as R0) and up to six subordinate activities (identified as R1 ... R6). An activity can either be executing the program string, or executing messages, such a **copy** instruction sent from its superior. The same machine code instructions are used in both cases. For example the **/->** addressing selector is a machine code instruction which can be executed as an address argument in the program string. Alternatively an activity executing an operator argument can send that selector instruction as part of a message to be executed by a subordinate activity, in order to position the subordinate at one of the operand arguments. The instruction set provides: data operations such as arithmetic functions on integers (of any length), logical functions on booleans and comparison and conditional selection functions on arbitrary objects; the addressing selectors **/<-**, **/->**, **/in**, **/out**, **/esc**, **/start**, **/end**, and **$Rn** (identifying a particular related activity as the base for base-relative addressing); creation of subordinate activities; communication between activities; **copy**, **take**, **replace**, **insert** and **execute** actions on objects (of any length); and the **?** and excl synchronisation primitives. An instruction is a variable length object [4 - 12 bits]. A full instruction set is described in

Appendix A.

A vacant processing element, i.e. one not currently occupied by an activity, will shift symbols from one of its DEQs to the other in order to balance the number of symbols held by each. A DEQ does the same with respect to its neighbour in the adjacent computing element and thus (for resource allocation reasons discussed below) the string will tend to spread evenly over the total memory between successive occupied processing elements.

Figure 38(b) outlines the functional organisation within a processing element. The processing element has external access to two of the symbols stored in the associated memory element, and to single-symbol input and output buffers (R0 - R6) for communication with the associated related activities. The environment [50 bits] accessible to the central processing unit (CPU) includes that memory and communications data, and some registers internal to the processing element. The CPU functions as a finite state machine (implemented as one or more PLAs) which on each cycle generates a new environment state dependent on the previous environment. [Typically 15 bits of environment is relevant to the set of transitions for executing a particular instruction.] As an illustration of the possible function of this design, consider the execution of a + data operation. This is performed bit-serially with its operands being received from R2 and R3 and its result being transmitted to R1. The principal state transition is to set new bit values for the R1 output buffer and an internal carry register, as determined by the values of the R2 and R3 input buffers, and replace the used R2 and R3 input values with "empty" markers. If however an input buffer is still empty

or an output buffer still full from a previous cycle then "no-change" transitions will occur until the communications system has filled/emptied the buffers.

The processing element's internal registers include: the current instruction (IR); the current instruction source (CIS) indicating whether the next instruction is to be taken from the program string or a particular Rn buffer (in either case the instruction is loaded into the instruction register a symbol at a time); some supplementary state (SSR), such as the carry bit; and an up/down counter (U/D). This latter is used to count matching brackets when compound objects are being processed. For example executing a /-> selector involves moving symbols from one DEQ to the other, counting up and down the level of nesting as (and) symbols are encountered, until the count returns to zero. [A counter of 10 bits, allowing upto 1024 levels of object nesting, is probably sufficient for any size machine].

5.1.3. Activity Migration and Resource Allocation

There are two types of activity migration, local and remote. Local migration is the migration of an activity from its current computing element to an immediately adjacent computing element. As described in detail below, this occurs in order to provide an activity with access to information (symbols of the program string), or to some hardware resource (storage or processing capacity) which is not immediately available in the activity's current computing element. The migration is negotiated using the connections between adjacent processing elements. If, say, the activity is migrating to the right then the sequence of

symbols occupying the right DEQ of the original computing element and the left DEQ of the new computing element are shifted, by the new computing element, to its right DEQ. This ensures that when the activity has migrated it will retain the same logical position in the information structure. Thus the two DEQs separating the two computing elements are emptied before the migration actually occurs. The actual migration is achieved by activity state information [8 bytes] being sent as a message from the old incarnation of the activity to the new incarnation. (The organisation of the message communication system is such that during the transmission of this message the new incarnation can safely migrate again if necessary.)

There are four possible causes of local migration to consider -

(i) To provide access to the string - An activity is popping symbols from the DEQ to its right or left, and the DEQs between itself and the next activity in that direction become empty. The activity migrates through the intervening processing elements until it is adjacent to that other activity and then swaps position with it. This type of migration occurs in traversing the program string (as a result of sequentially executing instructions, performing **copy**, **take** and **replace** actions, and executing addressing selectors).

(ii) To provide access to storage - An activity is pushing symbols onto the DEQ, to its right or left, and all the DEQs between itself and the next activity in that direction become full. In this case the rightmost of the two activities is forced to migrate to the processing element to its right. This produces a pair of empty DEQs between the two activities and so allows the symbol pushing to

continue. This type of migration can occur in traversing the existing string or inserting a new object.

(iii) To provide access to processing elements - An activity is always created by another activity, at the same position in the string, and at creation the new or original activity is forced to migrate to the processing element to the right (since only one activity can occupy the original processing element).

(iv) Propagated migration - An activity P is forced to migrate to the right (as in (ii) or (iii)) to obtain resources (for itself or possibly in (ii) for the activity to its left) and the required resources are not immediately available. Before P can migrate the destination processing element must be vacant and the intervening DEQs empty. If there is an activity Q occupying the required processing element then the original migration of P is propagated to Q - Q is forced to migrate to its right, possibly forcing further propagated migration. Once Q has migrated P's migration can be completed. (Emptying of the two DEQs can also cause further activity migration, as in (ii), since it will involve pushing symbols onto the DEQ to the right).

The local migration mechanism thus provides a simple scheme for the decentralised allocation of processing and memory resources which ensures that local resource requirements are always satisfied if there are sufficient free resources anywhere in the machine. This scheme relies on the circular connection of computing elements which means that a free resource can always be considered as being to the right of the particular computing element needing it.

An important aspect of the resource allocation scheme is that the program string expands and contracts not only in response to insertion and deletion of objects but also in response to the creation and termination of activities within them. For example consider the expression **(* (...A...) (...B...))**. The machine code program for this expression might initially be in one computing element's memory. The activity executing the multiply will create subordinate activities to execute the **(...A...)** and **(...B...)**. These activities must migrate to the next two computing elements and in doing so they, so to speak, sweep the two objects in front of them. Thus the **(...A...)** and **(...B...)** are shifted to two different computing elements where they are executed concurrently. Generally, an executing program fragment will expand over sufficient computing elements to provide the necessary processing resources, and will subsequently contract again when it no longer needs those resources but another expanding part of the program structure does. As mentioned above, the operation of vacant computing elements is such that the program string will tend to spread evenly over a number of adjacent vacant computing elements. Thus if there are spare memory resources in the neighbourhood of an inactive program fragment (i.e. one containing no activities) the program fragment will expand over those resources in anticipation of the expansion that will be needed if it does become active.

The other, remote, type of migration is initiated by execution of a **$Rn** type of selector supporting base-relative addressing. This selector positions an activity directly at another activity which may be anywhere in the program string. Remote migration is negotiated by a message to the specified destination activity (**Rn**) using the general communications

system, rather than by using the direct connections between adjacent processing elements which are provided to facilitate local migration. On receipt of this message, the destination activity's computing element initiates a new incarnation of the migrating activity in the adjacent processing element (as in case (iii) above) and then the state of the migrating activity is transmitted from the old incarnation to the new incarnation just as occurs in local migration.

5.1.4. Communications

The communications units together provide a communications system for the transmission of messages between related activities in possibly non-adjacent computing elements. This communication system functions as two slotted rings rotating in opposite directions (LR, left ring, and RR, right ring). Each symbol of a message is transmitted as a separate packet in a passing empty slot of one of the rings, and is acknowledged when it has been accepted by the destination processing element. The acknowledgement provides simple flow control - the destination has only a single-symbol buffer, and so a further symbol cannot be sent until the buffer has been emptied. One reason for using only single symbol buffering is that when an activity migrates from one computing element to another, any message symbols that had been received by the old computing element, but not yet processed, must be transferred to the new computing element. Single buffering for such unprocessed symbols reduces the amount of information to be transferred and simplifies the migration mechanism. If a buffer contains a symbol of a message then there cannot be another symbol packet in transit and thus the buffer contents of a migrating activity can be sent to its new incarnation by

the normal message passing mechanism with no danger of packets arriving there out of sequence. Also, as will be discussed later, programs can easily be organised to use the variable length object structure for efficient, unbounded, buffering.

Associated with each output buffer is a routing flag (=> or <=) to determine which of the two rotating rings is to be used for transmitting symbol packets from that buffer. (An acknowledge packet is always transmitted in the reverse direction from that in which was received the symbol packet being acknowledged.) This routing flag is an estimate of which direction provides the shorter path to the destination. The communication channel between two activities is established when one (the superior) executes an instruction to create the other as its Rn subordinate. The two activities will be initially in adjacent computing elements and the creating instruction specifies whether the subordinate is created to the right or left of its superior. That information determines the initial setting of the RO flag in the subordinate and the Rn flag in the superior. Subsequent migration of the activities can result in a routing flag indicating the "wrong" directions, i.e. the direction in which packets have to travel more than half way round the ring. One scheme for correcting the routing information is to divide the ring into a number of sectors and to provide in an acknowledge packet a count of the number of sector boundaries which that packet has crossed during its transmission. The receipt of an acknowledge packet with a sector count exceeding half the total number of sectors indicates that the packet which it acknowledges was (probably) transmitted in the wrong direction. This causes the routing flag of the appropriate output buffer to be inverted so that the next symbol packet will be sent in the "correct"

direction. This scheme requires that a number of computing elements (evenly spaced around the ring) are "sector boundary" elements. The communication unit of such an element will increment the sector count of any acknowledge packet passing through it.

There are two circumstances in which the routing information should be corrected. Firstly two related activities may cross. For example activity R2 initially to the right of its superior executes a /<- selector which re-positions it to the left of its superior. Secondly the two activities may migrate further apart until they are separated by more than half the total number of computing elements. The former circumstance, of activities crossing, is likely to be the more frequent, and also the more important to detect since the two activities will typically be very close in the "correct" direction, and thus very distant in the "wrong" direction. Three sectors is sufficient to always deal with this circumstance and is a particularly appropriate choice since the same amount of information (2 bits) is needed for the sector count in an acknowledge packet as is needed for the symbol carried by a symbol packet. Also the need for a "sector boundary" element to increment a 2-bit sector count in a passing acknowledge packet is unlikely to be a limiting factor on the packet transmission rate. For the other circumstance, two related activities migrating very far apart, the likelihood of correct detection by this scheme depends on the number of sectors. With only three sectors, the worst case is that a packet may traverse two-thirds of the ring in the "wrong" direction, rather than one-third in the "correct" direction. The expected locality properties of programs suggest that this circumstance will be relatively rare. Thus the additional complexity of increasing the number of sectors in

order to better handle this circumstance would be unlikely to be effective.

The addressing of messages in the communication system is in terms of logical activities rather than physical computing elements since an activity may at any time migrate from one computing element to another. For a message from an activity to a subordinate, the destination is identified as **A/n** where **n** specifies the particular subordinate and **A** is an activity identifier, unique within the computer. For a message from a subordinate to a superior, the destination is identified as **A/0**. An activity identifier, **A**, thus identifies a (bi-directional) logical channel carrying all communications between that activity and its subordinates. (Those, leaf, activities that do not currently have subordinates do not need identifiers.)

The computer maintains a decentralised pool of activity identifiers, the number of identifiers being equal to the total number of computing elements in the computer. (The number of activities needing identifiers is naturally bounded by the number of computing elements, although generally is much smaller). Each activity identifier is either allocated to a particular activity, in which case it is part of the state of that activity, or it is free in which case it is held in an identifier store within the communications unit of some computing element. Each identifier store can hold one such free activity identifier. When an activity needs to be allocated an identifier (i.e. when it first creates a subordinate) it places an "allocate" message on one of the rotating rings. That message will be serviced by the first encountered communications unit with a non-empty identifier store. The identifier

is taken from that store and sent to the requesting activity, leaving the store empty. When the activity terminates, it releases its activity identifier by emitting a similar de-allocate message carrying the freed activity identifier. This message is serviced by a communication unit with an empty identifier store into which is stored the freed identifier.

There are similar motivations for this scheme for the allocation of activity identifiers (logical communication channels) and the scheme for the allocation of processing and memory resources. Both resource allocation schemes are fully decentralised. Resource requirements will generally be satisfied from locally available resources. However a requirement will if necessary be satisfied by an available resource anywhere in the machine.

5.1.5. Control Element

The principal function of the control element, CEO, is to interface the ring of computing elements to its external environment. The external environment may be a higher level of recursive machine organisation with the computing element ring functioning as one component of a multi-level LEGO machine. The role of the CEO element in such an organisation is discussed in the next Section. Alternatively, as is the case for Figure 38(a), the computing element ring may function as a complete LEGO machine. In this case the external environment would be a user terminal or (during development) a conventional host computer. The CEO element of a complete LEGO machine has three functions. Firstly there is the machine's initialisation. Initialisation includes establishing a

different "activity identifier" in the free activity identifier store of each computing element's communications unit, and establishing particular computing elements as "sector boundaries". These initialisation functions could be implemented by constructing each computing element slightly differently. However (for at least manufacturing yield reasons) it is preferable to make all computing elements identical and establish these minor differences by messages transmitted from CE0 to other elements during an initialisation sequence. The initialisation sequence would also need to establish a "null" initial program state with all processing elements vacant and memory DEQs empty.

The second, principal, function of CE0 is interaction with the external system. The basis for this interaction is a "monitor" activity which is created as the final stage of initialisation. (It would be initially created in a particular computing element, say CE1, but could subsequently migrate as can any other activity.) The monitor activity behaves as the subordinate of a conceptual "user" activity residing in the external system. The monitor activity has its superior activity as instruction source and will respond to messages received from its superior in the same way as does any other activity. The communication unit of CE0 will inject into, and extract from, the communications system messages between the monitor and the user activities, in the same way as does the communication unit of any computing element. The messages generated for the monitor by the external system would be standard machine code instructions and other objects (as described in Appendix A). Typically there would be messages to: insert a program object into the object structure; create an independent activity to execute that program object; create subordinate activities to read different parts of the

program structure in order to monitor the execution progress of the program and retrieve results being generated for output; possibly insert, and create activities to execute, new "diagnostic" program fragments; finally, delete the entire program object.

The third function of the CEO element would be monitoring the behaviour of the computer itself (rather than the program it is executing). For example, it could collect and provide the external system with information about message traffic. Also it would be possible to incorporate in the computing element design facilities to allow CEO to determine, for example, how many computing elements were vacant at a particular point in time.

5.1.6. Multi-level organisation

Figure 38(c) shows part of a multi-computer organisation comprising a ring of computers each having the internal organisation of Figure 38(a). This extension involves changing the role of the CEO control element in each computer's local ring, but requires no changes to the design of other elements or their interconnections. The multi-computer organisation is exactly the same as that within an individual computer. Externally each computer, through its CEO, functions exactly as one of its computing elements, with memory, processor and communication connections for the shifting of objects, migration of activities and transmission of messages. The CEO elements of adjacent computers connect the memory and processing elements of one computer's low order computing element to those of the adjacent computer's high order computing element. The effect is exactly as if all the computing elements of all the

computers were connected in one large ring, with CEO elements forming a higher level communication system by which a message from, say, CEn in COMPUTER1 to CE1 in COMPUTER3, by-passes COMPUTER2's local communications system. Logically, migration of activities and shifting of program string over computer boundaries is exactly the same as within a computer, although there may be physical differences such as slower transmission and different transmission protocols (in which case the function of a CEO elements would include interfacing between internal and external transmission).

A CEO element has principally a communications function similar to that of a gateway node connecting different computer networks. It must recognise whether a message received from an adjacent computer is destined for a computing element within its computer, and if so switch it into the local communications system. Similarly it must when appropriate switch messages from the local system onto the more global system. Within a local communications system, a message's destination is identified by a locally unique activity identifier. Within the more global communications system there must be a different set of unique activity identifiers for identifying messages between superior and subordinate activities in different computers. (Locality considerations suggest that in the majority of cases an activity and its subordinates will all be in the same computer, and so the number of more global activity identifiers required is much less than the total number of local activity identifiers in all computers.) In switching a message from one communication level to the other the CEO also performs the mapping between local and global activity identifiers. Thus a CEO must maintain a (two-way) identifier mapping table which is updated as activities

migrate in and out of the associated computer. Within the multi-computer organisation there will be a control computer, COMPUTER0 (not shown), serving the same role as the CE0 control element within a single computer. That is, either connecting into a yet higher level of recursive machine organisation or interfacing to an external user system. This general scheme clearly extends to higher levels of machine organisation in the same way.

5.1.7. Discussion

The principal motivation for the LEGO design was to obtain a reasonably simple machine supporting the full conceptual sophistication of the RCF model and general recursive machine organisation. The machine instruction set provides all the basic mechanisms for the various forms of program representation and execution discussed in the last Chapter. In contrast, both the other major recursive systems implementations to be discussed in the following Sections, the UNIX United networking system and the R.M. computer architecture, support models somewhat less general than RCF (although the UNIX United and RCF model are remarkably close).

Three major aspects of the general recursive machine organisation were discussed in the preceding Chapter, namely extensibility, locality and special-purpose computing elements. The LEGO design is fully extensible to any level of recursive machine organisation and the resulting multi-level design provides a hierarchic communication system supporting program locality. Even within a single level design with a large number of computing elements, the two counter-rotating communications rings

provide some degree of locality. The executions of, say, two (adjacent) programs by two non-overlapping sub-trees of activities will proceed largely independently, with no communications interference. (The only interaction between two such disjoint computations is beneficial in that they effectively share a common pool of processing and memory resources around their mutual boundary.) The LEGO design cannot however support special-purpose computing elements in the way discussed for the general recursive machine organisation, since that would require a particular object to be permanently resident in a particular special-purpose computing element. Such a requirement would compromise the LEGO resource allocation scheme which depends on objects being able to freely shift around the ring of computing elements.

There are some particular aspects of the LEGO design which allow a computing element to be reasonably simple, considering its generality. Firstly, there is the small amount of external information directly accessible to a processing element. This information consists of the two symbols on either side of the activity in the program string, and one symbol each of messages to and from a few (typically three) related activities. Consequently the CPU has a simple cycle for the serial processing of data. An important aspect of this design is that the most basic level of a CPU cycle accommodates the asynchronous communication of information between computing elements, an essential element of the RCF model. At this level the main problem of the LEGO design is likely to be a processing/communication imbalance. The un-buffered communication of single-symbol packets via off-chip connections, is unlikely to keep pace with the CPU processing of, purely on-chip, information. There would be a closer balance for a VLSI realisation (much more in

keeping with the general design philosophy) where each chip could contain a number of computing elements with a significant proportion of the communications being via much faster, on-chip, communication paths. Such an implementation would use the multi-level design of Figure 38(c) with each **COMPUTER** being a single chip, and a **CEO** element providing the only off-chip connections (which are the same as those for a single computing element LSI chip). It might also be desirable to provide multiple buffering and to use multi-symbol packets transmitted in parallel (achieving a better balance between addressing information and useful data). Such a widening of the communication path would be more viable for a VLSI implementation than for an LSI implementation since in the latter the computing element design is constrained by off-chip bandwidth limitations.

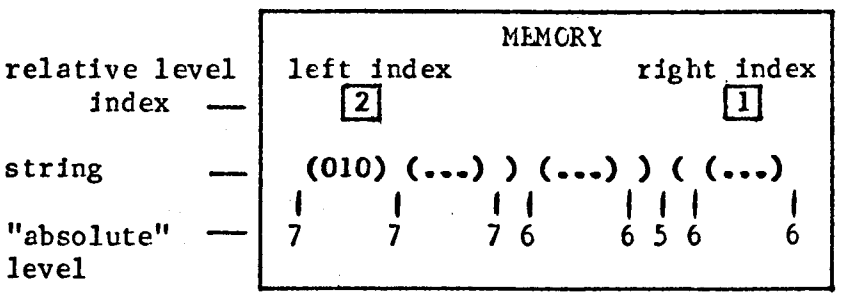
A second, related, aspect is the choice in the LEGO design of providing quite low level instructions, closely corresponding to basic machine operations, and the direct implementation of the general model without any optimisations. These aspects of the design can result in significant program representation and execution inefficiences. Taking, as an extreme example, the expression **(< 2 3)** which compares two simple constants. The operator would require a number of machine code instructions to create subordinate activities, position them at the operands, instruct them to execute those operands and finally to perform the actual (bit-serial) comparison of the returned results. The choice of providing such primitive machine code instructions, rather than say implementing each operator as a single machine code instruction, is partly to keep the computing element design simple and partly to facilitate experimentation with different operator types (such as stream) and

other constructs at the program organisation level.

The execution of (< 2 3) involves the full mechanisms for creating, and allocating computing elements to, subordinate concurrent activities. The advantage of this approach is that the same instructions and mechanisms deal with, say, (< (...A...) (...B...)) where (...A...) and (...B...) are complex program fragments returning large nested data structures. However for a realistic implementation it might well be necessary to complicate the design by incorporating some optimisations. For example, it may be desirable to provide special instructions for the common cases where an operator's operands are values or addresses short enough to be loaded into CPU registers and processed in a more conventional way.

Addressing is based on a very primitive (almost minimal) set of selectors which can give very inefficient program representation. For example selecting the 21st component of an object will require twenty /-> selectors. For a realistic implementation it would be necessary to include the fuller set of selectors given in Figure 25. The DEQ memories provide a simple hardware mechanism for storing and accessing variable length objects. One consequence of this implementation, reflected in the machine code definition, is that when an activity accesses, e.g. copies, an object, it is necessarily re-positioned at the following object. This is convenient when serially accessing adjacent objects, but inconvenient when repeatedly accessing the same object. The main consequence of the DEQ memory implementation is that interpreting an address to position past an object requires a scan of the whole sub-structure of the object (different parts of this scan may be per-

formed in sequence by different computing elements). An approach to alleviating this problem (adopted in the R.M. implementation to be discussed later) is for a computing element of incorporate an index to the structure of its part of the program string. One simple indexing scheme is the following -



The left (right) index records the difference of program string nesting level between the position at the memory's left (right) boundary and the least nested position in the memory. As, say, a /-> selector scans an object, say OB, a count is maintained of the current nesting level within OB. If, when the scan reaches this memory, the count is greater than the left index then the end of OB is not contained in this memory and the scan can immediately continue at the next memory (with the count being decremented by the value in the left index and incremented by the value in the right index). A simple algorithm for maintaining these indexes involves incrementing and decrementing them as delimiter symbols are shifted across the memory's boundaries. (This algorithm can result in the "safe" error of the indexes both being too high by the same amount, sometimes causing an unnecessary scan which can be used to correct them.)

The circular connection of computing elements and the choice of at most one activity per computing element results in simple, totally

decentralised, schemes for the integrated allocation of processing and memory resources to activities and program string, and for the allocation of logical channels (activity identifiers) for communication between activities. The restriction to a single activity per computing element, and the form of memory organisation, can lead to excessive migration of activities and shifting of program string, particularly in view of the frequent creation of subordinate activities implied by the direct implementation of the general RCF model for even simple instructions. This can clearly lead to poor performance, particularly for conventionally organised programs which do not exploit any of the machine's concurrency and communications potential.

Performance is not a major motivation for this design and the design choices discussed above were principally motivated by the resulting simplicity of the initial implementation. However it is worth discussing the machine's behaviour for a style of program better suited to its capabilities than is the conventional style. Performance will be best for program fragments in which a particular pattern of activities is maintained for a reasonable length of time. Figure 39 shows an example program fragment with this characteristic. This program fragment is a development of the data flow style of iteration using stream operands, shown in Figure 33 of Chapter Four, and is represented using the notation developed there.

The operator of **i1**, iter+ is similar to the stream+ of Figure 33. The executing activity **P** positions subordinates, Q and R, at the start of its "stream" operands (the value sequences **a1**, **a2**, **a3** , etc., and **b1**, **b2**, **b3**, etc.), and subordinate activity **S** at the end of its stream

result. For this iter type of operator the executing activity P then repeatedly performs **take** (destructive read) actions on the objects at which Q and R are positioned and adds the returned values, **an** and **bn**, to produce a result **cn**. This result is sent to S as part of an **insert** action to append that result to the end of **i2's** operand stream.

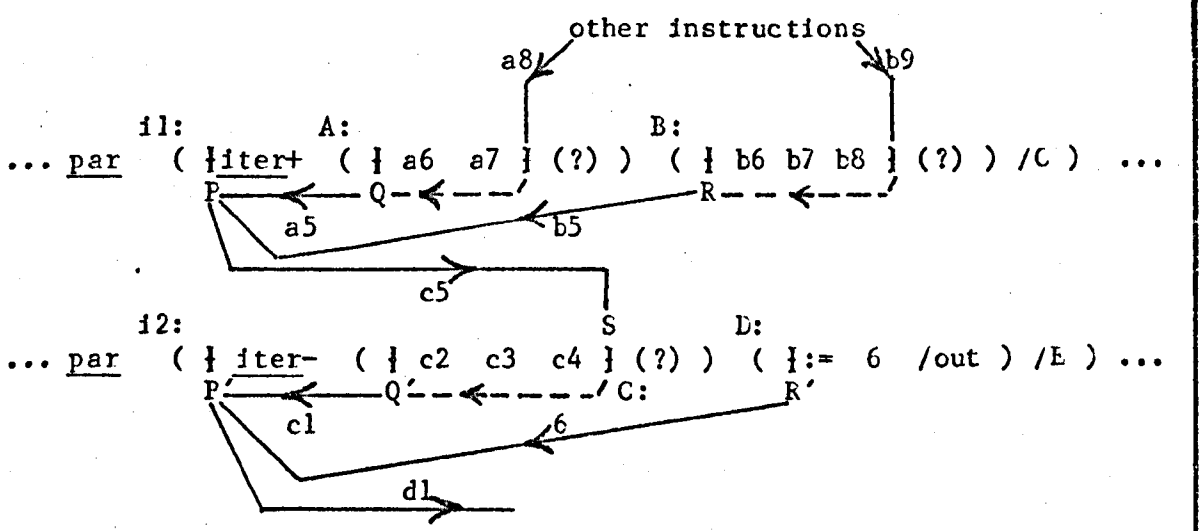


Figure 39 - Execution of Iterative Data Flow in RCF

Instruction **i2** has the same structure as **i1** except that the second operand is an embedded literal, implemented as the object **(:= 6 /out)**. This has the normal (as opposed to stream or iter) type of operator, **:=**, to return its single operand **6**. It is followed by an **/out** address to re-position the executing activity outside the object which it then re-executes, thus returning a stream of constant values to its invoker. An operator such as iter+ in this representation is actually a sequence of primitive machine code instructions to achieve the described behaviour. (Actual machine code for this example is given in Appendix A as an illustration of the instruction set defined there.)

Instructions i1 and i2 would be part of an iterated group of instructions, separated by par operators so that they are all executed in parallel. On the first execution of the group, the par operators create activities at all the group's components (such as i1 and i2) which in turn create subordinate activities at their operands and results. This creation of activities will cause the whole group to expand over sufficient computing elements to provide a processor for each activity (causing contraction in other, no longer active, parts of the program). The iteration actually occurs at the instruction argument level (rather than at the instruction level, as in Figure 33, or the instruction group level, as in conventional control flow). Each activity repeats a small sequence of machine operations, remaining essentially at the same position in the object structure. The initial iteration in effect sets up a logical, data flow, net connecting the computer's processing elements. As indicated by directed arcs in Figure 39, the flows of data pass through the communications system (solid arcs) implementing the connections between activities, and through the memory system (dotted arcs) implementing the stream operands. These streams and the DEQ memories supporting them act as buffers in the data flow paths, accommodating differences in execution rates between the different instructions. After the net has been established, there will generally be no shifting of the string or migration of activities between computing elements. The exception is in the case where a discrepancy in processing rates results in an operand stream expanding beyond the capacity of one computing element's memory and thus causes the shifting of the string to provide the necessary local space.

The data flow style communication of values from i1 to i2 is achieved by the value's generator P and its consumer P' being independent activities which communicate via memory. This, as in data flow architectures, allows maximum concurrency but may consume considerable resources (in this case the storage for the operand streams). In the case of the second operand of i2, the values are generated by an activity R' created as a subordinate of the consumer, P'. These operand values are returned directly on the (unbuffered) communications path from the generator to the consumer. In this case, flow control in the communications system means that the execution of the generator is in effect driven by demands from the consumer. As in reduction, this results in effectively less concurrency, but conserves machine resources. In this case if instruction D were represented in the same way as instruction i1, with i2's second operand being a stream, it could quite rapidly, and uselessly, fill up a large amount of storage with 6s. Thus the demand driven execution conserves storage resources. However the choice of one activity per computing element, with R' permanently occupying a processing element, means that the possible conservation of processing resources is not in fact exploited as it might be in a implementation which shared processing elements between activities.

The LEGO design discussed in this Section was developed specifically to obtain a complete realisation of the RCF model and principles of recursive machine organisation. The following Sections cover four computing systems which, in varying degrees, incorporate concepts similar to those in the recursive control flow model, recursive machine organisation and LEGO machine design. The similarities and differences between the LEGO design and these other systems illustrate a number of

general implementation issues which are discussed at the end of this Chapter.

5.2. UNIX United

The Newcastle Connection[63] is a software subsystem, developed at Newcastle University, which can be added to each of a set of physically inter-connected UNIX systems (computers running the UNIX[56] operating System) so as to construct a distributed system, referred to as UNIX United. The resulting UNIX United system is functionally indistinguishable, at both the user's system command language ("shell") interface and program's system call interface, from one of the UNIX systems from which it is constructed. Thus it meets a principal criterion for a recursively structured system. Although in no way directly part of this thesis work, UNIX United and UNIX itself exemplify many of the concepts discussed in this Chapter. The system is discussed here because it illustrates one end of the possible range of recursively structured computing systems and it provides a particularly familiar context for the discussion of implementation issues. Also, close analogies that can be drawn between (often independently arrived at) concepts and mechanisms in RCF and UNIX provide some evidence of their value. The UNIX United system, being an extension to the existing UNIX operating system, has been implemented quite rapidly and an initial version is currently available commercially.

Viewing UNIX and UNIX United as an illustration of the recursive architecture concepts, the recursive storage structure and contextual addressing is that provided by the UNIX filestore and the file naming scheme, the operational model is that embodied in the UNIX shell and system call interfaces, and a primitive computing element is a single UNIX system.

The filestore provides three levels of "objects": single, fixed-length bytes as primitive objects; files as compound objects each comprising a variable length sequence of bytes; directories providing a hierarchy of higher level compound objects each comprising a variable number of files and directories. (UNIX does provide the facility for one file to be linked into several directories, thus departing from a pure tree structure. However this feature will be ignored here as it is used relatively infrequently, causes some difficulty in UNIX United, and is not very general in that above the file level there must be a strict tree structure of directories.)

A file or directory (a filestore node) is specified by an identifier comprising a sequence of context-relative selectors, in the same way as an object is addressed in the RCF model. The principal type of selector is a name, e.g. **brian**, identifying a particular node within a directory (this corresponds to a **/n** RCF selector identifying a particular component by its position). There are no "global" names, each name being relative to and unique within a particular directory. The filestore tree structure is usually represented as in Figure 40, where the nodes represent files or directories and are labelled by their names within parent directory. The other selector which can be used is the parent selector, **..**, which corresponds exactly to the **/out** selector of RCF. A complete node identifier, of the form **C/sel/sel/...**, corresponds to the base-relative form of addressing in RCF. The starting context C of the identifier is one of two previously identified directories, namely the user's "root directory" or "current working directory". At the file level there is a discontinuity in the addressing scheme, with the addressing of bytes within a file being totally different from

the addressing of directories and files. Each byte is addressed by its absolute position in the file (corresponding exactly to a /n RCF selector) or relative to the current position in the file (corresponding exactly to a /+n or /-n RCF selectors).

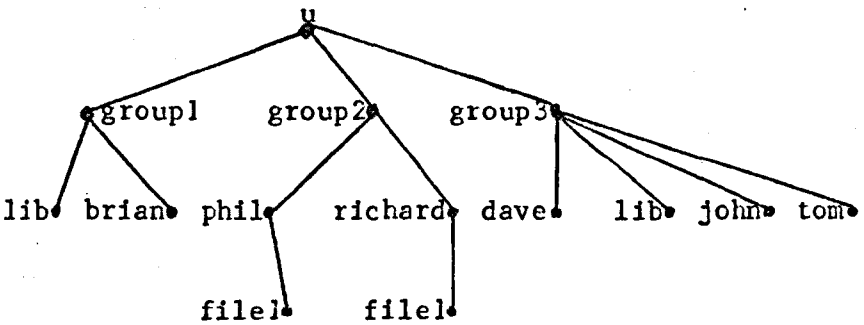


Figure 40 - UNIX and UNIX United Filestore Structure

UNIX United provides a generalisation of the UNIX filestore structure. Each individual UNIX system has its own tree of directories. The root node of the tree is logically an object containing that system's complete filestore (in Figure 40 the directories **group1** and **group2** could be roots of separate UNIX systems). There can then be a higher level directory (or more generally a superstructure of directories) which contains those individual root directories as its components. The complete UNIX United filestore structure thus forms one tree with a homogeneous contextual addressing scheme which is the same as that within an individual UNIX system. In fact any directory node in the tree, such as **john** in Figure 40, can be the root of a separate UNIX system. Node identification is the same regardless of whether UNIX system boundaries are crossed. In Figure 40 the complete tree might be contained in one UNIX system, or **group1** and **group2** might be separate UNIX systems. In either

case, **richard's filel** could be identified from the context of **brian** as **/../../group2/richard/filel.**

There are significant similarities between the RCF model of program execution and the model presented by the UNIX shell and system call interface. UNIX provides a tree structure of user processes, each of which is executing a program file (object) from the filestore structure. Associated with a process are a number of file descriptors, each giving read (RCF **copy**) and write (RCF **replace**) access to the byte at a current position in a file. A file descriptor is dynamically created by an **OPEN** command specifying a file identifier of the form discussed above. In UNIX United an **OPEN** executed in one machine can identify a file in a different machine, in which case read and write accesses are implemented by messages between the two systems. The tree of UNIX user processes corresponds closely a tree of RCF activities. A file descriptor corresponds closely to an RCF activity which is a leaf of the activity tree, used by its superior to access as data the object at which it is positioned.

A UNIX process can create subordinate processes by **FORK** commands. The subordinate is initially at the same position as its superior, executing at the same point in (a new copy of) the same program file. The subordinate process will then typically execute an **EXEC** command, moving itself to execute a different program file. The destination file is specified by a file identifier of the form discussed above. In the case of UNIX United the identified file can be in a different UNIX system, implicitly causing the process to migrate to that system. A subordinate process initially inherits the same connectivity with the filestore as

its superior, that is it is given copies of all its superior's file descriptors. This connectivity can subsequently be changed by closing files and opening different files. These mechanisms in UNIX are very similar to those in the LEGO implementation of the RCF model. In LEGO a subordinate activity is initially positioned adjacent to its creating activity and typically then executes an address to re-position itself at the actual object to be executed (which may cause it to migrate to a different computing element). When an activity is first created it effectively inherits the addressing contexts provided by the other subordinates of its superior (as required to support procedure calls discussed at the end of Section 4.2.6).

One significant difference between UNIX and RCF is that in the former there is no notion of a process being able to address relative to its own position in the filestore structure (e.g. relative to the directory from which was selected the program file it is executing). This results in a problem with, for example, a library program which uses a sibling program or data file in the same "library" directory. Such a library program must contain an identifier for that sibling file, but in UNIX there is no addressing context which can be robustly used for such an identifier. The omission of "program relative" addressing, which would easily solve this difficulty, is perhaps a result of the conventional view that main memory in which a program is executed is very distinct from the filestore memory from which the program is loaded for execution. (In contrast, RCF at least conceptually uses the same memory space for both object execution and "permanent" object storage.)

UNIX implements its variable length "object" structure and multiple

process model using standard operating systems techniques. The byte sequence comprising a file is stored in a number of (arbitrarily positioned) disc blocks which are dynamically allocated and released as the file expands and contracts. A directory (itself implemented as a file) contains a list of its component files and pointers to the physical blocks in which they are stored. The additional mechanisms provided by UNIX United are those for handling remote file execution and access, resulting from an **EXEC** or **OPEN** call with a file identifier that goes outside the current UNIX system. Both cases correspond to activity migration in RCF and involve the migration of a process (the user process or a system, file access, process) essentially as was described for the general recursive machine organisation in Chapter Four. The migrating process is embodied in a message by which the process state is transmitted to its destination. The destination is specified by the file identifier used in the system call to name the remote file. This identifier is incrementally interpreted as the process migrates through the UNIX systems on the path to the destination. As the process migrates a logical connection between its source and destination is established which is used for subsequent direct communication. (Each segment of the complete logical connection is implemented by a system "forwarding" process in the UNIX system through which it passes.) For a user process the state that has to be transmitted (of the order of 300 bytes) is principally that providing its connectivity to other parts of the filestore structure, that is its file descriptor information.

The UNIX United system is "transparent" in two important respects. Firstly it exhibits the extensibility properties of recursive systems discussed previously. Enhancing a UNIX system to be a component of a

larger UNIX United system has no effect on the user's interface, even though that interface then provides access to remote data and program files. Users (and their existing programs) which do not exploit the enhancement are completely unaffected by it (particularly, existing filenames retain their validity). Secondly, the enhancement is implemented without modification to the standard UNIX software, i.e. the operating system kernel and various utility programs. This is achieved by the Newcastle Connection software which implements the enhancement being inserted as a separate layer between the kernel and the user (and utility) programs. To user programs the Connection layer impersonates the kernel, providing the same system call interface. To the kernel it impersonates the user programs, using the standard interface provided by the kernel. The principal function of the Connection is to trap any system call relating to a file in a remote system and to implement the necessary inter-system communications required to service the call. (Other system calls are passed straight down to the kernel.)

5.3. BASIX

BASIX [18] is a computer programming language ("BASed on the unIX system command language") rather than a decentralised computing system. It is mentioned briefly here as it is the first working implementation of a RCF model. Also its implementation is based on exploiting the similarities between RCF and UNIX. Although developed as a separate project, by Isabel Gouveia Lima and David Mundy, there has been design work by members of the Computer Architecture Group, myself and Philip Treleaven, and by a member of the Unix United team, Lindsay Marshall.

The language follows closely the concepts of the RCF operational model developed in Chapter Four, but at the conventional programming language level. An object is either a unit of program structure (a "block", procedure, expression, etc.) or a unit of data structure (e.g. an array). An object's components (e.g. local variables, nested data) can be addressed by name or position. The execution model is that of a tree of processes (activities) each executing an object, with communication of results from subordinates to superiors. The language provides a fairly conventional syntax for expressions, FOR loops etc. The syntax for specifying relationships between concurrent processes (data communication and synchronisation patterns) is largely based on the syntax of the UNIX shell system command language.

The BASIX implementation is an interpreter which runs on the UNIX operating system. The UNIX filestore structure and processes are used to implement the object structure and concurrency (for reasons of implementation expediency rather than efficiency!). The main benefits found in the use of the language are the ease of organising concurrency and information structures.

5.4. RIMMS

The Reduced Instruction Set Multi-Microprocessor System (RIMMS[34]) is a parallel computer, supporting a multi-thread control flow model, which has been influenced by some of the RCF concepts. Its design attempts to apply the reduced instruction set design philosophy[64] to a parallel computer. The overall design of RIMMS has been worked on as a joint venture by members of the Computer Architecture Group, including

myself, with detailed design and implementation being done by Lewis Foti and L. Wang.

RIMMS has two, well separated, levels of machine organisation. The lower level is that of an individual computing element and the higher level is that of the parallel computer obtained by connecting a number (upto 255) of those computing elements together by a slotted ring type communications system. The higher level provides three primitives, **LOAD**, **STORE** and **EXEC**, for communication between its component computing elements. **LOAD** and **STORE** (corresponding to the RCF **copy** and **replace** primitives) allow a process (activity) in one computing element to access memory locations in a different computing element. **EXEC** supports the migration of a process to a different computing element. These primitives are implemented using messages between computing elements of the form -

type (LOAD, STORE or EXEC) : destination (16-bit) ; **operand** (16-bit)

The destination field is an address identifying the location being accessed by a **LOAD** or **STORE**, or the next instruction of the process migrating by an **EXEC**. At this, parallel computer, level an address has the form **/element/word** identifying a particular word within a particular computing element. The operand is the return address for a **LOAD**, the value for a **STORE**, or the migrating process's state for an **EXEC**.

An individual computing element consists of a 16-bit microcomputer with 256 words of local memory. Each computing element can support a number of concurrent processes. The instruction set, based on the reduced instruction set philosophy, has less than twenty different

instructions and only two addressing modes. There are two context registers, Data and Code, and the two forms of address are /D/d or /C/d where d is a displacement relative to one of those registers. The displacement is an 8-bit quantity sufficient to address the memory of one computing element, whereas the context registers are 16-bit quantities sufficient to address the entire system's memory. If the actual address used to **LOAD** an instruction operand or **STORE** its result is in a different computing element than that in which the instruction is being executed then a **LOAD** or **STORE** operation of the higher level machine is automatically invoked. The Data register, which can be explicitly set, identifies the process's current workspace. (The workspaces of different processes are not necessarily disjoint.) The Code register is the process's program counter, identifying its current instruction. (This register is principally used to access literal values embedded in the code). The Code register is automatically incremented as the process moves sequentially through its code and can be explicitly set by a branch instruction. In either case, if the new Code register value points outside the computing element, then the **EXEC** primitive of the higher level machine is automatically invoked to migrate the process to the computing element containing its next instruction. The only process state information that needs to be communicated to the new computing element is the Data register value.

There are two major features of the RIMMS design worth emphasising. Firstly it implements the notion of process migration between computing elements, and in order to do so efficiently the amount of process state is kept to a minimum. Secondly the design is intended to minimise the impact of the higher level, multi- computing element organisation on the

programming interface provided by an individual computing element. The design thus illustrates a possible approach to transparently extending an existing micro-computer design to a multi-microcomputer system. In this respect there is some similarity with the transparency of UNIX United. From within a computing element the only visible effect of combining several computing elements in the higher level organisation is that the address space is extended. There are no explicit mechanisms for programming interactions between computing elements. Instead such interactions are implicitly invoked by the program fragment in one computing element generating an address in a different computing element. The only effect on the design of a computing element is that such remote addresses must be trapped and mapped into corresponding communication messages. The implementation of this additional functionality in each computing element (corresponding to the Newcastle Connection software used to implement UNIX United) is entirely separable from the implementation of the rest of the design.

Currently there is a software simulator for RIMMS, produced mainly to evaluate the machine's programmability. The main success is the transparency of computing element communication and the minimisation of process state. The main shortcomings are the limited number of addressing modes and a restriction of literals and address modifiers to 8-bit quantities. Work has now started on a hardware design. It is expected that in the future RIMMS will be extended to include more of the concepts of the RCF architecture, such as allowing instruction operands to be variable length byte sequences.

5.5. R.M.

The concept of recursive computer architecture originally came from Wayne Wilner's Recursive Machine (R.M.) design[55,65]. The RCF architecture and LEGO implementation are in some ways very similar to the R.M. work and in some ways very different. The R.M. operational model provides the same hierarchic structure of delimited objects, represented as a sequence of (,), 0 and 1 symbols. There is a similar form of contextual addressing with an address being a sequence of selectors interpreted incrementally. There are however a number of significant differences in the addressing schemes. Firstly there is a form of absolute addressing where the starting context of the address can be specified as the outermost object of the entire object structure. (The provision of such addressing compromises the possibility of an un-premeditated joining together of existing systems, as can occur in for example UNIX). Secondly there is a much more sophisticated set of addressing selectors, including facilities for selecting a component object by specifying its content rather than its position. Thirdly there is nothing corresponding to the RCF's base-relative form of addressing, all addresses being relative to their point of use.

The major difference between R.M. and RCF is the model of program execution. The R.M. model[65], deriving from work on object-oriented computation[60], is a particular form of parallel control flow and does not directly support other models. An instruction can use an address to send a message (itself an object) to the addressed object. Such a message is a forked flow of control. The arrival of the message at the destination object cause the creation of a process or "activity". The

message is typically a structure of instructions which are executed by the new activity. This execution is in the context of the destination object so that addresses in the message are relative to that object and the message can cause modifications in the neighbourhood of its destination. Simple examples of the types of messages are -

(i) **(insert 20)** - A new component of the destination object is created.

(ii) **(copy /self to /address-of-X)** - A copy of the destination object (identified by the selector **/self**) is sent as a message to **X**. The copied object, say a procedure definition, is then executed in the context of **X**, thus achieving a procedure call using the components of **X** as parameters.

If there are two messages for one destination object then there will be some sequence to their arrival, and the activity associated with the second message to arrive is delayed until that associated with the first message has finished. This mutual exclusion is the one process synchronisation mechanism provided.

In effect, as in the RCF model, an activity can create another activity which is positioned at an addressed object and sent a message for execution. The fundamental difference between the two models is in the activity structure. In the RCF model the created activity can be a subordinate of the creating activity and there can be further direct communication between them, particularly the return of a result. The R.M. model is essentially a subset of the RCF model in which a created activity is necessarily independent and so no further direct communica-

tion is possible beyond the original message. If subsequent communication is required, as in example (ii) above, the created activity must send a message to an object in the neighbourhood of the creating activity. This requires that the original message contain the return address, /address-of-X, of its originator relative to the context of its destination. However it is not clear that there is a robust general scheme for constructing such return addresses within a dynamically changing object structure.

The R.M. implementation accommodates a tree structure of nested computing elements, as in Figure 37. The basic memory organisation is similar to that of the LEGO implementation with the expansion and contraction of the object structure being accommodated by DEQ memory elements and by the movement of objects between adjacent computing elements. As an additional mechanism for handling variable length objects, a shift register can overflow to a RAM associated with its computing element, which in turn can overflow to higher levels of storage hierarchy, such as discs, associated with higher level computing elements. A primitive computing element can contain several activities at objects in its memory. The execution cycle is to search the memory for an activity, execute that activity to completion and then search again. The communication system supports messages from an activity to a destination object. The sequence of selectors forming the address is interpreted incrementally by communications units on the route from the source to the destination. Each communication unit (of a non-primitive computing element) is itself actually a primitive computing element. All communications units keep a record of the absolute address of the start and end of the object structure contained in its computing

element's memory. This information provides a structure of indexes into the memory, thus avoiding some of the searching which occurs in LEGO. The communications units also perform an additional load balancing function, determining in conjunction with the neighbouring computing element whether objects should be shifted between them.

5.6. Discussion

The LEGO computer, UNIX United networking system, RIMMS computer and R.M. computer provide examples of various different approaches to the implementation of recursive systems. The most significant issue is the mapping of logical objects to physical storage structure. The basic choice is between a static storage structure with permanent objects allocated to particular units of physical memory; or a dynamic storage structure with objects being freely created and deleted and shifting between memory units as the object structure changes. RIMMS implements the extreme of completely static storage, whereas LEGO and R.M. implement the other extreme of completely dynamic storage. The UNIX United filestore is intermediate between these two. At the level of a directory representing a UNIX system there is a static storage structure. Although directory objects at this level can be created and deleted, that is not part of normal operations. Within a UNIX system component directories and files can be created, deleted, expanded and contracted, as a result of which there will be changes in the allocation of logical objects to physical storage. (For example, contracting and then expanding a file may cause a change in which actual disc blocks contain the file.) Dynamic storage is more flexible but requires a more sophisticated address interpretation mechanism and also a storage allocation

scheme which needs to be decentralised if (as in LEGO and K.M.) the dynamically allocated storage includes more than one computing element. Generally dynamic storage is likely to be inappropriate at the computer network level, being either impractical due to the inter-node communication required for storage allocation, or unacceptable due to the resources at different nodes being owned by different users. (With the recursive machine organisation there is no absolute distinction between a "parallel computer" and a "computer network" and in this context perhaps the distinction between dynamic and static storage structure provides the best definition of the "parallel computer" level of recursive machine organisation.)

For a system with dynamic storage structure there is the question of how physical storage is organised to implement the changing object structure. Here there is a strong distinction between serial access techniques, such as with the DEQs used in LEGO and K.M. (or, in a more conventional context, magnetic tapes), and direct access techniques, such as with discs used for the UNIX filestore. In the former, the structure of the physical storage devices directly supports the insertion and deletion of objects and the structure of information is encoded directly in the information, as delimiter symbols. These characteristics are very amenable to localised processing and impose no limit on object size. Direct access techniques involve additional mechanisms, beyond those implicit in the physical storage medium, for allocating and structuring the storage medium. Supporting a structure of variable length objects would typically require lists of free space, pointers to an object's components and the system functions to maintain that additional information. These characteristics lead to more centralised

processing. However, given these additional mechanisms, the addressing of information is easier than with serial access techniques. Intermediate between these two is the "index sequential" technique, as used in R.M., with supplementary information to aid addressing of information stored in sequential access devices.

A basically sequential storage technique is the more appropriate if a computer is to support the manipulation of variable length, arbitrarily long, objects at the level of individual data items (e.g. integers), particularly as the "bit" components of those objects do not generally need to be directly addressable. However at the level where larger objects are being stored and operated on (such as "files" being moved between "directories") it would be more appropriate to use direct access techniques with which such operations can be more efficiently achieved by manipulating pointers.

The second major issue is the implementation of activities (processes) and their migration, particularly organising the delivery of messages to "moving targets". In all four of LEGO, UNIX United, KIMMS and R.M. there are multiple activities and the processing element supporting an activity is that associated with the storage unit in which is stored the object being executed by the activity. This minimises the communications overhead of instruction access (which is predictably very frequent) at the possible expense of data access (which is not easily predictable). Three levels of, increasingly more autonomous, processing resource allocation can be identified. The highest level is that where there is a static storage structure which gives the user implicit control over which computing element will support an activity. For exam-

ple, in UNIX United, this can be deliberately controlled by copying a program file into the directory tree of a particular UNIX system and executing that copy. (This would be done either to exploit particular processing characteristics of that system, or minimise the amount of inter-system communication needed for the program's file accesses.)

The intermediate level is where there is a dynamic storage structure involving several computing elements. At this level there is automatic processing resource allocation which is an integral part of storage allocation. The influence of processing load on storage allocation may be very large (as in LEGO where creation of new activities can cause considerable shifting of objects) or minimal (as in R.M. where the only influence is that activity state information uses some general storage capacity and so creation of activities might require some shifting of objects to provide the space).

The lowest level is that internal to a primitive computing element where processing resource allocation is an autonomous system function. Apart form the degenerate case of a single activity per computing element, as in LEGO, there is the basic choice of whether (as in UNIX) a processing element is time-multiplexed between activities in the computing element. The alternative (as in both RIMMS and R.M.) is to allocate a processing element exclusively to one of the activities until it has terminated, migrated or become (temporarily) unable to continue (for example in RIMMS, waiting for the response from a remote operand LOAD). The inclusion of multiplexing will depend on the grain of concurrency and degree of physical parallelism. Multiplexing is desirable where, as in UNIX, activities may be whole programs interacting with different

users and running on the same physical processor. However multiplexing is not so necessary, and the additional complexity of implementation unlikely to be effective, where, as in R.M., each user program will be a separate object with exclusive use of many processing elements for the support of its activities and many activities will be very short-lived, for example created just to insert one data item into a structure.

The migration of an activity from one computing element to another requires the transmission of activity state data. In the case of LECO, RIMMS and R.M. activity migration is a basic part of a design which therefore attempts to minimise the amount of data that has to be transmitted. This data is principally the identification of the activity's destination and information about its connectivity with other activities and thus the rest of the object structure. There may also be some internal state information. In LECO the destination identification is a single selector. The connectivity information is its own activity identifier and that of its superior (to allow the emission and reception of messages to and from its subordinates and superior) and the content of the single-symbol buffers used for those messages. In RIMMS the destination identification is the Code register and the connectivity information is the single Data register. In R.M. the destination is identified by a sequence of selectors which contracts as the activity moves towards its destination. There is no direct connectivity to other activities or objects. Any objects, other than the destination, which the activity needs to access will be included in, or addressed by, the message sent with the migrating activity. This message, the activity's internal state, can be of any length.

The original UNIX design did not anticipate the process migration which occurs in UNIX United and quite a large amount of data has to be transmitted. As in R.M. the destination is identified by a sequence of selectors which contracts as the process moves towards its destination. Connectivity information is that associated with the process's open files. There can be quite a large number of these and this information constitutes most of the transmitted data.

The question of communication between activities only arises in LEGO and UNIX United where there is a persistent structure of activities (processes). These two system illustrate two general approaches to implementing communication between migrating activities. In LEGO the communication is achieved by the broadcasting of messages. A message is not physically addressed to a particular destination computing element. Instead it is logically addressed to a particular destination activity specified by an activity identifier, unique within the computer. The system relies on the computing element containing the destination activity being the only one to recognise the logical address and thus accept the message. In UNIX United remote communication is achieved by the forwarding of messages. The path from the source (user process) to destination (a system process supporting remote file access) uses a chain of forwarding agents (system processes), one in each UNIX system on the path. Thus at each stage of its transmission a message is physically addressed to a specific UNIX system - the one containing the next forwarding agent in the chain. The forwarding system is more appropriate at the computer network level since migration is relatively infrequent and so once the forwarding chain has been established it will last for some time. Also, at that level, the chain will typically involve

only a few stages, and in any case it would not be practicable to use system-wide unique process identifiers. In contrast, at the level of a parallel computer like LEGO, migration is very frequent and a chain of forwarding agents would involve a large number of stages. At the higher level of multiple LEGO computers there is effectively a forwarding system, for the same reasons as in a network organisation. Each CLO "gateway" computing element acts as a forwarding agent for messages between an activity in its computer and an activity in another computer.

The concepts of the RCF model and recursive machine organisation provide a general framework for decentralised systems at all levels. However, as the above discussion illustrates, there are different characteristics at different levels which motivate different implementations of those concepts. The principal distinctions are at the level separating a (possibly multi-level) "computer network" from constituent (possibly multi-level) "parallel computers" and at the level of an individual computing element. Above the parallel computer level there would be a static storage structure implemented using direct access techniques. The concurrency and communication that needs to be supported is that of long-lived user processes with relatively infrequent communication within a structure which is fairly stable, both logically in terms of connection between processes and "open files", and physically in terms of process residency in particular network nodes. Also at this level the connectivity of a process (the number of open files and subordinate processes) is likely to be relatively high.

At the lower levels there would be a dynamic storage structure employing serial access techniques. The concurrency and communication

that needs to be supported is that of concurrent activities executing individual instructions and accessing individual data items, with relatively frequent communication within a rapidly changing but simple structure (e.g an activity having two subordinates for its instruction's operands). The lowest, single computing element, level is typified by a separation of storage and processing resources, as in conventional computers. This separation principally effects the allocation of processing resource which at this level is not tied to storage allocation. Also of course messages between activities in the same computing element can be transmitted via the computing element's local memory rather than requiring physical communication between different computing elements. This might effect the processor allocation strategy. For example it might be appropriate for allocation of processing resources among a sub-tree of activities in the same computing element to be based on a pre-order traversal of the sub-tree - a processing element would be allocated to a subordinate activity as soon as it is created and reallocated to the superior activity when the subordinate terminates. This corresponds to conventional expression evaluation and would allow a simple stack to be used for the communications along the arcs of the activity sub-tree.

Given these various issues a major area of further research is the possible development of a general approach to recursive systems implementation which accommodates the different characteristics at different levels, with minimum compromise to the unity of the general concepts.

6. CONCLUSIONS

6.1. Summary and Discussion

This thesis has investigated various general-purpose decentralised architectures providing highly concurrent program execution. The principal motivations for developing such architectures are the utilisation of concurrency to improve performance, the support of novel implicitly concurrent programming languages, and the exploitation of VLSI technology. The architectures covered were the control flow, data flow and reduction classes of architectures surveyed in Chapter Two and the LCF and RCF architectures developed in Chapters Three and Four. In each case the basic concepts of the architecture were described in terms of an operational model of program representation and execution followed by a discussion of the way in which various forms of program organisation are supported and the way in which machine resources are organised.

The novel data flow and reduction architectures have operational models which are radically different from the control flow model. In data flow an instruction is executed when its inputs are available and in reduction an instruction is executed when its outputs are needed. Both these models are implicitly concurrent whereas in concurrent forms of control flow the initiation and synchronisation of concurrency is under explicit program control.

The main result of the analysis of the various architectures in Chapter Two was a classification of their underlying control and data mechanisms and an understanding of the consequences of an architecture

adopting a particular set of mechanisms. The control mechanisms identified were sequential, parallel and recursive. The main benefit of the sequential control mechanism is in providing the programmer with complete operational control and in the execution of programs with little inherent concurrency. The main benefit of the parallel control mechanism is in the parallel execution of highly concurrent programs. The main benefit of the recursive control mechanism is for the more sophisticated program structures of applicative programming languages and in conserving machine resources by only executing what is actually needed to produce the required result. The two principal data mechanisms identified were by-value which is at an advantage in manipulating simple data items, and by-reference which is at an advantage in manipulating data structures.

Whereas all the architectures surveyed in Chapter Two are based on a particular pair of control and data mechanisms, the architectures developed in subsequent Chapters incorporate combinations of those mechanisms, allowing the control flow, data flow and reduction models to be used as different styles for programming a single computer so that each can be used where its particular advantages are needed. Even if eventually data flow or reduction were to completely replace control flow as the dominant model for general-purpose programming, it would nonetheless be important for some time that a general-purpose computer continue to support existing languages and programs based on the control flow model.

Chapter Three described the combination of control flow and data flow in a single architecture, the DCF. Parallel control flow, rather

than sequential control flow, was used since parallel control flow and data flow architectures are very similar. Both have a parallel control mechanism using tokens for instruction activation and are particularly suited to implementation on a packet communication machine organisation. In the DCF architecture concepts from parallel control flow and data flow are very directly combined. The parallel control mechanism uses tokens to control instruction execution, each token being either a control token, as in parallel control flow, or a data token, as in data flow. An instruction's operand may either be provided by a token (the by-value data mechanism of data flow), or referenced from the instruction, as in control flow. The packet communication machine organisation has processors, a matching store (as in data flow) and a separate addressable memory (as in control flow). Apart from the inclusion of both control flow and data flow concepts, an important feature of the DCF architecture is that it allows "non-atomic" procedures which start executing as soon as any inputs are available. This gives greater concurrency than the "atomic" procedures provided in proposed data flow architectures.

In the architectures covered in Chapters Two and Three the principal emphasis is on organising concurrent instruction execution within a single program and supporting that concurrency on a single computer with some multi-processor machine organisation. Generally only the principles of instruction execution and machine organisation are different from those in conventional architecture, with the other von Neumann principles of storage structure, addressing and instruction representation remaining unchanged. In contrast the work covered in Chapter Four was based on systematically generalising each of the von Neuman

principles to a corresponding recursive principle. These recursive principles were formulated to provide a coherent framework for meeting the general requirements of future general-purpose computing systems, particularly using VLSI technology. With VLSI technology it is both possible and desirable to include greater processing power with memory and this should be used to support in hardware a more powerful storage structure, closer to the logical structure of the information being stored. Address spaces should be arbitrarily extensible and support locality by allowing addresses between structurally close entities to be relatively short. Program representation and execution should more closely reflect logical program structures and support modularity. There should be a flexible operational model accommodating a wide variety of programming languages and styles. The need for centralised functions, such as global resource management, should be avoided, and global communication minimised. The development of sophisticated VLSI design technology will enable increasingly diverse special-purpose sub-systems to be implemented in hardware and these should be easily incorporated in the overall general-purpose computing system. Different levels of both hardware and software in computing systems should be unified - for example by adopting the same machine organisation principles both for components on a chip and for computers in a network; and the same storage and addressing models both for program variables and for operating system files.

The most significant characteristic of computing systems is increasing complexity. There are strong arguments that hierarchic structuring is the only way to cope with complexity[66] and this is borne out by the (increasing) predominance of hierarchic structure in

all aspects of computing systems. The important concept embodied in a coherent "recursive" structure (as opposed to a hierarchy of heterogeneous levels) is that there is a general scheme for constructing any level in the hierarchy from the lower level. With the recursive architecture principles this concept is applied throughout the architecture. Although hierarchy does not adequately reflect actual structures (for example a two-dimensional matrix has to be represented, inadequately, as a vector of vectors) it is possibly the richest general structure of sufficient simplicity to be directly supported in hardware.

Chapter Four presented a computer architecture, the RCF architecture, based on and intended to illustrate the broad recursive principles but with the emphasis, as in previous Chapters, being on program representation and execution, and particularly on the combination of control flow, data flow and reduction styles of program organisation. The RCF operational model provides a general framework which can support the specific mechanisms of the other models (such as the self-modifying instructions of reduction, the "unknown" arguments and data tokens of data flow) without incorporating them as an essential part of the model. This approach to the synthesis of other models is very different to, and more successful than, that taken in the RCF architecture. The essentials of the RCF model are the execution structure of a tree of communicating activities which model both processes for program execution and registers (or "file descriptors") for data access; the concept that an instruction's arguments are program fragments to be executed rather than data to be manipulated; the ability to access an instruction and any of its components as an addressable memory cell.

Of the various principles of recursive structuring the most fundamental are the nested objects of the storage structure and nested computing elements of the machine organisation. The dynamic recursive storage structure naturally leads to variable length contextual addresses and a recursive form of program representation and execution, just as the conventional storage structure naturally leads to conventional addressing, program representation and execution. The recursive storage structure could be implemented on a non-recursive machine organisation (as occurs within a UNIX system). Conversely a recursive machine organisation could be used to support the conventional storage structure, addressing and operational model (as, to an extent, occurs in some computer network and parallel computer organisations).

Chapter Five considered a number of computing system designs which incorporate some of the recursive architecture principles. The principal design discussed was that of the LEGO computer which incorporates all the recursive principles and supports the full RCF model. Of the other systems, the UNIX United computer networking system and the R.M. (Recursive Machine) computer architecture are the most relevant to this discussion since their development was largely independent of the RCF work. The UNIX United system provides an example of the applicability of the recursive architecture principles to computer network organisation. It also illustrates a very different, more conventional approach to their realisation than the approach adopted in the LEGO design. The R.M. work provides an example of a computer design, other than LEGO, for which principles of recursive structuring were adopted from the outset. The principal differences between the two is that R.M. has a more sophisticated implementation (particularly in allowing several

activities per computing element, providing more powerful addressing selectors and incorporating indexes into the object structure) whereas LEGO has a more general operational model which provides for a recursive program execution structure.

The various systems covered in Chapter Five were used to illustrate a general discussion of a number of implementation issues arising in (recursively structured) decentralised computing systems. The most fundamental aspect of an implementation is the relationship between logical and physical structure, i.e. the relationship of objects, activities and logical communication channels to memories, processors and physical connections. These relationships are principally determined by the scheme used for allocating logical objects to computing elements' physical memory since activities within an object are allocated to processors in the same computing element, and this allocation in turn determines the communication structure. (Other resource allocation considerations such as the multiplexing of a computing element's processor between the activities that it is supporting are of less significance.) The main distinction identified was that between static allocation where a logical object is synonymous with a particular computing element, and dynamic allocation where objects can move between computing elements under control of some distributed resource allocation scheme. The former is appropriate for special-purpose computing elements and computer networks where the allocation of objects is generally fixed or needs to be under explicit user control. However the latter is more appropriate for the internal organisation of a computer where the structure of program and data objects changes fairly rapidly as programs execute.

Three motivations were identified for the development of novel general-purpose computer architectures, namely the need to improve performance by the use of concurrency, the need to support the increased expressive power claimed for applicative languages, and the need to exploit VLSI technology. The RCF architecture and its initial LEGO implementation provide the physical parallelism, asynchronous communications and decentralised control of resources needed to implement concurrency. They also support the programming structures and mechanisms needed to exploit and control that concurrency. However, as for other novel highly parallel architectures, it is not clear at this early stage of development the exact extent to which performance benefits will ultimately be gained from the potential for concurrency. The major benefits claimed for applicative languages are unbounded data structures, higher order functions and the general lazy evaluation on which these are based, all of which have been included in the RCF program organisation. The recursive machine organisation and LEGO implementation meet the main design requirements for VLSI, namely the need for localised and asynchronous communications and for highly repetitive designs which can be easily scaled down and replicated as the level of integration increases.

6.2. Current and Future Research

There are a number of directions for further investigation of the RCF architecture and the recursive principles on which it is based. Firstly there is the implementation of the particular LEGO design described in Chapter Five and Appendix A. The (LSI) circuit layout of the single chip computing element for this implementation is currently being designed. That work, being carried out by another member of the

Computer Architecture Group at Newcastle University, is partly motivated by the need of the Group to gain experience in the realities of integrated circuit design.

There are a number of possible developments of the RCF architecture in the areas of storage structure and addressing, program representation and execution, machine organisation and implementation. A realistic machine would have to at least include selectors such as "nth component" at the machine code level. It may also be found desirable for the addressing scheme to recognise, within the normal object hierarchy, a sparser hierarchy of particularly significant objects such as: one user's programs and data; one program; one program block. This could be achieved by using special delimiters, e.g. [...], for such objects and including a /out[selector to identify the lowest containing object with those delimiters. Thus an address could select relative to the context of such an object regardless of its own depth of nesting within the object. The inclusion of some form of content addressing would also be worth investigating.

In the operational model there is an important difference between the execution of instructions transmitted between activities and the execution of instructions in the program string. In the former case instructions are discarded as they are executed whereas in the latter case the instructions can form an iterative algorithm. If the model could be extended to allow a general program fragment to be transmitted for execution it would then, for example, accommodate variable instruction sets with the program fragment being an interpreter, and completely general addressing with the program fragment being a pattern matching

algorithm. In addition to programming languages based on control flow, data flow and reduction, there are two other classes of languages which a future general-purpose architecture may need to support, namely object-oriented languages such as Smalltalk[60] and logic languages such as PROLOG[67]. It will be necessary to investigate the suitability of the RCF model for these languages.

For the machine organisation the most important areas are the fuller development of a multi-level organisation employing different implementation techniques at different levels; the inclusion of special-purpose computing elements; and the inclusion of computing elements with differing memory/processor ratios (ranging from the equivalent of backing store with a very high ratio to cache with a relatively low ratio). The outstanding research issue is the development of a general resource allocation scheme for the dynamic mapping of a changing hierarchy of objects onto a fixed (heterogeneous) hierarchy of machine resources. Such a scheme would need to be highly decentralised, specialisable to accommodate and utilise effectively the differing characteristics at different levels of recursive machine organisation, and able to manage the movement of objects between computing elements with differing memory/processor ratios in response to the changing distribution of activities within a program structure.

There needs to be considerable experimental investigation into the programming of the RCF architecture and the combined use of control flow, data flow and reduction models in the same program. The RCF architecture incorporates the concepts underlying conventional languages and novel applicative languages used to program data flow and reduction

machines. Thus one line of investigation is to consider in detail the generation of RCF machine code from one or more such languages. A promising starting point for investigating the combination of various models would be to implement a common procedure call interface, based on the RCF model, for interfacing between (independently compiled) procedures programmed in, for example, existing control flow and applicative languages, with the other language features implemented in their standard ways.

The RCF architecture as discussed in this thesis and its proposed initial LSI implementation are principally concerned with applying the recursive architecture principles to the design and programming of a single multi-processor computer. The complementary approach is to investigate the application of the RCF concepts to a network of conventional single processor computers. An experimental network implementation of the RCF architecture, obtained by programming each computer in a network to emulate the RCF model, would also be a useful simulation of a single multi-processor RCF computer. Compared with an actual hardware implementation, such a "network" implementation would be quick to develop and very flexible, whereas compared with a normal simulation it would be a relatively realistic implementation supporting several simultaneous users and allowing investigation of genuine concurrency in program execution and genuine communications characteristics. The close similarities between the RCF architecture and the UNIX United system suggest that the latter could be initially used as the basis for a network RCF implementation.

It is intended that the next stage in the development of the ideas

covered by this thesis will be two implementation exercises. The first of these will be to implement a full RCF system on a network of conventional computers to provide a framework for investigating the programming and resource management issues, and for possible developments of the RCF model. The second will be further work on LSI and VLSI implementation of computing elements based on the RCF model.

Appendix A - A Machine Code for the LEGO Computer

This Appendix describes a machine code designed for the initial LEGO implementation of the RCF architecture discussed in Section 5.1. First there is a general description of the machine as seen by the machine code programmer, followed by a detailed description of the machine code. This is presented by defining a symbolic assembler code using a BNF syntax notation in which braces {...} indicate zero or more repetitions of the enclosed material. Where motivation of the machine code facilities seems necessary, notes in [...] brackets are included to relate those facilities to the programming constructs discussed in Chapter Four. Boldface is used for terminal symbols of the syntax.

A.1 - Machine Model

```
<program> ::= ( { <object> } )
<object> ::= <primitive object> | ( { <object> } )

<program string> ::= ( { <item> } )
<item> ::= <symbol> | <activity>
<symbol> ::= <delimiter> | <primitive object>
<delimiter> ::= ( | )

<primitive object> ::= <data item> | <instruction> ;
<instruction> ::= <access> | <addressing> | <operation> |
                  <creation> | <exception>
<activity> ::= ]
```

The two major elements of the machine are the symbols representing the program and the activities executing it. The program is structured as a hierarchy of objects comprising primitive objects (instructions and data) and matching pairs of delimiter symbols for the start (and end) of compound objects. As the program is executed it is modified and its current state is the <program string>, a sequence of items which includes the symbols of the program representation. (During a

modification of the program string, e.g. the insertion of an object, there may be a temporary delimiter imbalance in the string and thus the <program string> does not always conform strictly to the <program> syntax.)

Each activity has a current position in the program and is modeled as being itself an item in the string together with the symbols of the program. There are communications connections between activities allowing an activity to transmit objects to and receive objects from connected activities. These connections support tree structures of activities with each activity capable of being connected to a superior activity (identified as **R0**) and a number (6) of subordinate activities (identified as **R1**, **R2** etc.).

An activity is executing symbols from some instruction source, either the program string or the objects transmitted from a connected activity. [Messages such as "**copy**", "**take**" and " **execute**" sent from a superior activity to a subordinate activity are instructions executed by the subordinate activity as a result of its instruction source being the connection to its superior activity ("**execute**" switches the instruction source to be the program string); the effect of the "eval" operator, executed by an activity **P**, is achieved by **P**'s instruction source being the connection to the subordinate activity which is evaluating the operand.]

Normally execution is sequential, with the activity's position moving past each item in the string as it is executed, or successive items being executed as they are received from the connected activity. The following Sections describe the effects of executing the different types

of items from the instruction source, namely delimiter symbols, primitive data items and various types of instructions. The different types of instructions are: Access instructions which operate on the program string as a data structure, reading objects from the string to be transmitted to a connected activity and modifying the string with objects received from a connected activity; Addressing instructions which move the position of the activity within the string and also may suspend the activity and switch its instruction source; Operation instructions which perform for example arithmetic operations on objects received from connected activities and transmit the result to connected activities; Creation instructions which generate parallelism by creating new activities; Exception instructions for handling various exception conditions.

When an activity's instruction source is the program string the next item may be another activity rather than a symbol of the program. Generally two activities accessing the string (i.e. with the string as instruction source or executing an access instruction) will follow each other through the string with the activity on the left waiting for the other to progress. However the activity on the right, PR, must be passed by that on its left, PL, if the further progress of PR is dependent on an event that could possibly never occur (that is, if it has been suspended and thus is dependent on another activity to modify the string, or is dependent on a connected activity sending it an object, for example when its instruction source is a connected activity).

A.2 - Delimiters

An activity has a current object of execution in the string. If the instruction source is the program string and the next item is the closing delimiter of the current object then the activity and its subordinates terminate. In all other cases a delimiter is ignored and execution continues with the next item from the instruction source.

The current object is established as being the object to the right of the activity when (i) the activity is first created, (ii) the instruction source is switched from being the connection with its superior to be the program string, (iii) the activity executes the first non-addressing instruction after a series of addressing instructions one of which moved the activity outside the previous current object.

A.3 - Data Items

Data items have two roles. Firstly a data item may be the next item to be executed from the instruction source in which case the activity retains its current position in the program string and its instruction source is switched to be the connection to the superior activity [from which typically an access instruction, e.g. to "copy" the data item, will then be received].

Secondly data items form the operands of operators which are described, together with the syntax of data items, in A.6.

A.4 - Access Instructions

```
<access> ::= <action> <act>
<action> ::= copy | repl | take | insert
<act> ::= R0 | R1 | R2 ... R6 | .
```

An access instruction generally specifies some action on a target object in the program string and a connected activity (**R0**, **R1** etc.) to which that object is transmitted or from which a new object for the string is received. The target object is the object to the right of the activity's position (to the right of the access instruction itself if it is executed from the string). A **copy** transmits a copy of the target object. A **replace** replaces the target object with the new object. A **take** is a destructive copy, transmitting the target object and deleting it from the string. An **insert** inserts the new object in the string to the immediate left of the next symbol to the right of the activity. A **.**, indicating a "null activity", can be specified instead of an actual activity. An object transmitted to **.** is just discarded. An object received from **.** is always the empty object **()**. Thus, for example, **take.** just deletes the target object and **repl.** replaces it with **()**. On completion of any access instruction the activity is positioned to the immediate right of the target or new object in the string.

An action (**replace** or **take**) which deletes an existing object will proceed a symbol at a time, deleting each symbol in turn. There may be another activity positioned within the object being deleted and the deletion will generally wait at that point until the other activity moves. However, in circumstances mentioned in A.1, it may be necessary for the deleting activity to pass the other activity and in doing so that activity is terminated.

[In the example used in Section 4.1.4 to discuss the general kCF model it was necessary for the **replace** action to be acknowledged. In the LEGO implementation this acknowledgement is implicit in the communications flow control mechanism. However alternative implementations might not have that implicit acknowledgement. Explicit Acknowledgement can be programmed when required. The **replace** instruction, followed by the new object, is generally sent from an activity **P** to a subordinate **Q** positioned at the object to be replaced. After sending the new object, **P** can use routing operations (see A.6) to send instructions to **Q** which cause **Q** to send the required acknowledgement to **P**.]

A.5 - Addressing Instructions

```
<addressing> ::= <selector> <switch> | <selector> ? <switch> |
                 !<selector> <switch> | !<selector> ? <switch>
<selector> :: out | start | <- | / | in | -> | end | esc | $ <act>
<switch> ::=  /e | /<act>  | /
```

An addressing instruction includes a selector which specifies a new position for the activity and a switch to possibly change its instruction source. The selectors **out** ... **esc**ape are those defined in Figure 25 of Section 4.1, and are relative to the position of the activity (following the addressing instruction itself if that is executed from the program string). For selectors **$R0** ... **$R6** the new position is to the immediate right of the specified activity. If a subordinate activity **Rn** is specified but does not exist then the **Rn** subordinate of the superior is used, and if it does not exist then that of its superior, etc. [A **$Rn** will form the first selector of a base-relative address, used for example in addressing procedure parameters, and is defined such that in the sub-tree of activities executing the procedure,

all activities can address parameters relative to the position of a particular subordinate of the sub-tree's root activity.]

The switch in an addressing instruction may specify that following the execution of the instruction the instruction source is to become the program string at the new position (**/e**); the instruction source is to become the connection to a specified activity (**/Rn**) (if the null activity is specified, **/.**, then the activity terminates); or that there be no change (**/**). [In a normal address where the addressed object is executed, the last selector would have a **/e** suffix; In a quoted address where the addressed object is not to be executed, the last selector would have a **/RO** suffix switching control to the superior, as occurs when executing a data item.]

The selector may be suffixed by a **?** specifying that the activity be suspended at its new position. The suspended activity will be reactivated when another activity performs a **replace** or **take** action on the object at the new position or an **insert** action to insert a new object at that position. [The "unknown argument", "**(?)**", would be represented as **(out?/;).**]

The selector may be prefixed by a **!** in which case the rest of the instruction is not executed and the instruction (if it is in the program string) is modified by the **!** being removed. [This provides the exclusion argument, "(excl)", which in machine code is **(!out?/;).**]

A.6 - Operations

```
<operation> ::= <typed op.> | <compare op.> | <special op.>

<typed op.> ::= <int op.> | <bool op.> | <bit op.>
<int op.> ::= + | - | 0-
<bool op.> ::= & | ! | ^! | ^
<bit op.> ::= '& | '! | '^! | '^

<compare op.> ::=  = | ^= | > | >= | < | <=
<special op.> ::= if | == | := | <act> , <act> ::= <act>

<data item> ::= <value> | <error>
<value> ::= <int> | <bool> | '<bit> | "<char>
<int> ::= ... -2 | -1 | 0 | 1 | 2 ...
<bool> ::= T | F
<bit> ::= 1 | 0
<char> ::=  <any ASCII character>
<error> ::= #

<obj> ::= <instruction> | <value> | ( { <obj> } )
<any> ::= <object>
```

An operation instruction processes operand objects to produce a result object as defined by its operator. Each operand is received from a specific subordinate activity (usually there is one operand from R2 and one from R3). The result is transmitted to subordinate R1. If there is no R1 subordinate then the result will be sent to the superior R0 in response to any object from the superior. If there is no superior either then the result is just discarded. Figure A1 shows all the operators and the types of valid operands and results produced, these types being specified by the syntax for <data item>, <obj> and <any>. The typed operators are straightforward, each having as its operands and results primitive objects of the same type (integer, boolean or bit). The other operators may involve compound objects and the special error object. An error object is produced as the result of applying an operator to an operand of the wrong type (e.g. multiplying two characters or an integer and an error object). In defining the type of valid operands

there is a distinction between a general object, <any>, and an object, <obj>, restricted to exclude error objects.

oper-ator	operands				result	meaning
	R2	R3	R4	<act>		
integer						
+	<int>	<int>			<int>	add
-	<int>	<int>			<int>	subtract
0-		<int>			<int>	unary minus
boolean						
&	<bool>	<bool>			<bool>	and
!	<bool>	<bool>			<bool>	or
^!	<bool>	<bool>			<bool>	nor
^	<bool>				<bool>	not
bit						
'&	<bit>	<bit>			<bit>	and
'!	<bit>	<bit>			<bit>	or
'^!	<bit>	<bit>			<bit>	nor
'^	<bit>	<bit>			<bit>	not
compare						
=	<obj>	<obj>			<bool>	equal
^=	<obj>	<obj>			<bool>	not equal
>	<obj>	<obj>			<bool>	greater
>=	<obj>	<obj>			<bool>	greater or equal
<	<obj>	<obj>			<bool>	less
<=	<obj>	<obj>			<bool>	less or equal
special						
if	<any>	<any>	<bool>			conditional
==	<any>	<any>			<bool>	equivalent
:=	<any>				<any>	identity
::=				<any>	<any>	routing

Figure A1 - The Operators

A compare operator treats its operands as two strings of symbols (i.e. delimiters, instructions and data items) which are compared in pairs from left to right. The strings are equal if all pairs of

corresponding symbols are equal, otherwise the result is the result of comparing the leftmost unequal pair according to the following ordering -) (lowest); '0; '1; F; T; integers in numeric order; characters in standard order; instructions in arbitrary order; (. [The position of the delimiters in this ordering gives the effect of recursively comparing compound objects, for example -

(("e "v "e)) < (("e "v "e) "j) < (("e "v "e) ("j "i "m))]

The conditional operator (**if**) has three operands and the result is the first (**R2**) or second (**R3**) depending on whether the third (**R4**) is True or False. The equivalence operator (==) tests the operands for equality and allows an object to be tested for error. For the identity operator (:=) there is a single operand which is transmitted unchanged as the result. For the routing operator (::=) the result is its single operand from a specified source activity and copies are transmitted to two specified destination activities. If the null activity . is specified as the source then () is used as the operand object. If a destination activity is non-existent or is specified as . then its copy is just discarded.

The routing operator is used in conjunction with access instructions to replicate objects and to organise any communication that may be needed in addition to the simple operand/result communication pattern of the other operations. As an example the following sequence of instructions synchronise the executing activity P with its R1 subordinate, Q -

```
        i1          i2          i3
copy R1;  .,R0 ::=.;  .,. ::=R1;
```

The first, **copy**, instruction transmits to Q the following instruction,

i2. When **Q** has finished executing the previous instruction from **P** it will execute the **i2** which specifies that **Q** transmit to its superior, **P**, the empty object **()**. The next instruction executed by **P**, **i3**, requires an object from its **R1** subordinate, **Q**, and thus synchronises those two activities. When **P** receives that object, **()**, it is just discarded.

A.7 - Activity Creation

<creation> ::= **sub** <act> |
source <act> |
par

A **sub** instruction, such as **sub Rn**, creates a new subordinate activity **(Rn)** to the the immediate left of the creating activity. The new activity is connected to the creating activity as its superior and that is its initial instruction source. The new activity itself has no subordinates. A **source** instruction is the same except that the new subordinate activity is to the right of its superior, the instruction source for the created activity is implicitly switched to be the program string, and the instruction source for the creating activity is implicitly switched to be the connection with the created activity.

A **parallel** instruction creates a new independent activity which has the same current object as the creating activity, the same position as the creating activity (following the **par** instruction if that is executed from the program string) and with the program string as its instruction source. The creating activity then executes an implicit **->/** addressing instruction so that the next object it executes is that following the first object executed by the created activity.

A.8 - Exception Handling

<exception>::= **ok** | **err** | **skip** | **skip** <act>

Associated with an activity is an error status which indicates whether an error has occurred. This is cleared by an **ok** instruction, set by an **err** instruction and is also set by the occurrence of some of the exception conditions described below. A **skip** instruction tests the error status of the activity itself or a specified activity and executes an implicit **->/** addressing instruction if the status is clear (that is, the activity skips over the following object which would contain code to deal with exceptions).

The exception conditions that can occur are -

(i) Invalid instruction source - If the instruction source is switched to be the connection to a non-existent activity or a connected activity terminates while the connection to it is the instruction source, then the activity terminates.

(ii) Invalid selectors - A selector in an addressing instruction executed by an activity **P** is invalid if: it is an **in/** selector and the next symbol to **P**'s right is a primitive object or a **)**; it is a **->/** (or **<-/**) selector and the next symbol to **P**'s right (or left) is a **)** (or **(**); it is an **out/** or **escape/** selector and **P**'s enclosing object is the outermost object, the total program; it is a **$Rn** selector where **Rn** does not exist or the null activity, **.**, is specified. The error status is set and the effect on **P**'s position is that of a "null" selector, **//**, (except in the case of an invalid **out** or **escape/** selector which will act as a **start** or **end** selector).

(iii) Invalid access instructions - An access instruction is invalid if the activity to which the target object is to be transmitted, or the activity from which the new object is to be received, does not exist, or (except in the case of **insert**) the next symbol to the right of the activity is a **)**. The error status is set and where appropriate the error object is transmitted as the target object.

(iv) Invalid operation - An operation instruction is invalid if one of the operand objects is of the wrong type or the activity from which it should be received does not exist. The error status is set and the error object is transmitted as the result.

(v) Invalid creation - A **sub Rn** or **source Rn** instruction is invalid if the specified subordinate already exists or the null activity, **.**, is specified for **Rn**. The error indicator is set but otherwise the instruction has no effect.

A.9 - An Example

In order to illustrate the relationship between the machine code described here and the programming constructs used in Chapter Four, Figure A2 shows the machine code equivalent to some of those constructs. The example program fragment used (a) is part of that in Figure 39. The motivation for this example and its general operation was discussed in Section 5.1.7. The machine code instruction sequence corresponding to each construct in (a) is shown in (b) which contains comments explaining the detailed operation of that machine code.

(a) example

```
                P         Q                R              S
"( ¦ iter- ( ¦ c2 c3 (?) ) ( ¦ := 6 /out ) ¦ /E )"
                    C:              D:             r:
```

(b) machine code

```
program
construct                          machine code

   "iter"- [An iterative operator on stream operands - P sets up sub-
            ordinates at operands and result and repeatedly takes
            operand values and inserts result values]
           [Set up]
 1         ( sub R2; [creates R2 subordinate Q with P as instr. source]
 2           copy R2; [Q is sent the following object to execute]
 3           esc/e;  [address of C, specifying execution (/e)]
 4           sub R3; copy R3; ( esc/; ->/e; ) [as 1-3, for R,on D]
 5           sub R1; copy R1; ( esc/; ->/; ->/e; ) [similarly S,r]

           [Repeated operations]
 7         ( copy R2; take R0; [Q is sent instr. to take object to P]
 8           copy R3; take R0; [as 7, causing R to return its result]
 9           copy R1; insert R0; [as 7- S inserts next object sent]
10           -; [P performs subtraction on objects from Q and R, send-
                ing result for insertion by S]
11           out/; ) ) [return and repeat from last ( ]

   ":="  - [A normal (not iterative) operator]
12         (sub R2; copy R2; (esc/; ->/e; copy R0; /.;) :=;)
                  [as 1,2,7,10 but -
                    using copy instead of take with termination
                    of subordinate activity; single operand for
                    identity operation, := (instead of subtraction);
                    no result activity, R3, thus result to R0]
13          ->/e; [skip over single argument, the folowing 6 ]

14 "(?)"    ( out?/; ) [suspends activity activity to await operands]

   "E"    [A compound address executed by S]
15         (source R5; [A sub. activity T is created to execute rest
                        of this object]
16          copy R0;  [following object sent to T's superior, S]
17          (esc/; ->/; . . . <-/e;) [selectors corresponding to /E]
18          /.;) [terminate T]
```

Figure A2 - Machine code for part of example in Figure 39

References

1. P.C. Treleaven, "Exploiting Program Concurrency in Computing Systems," IEEE Computer, pp.42-50. (January 1979).

2. J. Darlington, P. Henderson, and D.A. Turner (eds.), Functional Programming and its Applications, Cambridge University Press (1982).

3. J. Backus, "Can Programming be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs," Comm. ACM Vol. 21(8), pp.613-641. (August 1978).

4. R. Kowalski, Logic for Problem Solving, Elsevier-North-Holland (1979).

5. D.A. Patterson and C.H. Sequin, "Design Considerations for Single Chip Computers of the Future," IEEE Transactions on Computers Vol. C-29(2) (February 1980).

6. C.H. Sequin, "Single Chip Computers, The New VLSI Building Blocks," CALTECH Conf. on VLSI, pp.435-445 (January 1979).

7. A.M. Despain and D.A. Patterson, "X-tree: A Tree Structured Multiprocessor Computer Architecture," Proc. Fifth. Int. Symp. Computer Architecture, pp.144-151 (1978).

8. H.T. Kung, L.J. Guibas, and C.D. Thompson, "Direct VLSI Implementation of Combinatorial Algorithms," Proc. Conf. on VLSI: Architecture, Design, Fabrication, California Institute of Technology, pp.509-525 (1979).

9. E.D. Lazowska et. al., "The Architecture of the Eden System," Technical Report 81-04-01, University of Washington (April 1981).

10. Elliot Organick, A Programmer's view of the INTEL 432 System, McGRAW-HILL.

11. I. Barron, "The Transputer," pp. 343-357 in The Microprocessor and its Applications, ed. D. Aspinall, Cambridge University Press (1978).

12. R.P. Hopkins et. al., "A Computer Supporting Data Flow, Control Flow and Updatable Memory," Technical Report 144, Computing Laboratory, University of Newcastle upon Tyne (September 1979).

13. R.P. Hopkins, "A Data Flow Computer with Addressable Memory," Proc. Data Driven and Demand Driven Languages and Machines Workshop, Tolouse France (1979).

14. P.C. Treleaven and R.P. Hopkins, "Decentralised Computation," Proc. Eighth Int. Symp. Computer Architecture (May 1981).

15. P.C. Treleaven, R.P. Hopkins, and P. Rautenbach, "Combining Data Flow and Control Flow Computation," Computer Journal Vol. 25(2), pp.279-290 (1982).

16. P.C. Treleaven, D.R. Brownbridge, and R.P. Hopkins, "Data Driven and Demand Driven Computer Architecture," Computing Surveys Vol. 14(1) (March 1982).

17. P.C. Treleaven and R.P. Hopkins, "A Recursive Computer Architecture for VLSI," Proc. Ninth Int. Symp. Computer Architecture, pp.229-238 (1982).

18. I. Gouveia Lima, R.P. Hopkins, L. Marshall, D. Mundy, and P.C. Treleaven, "Decentralised Control Flow - BAsed on unix," Proc. SIGPLAN 83 Symp. on Programming Language issues in Software Systems, SIGPLAN Notices Vol. 18(6) (June 1983).

19. T.L. Wat, "The Implementation of a JUMBO Computer on Three M6800 Microcomputer Systems," M.Sc. Dissertation, Computing Laboratory, University of Newcastle upon Tyne (1979).

20. D.R. Brownbridge, "A Simulator for Concurrent Architectures," M.Sc. Dissertation, Computing Laboratory, University of Newcastle upon Tyne (1979).

21. Z. Manna, Mathematical Theory of Computation, McGraw Hill (1974).

22. D.P. Friedman and D.S. Wise, "CONS should not evaluate its arguments," pp. 95-103 in Automata, Languages and Programming, ed. S. Michaelson and R. Milner, Edinburgh University Press, Edinburgh (1976).

23. P. Henderson and J. Morris, "A Lazy Evaluator," Proc. 3rd. ACM Symp. on the Principles of Programming Languages, pp.95-103 (1976).

24. Arvind et. al., "The Id Report: An Asynchronous Programming Language and Computing Machine," Technical Report 114, Department of Information and Computer Science, University of California, Irvine (May 1978).

25. J. McCarthy et. al., The LISP 1.5 Programmers Manual, Cambridge, Mass. (1962).

26. W.B. Ackerman , "A Structure Processing Facility for Data Flow Computers," Computation Structures Group Memo 165, MIT Laboratory for Computer Science.

27. A.L. Davis, "The Architecture and System Method of DDM1: A Recursively Structured Data Driven Machine," Proc. Fifth Int. Symp. Computer Architecture, pp.210-215. (April 1978).

28. J. McCarthy, "A Basis for a Mathematical Theory of Computing," pp. 33-70 in Computer Programming and Formal Systems, ed. P. Braffort and D. Hirschberg, North-Holland (1963).

29. D.P. Friedman and D.S. Wise, "An Indeterminate Constructor for Applicative Programming," Conf. Record of 7th Annual ACM Symp. on the Principles of Programming Languages, Las Vegas (January 1980).

30. W.A. Kornfield, "Combinatorially Implosive Algorithms," Comm. ACM Vol. 25(10) (October 1982).

31. J.H. Patel, "Processor-Memory Interconnection for Multiprocessors," Proc. Sixth Int. Symp. Computer Architecture, pp.168-177 (April 1979).

32. R.J. Swan, S.H. Fuller, and D.P. Siewiorek, "CM*: A Modular Multiprocessor," Proc. Nat. Comp. Conf., pp.637-644 (June 1977).

33. P.C. Treleaven et. al., "The Design of Highly Concurrent Computing Systems," Technical Report 126, Computing Laboratory, University of Newcastle upon Tyne (1978).

34. L. Foti, D. English, R.P. Hopkins, D. Kinnement, P.C. Treleaven, and L. Wang, "Design of a Reduced Instruction Set Multi-Microprocessor System - RIMMS," Internal Report, Computing Laboratory, University of Newcastle upon Tyne (February 1983).

35. J.B. Dennis and D.P. Misunas, "A Preliminary Architecture for a basic Data Flow Processor," Proc. Second Int. Symp. Computer Architecture, pp.126-132. (1975).

36. I. Watson and J. Gurd, "A Prototype Data Flow Computer with Token Labeling," Proc. Nat. Comp. Conf. Vol. 48 , pp.623-628. (1979).

37. W.E. Kluge and H. Schlutter, "An Architecture for the Direct Execution of Reduction Languages," Proc. Int. Workshop on High Level Language Computer Architecture, Fort Lauderdale, Fla., pp.174-180, University of Maryland and Office of Naval Research (May 1980).

38. P.C. Treleaven and G.F. Mole, "A Multi-Processor Reduction Machine for User-Defined Reduction Languages," Proc. Seventh Int. Symp. Computer Architecture, pp.121-129 (May 1980).

39. J.W. Clark et. al., "SKIM - The S K I Reduction Machine," LISP-80 (1980).

40. J. Darlington and M. Reeves, "A Reduction Machine for Parallel Evaluation of Applicative Languages," Proc. Conf. on Functional Programming and Computer Architecture, MIT (Oct. 1981).

41. R.M. Keller et. al., "A Loosely-coupled Applicative Multiprocessing system," AFIPS Conf. Proc. Vol. 48, pp.861-870. (1978).

42. J.B. Dennis, "The Varieties of Data Flow Computers," Proc. First Int. Conf. on Distributed Computing Systems, pp.430-439. (October 1979).

43. D.A. Turner, "A New Implementation Technique for Applicative Languages," Software Practice and Experience Vol. 9, pp.31-49. (1979).

44. B.H. Liskov, "A Design Methodology for reliable Software Systems," AFIPS Conf. Proc. Vol. 41, pp.191-199 (1972).

45. B.H. Liskov et. al., "CLU Reference Manual," Computer Science Memo 161, MIT (July 1978).

46. D.P. Friedman and D.S. Wise, "Aspects of Applicative Programming for File Systems," Proc. ACM Conf. on Language Design for Reliable Software, SIGPLAN notices Vol. 12(3), pp.41-45 (March 1977).

47. M.R. McLauchlan, "A Purely Functional VLSI Layout Language," Internal Report PMM 90, Computing Laboratory, University of Newcastle upon Tyne (Sept. 1980).

48. P. Henderson, Functional Programming Applications and Implementation, Prentice Hall International (1980).

49. D.S. Wise, Private Communication.

50. D.D. Chamberlin, "The Single Assignment Approach to Parallel Processing.," Proc. Nat. Comp. Conf. Vol. 39 , pp.263-269. (1971).

51. L.G. Tesler and H.J. Enea, "A Language Design for Concurrent Processes.," Proc. Nat. Comp. Conf. Vol. 32 , pp.403-408. (1968).

52. B. Randell, "The Structuring of Distributed Computing Systems," Technical Report 181, Computing Laboratory, University of Newcastle upon Tyne (1983).

53. V.M. Glushkov et. al., "Recursive Machines and Computing Technology," Proc. IFIP Congress, pp.65-70 (1974).

54. R.M. Barton, UK Patent Specification 1 503 321-325.

55. W. Wilner, "Recursive Machines," Internal Report, Xerox Palo Alto Research Centre (1980).

56. D.M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," Comm. ACM Vol. 17(7), pp.365-375 (July 1974).

57. The Conference on Data System Languages (CODASYL) DBTG report, October. 1969.

58. A. Hoare, "Communicating Sequential Processes," Comm. ACM Vol. 21(8), pp.666-677 (August 78).

59. D. May and R. Taylor, "OCCAM," 1983 Conf. on Parallel Processing (1983).

60. D. Ingalls, "The Smalltalk-76 Programming System Design and Implementation," Proc. SIGPLAN Conf. on the Principles of Programming Languages, pp.9-15. (1978).

61. D.A. Turner, Private Communication.

62. C.L. Seitz, "System Timing," in Introduction to VLSI Systems, ed. C. Mead and L. Conway, Addison Wesley (1980).

63. D.R. Brownbridge, L.F. Marshal, and B. Randell, "The Newcastle Connection or UNIXes of the World Unite," Software Practice and Experience Vol. 12 (1982).

64. D.A. Patterson and C.H. Sequin, "RISC I: A Reduced Instruction Set VLSI Computer," Proc. Eighth Int. Symp. Computer Architecture, pp.443-457 (1981).

65. W. Wilner, "Instruction Execution," Internal Report, Xerox Palo Alto Research Centre (1981).

66. H. Simon, "The Architecture of Complexity," Proc. American Philosophical Society Vol. 106(6) (1962).

67. M. vanEmden and R. Kowalski, "Semantics of Prolog as a Programming Language," Journ. ACM Vol. 7(3), pp.733-742 (1976).