Behavioural Specification and Simulation of Minimum Configuration Computer Systems

by

Ian Gorton BSc

A thesis submitted to the Council for National Academic Awards in partial fulfilment of the requirements for the degree of Doctor of Philosophy.

Sponsoring Establishment : Department of Computer Studies
Sheffield City Polytechnic

Collaborating Establishment: Motorola (UK) Ltd.

November 1988

Ian Gorton : Behavioural Specification and Simulation of Minimum Configuration Computer Systems

Abstract

The ultimate goal of Computer-Aided Design research in the area of digital circuits is the automatic synthesis of a complete solution from a behavioural specification. This thesis describes an attempt to attain this ideal in the more limited realm of designing single-board control systems, constructed from general-purpose microprocessor components. The difficulties currently encountered in designing and implementing microprocessor control systems are outlined, and the architecture of an integrated, knowledge-based design system is proposed as a method of overcoming these difficulties. The design system encompasses both behavioural and structural design functions. However, only the tools and techniques required to fulfil the behavioural design functions are considered in detail in this project.

A review of previous work in the field of automated digital circuit design and software and hardware specification languages is presented. The major features of a novel language for specifying and simulating control system behaviour are then described, together with an intermediate design description notation, which facilitates the generation of microprocessor assembly language code directly from behavioural specifications. The design and implementation of a fast, generalised microprocessor simulation facility constructed from transputers is discussed, and its performance potential analysed. The simulation facility enables the complete design for a given application to be tested, before any actual hardware construction takes place. Finally, an evaluation of the behavioural specification, synthesis and simulation techniques developed in this project is presented, and the benefits perceived from adopting such techniques are summarised. Issues concerning the integration of these techniques with the knowledge-based structural design tools are also dealt with, and suggestions for further developments and enhancements are identified.

Acknowledgements

I would like to thank my supervisors Dr J.M.Kerridge, Dr B.Jervis and Dr J.Travis for their help and advice during the period leading up to this report.

I would also like to thank Sally Sahni, Rebecca Strachan, Paula Fleetwood, Linda Fessler, Ursula Everson and Toni Olewicz who contributed to the work as part of their Masters Degree.

Contents

1 Project Overview

1.1 Minimum Configuration Systems

Microprocessors have made a significant impact on all aspects of control systems[1.1]. Direct digital control of machine processes has created production methods that are more reliable, economic and generally more efficient. The low cost, flexibility and processing speed of control systems constructed from Large Scale Integration (LSI) components enables them to be applied economically to even the simplest control tasks[1.2], often replacing the need for complex hard-wired logic[1.3]. Such possibilities have led to a move away from centralised mainframe or minicomputer control systems, towards decentralised control based upon many embedded microprocessors, each dedicated to performing a simple part of the whole control task[1.4].

In order to be cost-effective, or when the microprocessor is tightly coupled with other circuitry, it is often desirable to design special printed circuit boards (PCBs) to implement the different system functions. In these cases, the microprocessor system can be viewed as merely one set of components amongst others on the board. Such systems do not normally require a disk system or visual display unit, and are consequently difficult to develop and test[1.5].

Many microprocessor-based control systems, whether sub-components of a large decentralised system, or a stand-alone dedicated control system, are typified by containing the minimum number of components required to perform the control task. Systems in this category usually comprise a microprocessor and clock, small amounts of read-only and read-write memory and the capability to interface with the physical process being controlled. For this reason, such systems are referred to as minimum configuration systems (MCSs). Due to their small component count, MCSs can virtually always be constructed on a single

printed circuit board. More formally, an MCS may be defined as any size system within the basic load limitation of the microprocessor concerned[1.6].

1.2 The Design and Implementation of Minimum Configuration Systems

The design and implementation of an MCS is a complex task, requiring a wide diversity of expertise from the system designer. The designer must have a thorough knowledge of all the issues involved, from individual component characteristics to software design and implementation.

Once a requirements specification for the proposed control system has been finalised, a solution to the problem must be designed. The design describes how the processes in the specification are to be carried out. Such a design involves both the selection of a set of components and the design of the software routines needed to implement the solution. Decisions must be taken as to which functions to implement in hardware and which in software, and detailed algorithms and component interfaces must be specified[1.7]. The design is governed mainly by the performance levels required of the system in the specification, and the control functions that must be carried out[1.8].

The physical realisation of the design occurs during implementation. Software development and hardware construction may both proceed in parallel. When the hardware and software have been separately tested, they can be brought together for system testing. It is almost certain that some modifications and corrections will be necessary before the system satisfies its specification. However, if system testing reveals fundamental design errors, a considerable amount of redesign and implementation may be required.

1.3 The Difficulties of System Design

There are three main stages during which errors may be introduced into an MCS design[1.5]. These are:-

1. Interpretation
The requirements specification for the desired system may contain inconsistencies and ambiguities, which may lead the designer to make incorrect assumptions. Even complete specifications may be misinterpreted. This may result in logical errors being introduced into the hardware and software design.

2. Hardware
The task of designing the hardware for an MCS comprises two related activities: component selection and component interconnection. Component selection is complicated by the fact that many LSI components have the same functional characteristics but different operational characteristics. Further, components from one microprocessor family may not easily interface with components from others. Thus, in the absence of an integrated set of evaluation tools, the designer must rely on previous experience to ensure that the chosen components can fully satisfy the specification.

Component interconnection is a well understood, mechanical task, which is tedious and prone to errors. It requires the designer to check through the individual component data sheets, in order to identify the precise connections between them. Errors introduced during this process can be of a very subtle nature, making them difficult to locate during testing.

3. Software
The software requirements of MCSs vary according to the complexity of the application to be implemented. In general though, software should be simple to construct and test, efficient, maintainable and portable[1.9, 1.10]. Efficiency can be achieved by writing the control software in the

assembler language of the microprocessor in use. This approach however does not lend itself to satisfying the goals of maintainability, portability and ease of construction. These three requirements are best achieved by the use of a high-level language such as Pascal[1.11] or ADA[1.12]. These provide abstraction of control and data representation, and, coupled with modern compiling techniques, can give a level of efficiency approaching that of handwritten machine code.

Still, high-level languages do not provide a complete solution to the software development problem. Due to their general-purpose nature, they do not include constructs for accessing low-level processor facilities such as interrupts and input-output(I-O) interfaces. To achieve these, assembler subroutines have to be created. These subroutines are, however, processor-dependent. Any change or upgrading of processor or system configuration will require all the assembler routines to be rewritten in the assembly language of the new target processor.

Further problems arise during software testing. The initial tests take place on a software development machine, not upon the target hardware configuration. This means that a software test harness needs to be built, which simulates the behaviour of the environment in which the control software is to operate. The important aspect of test harnesses is that they are application specific, and do not form part of the final product. When the software seems to function correctly and is ready to be tested on the target hardware, the test harness becomes effectively obsolete.

Consequently, much effort is expended in designing, building and testing a prototype solution. Many of the errors located during testing may be simple implementation mistakes, which are relatively straightforward to correct. However, errors introduced during the design phase are often of a much more serious nature, and may only be detected when the prototype hardware and software are put

together[1.5]. For this reason, design errors can be costly and time-consuming to correct.

1.4 A Solution to the Problems of MCS Design

Clearly then, due to the lack of suitable techniques, the process of developing MCSs is complex and potentially expensive. Therefore it seems there is a requirement for a set of integrated software tools, which can design and simulate both the hardware and software for MCSs from a high-level system specification. This approach would ensure that all serious design faults have been removed before implementation proceeds[1.5].

Given a specification of the hardware and software requirements of a system, and knowledge of an appropriate set of components, the design system could construct a simulation of a possible solution. The simulation could then be evaluated and, if necessary, modified by the designer. When the simulation satisfies its specification, the design system could automatically generate the printed circuit board (PCB) layout for the solution, so that the implementation can begin. Figure 1.1 shows the proposed architecture of the design system.

The five major components of Figure 1.1 are:-

1. Problem Specification

The specification mechanism employed must allow the designer to specify completely the structural (hardware) requirements of the system, together with the behaviour that the system is to exhibit (the software).

2. Software Generation

The behavioural specification must be analysed and transformed into an equivalent representation, expressed in

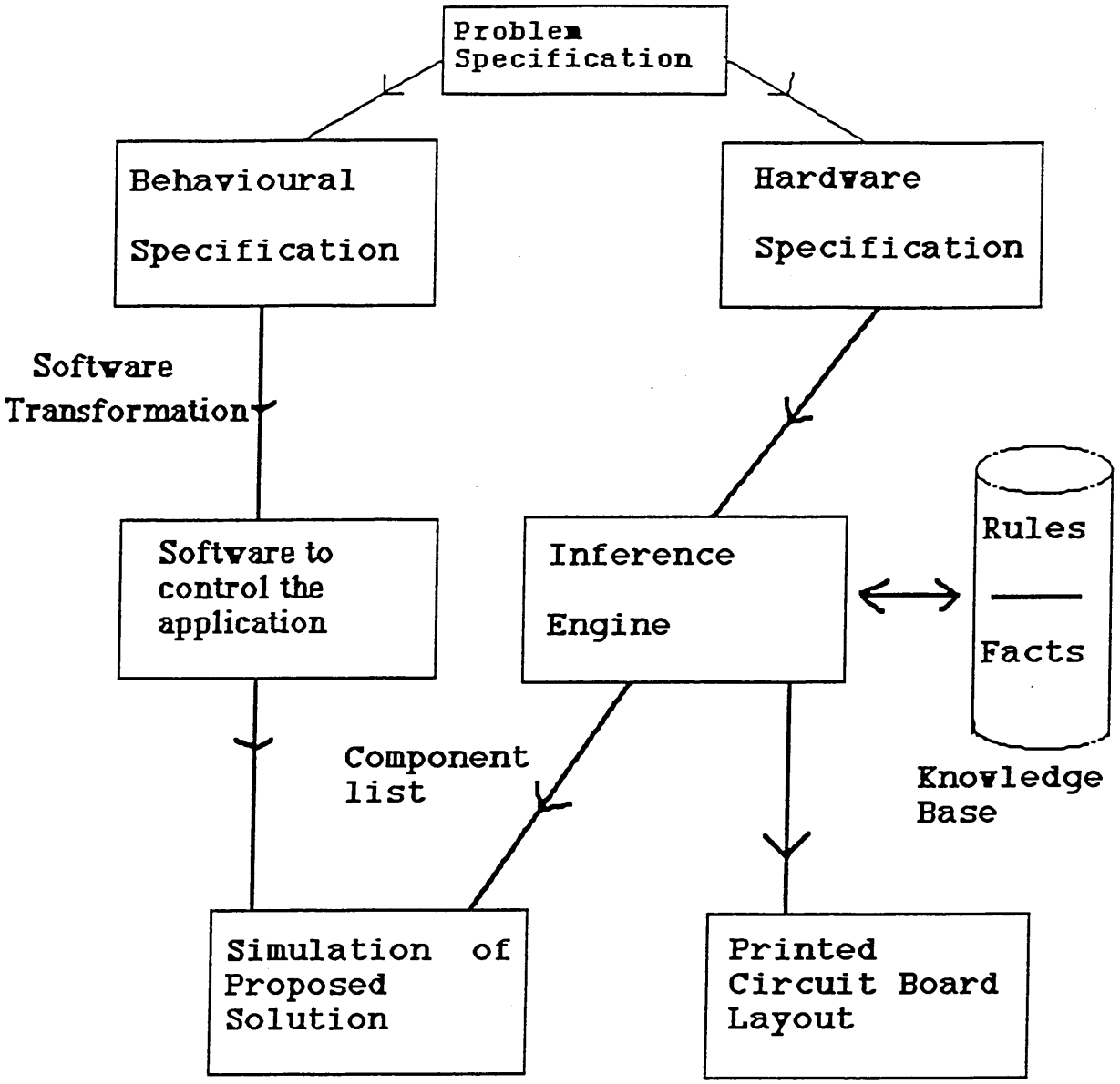


Figure 1.1

Expert System Architecture

the assembly language of the microprocessor chosen to implement the system.

3. Knowledge Base and Inference Engine

The knowledge base will contain factual data on LSI components, a collection of rules which govern their application, and a generalised functional simulation of each component. The inference engine will attempt to find a solution to a given problem by applying rules in the knowledge base. The result of this process will be a configurable MCS.

4. Simulation

Individual component simulations will be stored in the knowledge base, and be combined when required to form simulations of the proposed MCSs. A language suitable for component description and simulation must be used to construct simulations, together with hardware which facilitates efficient simulation execution. If sufficiently fast ('almost' real-time) simulations can be generated, it may be possible to interface the simulation with the physical system to be controlled.

5. PCB layout

The PCB layout for the final solution can be produced from the detailed data stored in the knowledge base.

The nature and operation of each of the above system components is diverse: each requires a fundamentally different set of techniques for its design and implementation. For example, the hardware selection element of the system should greatly benefit from the use of expert system techniques[1.13]. The explicit definition in the knowledge base of the heuristics used to design MCSs should make the design system much simpler to implement, maintain and enhance. However, the component and system simulations

would be best constructed in a language which contains features appropriate for describing hardware.

The adoption of an integrated, rule-based approach to the design process has several potential advantages. Firstly, the high-level behavioural specification could be totally processor-independent, making it possible to automatically generate machine code for any given processor in the knowledge base. Second, the expert system could select appropriate components from its knowledge base, and perform the task of component interconnection. Given an extensive and regularly updated knowledge base, the expert system should perform the selection task at least as well as a human designer. Moreover, it should not be error-prone when configuring the system components. Third, a potential solution could be evaluated early in the development cycle by comparing the behaviour of its simulation against the requirements specification. This allows the designer to be confident of the correctness of the design before any implementation activity takes place.

1.5 Objectives

The work reported in this thesis describes a set of tools and techniques which can perform the synthesis and simulation of low-level machine code for an MCS from a design-level behavioural specification. No attempt is made to design or construct the knowledge base or inference engine of the expert system, as this work forms the focus of another research project. Only the interface and information interchange between the inference engine and the behavioural specification system is considered in some detail.

Chapter 2 surveys some of the most important work in the field of CAD and expert systems for designing computer systems, together with the languages used to specify and describe software and hardware requirements. Chapter 3 discusses the requirements of a behavioural specification

language for MCSs, and presents the features of a language designed to meet these requirements. Chapter 4 explains how behavioural specifications can be automatically transformed into machine code implementations for particular microprocessors. The structure and features of a processor-independent assembly language which facilitates this transformation is also described. In Chapter 5, the design and construction of a generalised simulation facility for MCSs is presented, and the performance of some example simulations is discussed. Through the use of several examples, Chapter 6 attempts to evaluate the specification and simulation techniques described in the earlier sections. In this manner, a classification of applications where such techniques are primarily suitable is devised. Finally, Chapter 7 presents areas which may be considered for further work, and provides a detailed outline design specification for the operation of the complete expert system.

1.6 Summary

Microprocessor systems are used widely to implement process control functions. The complexity of such control systems varies greatly. However, current hardware and software development tools do not facilitate fast prototyping and evaluation of possible solutions. Consequently any basic design errors carried through to prototype implementations are costly and time-consuming to correct. In an attempt to solve these problems, the essential elements of an integrated design environment for minimum configuration systems have been presented. The design environment would allow applications to be described at a high-level, relieving the designer of much detail. Simulations of possible solutions would be generated from system specifications. Thus potential solutions could be evaluated before any actual implementation takes place. This should greatly lower the occurrence of serious and expensive design faults in MCS implementations.

References

1.1 House,C.H.:'Perspectives on Dedicated Control'. IEEE Computer, Dec 1980, vol 7, no 12, pp 35-48

1.2 Arnold,J.T. :'Simplified Digital Automation with Microprocessors',ch.1, pp 2-10, Academic Press, New York 1979

1.3 Johnson,C.D.:'Microprocessor-based Process Control', ch.1, pp 1-32, Prentice-Hall, INC., New Jersey 1984

1.4 Simons,G.L. : 'Uses of Microprocessors',ch.4, N.C.C. Publications, 1980

1.5 Hudson,C. : 'Techniques for developing and testing microprocessor systems', Software and Microsystems, August 1985, Vol 18, No. 4, pp 81-92

1.6 Streitmatter,G.A. and Fiore,V. :'Microprocessors - Theory and Application',ch.12, pp 219-253, Reston Publishing Company, Virginia, 1979

1.7 'System Specification, Design and Implementation Tools', Dept. of Computation, University of Manchester Institute of Science and Technology

1.8 Zaks,R. : 'Microprocessors',ch.9, pp 363-387, SYBEX Inc, USA, 1977

1.9 Saxena,S. and Field,J.A.:'Portable Real-Time Software for 8-bit Microprocessors', Software - Practice and Experience, 15, (3), 277-303 (1985)

1.10 Welsh,P.H.:'Managing Hard Real-Time Demands on Transputers',Procs 7th Occam User Group Conference, Grenoble, 14-16 Sept 1987

1.11 Jensen,K. and Wirth,N.:'Pascal User Manual and Report', Springer-Verlag, New York-Berlin, 1975

1.12 Buhr,R:'System Design with ADA', Prentice-Hall, 1984

1.13 Black,W.J.:'Intelligent Knowledge Based Systems - An Introduction', Van Nostrand Reinhold(UK), 1986

2 Related Work

2.1 Computer-Aided Design of Digital Systems

2.1.1 Introduction

Many integrated Computer-Aided Design (CAD) systems have been developed to assist with the design of digital systems. Existing CAD applications range from the development of mask descriptions for Very-Large Scale Integration (VLSI) components, to the configuration of the components necessary to fully implement a minicomputer system[2.1]. However, irrespective of the precise nature of the application, all such design systems share a similar goal: the synthesis of a low-level, manufacturable solution from a high-level statement of a problem. The synthesis process usually involves the analysis and manipulation of an abstract behavioural or structural specification through several progressively more detailed levels of design description, until the necessary level of complexity is attained[2.2]. This process, by definition, implies a large search space: the key problem for a design system is to choose amongst the many possible designs, selecting the one which best satisfies the specification. For this reason, most recent CAD systems have incorporated knowledge-based techniques into their operation, in an attempt to reduce the complexity of the design process to a manageable scale[2.3]. It is claimed that the use of modern CAD systems may increase the rate of development of future digital systems by as much as twenty times[2.4].

The following sections review the aims and operation of a number of CAD systems, which aid in the design of digital systems at a wide range of levels of application.

2.1.2 The RT-CAD System

The RT-CAD system developed at Carnegie-Mellon University (CMU) represents an attempt to accelerate the design

process of integrated circuits (ICs)[2.5]. The major aim of the system is to minimise the effect of advancing implementation technologies for ICs by constructing a system which provides a technology-relative design process. It builds upon earlier design automation systems which concentrated on the synthesis of various levels of design description into purely gate-level implementations[2.6]. By providing libraries of different implementation modules, the system can generate alternative solutions for a given problem description. Thus, the inclusion of alternative module sets allows the system to perform designs independent of any particular implementation technology. Moreover, this approach should encourage the incorporation of new technology into the design process.

The system operates by accepting a behavioural description of the IC to be designed expressed in the ISP[2.7] hardware description language(HDL). This is then compiled and the object code produced is loaded into the system data base, where it can be manipulated by other design tools in the system. The structure of the system is shown in Figure 2.1.

The most important of these tools is EXPL[2.5]. It takes as input the object code from the ISP compiler, together with a set of user supplied cost and performance constraints. From the compiler output, EXPL generates a graph which represents the behaviour of the required system. It then attempts to manipulate the original graph to establish alternative design possibilities. Essentially EXPL tries to determine which operations can be performed concurrently, and which must remain in a definite sequence. Each alternative design is passed on to the module set evaluators. These complete and evaluate the design for each alternative in terms of its hardware module set. The evaluation of each design is passed back to EXPL, which decides, by applying a set of heuristics, which solutions

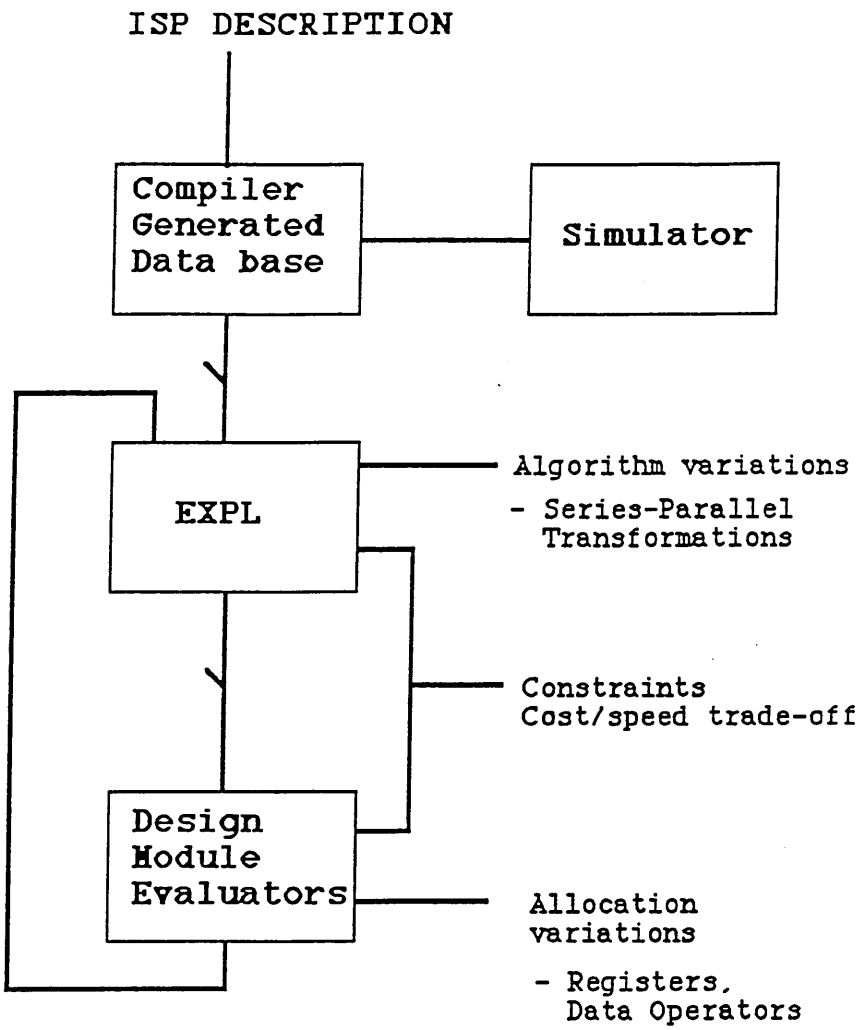


Figure 2.1 The design process in the RT-CAD system

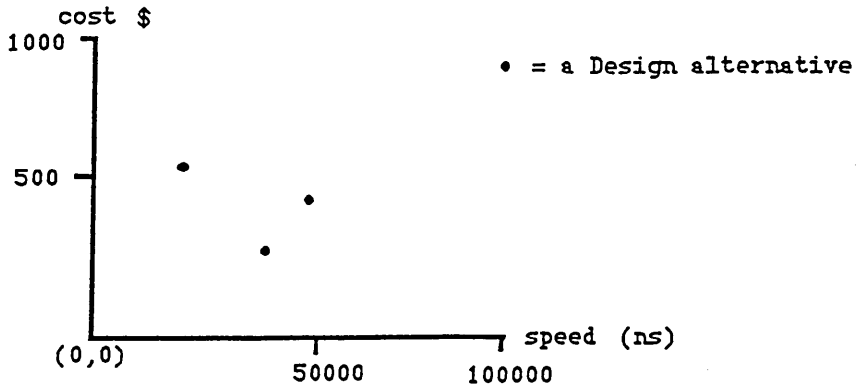


Figure 2.2 An Example Design Space

to discard as impractical, and which to keep in order to generate further solutions by yet another application of graph transformations. In this manner, the process is continued until EXPL finds the optimal implementation within the given constraints.

Thus, EXPL searches through the set of possible designs for a circuit. With the assistance of the technology-dependent module evaluators, it attempts to find the best solution which satisfies the problem specification. The set of possible designs of a circuit is known as its design space. The design space, explored automatically by EXPL, can be depicted by a two dimensional graph, as in Figure 2.2. Each alternative design is represented on the graph by its cost and time co-ordinates. These are computed by the technology-dependent module evaluators.

The exploration of the design space by EXPL is driven by the goals (cost and speed) set by the designer. Ideally the goal is to find an alternative design whose position in the design space is as close as possible to the origin (0 cost, 0 time). Realistically though, the least expensive solutions are not the fastest, and vice versa. Therefore the aim of EXPL is to find a solution which has an acceptable level of performance for minimum cost.

The RT-CAD system also includes design tools which perform the verification and simulation of ISP descriptions. It is possible to develop syntactically correct ISP descriptions which make no sense semantically. The system guards against this happening by checking the correctness of the semantics of each design specification. Simulation of ISP designs is performed by stepping through the flowchart which is produced by the ISP compiler. An interactive command language allows the user to set and display the contents of registers and define arbitrary breakpoints.

2.1.3 ULYSSES

ULYSSES is a sophisticated VLSI CAD environment developed at CMU[2.8]. It aims to automate the design process, and consequently lower the design cost, of complex VLSI components. Other CAD systems[2.9] require the designer to manually execute individual design tools, and to manage the various files used as input to CAD tools, or created as output by tools. In ULYSSES, all the required CAD tools are fully integrated and controlled by the system itself. Thus the designer interacts with the system at a higher level. ULYSSES completes a design by automatically invoking the required tools, and managing the various intermediate files produced by the individual tools. This allows the designer to concentrate on the high-level design, without needing to become an expert user of a complex CAD system.

In order to effectively address the problems of CAD tool integration, ULYSSES employs Artificial Intelligence(AI) techniques. It functions as an interactive expert system, interpreting design descriptions and initiating design tool executions. Specifically, ULYSSES has the ability to:-

1. manipulate a variety of hardware description languages
2. describe and automatically execute a diverse set of design tasks
3. allow the designer to arbitrarily interrupt, restart, or redirect a sequence of design tasks
4. represent important design decisions
5. explain its sequence of design activities and provide reasons for specific design decisions
6. maintain the design history for each significant design point
7. evaluate competing design points during design elaboration
8. easily facilitate the integration of new CAD tools

ULYSSES incorporates a sophisticated, knowledge-based scheduler to control the execution of the individual CAD tools. The knowledge associated with the scheduler enables ULYSSES to achieve much of the flexibility that it requires to successfully complete a design. The scheduler can be controlled by the designer, who may wish to interrupt a sequence of design tasks which are performing erroneously. In such cases the designer may choose which tools to execute to continue with a particular design.

The need to control the vast volume of data generated as a circuit design is synthesised presents a significant problem in ULYSSES. Each stage in the design process produces an intermediate description in some appropriate notation. Further, alternative implementations may be produced at each design stage. These arise when different trade-offs of design parameters, such as cost, speed and power consumption, are considered. Thus these alternatives represent competing design points in the design space of a circuit.

In ULYSSES, a frame-based[2.10] tree structure is employed to characterise the design space[2.11]. Each node in the tree represents a particular state in the evolution of a design. As CAD tools are used to add details to a design, child nodes are created which correspond to the new situation. As several alternatives may be produced at each stage of the synthesis process, a node may have several children emanating from it, each of which corresponds to a different design decision. Relations link design points to their parents, so that a child node can inherit design information from its parent. This means that only new or altered data needs to be stored at a child design point. The use of a tree structure has the advantage that it is possible to backtrack to a parent design point when further progress from the current design cannot be made. Also, a terminal node in this design space will represent a complete solution to a VLSI design problem.

ULYSSES adopts the blackboard model of the Hearsay 2 system[2.12] as a model for its architecture. In ULYSSES, each CAD tool is viewed as a knowledge source. All knowledge sources are considered to be self-activating, asynchronous parallel processes. Knowledge sources communicate via a blackboard, which is effectively a global data base. The blackboard supports the many levels of representation necessary in a circuit design, and in particular, it holds the whole design space. Each knowledge source has a set of pre-conditions associated with it, which must be satisfied by data in the design space before it can execute. Consequently, the knowledge sources periodically monitor the evolution of the design space. When the pre-conditions of a knowledge source are met, that knowledge source is activated. A special knowledge source, the Rating Policy Module, is activated by the scheduler whenever the design space is altered. It provides a uniform basis for comparing alternative designs and for pruning unpromising designs from the design space.

Another important aspect of ULYSSES is the ability to describe a wide variety of design tasks and methodologies. This mechanism is provided by the Scripts language, which, while retaining all the facilities normally associated with a production system[2.21], also allows some tasks to be specified procedurally[2.13]. Essentially a script is a set of instructions which realise a given design task. Thus a script can be said to provide a method of composing a very large production rule out of a sequence of individual knowledge source executions.

2.1.4 Synapse

Synapse[2.14] is an experimental expert system intended to support VLSI design. The goal of the system is to enable a very high level specification of a problem, including performance constraints, to be mapped into custom VLSI circuits. In Synapse, design descriptions at all levels of abstraction are represented as algebraic expressions. The

design of a VLSI circuit in Synapse involves the repeated transformation of algebraic expressions until an expression is reached that represents a viable solution. This approach to VLSI design is novel, representing a significant departure from most existing paradigms.

The input to Synapse is a behavioural specification of the desired circuit: this forms the initial expression. Any number of transformations may be applied to this expression, and all result in either legitimate designs or intermediate design states. In general, transformations either change the dimension or the level of abstraction of an expression (usually increasing the amount of detail), or they improve the system's performance attributes. All transformations are formally proved to leave the functional behaviour of the system unaltered. When an expression has the desired performance characteristics, and is in the proper dimension such as a mask description, it may be viewed as a possible solution.

For any given specification, several possible implementations may be generated if all the applicable transformations are applied. These represent different design points in the design space for the circuit. Synapse therefore utilises strategies to focus the search of the design space into the most promising areas. Synapse also allows the designer to perform transformations on expressions. In this case, the designer is primarily responsible for proving that the transformations are correct, and for ensuring consistency between representations.

Synapse uses KEE[2.15], an expert system development tool, to implement the inference engine. KEE provides facilities to maintain consistency between alternative design representations, and both forward and backward chaining mechanisms. These are used to apply the transformation rules to the specification, and to implement machine learning capabilities.

Synapse is part of a long-term research project to explore expert system issues in CAD. Its novel approach to the domain of VLSI design has shown that AI techniques can be used successfully to automate the design process. Further work on Synapse is concentrating on enriching the system's knowledge base and increasing its learning capabilities.

2.1.5 MAPLE

MAPLE[2.16] is an expert system developed at the University of Reading. Its purpose is to automate the design of hardware for dedicated microprocessor applications. It attempts to satisfy the hardware requirements of a system by choosing the most appropriate combination of pre-designed boards from its knowledge base. MAPLE does not attempt to design new boards for applications. However, MAPLE's designers intend to extend its capabilities to enable the design of boards from individual components, and to provide limited assistance with software design.

MAPLE is implemented as an interactive system, and each 'consultation' has three distinct stages: INTERVIEW, DESIGN and REPORT. The INTERVIEW stage enables MAPLE to acquire the design goals and constraints of the application. The user is requested to specify the hardware requirements and constraints such as power consumption and cost. During the DESIGN stage, MAPLE uses its knowledge to design a system to meet the applications requirements. Finally, during the REPORT stage, MAPLE issues a complete set of documentation for the design, and a list of any assumptions that were taken.

MAPLE is implemented in a procedural language (PASCAL). This means that the rules MAPLE applies during the design phase are embedded in the code of the search algorithm. Therefore, in order to add or modify rules, the algorithm itself must be altered. This is a considerably more difficult task than that encountered in other systems such as XCON(see below), where adding new heuristics merely

involves providing more data. Altering the design algorithm requires in depth knowledge of the program, and could involve extensive testing to ensure that any changes have not introduced errors.

Although MAPLE's capabilities warrant classification as an expert system, its internal architecture does depart from the expert system paradigm of a separate knowledge base and inference engine. So, in many respects, MAPLE is similar in construction to many of the algorithmic-based CAD systems which exist[2.17]. In justification of their implementation strategy, MAPLE's creators claim that the problem of microprocessor hardware design can be solved by a well-defined, compact strategy, which is unlikely to change dramatically[2.18]. While this may be correct[2.19], a procedural implementation will almost certainly complicate the task of extending MAPLE's abilities. The added complexity of designing software as well as hardware for applications, and the creation of new boards from individual components will require the adoption of a much more flexible approach. It is doubtful that a well-defined algorithm could be confidently developed to perform such a collection of integrated design tasks. Consequently, it is likely that the full utilisation of knowledge-based techniques would allow a better solution to be reached.

2.1.6 XCON

XCON (originally known as R1) was developed by a research team at Carnegie-Mellon University(CMU)[2.20]. Its domain of expertise is the configuration of VAX 11-780 minicomputer systems, and it has been used successfully by Digital Equipment Corporation (DEC) since 1982. VAX systems are not offered to customers in standard configurations. Rather a customer may order a specific configuration of input, output, storage, processor and software. XCON's task is to determine a correct configuration for an order. This involves recognising any interdependences between components, and adding extra components when necessary. The

output produced may be used directly by technicians to assemble the system.

XCON takes a set of components as input and produces diagrams showing the required physical relationships between the components. Although XCON cannot perform the task of selecting components to satisfy a functional specification, it is capable of determining which components require others in order to be configured. If the component set given to XCON is incomplete, it adds whichever subsidiary components are required (e.g. cables, cabinets).

XCON is implemented in OPS5[2.21], a general-purpose rule-based language developed at CMU. OPS5 provides a rule memory, a global working memory and an interpreter that tests the rules to determine which are satisfied by the data held in working memory. Rules in OPS5 are expressed as IF-THEN statements. These consist of a set of conditions which can be matched against the descriptions in working memory, and a set of actions which modify the data in working memory. On each cycle, the interpreter selects a rule which is satisfied and applies its actions to the data in working memory. Actions always add to or modify working memory. Thus in XCON, the rules have conditions that recognise situations in which an extension is required to an incomplete configuration; the actions then effect that extension.

In XCON, OPS5's two memories are augmented by a third, This memory, the data base, contains descriptions of each of the components supported for VAX systems. Each data base entry comprises the name of a component and a number of attribute/value pairs which describe the important properties of the component for the configuration task. As XCON configures an order, it retrieves the relevant component descriptions from the data base and places them in working memory.

Production, or rule memory contains all of XCON's knowledge of how to perform the configuration task. These rules can be viewed as state transition operators. The conditional part of each rule describes the properties that a state must possess in order for the rule to be applied. The action part of a rule specifies which features of the current state must be modified or augmented in order to reach a new state on the solution path. Each rule is a more or less autonomous piece of knowledge that waits for a state it recognises to be generated. When this happens, it effects a state transition. The new state generated should subsequently be recognised by one or more other rules, which in turn effect a state transition. This process continues until the system is configured.

XCON differs from other domain-specific systems primarily in its use of Match[2.22] as opposed to Generate-and-test as its central problem-solving strategy[2.20]. Rather than exploring several hypotheses until an acceptable one is found, it exploits its knowledge of its domain to generate a single acceptable solution. With Match, the conditions associated with each state are sufficient to guarantee that if a state transition is permissible, then the new state will be on the solution path. Thus with Match, false paths are never followed, and so backtracking is never required. In the configuration task, the knowledge available at each step is normally sufficient to distinguish between acceptable and unacceptable paths. There is only one subtask in XCON for which several alternatives must be generated before the optimum solution is found.

The significance of XCON is mainly due to the fact that it was the first knowledge-based CAD system to be used in the commercial world. It proved that expert systems could be used to automate design tasks, and provided a development methodology that should be applicable to other systems[2.23]. XCON also demonstrates that OPS5 is an appropriate tool for the development of domain-specific systems, and that the use of production rules can simplify

the task of refining and extending the knowledge base[2.20].

2.1.7 Others

The vast range of expert system developments in CAD makes it impractical to cover each system in detail. However there are several important examples which are worthy of note. EL and SYN are two expert systems developed at the Massachusetts Institute of Technology. They are intended to help a designer analyse analogue circuits[2.24]. Palladio is a prototype expert system under development at Stanford University. The major goal of Palladio is to enable designers to construct VLSI circuits, and at the same time explicitly express the design heuristics that were used. Designers may create personal knowledge bases which can be incorporated into circuit designs[2.25]. This capability should encourage experimentation with VLSI design methodologies.

2.2 Software Specification Techniques

2.2.1 Introduction

The need for languages which provide precise specifications at all stages of the system development life-cycle is widely recognised[2.26]. A considerable amount of research has been carried out in this area, but there are still no generally accepted tools or methodologies. A possible reason for this lies in the fact that each stage of system development requires a specification at an appropriate level of detail, expressed in a suitable notation. Further, different types of applications have unique characteristics, which are best described by specialised language features[2.27]. Thus, most current software specification languages are dedicated to one particular aspect of system development.

It is possible to identify three categories, which provide a meaningful framework for the investigation of the many existing languages[2.28]. These are:-

a) phase of applicability
b) area of application
c) language model

The following sections elaborate on these categories.

2.2.2 Phase of Applicability

Specification languages can be used to describe target software systems from the first phase of requirements definition up to the physical specification of program design. An initial requirements specification may be abstract and vague. But as successively more information is added, the level of detail increases, and this continues until the development of the actual computer programs is complete. Within this development spectrum, three broad areas of specification languages can be defined[2.28].

2.2.2.1 Requirements Specification Languages

Requirements specification languages(RSLs) are used to describe the initial user requirements for a computerised system. RSLs describe the basic functions of the system, together with constraints such as structure and performance. Thus RSLs provide a problem-oriented description of systems, stating what needs to be done, not how. An early and well-known example of an RSL is PSL - Problem Statement Language[2.29]. More recent examples include formal specification languages such as Z[2.30] and OBJ[2.31].

2.2.2.2 Design Specification Languages

At the next level of refinement, the requirements definition is taken and the overall design of the system is

carried out. The major functions of the system and their relationships are identified. Thus design specification languages(DSL) can be said to specify how a system can achieve its aims. It is a solution-oriented description of a software system. A typical representative of DSLs is DDN[2.32].

2.2.2.3 Program Design Languages

Once the overall design of the system has been finalised, algorithms and data structures have to be developed and precise interfaces between modules established. Program design languages(PDL) therefore provide facilities which are specifically related to data structure specification and module interaction, and are implementation-oriented. GYPSY[2.33] is an example of a PDL.

2.2.3 Area of Application

Software systems can be classified as being sequential, concurrent or real-time[2.34]. Sequential software systems can be specified as sequences of actions, always performed in the same order, with no two actions performed together. Concurrent systems consist of several activities occurring in parallel and communicating in some controlled manner. In real-time systems, activities may occur sequentially or concurrently. However real-time software must be capable of responding to external stimuli within a specified time period, and further, the order in which stimuli arrive may not be predictable.

Real-time and concurrent systems tend to be considerably more complex than comparable sequential systems. Consequently a specification language intended for sequential data processing systems would be totally inappropriate for specifying a complex real-time multi-variable control system. So the type of system at which a specification language is aimed heavily influences the characteristics and features included in the language.

Thus, the description of the relationships between external stimuli and responses is vital in real-time and concurrent systems, whereas file formats, data integrity and validation criteria may be the most important aspects of a sequential data processing system specification.

2.2.4 Language Model

Three separate concepts can be identified which form the basis of existing specification languages[2.27]. These are described below:-

2.2.4.1 State-based Languages

State-based languages are based upon the model of finite state machines[2.35]. They provide a method of specifying the set of possible states of a system, and the state transition functions which enable the system to move from one state to another. The major advantage of state-based languages is their use of abstraction. It is possible to abstract the state space so that it reveals details only of particular interest. This can be done at each stage of the development process, allowing specifications to be hierarchically structured. Examples of state-based languages are GYPSY[2.33] and DREAM[2.36].

2.2.4.2 Event-based Languages

In event-based languages, specifications are stated in terms of actions which must be performed when a certain event occurs. Events may be specified to occur in a certain sequence, or may occur non-deterministically. This approach was first used in specifying requirements in the telephone industry[2.37], and from subsequent developments languages such as RLP[2.38] emerged. Advantages claimed for RLP and the event model are facilitation of test plan generation and enhanced readability through isolation of system features[2.38].

2.2.4.3 Relational Languages

Relational languages allow systems to be described in terms of the required relationships between important properties of the desired system. These properties usually include the inputs to be processed, the outputs to be produced, the functions to be performed and the events that may occur. Relationships may then be created between inputs, functions and outputs. Two specification tools which have been built around the relational model are RSL[2.39] and PSL[2.29].

In general terms, it appears that state-based languages find their most natural application in design and implementation specifications, whereas relational languages are best suited to requirements specification. However, it seems that event-based languages are not so easy to classify, and may be useful at several levels of the development process[2.27].

2.2.5 Desirable Features of Software Specification Languages

Irrespective of their intended application, it is possible to define a set of common goals which a specification language should meet[2.40].

a) Understandability
b) Analysability
c) Maintainability

These criteria have many implications on the facilities which a specification language should provide. A number of the more desirable features are summarised below:-

2.2.5.1 Understandability

1) Dimension of language -- Specification languages may be characterised as one-dimensional (character string languages), two-dimensional(graphical languages) or

hybrid(written/graphical languages)[2.41]. Although graphical languages seem to be superior in terms of overall clarity, problems exist because of their limited 'processability'. However recent advances such as compilable graphics have to some extent alleviated these problems[2.42].

2) Level of detail -- It is important that a specification only contains information that is relevant to the current phase of development. A language should facilitate the suppression of irrelevant details, enabling the overall structure of the specification to be easily visible. In this way, specifications can be developed incrementally from a vague statement of requirements to a complete physical design. Good examples of languages which employ this approach are PDL[2.43] and SPECLE[2.44].

A commonly used abstraction mechanism which allows the hiding of unimportant data is modularisation. Specifications can be decomposed into small meaningful units which are defined at a lower level of detail. A specific modularisation technique known as data hiding has been included in many implementations[2.45]

3) Formality -- Formal specifications of systems eliminate all sources of imprecision by using precise syntactic and semantic definitions. Unfortunately specifications written in formal notations are difficult to comprehend. In contrast, informal specifications incorporate abstraction techniques to focus on important issues and increase understandability. This fact makes informal notation difficult to analyse and verify. Obviously then, a proper balance must be sought between formal and informal notations in order to maximise the benefits of each approach[2.46, 2.47].

2.2.5.2 Analysability

1) Static validity -- It should be possible to analyse a specification and check for such properties as conflict, ambiguity and redundancy. This is analogous to syntax checking in conventional languages.

2) Traceability -- This refers to the capability of verifying a specification against its successor or predecessor. As already stated more than one level of specification will generally be needed during the development of a system. It is therefore important that each level of specification is functionally equivalent, and that no errors or inconsistencies are introduced.

3) Dynamic validity -- This objective is concerned with the evaluation of the behaviour of specifications before implementation proceeds. Specifications should be executable, forming a simulation of the required system. In this manner, a specification may be modified until its behaviour is deemed satisfactory. This facility is of particular importance in the realm of real-time systems, where performance and efficiency are of particular interest, and is demonstrated by the SREM project[2.39].

2.2.5.3 Maintainability

1) Modification -- Languages should allow specifications to be easily extended or adapted. Useful facilities which simplify updates to software include data hiding and modularisation[2.45]

2) Document generation -- Automatic documentation generation is a very useful feature of specification languages. When a system is modified, the documentation must be altered accordingly to reflect the changes. This is a time-consuming and tedious manual process. However if the documentation for a system has been generated automatically from the specification, it should also be possible to

produce updated documentation when the specification is altered.

2.2.6 Examples

2.2.6.1 Vienna Development Method (VDM)

VDM is a formal specification language[2.48]. A VDM specification defines a system in an implementation-independent manner. This is achieved by using mathematical models to describe objects and structures, as well as the meaningful operations which may be performed upon them. In VDM, the mathematical models of structures are abstract data types, described using the ideas of sets, functions and relations. Operations allowed upon structures are also specified using such mathematical notations. A VDM specification has three distinct components: a state definition, the definition of invariants, and the definition of operations.

The state definition describes the structures required in terms of basic types (real, integer, Boolean), which can be combined using special constructors to give the mathematical notions of sets, sequences and functions. Composite types can also be formed from these basic types. The individual elements of a composite type do not need to be of the same basic type; composite types may in fact be defined recursively.

Invariants are constraints which must be preserved by operations. Invariants thus represent properties of the system which must always hold true. In VDM, invariants should be proved for each operation.

Operations, similar to functions, are defined by predicates. There are two predicates defining each function, a pre-condition and a post-condition. The pre-conditions define the circumstances in which the

operation produces valid results. The post-conditions defines the effect of performing the operation.

VDM specifications are unambiguous and free of design and implementation directives. They are also not directly executable. The execution of formal specifications is a desirable feature of a specification language. Specification execution provides a prototype implementation of the desired system. This can be used to remove syntactic errors, and increase confidence in the correctness of the specification. In order to execute VDM specifications, the structure definitions must be reified and decomposed into code for a programming language. Reification consists of moving from abstract data types to the sorts of data structures available in a target programming language. For example, a VDM sequence may be represented as a linked list or an array in most common programming languages. Further, the operations specified for the original abstract data types must be respecified to operate on the reified, more realistic data types, and the abstract operations must be decomposed into statements in the target language. Still, this is not sufficient to guarantee that the implementation inherits all the desired properties of the specification. Consequently, proof obligations must be provided, to show that the implementation is indeed correct with respect to its specification.

Clearly, the process of reification, decomposition and proof of correctness is extremely complex, and may not be generally feasible or applicable[2.49]. Certainly a comprehensive set of development tools such as syntax-checkers and theorem-provers are required to support such a transformation. This however, does not detract from the advantages which can currently be derived from using VDM, namely the unambiguous, concise, abstract description of a system.

2.2.6.2 Espreso

Espreso is a language for specifying the requirements of complex, real-time process-control systems[2.50]. A fundamental assumption of the language design is that system requirements have an inherent hierarchical structure, which should be detected and encoded in a specification. Espreso thus allows specifications to be built hierarchically by providing extensive module packaging and data hiding facilities.

Logical and arithmetical expressions are not permitted in Espreso specifications. Rather, the language constructs are limited so that only the high-level aspects of the problem may be addressed. This has the effect of reducing the number of concepts in the language, and of preventing the user from dealing with low-level details and algorithms too early. The language does however facilitate the expression of parallel operations at an abstract level, and includes well-understood mechanisms for controlling message passing and access to shared resources. It also allows informal, text descriptions to be entered. Although these cannot be analysed, they do provide a convenient method for the user to express ideas which are not fully formulated.

The Espreso language is formally defined by an extended attribute grammar, which describes the complete language syntax[2.51]. The provision of a formal language definition helps guard against unforeseen inconsistencies and ambiguities in the language, and makes specifications more amenable to checking and verification. Each Espreso construct is expressed in a Pascal-type manner. This has the consequence that Espreso specifications resemble skeleton Pascal programs. By refining the specification, manually or partially automatically, a prototype implementation of the specification can be produced.

The Espreso system is fairly typical of many RSLs and DSLs. It is rigorously defined, and contains constructs and

features which are appropriate to its intended level of detail and application area. Although Espreso specifications are not executable, many aspects of static validity can be checked for, and the specification forms a reliable high-level source for further refinement of detail.

2.3 Hardware Description Languages

2.3.1 Introduction

A hardware description language(HDL) is a notation which may be used to depict particular aspects of digital systems[2.52]. The complexity of digital systems design has led to the development of many HDLs. The aim of HDLs is to conquer complexity by the systematic use of abstraction at each level of the design process[2.52]. There are many levels in the process of hardware design, ranging from circuit and logic design to behaviour and system specification. Each level has its own purpose, and each level needs to be described. Therefore different HDLs reflect different levels of abstraction of computer hardware[2.53]. In general terms though, HDLs may be classified as providing either physical information on the structure and interconnection of components, or behavioural information on the function of circuits[2.54]. Many languages attempt to provide both structural and behavioural descriptions. However these two aspects of hardware are best described by different notations. Consequently, languages which incorporate facilities for both types of descriptions usually include two distinct types of notation[2.55].

2.3.2 Structural HDLs

Many situations arise during hardware design where there is a need to describe the structure of a circuit without giving any behavioural information[2.54]. Structural descriptions of circuits may be input into CAD systems,

producing a geometrical layout of the circuit[2.4]. Current structural HDLs allow systems to be described at many levels. At each level the description of the circuit must show the basic components and their interconnections[2.52].

As is common with most HDLs, purely structural HDLs have borrowed many features from high level programming languages. Components may be described as functions, with the function's arguments representing the inputs to the component, and the function's result representing the components output. When a particular component is required in a description, it may simply be 'called' in the appropriate place. Conventional loops are used to precisely describe regular structures, and some languages allow types to be declared to aid in the verification of circuits[2.54]. Examples of HDLs which exhibit these features (and many others) are MODEL[2.56] and ELLA[2.57].

2.3.3 Behavioural HDLs

The behaviour of digital circuits may be described at many different levels, from individual gates to whole components or systems. Behavioural HDLs currently serve two main purposes[2.54]. They allow the desired function of a circuit to be stated by the designer at the inception of the project, and they provide a means of verifying a component's performance after it has been fabricated. Checking that the manufactured component does implement its intended behaviour is a significant problem. Existing tools perform this task by comparing the results of a simulation with the results produced by the actual component. This is generally a complex and error-prone task, resulting in components which are only as reliable as the data used to test them[2.58]. In attempts to alleviate the problem of establishing design correctness, formal notations are being proposed as languages to model behaviour. Verification could then be carried out by rigorous mathematical proofs[2.54].

Existing behavioural description languages contain many common features. Most provide some method of representing global time and propagation delays[2.55], and this is often a crucial factor of a design. Digital systems also exhibit highly parallel behaviour, and facilities to describe parallel operations are included in many languages[2.59]. Another important characteristic of digital systems is the requirement to examine the state of an input or register, and perform an action corresponding to that state. A suitable construct for describing this situation is the generalised CASE statement, as found in many programming languages. CASE statements are found in ELLA and VHDL, and ISPS contains an OPERATE statement which is semantically identical.

2.3.4 Hardware Synthesis

Synthesis may be defined as the translation of a higher level of description of a design object into a lower one[2.60]. A complete synthesis system should generate layout masks from a high level behavioural description of a system[2.2], with all intermediate levels of structural and behavioural descriptions constructed automatically. Physical synthesis from a structural description is a reasonably well-understood process, with many design tools available for gate-arrays[2.56] and standard cell arrays[2.61]. However structural synthesis from a behavioural description is a much more complex task, due to the difficulty of maintaining the correct functionality of the hardware structure[2.62].

Silicon compilers have been proposed to carry out the entire synthesis process. Still, due to the complexity of the task, interaction with designers is required at many of the intermediate stages[2.2]. Most silicon compilers therefore accept a relatively low level behavioural description and translate it into a fixed target architecture[2.62].

More ambitious attempts at silicon compilation from a high level description are now the subject of much research activity. One trend is to try to translate true behavioural descriptions of digital systems using high level programming languages such as ADA[2.63], occam[2.64], Modula2[2.65] and Concurrent Prolog[2.58]. The rationale behind this approach lies in the fact that sophisticated programming environments have been developed for many general purpose languages. Therefore, where applicable, existing languages should be used as a basis for development[2.65]. Languages that contain constructs to express parallelism and communication between parallel components are suitable for the task of high level behavioural descriptions[2.66]. An added advantage of using a programming language for this purpose is that the description of a device can be compiled into an efficient simulation.

2.3.5 Examples

2.3.5.1 Instruction Set Processor Specification (ISPS)

ISPS, an improved version of the earlier ISP language, is essentially a behavioural HDL[2.59]. It is intended that ISPS descriptions should be amenable to a wide range of design applications, rather than supporting a wide range of design levels. The language itself is designed to be both flexible and simple, and it incorporates many constructs that are usually found in high-level programming languages.

An ISPS description of a hardware component comprises both an interface and a behavioural description. The interface gives the external structure of the component in terms of the number and type of registers which are used to transmit and receive data. The component's behaviour is described by procedures which specify the sequence of the control and data operations. Although ISPS is mainly intended as a behavioural HDL, it does allow some structural information to be included. This manifests itself in the need for

specifying the width of registers and data paths, and the connections between registers and functional units.

ISPS procedures may contain, together with control and data operations, declarations of local hardware units of arbitrary complexity. This allows machine descriptions to be constructed in a hierarchical fashion. Specialised control constructs are included in the language to enable the clear expression of the decoding of machine instructions, and to allow operations to take place concurrently. Procedures may be parameterised to allow multiple invocations of the same component within a circuit description.

Complete ISPS descriptions are analysed and transformed into a formally defined intermediate representation known as the global data base. This intermediate format is sufficiently generalised to enable it to be used for many diverse design applications. Such applications include simulation, fault analysis, architecture evaluation and design automation. By using this approach, the designers of ISPS hope to create a unified environment for research and development in multiple application areas. Thus ISPS descriptions would serve as a common vehicle for investigation into many aspects of the analysis and design of digital systems.

2.3.5.2 VHDL

VHDL has been developed as a standard HDL for the Very High Speed Integrated Circuit (VHSIC) project sponsored by the American Department of Defence[2.67]. VHDL supports the design, documentation and simulation of hardware from the digital system level to the gate level. VHDL is designed to be independent of any underlying technology or design methodology. This feature should enable the very latest advances in technology to be quickly and easily incorporated into the development of VLSI systems.

The primary abstraction mechanism in VHDL is the design entity, which is used to represent hardware components. Design entities are composed of an interface description and one or more bodies. The interface defines the external characteristics of a component, such as ports and generic parameters, and each body represents an alternative design approach consistent with those characteristics. Design entity bodies may be either architectural or behavioural in nature. Architectural bodies contain essentially structural information, and are intended to convey details about possible implementations of a component. Behavioural bodies give a control flow description of the desired behaviour of a component. They contain data structure definitions and define sequential algorithms that operate on the data structures to determine the values of the output signals. Data structures and algorithms are specified using a collection of common programming language constructs such as 'IF' and 'CASE' statements. A behavioural body may not contain any structural information about the component.

One feature of VHDL which distinguishes it from most other languages is the provision of two types of time. These are referred to as macro and micro time respectively. The macro-time scale represents real time units (e.g. microseconds), and is used to describe the temporal interaction among all the components in the circuit. The micro-time scale is used to specify short delays through combinational circuitry and is essentially not measurable. Thus, when the time between two hypothetical events A and B is described in micro-time, the implication is merely that 'B happens shortly after A', while macro-time is used to specify the time between events in a precise manner. Consequently, any number of micro-time units may exist between any two consecutive macro-time units.

VHDL provides a very specialised framework for the design of VLSI circuits. The overall organisation of the language reflects the hierarchical structure of hardware designs, and the design entity concept provides a suitable

abstraction for describing hardware components. The language allows all levels of hardware to be described independently of the the implementation technology or design methodology, and many circuit concepts, such as propagation delays and buses, are already built in. This gives a language which is initially straightforward to use, but which may lack a certain amount of generality and flexibility[2.54].

2.4 Discussion

In this chapter the most important and novel aspects of a number of experimental CAD systems have been presented, together with a general classification and examples of both software and hardware specification languages. Much state-of-the-art research in CAD is aimed at the behavioural specification and automatic synthesis of VLSI circuits. Many of the tools and techniques used in such systems are however applicable in a more general sense to other application areas in the wider realm of CAD for digital systems. It seems that this will especially apply to the adoption of expert system techniques. Expert systems can be used to reduce the complexity of CAD systems to a manageable scale, and mostly remove the need for intervention from human designers.

It further appears that the major difficulty encountered in both hardware and software specification is similar. This is namely that different applications are often best described by radically different specification notations. Also, if automatic synthesis is to be achieved, each of the intermediate levels of design description generated requires its own notation, which is appropriate to the level of abstraction needed at that particular stage of the synthesis process. This problem presents a significant challenge to the designers of CAD systems. Specification languages need to be defined which enable the major aspects of a required system to be adequately and concisely expressed. The underlying language model should closely

match the natural structure of the application to be described, and the language should contain constructs which are of an appropriate level of abstraction. Intermediate design notations then need to be developed, together with consistency-preserving transformation techniques, which can automatically derive successively more detailed descriptions of the application, until a complete implementation is reached.

The remainder of this thesis relates how some of these concepts and ideas have been applied to the specific area of a CAD system for single-board microprocessor controllers. Many of the software and hardware specification techniques described above are brought together to form a behavioural specification language for control systems. A method of transforming behavioural specifications through an intermediate design description into software implementations is also presented. This process involves interaction with a knowledge-based hardware design system, which is under development in another project. Finally the parallel programming language occam is used as a hardware description language to construct a generalised simulation environment for microprocessors. Such simulations can be used to validate designs which are produced by the CAD system.

References

2.1 Stefik,M.J. and Kleer,J.D.:'Prospects for Expert Systems in CAD', Computer Design, pp 65-76, vol 22, no 5, 1983

2.2 Newton,A.R. and Sangiovanni-Vincentelli,A.L. : 'Computer-Aided Design for VLSI Circuits', IEEE Computer, pp 39-59, vol 19, no 4, April 1986

2.3 Carter,H. :'Computer-Aided Design of Integrated Circuits', IEEE Computer, pp 19-36, vol 19, no 4, April 1986

2.4 Werner,J. :'The Silicon Compiler : Panacea, Wishful Thinking, or Old Hat?', VLSI Design, Sept/Oct, 1982, pp 46-52

2.5 Siewiorek,D.P. and Barbacci,M.R. :'CMU RT-CAD System: An innovative Approach to CAD', Procs AFIPS NCC, vol 45, 1976, pp 643-655

2.6 Darringer,J.A. :'The Description, Simulation and Automatic Implementation of Digital Computer Processors', PhD Thesis, Department of Electrical Engineering, Carnegie-Mellon University, May 1969

2.7 Bell,C.G. and Newell,A. :'Computer Structures, Readings and Examples', McGraw-Hill, New York, 1971

2.8 Bushnell,M.L. and Director,S.W. :'ULYSSES - a Knowledge-based VLSI Design Environment', Artificial Intelligence, vol 2, no 1, Jan 1987, pp 33-41

2.9 Steinberg,L.I. and Mitchell,T.M. :'A Knowledge Based Approach to VLSI CAD - The Redesign System', Procs. 21st Design Automation Conf, ACM (SIGDA) and IEEE (Computer Society), 345 East 47 St., New York, NY 10017, pp 412-418, 1984

2.10 Barr,A. and Feigenbaum,E.A. :'Frames and Scripts', in The Handbook of Artificial Intelligence, HeurisTech Press and Williams Kaufmann, Inc., Stanford, California, volume 2, pp 54-57, 1981

2.11 Bushnell,M.L. and Director,S.W. :'ULYSSES - an Expert System Based VLSI Design Environment', Procs 1985 International Symposium on Circuits and Systems, IEEE Service Centre, Piscakiway, N.J., pp 893-896

2.12 Reddy,R, et al :'The Hearsay2 System', in Speech Understanding Systems, Dept. of Computer Science, Carnegie-Mellon University, 1976, ch 1, pp 6-7

2.13 Schank,R. et al :'SAM - A Story Understander', Research Report 43, The Yale A.I. Project, Yale University, August 1975

2.14 Subrahmanyam,P.A. :'Synapse: An Expert System for VLSI Design', IEEE Computer, pp 79-89, vol 19, no 7, July 1986

2.15 Intellicorp :'KEE Software Development System User's Manual', Jan 1985

2.16 Bowen,J.A. and Smith,M.F.:'Expert Systems for the Analysis and Design of Microprocessor Applications', Journal of Microcomputer Applications, (1983)6, pp 155-161

2.17 Flake,F.P. et al :'HILO MARK 2 Hardware Description Language', Procs. of the IFIP 5th International Conference on CHDLs, Kaiserslautern, Sept 1981

2.18 Bowen,J.A. :'Automated Configuration of Backplane-based Microcomputers', Procs Conf on CAD, 1984

2.19 Davis et al:'An Overview of Production Systems', Machine Intelligence, pp 300-332, vol 8, 1977

2.20 McDermott, J. :'R1: a Rule-based Configurer of Computer Systems', Carnegie-Mellon University, Department of Computer Science, 1980

2.21 Forgy.C.L.and McDermott,J.:'OPS, a Domain Independent Production System Language', Proceedings of the fifth International Joint Conference on AI, MIT, 1977, pp 933-939

2.22 Newell.A, Shaw,J.C. and Simon,H.A. :'Empirical Explorations with the Logic Theory Machine', in Computers and Thought, edited by Feigenbaum,E.A. and Feldman,J., McGraw-HIll, New York, 1963

2.23 McDermott,J. :'Domain Knowledge and the Design Process', Procs. of ACM IEEE Design Automation Conference, Nashville, 29 June - 1 July, 1981

2.24 Sussman,G.J. :'Electrical Design, A Problem for Artificial Intelligence Research', IJCAI no 5, 1977, pp 894-900

2.25 Stefik,M.J. et al :'The Partitioning of Concerns in Digital System Design', Procs. Conference on Advanced Research in VLSI, pp 43-52, Jan 1982

2.26 Hekmatpour,S :'The Execution of Formal Specifications', Computing Discipline, Open University, Technical Report 85/2, June 1985

2.27 Riddle,W.E. and Wileden,J.C.:'Languages for Representing Software Specifications and Designs', ACM SIGSOFT Notices, no 3 Oct 1978, pp 7-11

2.28 Stoegerer,J.K. :'A Comprehensive Approach to Specification Languages',Australian Computer Journal, vol 16, no 1, Feb 1984, pp 1-13

2.29 Teichroew,D. and Hershey,E.A. :'PLA/PSA - a Computer-aided Technique for Structured Documentation and Analysis of Information Processing Systems', IEEE Trans. Software Engineering, SE3 Jan 1977, pp 41-48

2.30 Ince,D. :'Z and System Specification', Information and Software Technology, vol 30, no 3, April 1988, pp 139-145

2.31 Coleman,D. and Gallimore,R.M. :'Software Engineering Using Executable Specifications', Department of Computation, UMIST, 1984

2.32 Riddle,W.E. et al :'An Introduction to the DREAM Software Design System', ACM SIGSOFT Notices, no 2, July 1977, pp 11-24

2.33 Ambler,A. et al :'GYPSY - A Language for Specification and Implementation of Verifiable Programs', Procs. Conf. on Language Design for Reliable Software, ACM SIGPLAN notices, vol 12, no 3, March 1977, pp 1-10

2.34 Wirth,N. :'Towards a Discipline of Real-time Programming', CACM, vol 20, Aug 1977 pp 577-583

2.35 Minsky,M.L.:'Computation. Finite and Infinite Machines', Prentice-Hall 1967

2.36 Riddle,W.E. :'Behaviour Modelling During Software Design', IEEE Trans. Software Engineering, SE4, July 1978, pp 283-292

2.37 Kawashima,H. et al:'Functional Specification of Call Processing by State Transition Diagrams', IEEE Trans. Communication Technology, vol 19, no 5 Oct 1971, pp 581-587

2.38 Davis,A.M. and Rataj,W. :'Requirements Language Processing for Effective Testing of Real-time Systems', Proc. Software Quality and Assurance Workshop, Nov 1978 San Diego, pp 61-66

2.39 Alford,M.W. :'A Requirements Engineering Methodology for Real-time Processing Requirements' IEEE Trans. Software Engineering, SE-3, Jan 1977 pp 97-108

2.40 Balzer,R. and Goldman,N. :'Principles of Good Software Specification and Their Implications for Specification Languages',Proc. Specification of Reliable Software, 1979 pp 58-67, IEEE cat no 79, Ch 1402-9C

2.41 Jones,C. :'A Survey of Programming Design and Specification Techniques', Proc IEEE Conf. Specifications for Reliable Software, pp 91-103,1979

2.42 Willis,R.R. :'AIDES - Computer-aided Design of Software Systems', Software Engineering Environments, GMD, North Holland, 1981, pp 27-48

2.43 Caine,S.H. and Gordon,E.K.:'PDL - A Tool for Software Design', Proc. AFIPS NCC, pp 271-276, 1975

2.44 Biggerstaff,T.J.:'The Unified Design Specification System(UDS)',Proc. IEEE Conf. Specifications for Reliable Software, pp 104-118, 1979

2.45 Parnas,D.L.:'The Use of Precise Specifications in the Development of Software', Information Processing 77, IFIP, pp 861-867, North Holland Publishing Company, 1977

2.46 Zemanek,H.:'Formalism: History, Present and Future',Proc 4th Informatik Symp., Sept 1974, Springer-Verlag Lecture Notes in Computer Science no 23, pp 477-501

2.47 Jones,C.:'Software Development - A Rigorous Approach', Prentice-Hall, 1980

2.48 Duce,D.A. and Fielding,E.V.C.:'Formal Specification - A Comparison of Two Techniques', The Computer Journal, vol 30, no 4, April 1987, pp 316-327

2.49 Gibbins,P.F.:'What are Formal Methods?', Information and Software Technology, vol 30, no 3, April 1988, pp 131-137

2.50 Ludewig,J.:'Computer-Aided Specification of Process Control Systems', IEEE Computer, vol 15, no 5, May 1982, pp 12-21

2.51 Naur,P. (ed):'Revised Report on the Algorithmic Language ALGOL 60', Communications ACM, vol 6, no 1, Jan 1963, pp 420-453

2.52 Barbacci,M. :'Structure and Behaviour of Digital Systems' in New Computer Architecture, edited by J.Tiberghien, 1984

2.53 Chu,Y :'Why do we need Computer Hardware Description Languages', IEEE Computer, vol 7, no 12, Dec 1974, pp 18-22

2.54 Davie.B.S.:'Hardware Description Languages: Some Recent Developments', report no. CSR198-86, Dept. of Computer Science, University of Edinburgh, April 1986

2.55 Shahdad,M. et al:'VHSIC Hardware Description Language', IEEE Computer, vol 18, no 2, Feb 1985 pp 95-103

2.56 Lattice Logic Ltd:'CHIPSMITH, a Random Logic Compiler for Gate-arrays, Optimised Arrays and Standard Cells', Edinburgh 1985

2.57 Praxis Systems Ltd.:'ELLA : Design, Modelling, Simulation - System Overview', Bath 1985

2.58 Suzuki,N.:'Concurrent Prolog as an Efficient VLSI Design Language', IEEE Computer, vol 18, no 2, Feb 1985, pp 33-39

2.59 Barbacci,M.R. :'Instruction Set Processor Specifications(ISPS): The Notation and its Applications',IEEE Trans. on Computers, vol c-30, no 1, Jan 1981, pp 24-39

2.60 Collis,G.V. and Edwards,M.D.:'Automatic Hardware Synthesis from a Behavioural Description Language: occam', Microprocessing and Microprogramming, vol 18, pp 243-250, 1986

2.61 Plessey Semiconductors:'Plessey Megacell', 1985

2.62 Johannsen,D.:'Bristle Blocks : A Silicon Compiler' Caltech Conference on VLSI, 1979, pp 303-310

2.63 Organick,E.I. et al:'Transforming an ADA Program Unit to Silicon and Verifying its Behaviour in an ADA Environment: A First Experiment', IEEE Software, vol 1, no 1, 1984, pp 31-48

2.64 May,D. and Keane,C.:'Compiling occam into Silicon', Communicating Process Architecture, Inmos Ltd, 1986

2.65 German,S.J. and Lieberherr,K.J.:'Zeus: A Language for Expressing Algorithms in Hardware', IEEE Computer, vol 18, no 2, Feb 1985, pp 55-65

2.66 May,D. :'occam (Hardware Description Language)', Procs. IEE Colloq. on Software Tools for Hardware Design, London 1985, Digest no 98 P5/1-5

2.67 Waxman,R.:'Hardware Design Languages for Computer Design and Test', IEEE Computer, vol 19, no 4, April 1986, pp 90-97

3. A Behavioural Specification Language for Minimum Configuration Systems

3.1 Language Requirements

The aim of the Behavioural Specification Language(BSL) described in this section is to provide a notation which allows a system designer to clearly express the design decisions taken from a given MCS requirements specification. These design decisions include the specification of the precise control algorithm chosen for each system variable, the strategies to be used to achieve input and output, and any important time and performance constraints. All this information however should be described at a sufficiently abstract level, which does not constrain the number of possible implementation alternatives, particularly in terms of hardware/software trade-offs. Thus the following set of requirements for the BSL can be defined.

3.1.1 Event-Based Model

The behaviour of an embedded MCS can be described in terms of actions which must be carried out in response to external stimuli[3.1]. Such systems continuously monitor and alter their external environment in order to maintain some desired stable state. There are two mechanisms which can be employed by microprocessors to detect the arrival of external stimuli[3.2]. These are :-

1) interrupts
2) polling

At the design specification level, the designer must have decided whether the input lines will be interrupt driven or polled. If there are several interrupt driven lines, some form of priority may need to be imposed on the exact order of interrupt processing. In exceptional cases the designer may wish to mix the polling of inputs with interrupt driven lines. Interrupts and polling systems are inherently event-based, and therefore a specification language for

such systems must be able to reflect this mode of behaviour in a simple, natural and abstract fashion.

3.1.2 Actions

The language must provide a mechanism for associating a sequence of actions with the arrival of an external stimulus. Each sequence of actions would typically consist of individual assignment, calculation, repetition, comparison and output actions.

3.1.3 Representation of Time Constraints

It must be possible to specify a maximum permissible time period for the execution of a set of actions associated with a particular input. This may be necessary when an external device must be serviced in a short time period. Further, the language must allow the specification of the sampling of an input at a regular time interval. This is a common feature of control systems, which can be applied when the rate of change of a controlled variable is well understood[3.3].

3.1.4 Formal Definition

The BSL should be formally defined to ensure that it contains no unforeseen inconsistencies or redundancies, and is completely unambiguous[3.4].

3.1.5 Analysable

It must be possible to check BSL descriptions for static validity. To this end, the language should enforce strong typing rules for all variables and expressions. Scope rules similar to those found in most block-structured high-level languages should also be incorporated. These would allow the values of variables to be shared amongst several routines or to remain private to just one particular routine.

3.1.6 Executable

It must be possible to transform a design specification into a high-level behavioural simulation of the desired system. This allows the designer to ascertain at a very early stage that the design meets its requirements. The simulation at this level is less detailed than the component level simulations which may subsequently be generated by the CAD system. However, a general behavioural simulation provides a useful prototyping tool to validate designs before the selection of specific components and algorithms is initiated.

3.1.7 Familiarity

The language should contain constructs and expressions which are common in existing design specification and programming languages. This would enable designers to quickly become familiar with the language and consequently make its use a more practical proposition.
Specifications should also be expressed in a modular fashion. This would enhance both the readability and maintainability of specifications.

3.2 Language Basis

It was decided to base the BSL upon the programming language occam[3.5]. Other specification languages[3.6,3.4] have been successfully built around the features of existing high-level languages such as Pascal. This approach makes sense from a software engineering viewpoint. Much research has gone into the design of the many widely-used high-level languages. Therefore, as consolidation is a major virtue of a language designer, existing languages should be used as a basis for development wherever they meet the desired criteria[3.7, 3.8]. A brief introduction to occam is given in Appendix A.

Occam was chosen for the following reasons:-

1) It provides a simple and clear notation for expressing input and output operations, and the semantics of the input operation precisely match the event model of specification languages.

2) The occam ALT construct provides a convenient abstraction for describing interrupts, priority and non-determinancy.

3) Occam is a simple and secure language, built upon a strong formal basis[3.9, 3.10]. It contains control constructs which are similar to those found in most high-level languages.

4) Occam channels provide a natural, abstract mechanism for representing the input and output lines from an MCS to the environment under control.

5) Existing software tools can be utilised to verify and compile specifications to give a high-level simulation of the system being designed.

However, the occam language is not used in its pure form for behavioural specifications. Rather, constructs have been added to the language to enable designers to operate at a more abstract level, with notations which closely match the features of the problem at hand. Conversely, several of the fundamental features of occam have been omitted from the BSL. The most important of these is the occam PAR statement. The expression of parallelism is not necessary in single-board embedded systems, which are implemented by conventional 8- or 16-bit microprocessor technology. Parallelism in control systems becomes of greater importance in larger minicomputer or mainframe systems, where the control software tasks may be multiprogrammed under the supervision of a multi-tasking operating system, such as UNIX[3.11] or VMS[3.12].

In order to execute behavioural specifications, the additional BSL constructs must be transformed into occam. The resulting syntactically correct occam program is then compiled (by an existing occam compiler) and executed to give a behavioural simulation of the system.

3.3 Language Features

3.3.1 Specification Structure

This section describes in depth the most important features of the BSL. Examples of the constructs are given together with a syntax description in Backus-Naur Form(BNF)[3.13]. However, so as not to include superfluous detail at this stage, the syntax description in this section is not complete. A complete BNF for the language is given in Appendix B.

A behavioural specification comprises a title section and a specification section.

```
<BSL Specification> ::= <title> <specification section>
<title> ::= TITLE <text> :
<specification section> ::= <channel declarations>
                            <routine declarations>
                            <control section> :
```

The title section is included purely for documentation purposes. The text which appears between the keyword TITLE and the terminating colon is treated as a comment, and thus may contain anything the specification author desires.

The specification section comprises three distinct phases. These are the channel declarations, the interrupt service routine declarations, and the control section which states how each individual input channel is to be handled.

3.3.2 Channel Declarations

```
<channel declarations> ::= CHAN <direction>
                           <channel id list>  :
                           { <channel declarations> }
<direction> ::= IN | OUT
<channel id list> ::= <channel id> { , <channel id> }
<channel id> ::= <valid variable name>
```

The BSL uses a slight variation on the occam channel to represent input and output lines to the system. Channels are declared, as in occam, in a CHAN statement. In addition, channels must be explicitly defined as input or output. This enables stringent checks to be made on channel use. An example of a channel declaration is given below:-

```
CHAN IN temperature, pressure:
CHAN OUT valve, heater :
```

3.3.3 Service Routine Declarations

```
<Service routine declarations> ::= PROC <routine id> :
                                   <type>
                                   <formal parameter
                                    list>
                                   <time constraint>
                                   <routine body> :
                          {<Service routine declarations>}
<routine id> ::= <valid variable name>
<type> ::= INTERRUPTABLE | UNINTERRUPTABLE
<formal parameter list> ::= () |
                            ( <typed variable list> ) |
                            (  <typed variable list>
                            { , <typed variable list>} )
<time constraint> ::= <empty> | <integer value> <units>
<routine body> ::= <local declarations> <processing>
<typed variable list> ::= <variable type>
                          <variable list>
```

A service routine declaration is based upon the process and procedure definition in occam. Each service routine may consist of occam declaration, assignment, calculation, repetition, comparison, input and output statements. A RESULT statement, similar to that in occam, is used to return from the service routine the value that is to be output to control the environment. A simple example of a single service routine is given below.

```
PROC Temp.Regulation : INTERRUPTABLE
                       (BYTE temp)
  VAL critical.temp IS 150:
  VAL heater.on     IS 1:
  VAL heater.off    IS 0:
  SEQ
    IF
      temp < critical.temp
        RESULT(heater.on)
      temp >= critical.temp
        RESULT(heater.off)
:
```

Routines are typed as 'INTERRUPTABLE' or 'UNINTERRUPTABLE'. This enables the designer to specify whether or not a particular service routine may be halted in order to service a more important interrupt request. It is analogous to the masking and unmasking of interrupts at assembly language level. The parameters passed to a service routine are regarded as passed by reference parameters, and thus their value may be altered within the service routine itself. An optional time constraint parameter may be associated with a service routine. This states the maximum time period that a routine may take to process a particular input value. The time period may currently be specified in units of either milliseconds or seconds. These should provide sufficient and convenient expressive power to cover all but the most time-critical of applications, in which a routine must service an input in less than one millisecond.

3.3.4 Control Section

```
<control section> ::= <global variable declarations>
                      <execution condition>
                      <input structure>
<global variable declarations> ::= <array definition>
                                   <variable type>
                                   <variable list> :
<array definition> ::= <empty> | [ <integer value> ]
<variable type> ::= BOOL | BYTE | INT | REAL
                  | CHAN <direction>
<variable list> ::= <valid variable name>
                    { , <valid variable name> }
<execution condition> ::= WHILE <Boolean condition>
<input structure> ::= <control statement>
                      <event statement>
                      { <event statement> }
                      { <input structure> }
```

```
<control statement>::= INTERRUPT | PRI INTERRUPT | POLL
<event statement> ::= <input variable declaration>
                      <input statement>
                      <output statement>
<input variable declaration> ::= <empty> |
                                 <variable type>
                                 <valid variable name>:
<input statement> ::= <channel id>     ?
                      <valid variable name>
                      <sample statement>
<sample statement> ::= <empty> |
                       SAMPLE <interval>
<interval> ::= <empty> | <integer value> <units>
<output statement> ::= <channel id>    !
                       <routine id>
                       <actual parameter list>
```

The control section specifies the manner in which values are to be received on the input channels from the environment. Global variables, whose value is of importance to more than one particular service routine, are declared at this stage. Restrictions are imposed on the occam variable types, both global and local, that may be used in the BSL. The full range of variable types offered in occam is not required because dedicated controllers do not usually need the precision of 64-bit floating-point numbers or 32-bit integers. Also these data types cannot be efficiently implemented on 8-bit, or to a lesser extent, 16-bit microprocessors. For these reasons, the only data types permitted in the BSL are BOOL, BYTE, INT (16-bit integer) and REAL (32-bit real). These should provide sufficient expressive power and accuracy for all intended applications.

Channel input and output operations are expressed using the '?' and '!' notation of occam. However, while the syntax of the input statement is not altered, the output statement is modified to give a more abstract, functional representation. This is illustrated below.

```
temperature ? current.temp
  heater ! Temp.Regulation (current.temp)
```

The behaviour of embedded systems is inherently event-based. These systems continuously monitor and alter

their external environment in order to maintain some desired stable state. Thus, the above pair of input and output statements show how this situation can be expressed. They state that the system receives a value on the channel 'temperature', and subsequently outputs a value on the channel 'heater', which is determined by applying the service routine 'Temp.Regulation' to the current temperature value. This notation provides a simple, abstract mechanism for associating a particular sequence of actions, a service routine, with the arrival of an external stimulus. In many applications a system will be dedicated to controlling more than one physical device, the values from which may be inter-related. It must therefore be possible to place constraints upon the order in which interrupts are serviced, and allow some devices, if necessary, to have priority over others. To achieve this, two constructs, INTERRUPT and PRI INTERRUPT have been included in the BSL. Syntactically they are similar to the occam ALT and PRI ALT statements, but semantically they are radically different. In addition, a POLL construct has been incorporated to allow the polling of input lines to be expressed.

For example, consider a system which waits for inputs on a single input channel. If the arrival of values on this channel is to be signalled by interrupts, the situation can be described as :-

```
CHAN IN input:
CHAN OUT output:
WHILE TRUE
  INTERRUPT
    BYTE value:
    input ? value
      output ! process.value ( value )
```

INTERRUPT clearly portrays the purpose of the statement in a vocabulary that is familiar to microprocessor system designers. If it is desired to continuously poll a single input line, the following construct could be used:-

```
CHAN IN input :
CHAN OUT output:
WHILE TRUE
  POLL
    BYTE value:
    input ? value
      output ! process.value ( value )
```

The POLL and INTERRUPT constructs therefore convey the implementation strategy decided upon by the designer. These constructs can easily be extended to cater for multiple inputs. For example, using the INTERRUPT construct:-

```
CHAN IN  temp1, temp2, emergency:
CHAN OUT heater1, heater2, alarm:

INTERRUPT

  BYTE any:
  emergency ? any
    alarm ! sound.siren ( any )

  BYTE value1:
  temp1 ? value1
    heater1 ! process.value1 ( value1 )

  BYTE value2:
  temp2 ? value2
    heater2 ! process.value2 ( value2 )
```

In the above example, all three interrupt lines are of equal priority. However, the designer may impose a simple two-level priority ordering on inputs by defining the procedures associated with each input as interruptable or uninterruptable. An uninterruptable procedure is consequently of a higher priority than one that may be interrupted.

General interrupt priority can be expressed by utilising the PRI INTERRUPT construct. Inputs are assigned a priority according to the textual order in which they appear in the construct - the first has the highest priority and so on. When an interrupt occurs, the priority of its input channel is compared with the priority of any currently active interrupt service routine. If the executing routine is of a lower priority, it is suspended and the routine to process the interrupt is executed. If the converse is true, the

active routine continues to execute. Thus PRI INTERRUPT essentially provides a daisy chain system in that the input which invokes the construct from an inactive state may not be the one that is immediately satisfied if another input of higher priority arrives. An example of the use of a PRI INTERRUPT is given below:-

```
CHAN IN  temp1, temp2, emergency:
CHAN OUT heater1, heater2, alarm:

PRI INTERRUPT

  BYTE any:
  emergency ? any
   alarm ! sound.siren ( any )

  BYTE value1:
  temp1 ? value1
    heater1 ! process.value1 ( value1 )

  BYTE value2:
  temp2 ? value2
    heater2 ! process.value2 ( value2 )
```

3.4. An Example

Consider a water tank which is used to supply water at a certain temperature to an industrial process. The water flows into the tank at a constant rate, is heated to within a given temperature range, and flows out at a rate determined by the process which consumes the water. In normal operation, the output flow will be equal or slightly greater than the input flow. Importantly though, the output flow is never less than the input. This means that the tank cannot overflow, but could become empty if the output flow remains greater than the input for a significant period. To guard against this possibility, a level detector is placed in the tank, which indicates that the water level is dangerously low. When this happens, the valve which controls the inflow of water can be opened wide for a short time, thus refilling the tank to a safe level. A further level detector is used to indicate that the water level is satisfactory, enabling the valve to be closed to its normal setting. A diagram of this system is given in Figure 3.1.

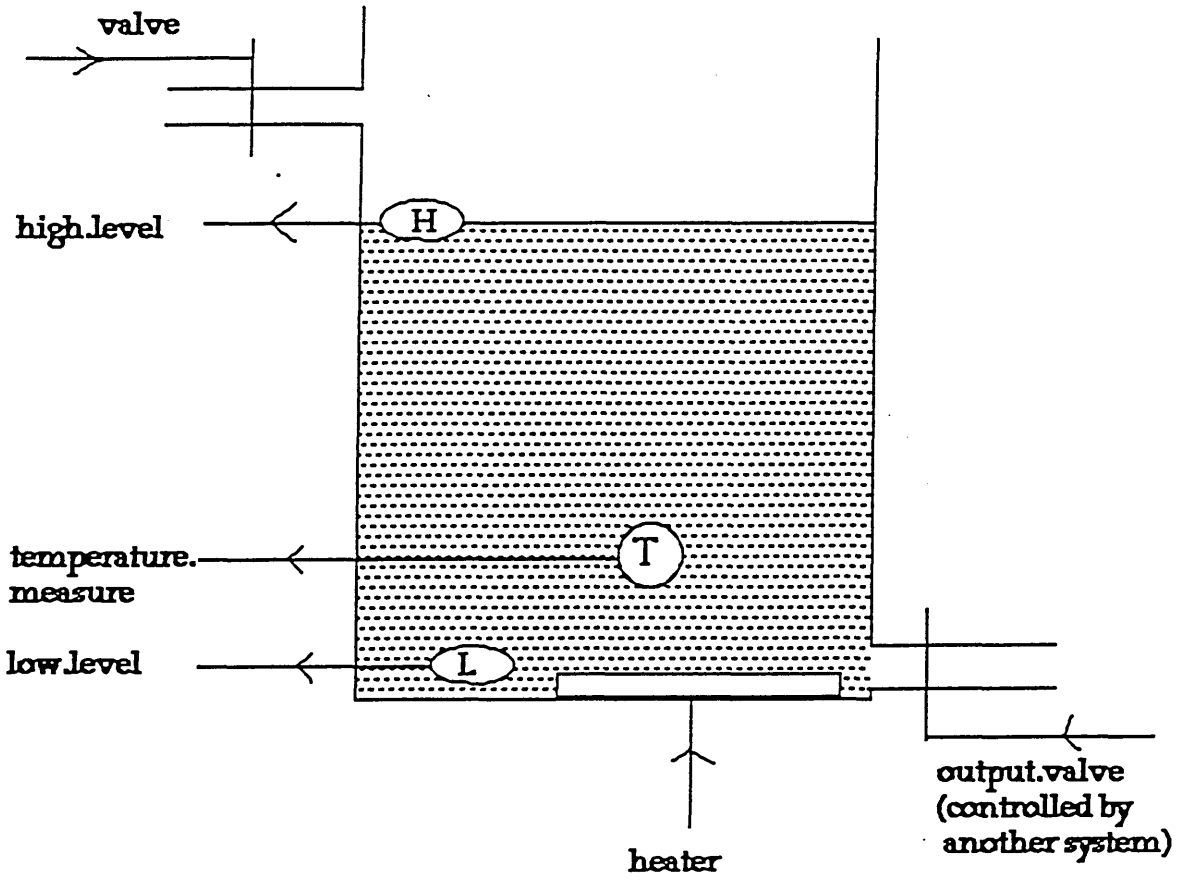


Figure 3.1 A Water Tank Controller

The behavioural specification of the system required to control the tank is shown in Figure 3.2. The algorithm adopted to control the water temperature is deliberately simplified, in order to illustrate the main features of the specification. In a complete implementation, a more complex control algorithm may be better suited[3.3]. Examples in Chapter 6 illustrate how the features of the BSL can be utilised to describe systems which exhibit greater complexity of behaviour.

In Figure 3.2, three distinct input channels are shown which may generate interrupts. The sources of these interrupts are the two water level measuring devices, and a temperature sensing device. Note that in the INTERRUPT statement, the input for the channel 'temperature' is followed by the SAMPLE option. SAMPLE followed by a time period in seconds, states how often the interrupt for this channel is to occur. In this case then, the temperature of the water in the tank is to be measured every five seconds. The sampling of physical variables in this manner is a very common feature of this kind of control application. Sampling may be implemented in software, using a loop to cause the required time delay, or in hardware, using a programmable timer chip to generate interrupts when the sampling period is complete. Both approaches have advantages and disadvantages, and the relative merits of each approach are dictated by the nature and requirements of the control system under consideration. In certain applications, when the exact rate of change of the environment is uncertain, it may be desirable to sample an input as often as possible. This can be expressed in the BSL by a SAMPLE statement which is not followed by any specific time interval. For example:-

```
pressure ? value    SAMPLE
```

The SAMPLE statement provides a simple, concise syntax for expressing this mode of operation. More importantly though, SAMPLE conveys only design level information. It does not enforce a particular implementation strategy to be adopted at a later stage of the development process.

```
CHAN IN temperature.measure, low.level,
        high.level:
CHAN OUT heater, valve:

PROC temperature.control: INTERRUPTABLE
                          (BYTE temp)
  VAL high.temp     IS 90:
  VAL low.temp      IS 80:
  VAL heater.off    IS 0:
  VAL heater.full   IS 2:
  VAL heater.warm   IS 1:
  SEQ
    IF
     temp > high.temp
       RESULT (heater.off)
     temp > low.temp
       RESULT (heater.warm)
     temp <= low.temp
       RESULT (heater.full)
:

PROC open.valve: UNINTERRUPTABLE
                 ()
  VAL open.valve IS 1:
  SEQ
    RESULT (open.valve)
:

PROC close.valve: UNINTERRUPTABLE
                  ()
  VAL close.valve IS 0:
  SEQ
    RESULT (close.valve)
:

WHILE TRUE

  INTERRUPT

    BYTE temp:
    temperature.measure ? temp    SAMPLE 5 seconds
      heater ! temperature.control (temp)

    BYTE low:
    low.level ? low
      valve ! open.valve ()

    BYTE high:
    high.level ? high
      valve ! close.valve ()
:
```

Figure 3.2 Behavioural Specification for the Water Tank Controller

In contrast, no time requirements are placed on the interrupts generated on the channels 'low.level' and 'high-level'. The control system does not need to continuously monitor the level in the tank. However, it must respond immediately when the external hardware informs it that the level is dangerously low or high. For this reason, the routines which open and close the valve to refill the tank are defined as UNINTERRUPTABLE. The arrival of an interrupt signal on either of the level monitoring channels must result in the control system instantly issuing the appropriate command. Failure to do this could lead either to the tank becoming empty, or overflowing.

Note that in this particular application, a new reading on any of the input channels is used to calculate a new value for just one corresponding output channel. The service routines can thus be regarded as functionally distinct, and do not share common data. The value read from one input channel is only of relevance to one particular service routine and can be regarded as private to that routine. Such locality of data can be expressed in the BSL by declaring an input variable immediately prior to its use in an input statement, as in Figure 3.2.. In applications where input values must be referenced by more than one service routine, the input variables must be declared globally.

The 'WHILE TRUE' statement in the specification is used to signify that the control system has no special terminating condition: essentially it operates until the power supply is removed. A more controlled method of termination can be specified easily using Boolean variables and termination signals. For example:-

```
BOOL running:
SEQ
  running := TRUE
  WHILE running
    INTERRUPT
      -------
      -------  input statements
      -------
      BYTE any:
      shut.down ? any
        running := FALSE
:
```

Thus, receipt of a signal on the channel 'shut.down' would cause the system to terminate. This is virtually identical to the technique employed in occam programs to terminate a set of concurrent processes.

3.5 Executing Behavioural Specifications

The ability to execute behavioural specification at a generalised, implementation-independent level is seen as an important phase of the development process[3.14]. The simulation of the desired MCS, which is created by running the behavioural specification, forms a further level at which verification of the design can take place. The designer may test the simulation by supplying example values on each of the input channels. In this manner, it can be determined whether each service routine produces the correct output values for a given set of inputs, and whether inputs for different channels are processed in the required order.

To transform behavioural specifications into an executable form, it was decided to convert the specifications into occam. The occam representation of the system could then be compiled into executable code by an existing occam compiler. An alternative approach would have been to write an interpreter for the BSL. This however was deemed less attractive. Much of the BSL is in fact a precise subset of occam. It therefore seems more sensible to convert the additional BSL features into occam. Using this approach, all the BSL features which are identical in occam can be

initially ignored, and left to be dealt with by the occam compiler when all the other BSL features have been converted. This method should reduce the effort required to convert behavioural specifications into an executable form, as it makes considerable use of existing software tools.

The strategy adopted to transform BSL descriptions into occam is to regard each service routine as a concurrent process, whose execution is controlled by a supervisor process which executes in parallel with the service routines. The role of the supervisor is to accept test values for the input channels in the specification, and to impose the required ordering constraints on the processing of these inputs. Information defining the exact characteristics of each service routine in specification is passed to the supervisor process as parameters: the code which comprises the supervisor is sufficiently generalised, so that it can be included without modification into any system simulation.

As an illustration, consider the specification of the water tank controller given in Figure 3.2. When this is converted into occam, the three service routines become occam processes which execute in parallel with the supervisor. Each service routine is connected to the supervisor process by a pair of occam channels, which are used to communicate data between the service routines and the supervisor. This information represents values for the service routines to process, results for the supervisor to display, and control and status information to enable routines to be interrupted and restarted. The structure of the transformed occam representation of this system is shown in Figure 3.3. The circles represent occam processes, and the lines represent the channels used to communicate between processes. It is worth noting that all the parallel behaviour incorporated in Figure 3.3 is automatically generated during the transformation process from the BSL to occam. Concurrent execution of processes is consequently never a complexity which the system designer needs to consider. Further, a

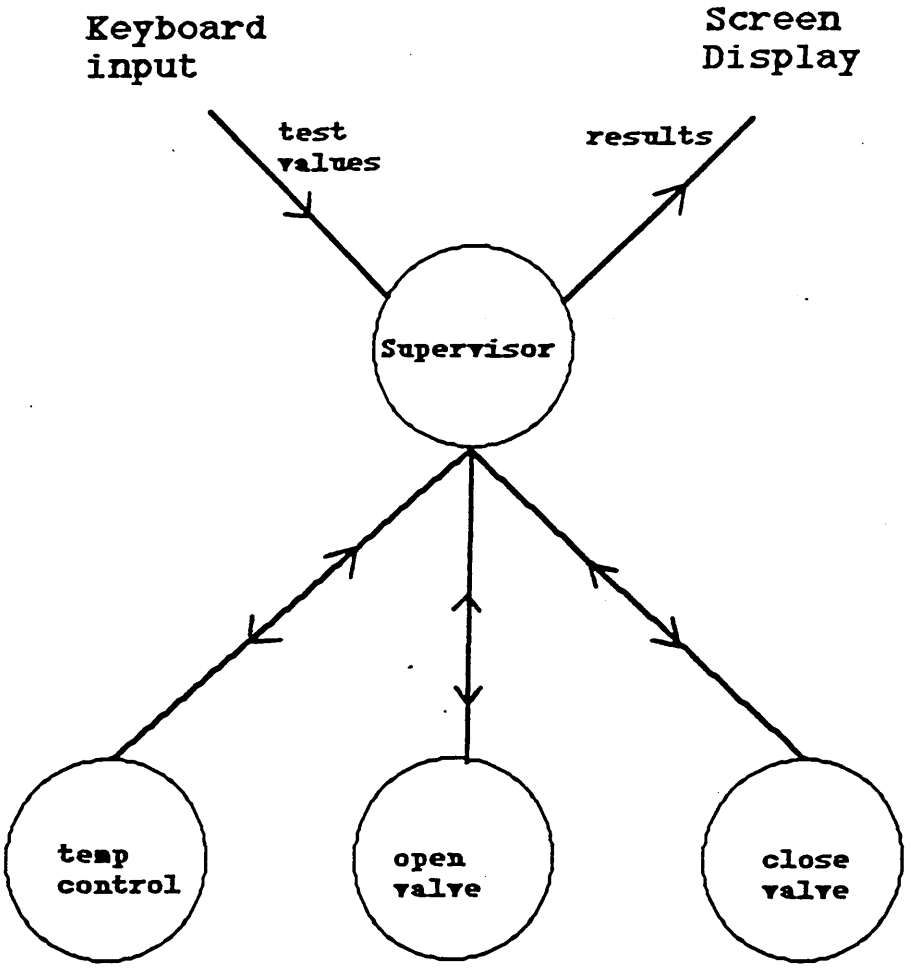


Figure 3.3 Occam Process Structure for the Water Tank Controller Specification

simple, generalised user interface is generated and managed by the supervisor process. This provides a user-controlled test environment which can be used regardless of the specific application under design. Figure 3.4 shows the initial screen display given when the specification in Figure 3.2 is transformed, compiled and executed.

The user interface allows the designer to initiate inputs on any of the input channels defined, to continue the execution of the currently active routine, or to end the simulation. Firstly then, when a simulation is started, an interrupt must be initiated and an input value entered by the user. This value is read by the supervisor process, and subsequently passed to the service routine associated with that input channel. The 'processor state' field on the screen is updated to 'active', and the supervisor waits for the next instruction from the user. If the user decides to continue with the active routine, the supervisor sends a message to the routine, requesting that it completes its processing and returns the result. The supervisor then displays the result on the screen (together with the input value which caused that result to be reached), signifies that the simulation is once again inactive, and awaits further instruction.

However, if the user chooses to initiate another input before the active routine is allowed to complete, the supervisor must decide whether to reject the input, or to interrupt the current routine and allow the most recent input to be processed immediately. This decision is based upon the relative priorities of the input channels involved, and the 'type' of the currently active service routine. If the active routine is defined as uninterruptable in the specification, the newly initiated input is always rejected. But should the active routine be interruptable, it is suspended if the priority of the requested input is equal to or greater than the active routine. A suspended routine is immediately resumed when the processing of the higher priority interrupt is

```
OPTIONS   1. Initiate Interrupt or Input
          2. Continue Currently Active Routine
          3. End System Execution

Enter Option:

Processor State  -- Inactive
Currently Active Routine      :
                                Input Value:
                                Result      :

Routine Identification Table
   0 = temp.control      1 = open.valve      2 = close.valve
```

Figure 3.4 User Interface for Behavioural Simulation

completed. Nesting of interrupts to greater depths is allowed, up to an arbitrarily imposed limit (i.e. the size of the interrupt stack).

The data which defines the priority level and type of each service routine is built up as specifications are analysed and subsequently transformed into occam. This data is then converted into constants in the form of occam tables, and written out into the occam representation of the specification. These tables, together with an integer constant which represents the number of service routines in the specification, are then passed as parameters to the supervisor process. More precisely, the parameters passed are:-

1. The number of service routines in the simulation.

2. A two-dimensional byte array which holds the name of each routine in the simulation. It primary purpose is to enable the identification of each routine to the user on the screen display.

3. A Boolean table which states whether a routine may be interrupted. If the entry for a routine is TRUE, then it may be interrupted by a higher or equal priority input: if the table entry is FALSE, the routine cannot be interrupted under any circumstances.

4. An integer array which defines the priority level allocated to each routine, in accordance with the behavioural specification.

The exact priority levels which are allocated to the service routines are determined according to the following rules.

If an INTERRUPT construct is recognised during the transformation process, each routine referred to in that

construct is allocated a priority level of 0. This method of priority allocation defines a system in which, if all the service routines are interruptable, an input occurring on any channel will cause any active routine to be suspended. However, if some routines are defined as uninterruptable, a simple two-level priority scheme is created, in which routines that may not be interrupted are of a higher priority than interruptable routines.

If a PRI INTERRUPT construct is recognised, the first routine to appear in textual order is allocated a priority level of 99, the next 98, and so on. (This, of course, restricts the number of inputs that may be currently specified in a PRI INTERRUPT construct to a maximum of 99. This value is considered unlikely to be exceeded.)

If a POLL construct is recognised, all the enclosed routines are allocated a priority level of -1. This is because the actual process of polling an input line implies that the arrival of an interrupt on that line cannot generate an interrupt. Therefore inputs to polled lines can only be accepted when the processor is not dealing with any other routines. Thus by giving polled routines the lowest possible priority level of -1, they are easily distinguished from other, interrupt driven routines. It also ensures that polled routines cannot interrupt any active routines, as they always have a lower priority level.

For the example in Figure 3.2, the following occam tables were generated during the transformation process and passed to the supervisor process:-

```
VAL service.routines IS 3:

VAL Name.table IS ["temp.control     ",
                   "open.valve       ",
                   "close.valve      "]:

VAL Interrupt.table IS [ TRUE, FALSE, FALSE ]:

VAL Priority.table  IS [ 0, 0, 0 ]:
```

These tables effectively form a record structure for each routine. The entries in the first element of each table (subscript 0) refer to the 'temp.control' routine, the second set of entries refer to the 'open.valve' routine, and so on. Together, they provide the supervisor process with all the information necessary to enforce the ordering restrictions imposed on the processing of inputs, as stated by the system designer in the behavioural specification.

One feature of behavioural specifications which is neglected at this idealised level is the aspect of time. The simulation assumes that all input values can be processed within the specified time constraints, and that the sequence of inputs conforms to any sampling requirements for the application.

3.6. Conclusions

A behavioural specification language for embedded microprocessor control systems has been presented. Specifications written in this notation can be automatically transformed into a semantically equivalent form, expressed in the concurrent programming language occam. The occam representation can subsequently be compiled and executed, to give an implementation-independent simulation of the behaviour of the desired control system. A simple, generalised user interface is automatically incorporated into the simulation, through which the user can input test data and observe the results obtained. This removes the requirement for system-specific software test harnesses to be constructed for each system under design.

The main advantages perceived from adopting this approach over existing techniques are:-

1. the explicit, abstract definition of the interrupt, polling and priority structures required in a control system. General-purpose high-level languages do not provide this specific capability.

2. the specification of the sampling and performance requirements of a system at an abstract, implementation-independent level. Again, such facilities are not present in general-purpose languages.

3. it allows the specification of the precise strategies which are to be used to control the systems environment.

4. the automatic production of a software test harness, that is application specific, but constructed using an application independent mechanism. The test harness enables specifications to be thoroughly tested at an early stage of the system development.

5. the capability to automatically transform behavioural specifications into actual software implementations of the required control system. The abstract nature of the BSL opens up a large design space of possible implementations, especially in terms of hardware/software trade-offs, and the selection of which microprocessor to use to construct the system.

References

3.1 Wirth,N. :'Towards a Discipline of Real-time Programming', CACM, vol 20, no. 8, Aug 1977, pp 577-583

3.2 Zissos,D. : 'System Design with Microprocessors', Academic Press Inc, London, 1978

3.3 Johnson,C.D.:'Microprocessor-based Process Control",Prentice-Hall, INC., New Jersey, 1984

3.4 Ludewig,J.:'Computer-aided Specification of Process Control Systems', IEEE Computer, vol 15, no. 5, May 1982, pp 12-20

3.5 INMOS Ltd. :'occam 2 reference manual', Prentice-Hall 1988

3.6 Ambler,A. et al:'GYPSY - A Language for the Specification and Implementation of Verifiable Programs', Procs. Conf. on Language Design for Reliable Software, ACM SIGPLAN Notices, vol 12, no.3, 1977, pp 1-10

3.7 Suzuki,N.:'Concurrent Prolog as an Efficient VLSI Design Language', IEEE Computer, vol 18, no 2, Feb 1985, pp 33-39

3.8 German,S.J. and Lieberherr,K.J.:'Zeus: A Language for Expressing Algorithms in Hardware', IEEE Computer, vol 18, no 2, Feb 1985, pp 55-65

3.9 Hoare,C.A.R. : 'Communicating Sequential Processes', Prentice-Hall 1985

3.10 Roscoe,A.W. and Hoare,C.A.R.:'The Laws of occam Programming', Oxford University Computing Laboratory, PRG, Technical Monograph PRG53, Feb 1986

3.11 Ritchie,D.M. and Thompson,K.:'The UNIX Time-Sharing System', CACM, vol 17, no 7, July 1974, pp 365-375

3.12 Digital Equipment Corp.:'VAX Software Handbook', Maynard, Mass., 1982

3.13 Naur,P.(editor) :'Revised Report on the Algorithmic Language ALGOL 60', CACM, vol 6, no 1, Jan 1963, pp 1-17

3.14 Duce,D.A. and Fielding,E.V.C. :'Formal Specification - A Comparison of Two Techniques', The Computer Journal, vol 30, no 4, April 1987, pp 316-327

4. Generating Microprocessor Control Software from Behavioural Specifications

4.1. Introduction

This section describes a set of tools and techniques which enable behavioural specifications of MCSs (expressed in the notation described in the previous chapter) to be analysed and transformed into machine code implementations for target microprocessor systems. A machine-independent assembler level representation has been devised, into which behavioural specifications can be compiled. The generalised, processor-independent code produced by the compiler can be translated into the instruction set of the microprocessor which is eventually selected to implement the system. During the compilation process, information is extracted from the specification which is of relevance to the hardware design phase of the system development. Such information includes the number of input and output channels, how each input channel is to be driven, including any sampling details, the relative priority of each input channel, and any time constraints imposed on the processing of values from particular inputs. The structure of the whole scheme is illustrated in Figure 4.1.

4.2 Processor-Independent Assembly Language

4.2.1 Design Rationale

Previous attempts to define a portable, processor-independent assembly language have used the concept of an abstract machine[4.1,4.2]. This entails the definition of an imaginary processor architecture, in terms of registers, addressing modes and data manipulation operations. Given a high degree of generality in the abstract machine design, it is possible to implement the operations efficiently in the target machine code of an existing microprocessor. In order to achieve acceptable efficiency levels, abstract machine operations tend to be

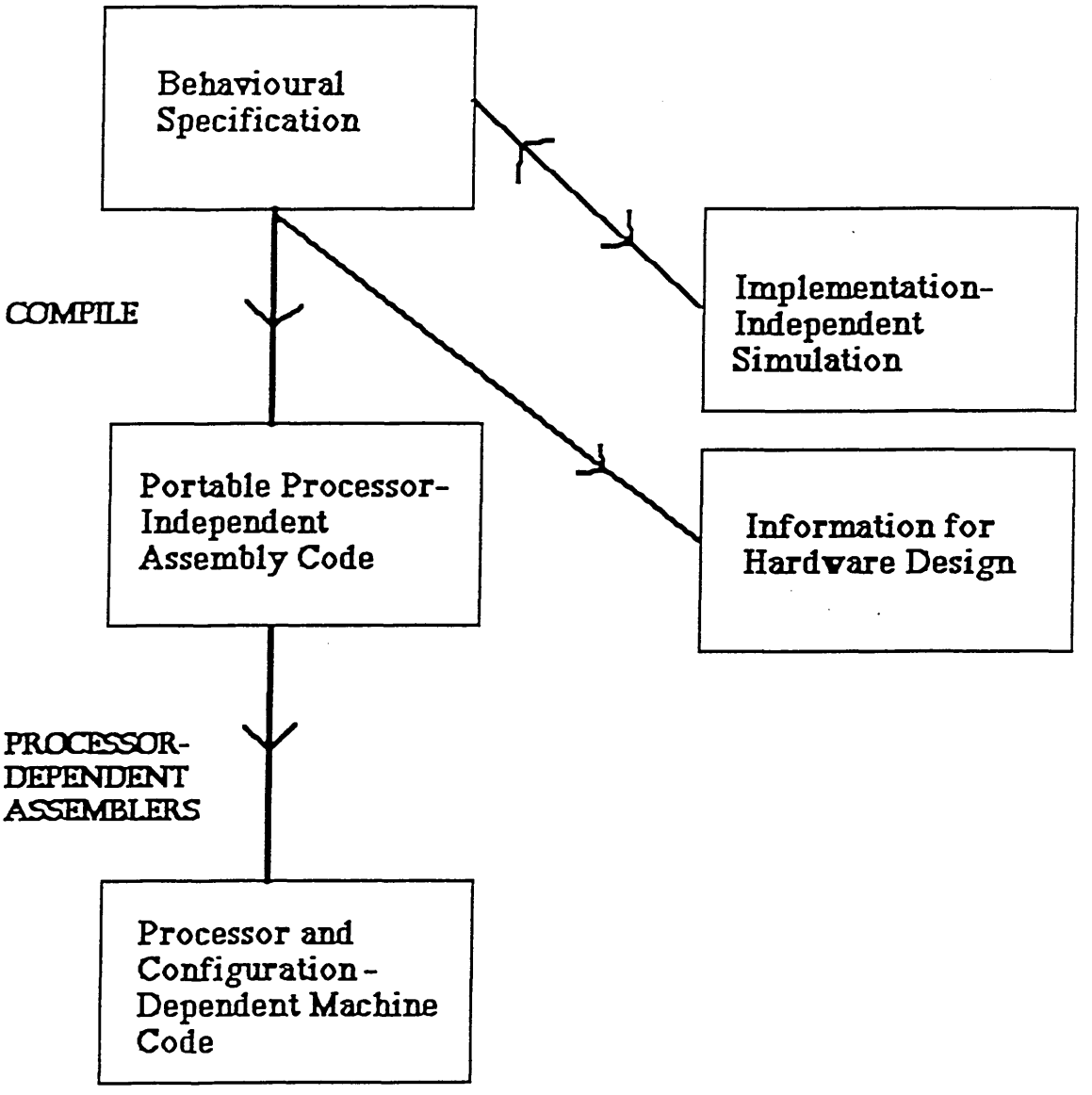


Figure 4.1 Transforming Behavioural Specifications into Processor-level Implementations

low-level, resembling the actual operations of their target microprocessor assembly languages. Consequently, the compilers used to translate high-level programs into abstract machine representations tend to be complex, while the process of generating target machine code is trivial[4.1].

However, as the internal architectures and operations of microprocessors vary greatly, it is extremely difficult to define a sufficiently general abstract machine architecture. Abstract operations which can be implemented by one machine instruction on one particular processor may take many more on another. It was therefore decided to define a higher-level, macro-assembly language, which does not rely upon any underlying architecture. This approach simplifies immensely the translation of the BSL into a portable assembly level representation, but it complicates the generation of the target processor machine code. Still, by the careful coding of the expansions of each macro operation, and the use of limited optimisation, relatively efficient object code can be produced for a wide range of microprocessors.

4.2.2 Language Features

4.2.2.1 Control Structures

The control structures included in the language closely resemble the control structures of the BSL. They are therefore simple to derive from the BSL description. Also, they can all be implemented very efficiently in any target machine code. The complete set of control structures is given below:-

1.
```
WHILE( operand, condition, operand)
        ---loop body-
ENDWHILE
```

2.
```
SEQ(operand, base, limit)
        ---loop body--
ENDSEQ
```

```
3.        IF
            (operand, condition, operand)
                  ---action-
            (operand, condition, operand)
                  ---action--
                  ------------
                  ---etc-----
                  ------------
          ENDIF

4.        CALL( subroutine-name)
          RETURN  -- returns from subroutine call
```

In addition, some simple instructions are required to give full control of interrupts and their arrival, and to enable time delays to be specified. These instructions are:-

```
INTON       -- enable interrupts

INTOFF      -- disable interrupts

WAIT        -- wait for an interrupt

POLL        -- poll an input line(s)

DELAY (n)   -- 'do nothing' for  n  time units
```

The INTON and INTOFF instructions are used to control the masking of interrupts. They are needed to implement procedures which are defined as uninterruptable in the specification. The WAIT instruction is required in situations when the system has no useful work to perform, and must wait for an indeterminate period until some external event occurs. A WAIT instruction is followed by input statements for all the possible events that may occur, and calls to the subroutines which deal with the events. POLL is identical to the WAIT instruction, except that it implies the processor is active, checking for inputs to arrive at all the enclosed input channels. The DELAY instruction enables the system to idle for a specified time period, expressed in microseconds, milliseconds or seconds. This is useful in situations when the sampling of input channels is to be implemented in software, or when the processor must wait a certain number of machine cycles for a valid value to be read from a channel.

4.2.2.2 Data Types

There are three data types provided, which implement the variable types of the BSL. These are:-

1. BYTE (8 bits)
2. INT (16 bits)
3. REAL (32 bits)

Note that the BOOL type of the BSL is not explicitly defined at this level. Boolean variables can more simply be represented as a byte value, which can only have the value 0, FALSE, or 1, TRUE. This method of handling Boolean variables is performed automatically by the BSL compiler.

A special set of macro operations facilitate the changing of a value from one variable type to another. These are required in most applications which apply mathematical functions to input values in order to derive the output value required. The operations, referred to as the Re-typing operations, are defined as:-

```
BTI  op1, op2    byte    (op2) to integer (op1)
ITR  op1, op2    integer (op2) to real    (op1)
ITB  op1, op2    integer (op2) to byte    (op1)
RTI  op1, op2    real    (op2) to integer (op1)
```

4.2.2.3 Operations

Macro operations have either one or two operands, with the exception of the INDEX operation, which has three. All of the two-operand instructions require that both operands must be of the same type. Some operations can manipulate operands of any of the three defined types, whereas others are restricted to a subset of these. Distinct opcodes for each individual operation are formed by prefixing an R, I, or B to the operation name to indicate the data type which is expected. For example, the general form of the ADD operation is :-

```
ADD operand1, operand2
```

This indicates that the value of operand2 is to be added to the value of operand1, leaving the result in operand1, and the value of operand2 unaltered. As the ADD operation can accept operands of any type, there are three ADD macro operations defined in the language; R-ADD, I-ADD and B-ADD. In the description of the operations which follows, only the general form is shown. The actual data types which each operation may accept are indicated in brackets following the general form.

The data manipulation operations can be divided into five categories:-

1. Assignment
2. Input-Output
3. Arithmetic
4. Logical
5. Index

There are only two operations in the assignment category. These are :-

```
ASSIGN  operand, constant      (R,I,B)
COPY    operand1, operand2     (R,I,B)
```

These define the load and store functions which are found in microprocessor instruction sets. COPY is a

general-purpose operation which can be used in any circumstance. ASSIGN is a special case of the COPY operation, and is only used when operand2 is a constant. Due to this particular characteristic, it is possible to implement ASSIGN more efficiently than COPY by using the immediate addressing mode of most microprocessors. The main use of ASSIGN is to initialise variables before they are used in calculations.

Input and output operations are defined simply as:-

```
INPUT   channel, operand       (B)
OUTPUT  channel, operand       (B)
```

The implementation of the I-O instructions varies greatly between microprocessors. Processors such as the Intel 8080 have special instructions to achieve I-O, whereas others, such as the Motorola 6800, use a memory-mapped I-O mechanism. Therefore, logical channel names are used in the I-O instructions at this level, leaving their eventual implementation, as ports or memory addresses, to be decided when the processor and system configuration are known.

The arithmetic operations encompass all the fundamental mathematical functions required to implement the BSL. Most of the operations exist in some form in microprocessor instructions sets. The arithmetic operations are:-

```
ADD op1, op2        (R,I,B)   -- add
SUB op1, op2        (R,I,B)   -- subtract
MUL op1, op2        (R,I,B)   -- multiply
DIV op1, op2        (R,I,B)   -- divide
REM op1, op2        (I,B)     -- remainder
INC op1             (I,B)     -- increment
DEC op1             (I,B)     -- decrement
```

INC and DEC are special cases of the ADD and SUB operations. Their inclusion enables a more efficient implementation of common occurrences such as loop counters and array indexing.

The logical operations provide bit-manipulation facilities to implement the logical functions of the BSL. The equivalent of these operations can be found in any microprocessor. They can therefore be implemented efficiently, with the minimum of effort. The operations are:-

```
AND op1, op2        (I,B)     -- logical and
OR  op1, op2        (I,B)     -- logical or
NOT op1             (I,B)     -- 1s complement
NEG op1             (I,B)     -- 2s complement
XOR op1, op2        (I,B)     -- exclusive or
SLL op1             (I,B)     -- shift left
SRL op1             (I,B)     -- shift right
```

Finally an operation to access one-dimensional arrays has been defined. Given the base address of the array and the

position of the element required, the operation extracts the value from the array and stores it in a temporary variable. The format of the operation is:-

INDEX base, position, variable (R,I,B)

So, for example, in the operation

INDEX table, 3, temp1

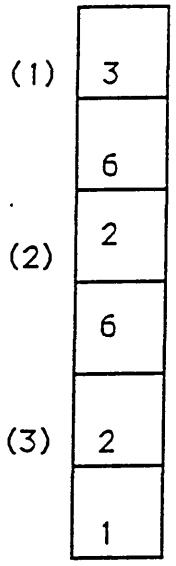
where table is an integer array, the third element of the array would be stored in the variable temp1. Note that the position of the element in the array is expressed in logical units, not a byte offset. It is the task of the implementation to calculate the exact position of the data in the array. Thus, in the above example, although the third element of the table is specified in the INDEX operation, the value of this element actually resides in bytes five and six of the array. This is illustrated in Figure 4.2.

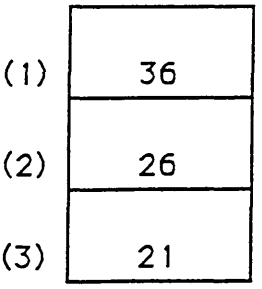
It is worth noting that in the definition of the macro operations, no mention is made of the effects of each operation on the condition codes which are present in microprocessors. The most important use of condition codes is when a program performs a test or comparison, and then inspects the appropriate condition code bit to decide whether to execute a jump instruction. This is the technique used to program repetition and selection constructs in machine code. However, low-level test and jump operations are not included in the processor-independent language. Rather, repetition and selection constructs are specified at a much higher level. This has the advantage of allowing the details of the condition code manipulation to be hidden in the implementation of the abstract control structures.

4.3. Compiling the BSL into Macro-Assembly Language

4.3.1 Overview

Figure 4.3 shows the macro-assembler code which results from compiling the specification in Figure 3.2. In this





Logical Table Representation

Actual Table Representation

Figure 4.2

simple example the two are not radically different. This is because the specification does not include any calculations or expression manipulation. It is essentially constructed from input, output and selection statements, which tend to compile on a one-to-one basis at this level. The main aspect to note is the removal of the more sophisticated features such as parameters and local variables from the BSL representation. This is because in general existing assembly languages for microprocessors do not include these features. All variables in a program are global, and thus no parameter passing is needed.

The BSL compiler operates in three passes. During the first pass, the channels and variables used in the specification are written to the macro-assembler code file. The second pass produces the main control logic, which states exactly how each input channel is to be handled. Finally, the third pass produces the code for each of the service routines in the system. This structure for the output file can be clearly seen in Figure 4.3. This format has been adopted because it adheres closely to the code format required by most existing microprocessor assembly languages. It therefore makes the subsequent translation of the macro-assembler code into actual machine code a much more straightforward process. There follows a brief description of the tasks performed by each pass of the compiler. The description focuses upon the aspects of the compilation process which are particular to this specific system. A more comprehensive report, which also states the current implementation restrictions, can be found in [4.3].

4.3.2 Pass One

The most important task performed by the first pass of the BSL compiler is to construct the symbol tables required to perform complete syntax checking of the specification. Much of the syntax is assumed to be correct, as most violations of the syntax rules will have been discovered when the specification is transformed into an executable form. All

```
CHAN temperature.measure
CHAN heater
CHAN low.level
CHAN high.level
CHAN valve

BYTE temp
BYTE low
BYTE high

INTON
WHILE TRUE
WAIT
  INPUT temperature.measure, temp
    CALL temperature.control
  INPUT low.level, low
    CALL open.valve
  INPUT high.level, high
    CALL close.valve
ENDWHILE
END

temperature.control:
INTON
IF
  temp, >, 90
    OUTPUT heater, 0
  temp, >, 80
    OUTPUT heater, 1
  temp, <=, 80
    OUTPUT heater, 2
ENDIF
RETURN

open.valve:
INTOFF
OUTPUT valve, 1
INTON
RETURN

close.valve:
INTOFF
OUTPUT valve, 0
INTON
RETURN
```

Figure 4.3 Macro-Assembler Code for the Water Tank Controller

the variables, channels and service routine identifiers are located during this pass, and stored for later reference in the appropriate symbol table. Further, all the constants declared in the specification are placed in a symbol table, together with the value that they represent. This enables the actual values of constants to be generated in the macro-assembler code. For this reason, the macro-assembler language does not include a method for defining symbolic constants. Finally, the output produced from the first pass is simply the macro-assembler definitions for each of the channels and variables used in the specification.

4.3.3 Pass Two

The second pass of the compiler is concerned with analysing and producing code for the control section of the specification. This acts as the main control logic for the macro-assembler code. Each input statement in the BSL is replaced by a macro-assembler INPUT statement, and each BSL output statement is replaced by a CALL statement. If the specification is interrupt-driven, a macro-assembler WAIT statement is generated. For polled channels, a POLL statement is issued. An END instruction is placed at the end of the control block to signify the extent of its scope.

4.3.4 Pass Three

The third pass is responsible for compiling the service routines which appear in a specification. This is the most complex phase of the compilation process. It involves analysing the expressions which make up a service routine, and generating the equivalent macro-assembler code. When a service routine header is found, it is replaced by a label, identical to its name in the specification. If a routine is interruptable, a INTON instruction is generated before the rest of the routine is analysed: for routines which may not be interrupted, an INTOFF instruction is generated. All RESULT statements found in a routine are replaced by an

output statement. The end of a routine is marked by a RETURN instruction. However, for uninterruptable routines, an INTON instruction is generated before the RETURN.

4.4 The Implementation of Macro Instructions

4.4.1 Data Types

The three variable types, BYTE, INT and REAL, defined in the macro-assembly language are implemented simply as one, two and four bytes respectively. All arithmetic operations are assumed to use values in signed twos-complement format. Thus the BYTE data type can be implemented directly by most 8-bit microprocessors, such as the 6800[4.4] and the 6502[4.5]. These processors are specifically designed to perform twos-complement arithmetic, using the left-hand, most significant bit to represent the sign of the variable. (see Figure 4.4). Flags in the condition code register of the 6800 and 6502 are automatically set when the result of an operation is negative, or overflow occurs. However, microprocessors such as the Intel 8080[4.6] have no way of indicating the sign of the result of an arithmetic or logic operation. Therefore an 8080 implementation of the macro-assembly language requires additional software and execution time, in order to 'remember' the sign of each variable in use[4.7]. The BYTE data type can represent values in the range +((2E7)-1) to -(2E8).

The INT data type is implemented as a double-precision 16-bit value. Two consecutive 8-bit locations are allocated to store integers; the address of the variable always gives the most significant, or high order byte. The low order byte is stored at the next memory location, given by adding one to the address of the high order byte (see Figure 4.5). The manipulation of INTs is consequently less efficient than that of BYTEs, as it essentially requires operations to be performed on each of the two bytes which constitute the value. For example, the addition of two INTs firstly requires the addition of the two low order bytes, followed

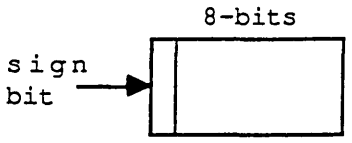


Figure 4.4 BYTE Representation

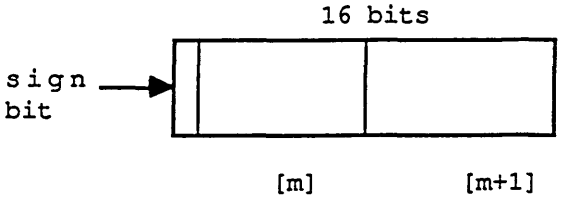


Figure 4.5 INTEGER Representation

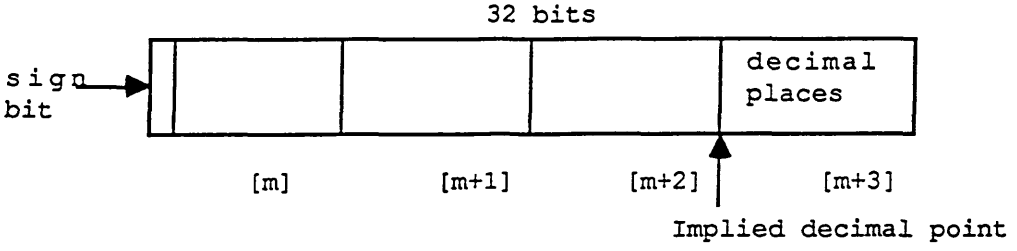


Figure 4.6 REAL Representation

by the addition of the two high order bytes, including any overflow from the initial addition. Overflow is indicated by the microprocessor's carry flag. The addition of the high order bytes is thus carried out using an 'add with carry' instruction, or by simulating such an instruction if one does not exist in the instruction set. An example 6800 implementation of the I-ADD macro instruction is given below:-

```
      I-ADD op1, op2        op1 := op1 + op2

I-ADD LDA A  op1+1          load low byte of op1
      ADD A  op2+1          add low bytes in A reg
      STA A  op1+1          store result in op1 low byte
      LDA A  op1            load high byte of op1
      ADC A  op2            add high bytes and carry flag
      STA A  op1            store result in op1 high byte
```

Many microprocessors give performance advantages for storing variables in the first 256 bytes of memory, often referred to as the zero page. The reason for this is that the address of a variable residing in this area can be specified by a single byte in a memory reference instruction. The variable address can therefore be fetched in one memory reference, instead of the two references which are required for all addresses greater than 255. Consequently, the translation process always tries to allocate variables to the zero page of memory. Some microprocessors however, such as the 6502, automatically allocate the stack area to the zero page. While this makes subroutine calls more efficient, it limits the segment of the zero page which may be safely used for variable storage.

REAL data types are allocated four consecutive bytes of memory, with the address of the variable giving the most significant byte. This is illustrated in Figure 4.6. The three high order bytes are used to store the integer portion of the real number, with the low order byte containing the fractional part. This gives an integer range of +(2 E 23)-1 to -(2 E 24), and a maximum precision of six decimal places. This implementation of REALs was chosen for

its simplicity, and efficiency of manipulation. Operations on REALs can be carried out using the same technique that is used for integer operations, the only difference being that the operations function on four bytes as opposed to two. For most MCS applications the range and accuracy provided by this representation will be easily adequate. Should an application need a greater range or more precision, the REAL operations could be redefined to use the more common three byte mantissa and one byte exponent floating point representation[4.8]. This however would be at the expense of considerably greater software complexity, size and execution time.

4.4.2 Operations

Most example operations given in this section are in Motorola 6800 assembly language[4.8]. Other 8-bit language examples are shown only when they offer considerable advantages or disadvantages over the 6800 implementation. However a comparison of the efficiency of the different implementations is not attempted, as this is dependent upon the precise clock rate of the processor used, and the amount of work a processor performs during each clock cycle. To illustrate this difficulty, consider the 8080 'load high and low direct' instruction and the 6800 'load index extended' instruction. Both perform the same function; they load a register with a 16-bit value from memory. The 8080 requires sixteen clock cycles against the five cycles needed for the 6800 to execute the instruction. However, a 1 MHz 6800 processor executes the instruction in two-thirds the time required by a 2 MHz 8080 component. Thus the clock speed is not by itself a valid indication of performance between different microprocessors[4.7].

The assignment operations can be implemented in 6800 assembler merely by using the accumulator load and store instructions. The constant value in the B-ASSIGN operation is always expressed as a two-digit hexadecimal number. The examples in this section use the hexadecimal digits 'F' and

'E' to represent arbitrary values. This is illustrated in the implementation of the B-ASSIGN operation given below:-

```
B-ASSIGN  op1, FF        op1 := FF

B-ASSIGN  LDA A #$FF     load hex constant
          STA A op1      store in memory
```

The allocation of a 16-bit value to an integer variable, the I-ASSIGN operation, can be implemented in much the same manner as above, except that the 16-bit index register can be utilised to make the operation more efficient. The constant value is this time expressed as a four-digit hexadecimal number.

```
I-ASSIGN  op1, FFFF      op1 := FFFF

I-ASSIGN  LDX  #$FFFF    load index reg
          STX  op1       store in memory
```

Assigning values to REAL variables is performed by duplicating the I-ASSIGN operation, so that the eight-digit constant value is transferred into the 4-bytes reserved for the variable. The decimal point in the constant is not explicitly shown.

```
R-ASSIGN op1, FFFFEEEE   op1 := FFFFEEEE

R-ASSIGN LDX  #$FFFF     load high order bytes
         STX  op1        store in memory
         LDX  #$EEEE     load low order bytes
         STX  op1+2      store in memory
```

The implementation of the COPY operation is less efficient than ASSIGN. This is because the value of the second operand has to be fetched from memory, whereas in the ASSIGN operation the value of the second operand is available in the operation itself. Implementations of COPY are shown below.

```
COPY      op1, op2       op1 := op2

B-COPY    LDA A op2
          STA A op1

I-COPY    LDX op2
          STX op1

R-COPY    LDX op2
          STX op1
          LDX op2+2
          STX op1+2
```

Table 4.1 shows the number of clock cycles required to perform each of the assignment operations on a 6800 microprocessor. This information shows the relative efficiency of the operations. Assuming that a 1 MHz 6800 component is used to execute the software, the number of clock cycles stated in the table also represents the time in microseconds for each operation to execute.

macro operation	operands located in zero page (cycles)	operands located outside zero page (cycles)
B-ASSIGN	6	7
I-ASSIGN	8	9
R-ASSIGN	16	18
B-COPY	7	9
I-COPY	9	11
R-COPY	18	22

TABLE 4.1 Comparison of Execution Times For Assignment Operations on a 6800

Table 4.1 clearly shows the performance benefits accrued from locating variables in the zero page of memory. Further, it highlights the cost in terms of efficiency of using REAL variables as opposed to INTs and BYTEs.

As the 6800 uses a memory-mapped I-O scheme, the implementation of the INPUT and OUTPUT operations is very similar to COPY. The channel name in the operation represents a hexadecimal address at which an input or output port is located. For example,

```
INPUT     channel, operand
```

can be implemented as

```
INPUT     LDA A channel   input from port
          STA A operand   store value
```

In the same manner,

```
          OUTPUT    channel, operand
```

becomes

```
          OUTPUT    LDA A operand   load value
                    STA A channel   output to port
```

However, an implementation of these operations on an Intel 8080 would require the channel name to represent an 8-bit I-O port identifier. The actual I-O task is then performed by the 8080's special purpose IN and OUT instructions (see below)

```
INPUT     IN  channel
          STA operand

OUTPUT    LDA operand
          OUT channel
```

It should be noted that the 8080 does not possess a single byte, direct addressing mode such as that included in the 6800 or 6502. The LDA and STA instructions in the above 8080 examples both require 16-bit addresses as operands. Consequently an 8080 implementation of the macro assembly language would not benefit from allocating variables to low addresses in memory.

The implementation of the arithmetic operations is straightforward, but can vary greatly between processors. This is clearly illustrated by considering the I-ADD operation. An implementation of I-ADD in 6800 assembler has been given earlier in this section. It essentially comprises two eight-bit add instructions, the second of which incorporates the carry bit. However, the 8080 has a 16-bit register pair add instruction, which can be utilised to perform the I-ADD operation. This is illustrated below:-

```
I-ADD     op1, op2    op1 := op1 + op2

I-ADD     LHLD op2    HL := op2
          XCHG        DE := HL
          LHDL op1    HL := op1
          DAD  D      HL := HL + DE
          SHLD op1    op1:= HL
```

Multiply, divide and remainder operations can be simply implemented by the successive application of the appropriate ADD or SUB operation. The other single-byte operations can all be constructed from simple, native instructions of the microprocessors in question. Implementations in 6800 assembler for each operation are given in Appendix C.

The group of logical operations can also be easily and efficiently implemented in 8-bit assembly languages. For example, consider the following 6800 implementation of the I-AND operation:-

```
I-AND     op1, op2      op1 := op1 AND op2

I-AND     LDA A op1+1   load low order byte
          AND A op2+1   'And' low bytes
          STA A op1+1   store result
          LDA A op1     load high order byte
          AND A op2     'And' high bytes
          STA A op1     store result
```

The major difficulty in implementing the re-typing instructions occurs in preserving the sign of a value as it alters from one type to another. For example, to convert a value held in a BYTE variable into an integer, the byte-to-integer instruction, as shown below, must be used.

```
BTI op1, op2          op1 := INT op2
```

The exact behaviour of this operation is described by the following algorithm:-

```
if op2 < 0 then
   op1 [high byte] := #FF
else
   op1 [high byte] := #00
endif
op1 [low byte ] := op2
```

Essentially the operation tests the sign of 'op2'. If it is negative, it is preserved in the twos-complement value of 'op1' by setting its high order byte to #FF: if 'op2' is positive (or zero), it sets the high order byte of 'op1' to zeros. The implementation of the other re-types operations can be performed in a similar manner, as shown in Appendix C. Note however that the real-to-integer and integer-to-byte operations do not take any precautions for dealing with potential overflows. The detection and possible correction of overflow must be explicitly included in the behavioural specification, as recovery strategies to deal with overflows will vary according to the application.

The implementation of the INDEX operation is most concisely expressed in terms of other, previously defined macro operations. The operation

```
I-INDEX   table, element, temp1
```

can thus be defined as:-

```
I-INDEX   I-ASSIGN disp, 0002      disp :=2
          I-MUL    disp, element
                       disp := disp * element
          I-DEC    disp
                       disp := disp - 1
          I-ADD    disp, table
                       disp := disp + table
          I-COPY   temp1, disp
                       temp1:=table[element]
```

This operation functions by calculating the displacement of the required element in the table, and copying this element to the variable specified as the third operand. However, a simple macro expansion of this definition would contain several redundant load and store instructions for the value of the variable 'disp', which could be kept in a register throughout. Hence the actual implementation, though remaining semantically equivalent, has been optimised by the removal of these redundant instructions.

The precise reason for the presence of such inefficiencies lies in the fact that the implementation of each macro instruction must be self-contained. That is, no assumptions

are made as to the contents of the processor's registers between instructions. The consequence of this is that each instruction has to load its operands into registers before performing any manipulation on them. When the function of the instruction is complete, the operands must be stored back into memory, where they can be found by subsequent instructions. Thus this macro expansion approach offers many opportunities for simple but effective optimisation strategies to be performed on the object code.

4.4.3 Control Structures

The major complications encountered in implementing the macro control constructs lie in the definition of comparison operations. These are needed to evaluate conditions that appear in the iteration (WHILE, SEQ) and selection (IF) constructs. Six comparison operators are valid in behavioural specifications. Therefore an implementation for each of these must be provided. Further the implementations must be able to compare values for each of the three valid types in the macro-assembly language. The six comparison operators are:-

EQ	equal
NE	not equal
LE	less than or equal
GE	greater than or equal
LT	less than
GT	greater than

To give a simple illustration of the use of one of the comparison operators, consider the segment of macro-assembler code below.

```
INT op1
INT op2
WHILE ( op1 < op2)

  -- loop body --

ENDWHILE
```

In order to implement the WHILE loop in 6800 assembler, a macro needs to be defined which compares two 16-bit values,

and branches according to the result of the comparison. If the condition under test is false (ie op1 >= op2), a branch must occur to the end of the loop. However, if the condition is satisfied, no branch occurs, and the next operation in sequence is performed. Such a comparison operation can be defined as

```
I-LT  op1, op2, addr
```

This operation behaves according to the following algorithm:-

```
if op1 < op2 then
    SKIP
else
    goto addr
endif
```

and can be implemented in 6800 assembler as

```
LDX  op1     load op1 into index register
CPX  op2     compare op1 and op2
BEQ  addr    if equal goto addr
BGT  addr    if op1 > op2 goto addr
```

This macro can now be utilised to perform the required comparison in the WHILE loop of the above example. eg

```
WHILE     LDX op1
          CPX op2
          BEQ ENDWHILE
          BGT ENDWHILE

            -- loop body --

          BRA WHILE
ENDWHILE  NOP
```

Implementations of all the comparison operators for BYTEs and INTs can be formed in a similar fashion. The comparison of REALs though is more complex, due to the lack of a 32-bit compare instruction in 8-bit processors. Using a 6800, a comparison of two real values essentially involves examining each constituent byte in turn, starting with the most significant, and remembering the result of each comparison. Complications arise when leading zeros appear in both operands. Consequently the macros must check for these and ignore them. Full details of the 6800

implementations for the comparison operations are given in Appendix C.

4.5 Processing the Macro Assembly Language

To generate processor-specific assembler code from an intermediate macro representation, a macro expansion routine is required. Implementations of each of the macro operations are stored in the macro expansion program. When an operation is recognised in the input file, it is replaced by its assembler implementation, with the correct arguments substituted in, and written to the output file.

In all the above examples of the two-operand macro operations, the second operand has been assumed to be stored in a memory location. This though is not always the case. Where constants are defined and used in the BSL, they are included in the macro-assembler operations as absolute hexadecimal values. In this way, the need for constants to occupy memory locations is obviated. This has the consequence that the second operand in a macro operation may in fact be a hexadecimal number. In order to deal with this eventuality, the macro expansion routine checks the nature of second operand of an operation, and outputs the correct assembler code. When the second operand is a number, more efficient code is generated using ASSIGN, because the value of the second operand is already present in the instruction. This dispenses with the requirement to read the value from memory, enabling a less time-consuming addressing mode to be utilised. The macro expansion routine implements this behaviour simply by storing two possible implementations for each operation, and outputting the correct one as required.

However, the processor-independent code does not contain all the details necessary to produce a complete software solution for the application. The intermediate code is still only a representation of the desired behaviour of the application. It therefore needs supplementing by structural

information concerning the hardware on which the control system is to eventually execute. Specifically, this means disclosing such facts as the model of the target microprocessor, the address map for the configuration, and the mapping of logical channels on to the processor's interrupt lines. The results of any hardware/software design trade-offs must be reported, as, for example, the macro expansion routines must know whether or not it is required to generate code to handle interrupt priorities or time delays. Also, details of the initialisation sequences for each I-O port need supplying, as well as any unusual details about the precise manner of accessing each port. Figure 4.7 provides a diagrammatical representation of this scheme.

Returning to the water tank controller example of the previous chapter, Figure 4.8 shows the 6800 assembler code that is generated from the processor-independent code in Figure 4.3. In this example, only additional information about the address map of the configuration has been supplied. Consequently the code in Figure 4.8 is not complete. The initialisation of the I-O ports and determination of which external device caused an interrupt still need to be included. However, these functions are relatively simple and require only a small number of machine instructions to implement. In almost all cases the interrupt service routines will comprise the bulk of the machine code for an application. The automatic provision of the additional structural information is the task of the knowledge-based system which performs the overall hardware design. While this is under construction however, the hardware design is performed manually and supplied interactively to the current implementation.

The macro expansion technique used to produce the 6800 code in Figure 4.8 has introduced some inefficiency. This is most apparent in the 'temperature.control' ('tempco') routine. The temperature value is read from the input port (tempem) into register A, stored in memory (at location

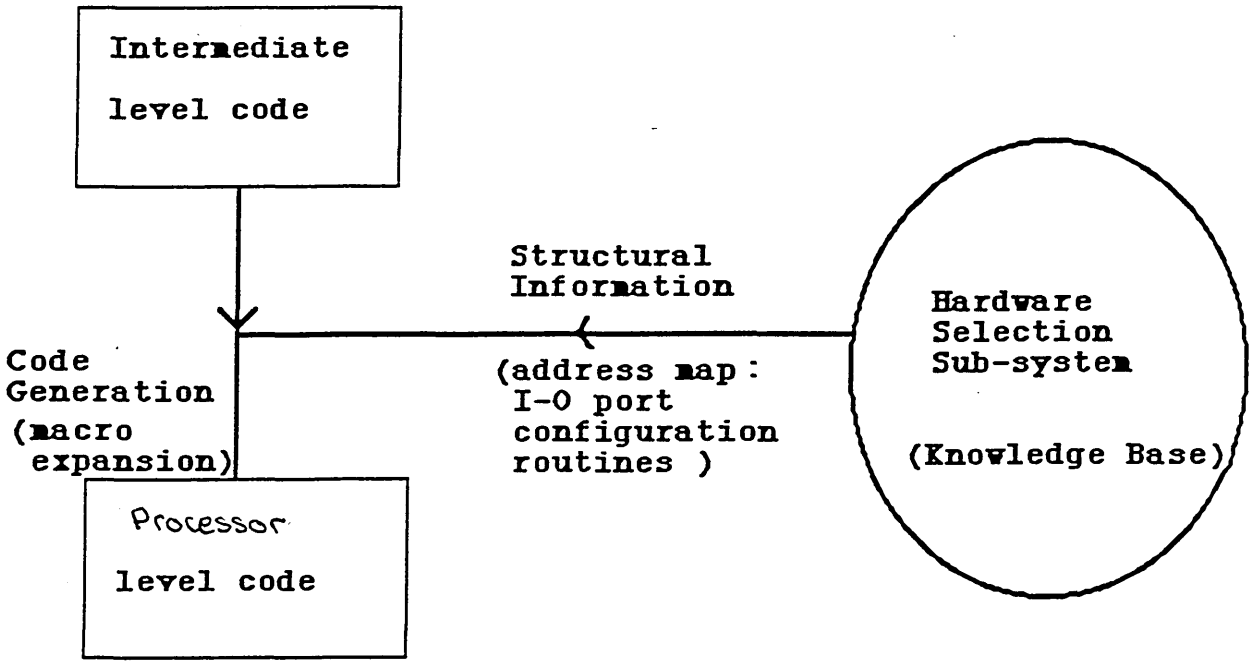


Figure 4.7 Processing the Intermediate Level Macro Assembler Code

```
*           (example) port addresses
TEMPME      EQU  $4000          temperature.measure
HEATER      EQU  $4002          heater
LOWLEV      EQU  $4004          low.level
HIGHLE      EQU  $4006          high.level
VALVE       EQU  $4008          valve
*           variables
            ORG  $0
TEMP        FCB  0              temp
LOW         FCB  0              low
HIGH        FCB  0              high
            ORG  $FC00
*           main program
BEGIN       CLI
            LDS  #00F0          initialise stack pointer
LAB1        WAIT                wait for an interrupt
            JMP  LAB1
*           temperature.control
TEMPCO      CLI                 interruptable
            LDA A TEMPME        input temperature level
            STA A TEMP
TEST1       LDA A TEMP          IF
            CMP A #90             temp > 90
            BLE TEST2
            LDA A #0                output heater, 0
            STA A HEATER
            BRA ENDIF1
TEST2       LDA A TEMP
            CMP A #80             temp > 80
            BLE TEST3
            LDA A #1                output heater, 1
            STA A HEATER
            BRA ENDIF1
TEST3       LDA A TEMP
            CMP A #80             temp <= 80
            BHI ENDIF1
            LDA A #2                output heater, 2
            STA A HEATER
            BRA ENDIF1
ENDIF1      NOP                 ENDIF
            RTI                 return
*           open.valve
OPENVA      SEI                 uninterruptable
            LDA A #1            output valve, 1
            STA A VALVE
            CLI                 enable interrupts
            RTI                 return
*           close.valve
CLOSEV      SEI                 uninterruptable
            LDA A #0            output valve, 1
            STA A VALVE
            CLI                 enable interrupts
            RTI                 return
            END
            MON                 end of program
```

Figure 4.8 6800 Assembler Code for the Water Tank Controller

'temp'), and then immediately loaded back into register A. In hand-written code, the intermediate store instruction would not be included. Obviously therefore, there is some scope for optimisation of the code produced by the macro expansion technique.

4.6 Conclusions

An intermediate level macro-assembly language has been described into which behavioural specifications of microprocessor control systems may be compiled. Although the macro operations which constitute the language may be implemented in any assembly language, the efficiency of the each implementation varies. In the case of 8-bit components, it seems that the Motorola 6800 and other similar microprocessors are well-suited and provide an efficient implementation. This is mainly due to the presence of a one-byte direct addressing mode and twos-complement arithmetic. Processors such as the Intel 8080 do not have these features, and thus provide a less than optimum implementation language. Criteria such as these may be of importance to the overall design of a control system, and thus may be regarded as of major influence in the choice of processor in applications where efficiency of operation is vital.

References

4.1 Saxena,S. and Field,J.A.:'Portable Real-Time Software for 8-bit Microprocessors', Software - Practice and Experience, 15, (3), 277-303 (1985)

4.2 Bowles,K.L.:'UCSD Pascal: a (nearly) machine independent software system (for microcomputers and minicomputers)',BYTE, 3, (5), 170-173 (1978)

4.3 Olewicz,T. :'A Behavioural Specification Language (BSL) Compiler', MSc Dissertation, Dept. of Computer Studies, Sheffield City Polytechnic, Sept 1988

4.4 Bishop,R.:'Basic Microprocessors and the 6800', Editions Mengis, 1981

4.5 Tully,A. :'6502 Reference Guide', Melbourne House Ltd, London 1985

4.6 Larsen,D.G., Titus,J.A. and Titus,C.A. :'8080 8085 Software Design', Howard W. Sams & Co., Inc., Indianapolis, 1981

4.7 Artwick,B.A. :'Microcomputer Interfacing', Prentice-Hall, Inc., 1980

4.8 Findlay,R. :'6800 Software Gourmet Guide and Cookbook', Hayden Book Company, Inc., New Jersey, 1976

5 The Design and Construction of the Microprocessor Simulation Facility

5.1 System Design

The simulation of a complete (hardware and software) microprocessor system design provides the most precise technique available for design verification[5.1,5.2]. Many general-purpose digital logic simulators exist[5.3]. However, most simulators operate at register-transfer level[5.4], and are not capable of effectively simulating a microprocessor system in terms of interactions between components. There are simulators which perform simulations of user programs[5.5], but these do not attempt to accurately model the timings or signal interface of the microprocessor concerned.

Still, there are logic simulators which can simulate arbitrary microprocessor system configurations[5.6,5.7]. Nearly all of these are implemented using a sequential programming language (e.g. FORTRAN) on a single-processor machine, with the component-level parallelism present in microprocessor systems simulated. For these reasons, most simulators suffer from slow execution speeds. This makes it impractical in real-time applications to consider interfacing the system simulation with the physical environment to be controlled. To thoroughly and efficiently test a real-time control system with its environment, in-circuit emulation must be used[5.2,5.8]. Unfortunately, this technique requires at least a minimum investment in hardware, and is therefore unsuitable for verifying the design of a system before any implementation begins.

Consequently, a major design objective of the simulation facility was to investigate the feasibility of constructing a fast ('almost' real-time) and accurate environment for simulating embedded, single-board control systems. Simulations could then be connected to the actual inputs and outputs of the physical system. This would enable the

design to be tested, and where necessary modified, without any prior commitment to hardware. In an attempt to achieve this aim, the parallel programming language occam[5.9] was chosen as the hardware description and simulation language. It is known that occam is a suitable tool for hardware description[5.10,5.11,5.12], and it was in fact used by INMOS in the design of the transputer[5.13]. Occam contains constructs to express concurrent behaviour in programs, and these provide a simple and natural way of simulating the parallelism present in digital systems. Simulations constructed using occam may also be divided into appropriate sections and mapped on to a network of transputers. In this way, the parallelism in the simulation could be exploited to give a significant increase in performance[5.14].

5.2 Component Simulation

An individual component simulation must model exactly the behaviour of the actual hardware component being described. An LSI component can be regarded as a black box with internal state, which communicates in a synchronised manner with other components via a common system bus. Similarly, an occam process can be regarded as a black box with internal state, communicating with other processes via point to point channels. Hence it appears that occam processes provide a natural method of representing hardware components[5.15,5.16]. The channel interface of a component simulation process can be used to model the physical wires which connect the component to the system bus. Then, by constructing a suitable bus simulation, components can simply be 'plugged in' to form a simulation of any desired MCS.

It was decided initially only to simulate components from the Motorola 6800 family[5.17]. 6800-based systems have been widely used in dedicated control tasks[5.18], and allow minimum systems to be constructed from a small number of components[5.2].

At the highest level of abstraction, component simulations consist of two separate occam processes. One of these acts purely as an interface to the system bus simulation, the other implements the particular function of the component. This scheme is illustrated in Figure 5.1, in which it is applied to an MC6821 Peripheral Interface Adapter (PIA). It is interesting to note that a similar approach is used in the ISPS[5.19] and VHDL[5.20] hardware description languages.

```
PROC MC6821 (CHAN OF ANY address.bus, data.bus.in,
                         data.bus.out, control.bus)
  ... process declarations
  CHAN OF ANY to.device, from.device:
  PAR
    interface( address.bus, data.bus.in,
               data.bus.out, control.bus,
               to.device, from.device   )
    PIA.body ( to.device, from.device   )
:
```

In this example, the interface inputs the address and control information from the bus channels at the start of each clock cycle and decodes the address. If this particular PIA is enabled, the interface process initiates communication with the PIA body process. Together, the two processes will simulate the actions required of the PIA, the exact behaviour being determined by the values sent on the control bus by the microprocessor (e.g. read/write). Conversely, if this PIA process is not addressed during a clock cycle, all subsequent bus signals are ignored by the interface until the start of the next cycle.

The reasons for adopting this strategy are threefold. Firstly, it enables a standard interface process to be constructed for all similar devices. For example, all memory and input-output components will have virtually identical interfaces. Differences will only occur when a component requires an extra channel and some processing to, say, generate interrupts.

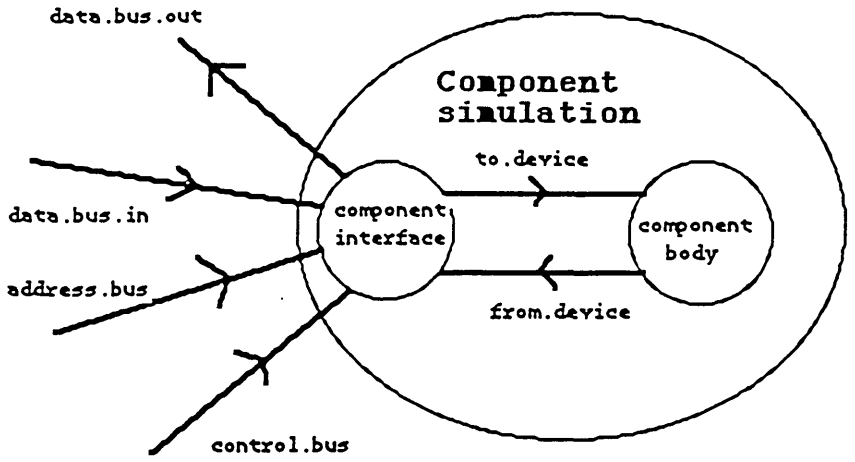


Figure 5.1 Structure of a Component Simulation

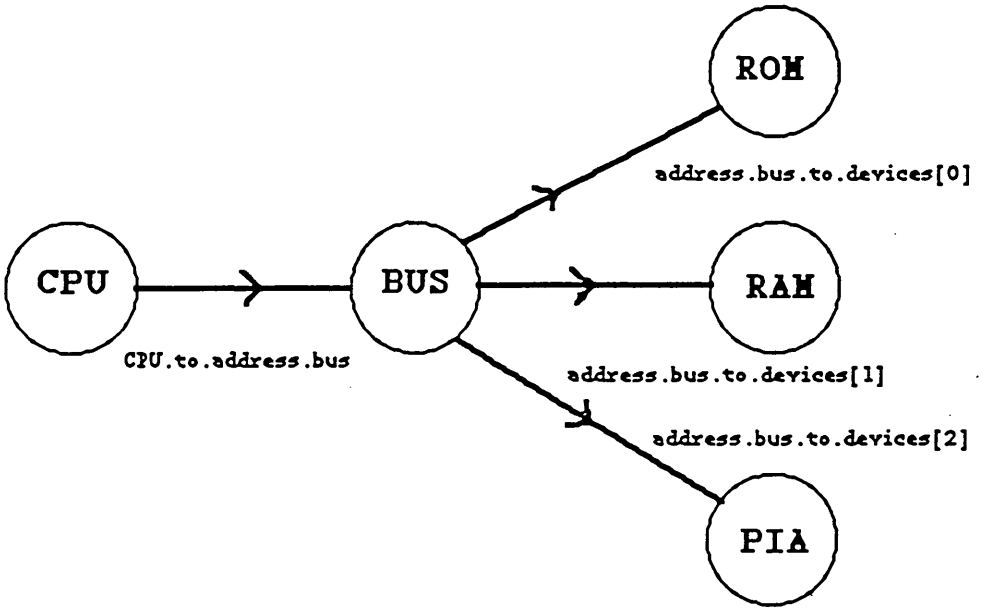


Figure 5.2 Occam Process Structure for the Address Bus

Second, if changes were made to the system bus simulation, only the component interfaces would require changing. The bulk of the code which describes a component resides in the device process, the interface generally being small in comparison. Therefore changes in the bus structure would only require fairly simple additions to the interface, leaving the majority of the code unaltered. This situation may arise when, for instance, components from the 6800 family need to be interfaced to an Intel 8080 microprocessor[5.21].

Third, this approach enables the design of a component simulation to be decomposed into a hierarchy, with parallelism expressed at each level of the hierarchy. LSI components consist of many sub-components operating in parallel, each of which may consist of many other sub-components, and so on. Clearly then, to effectively model such devices, a language is required which allows systems to be described as a hierarchy, with possibly many interacting concurrent processes at each level of the hierarchy. Occam is such a language, allowing component simulations to be constructed in a step-wise, hierarchical manner. This reduces the complexity of the programming task and increases readability through the use of abstraction.

A further advantage of occam is that it allows multiple instantiations of the same process definition to execute simultaneously. The different instantiations are differentiated by the values of their actual parameters. This can be illustrated by considering the internal characteristics of the MC6821 PIA. It consists of two 8-bit ports, each with two control lines. The function of each port is identical, with the same internal registers and control commands. The behaviour of the PIA may therefore be modelled by two identical processes operating in parallel. In occam however, there is no need to create and explicitly name two separate processes. Only one process needs to be constructed, which models the behaviour of the port. This single process may then be instantiated the required number

of times within a PAR statement. This feature of occam is shown in the example below :-

```
PROC PIA.body (CHAN OF ANY from.interface,
                          to.interface)
  ... PROC port(CHAN OF ANY in, out)
  CHAN OF ANY PortA.in, PortA.out, PortB.in,
              PortB.out:
  PAR
    Port ( PortA.in, PortA.out )
    Port ( PortB.in, PortB.out )
:
```

5.3 System Bus Simulation

5.3.1 Requirements

In Motorola 6800 systems, memory and input-output chips are connected to the central processing unit (CPU) via a series of wires. Essentially these wires fall into 3 categories: the address bus, the data bus and a set of control wires. The address bus comprises sixteen wires and carries the address of the location in memory which the CPU wishes to access. When the CPU outputs an address, it is received by each component connected to the address bus, but only one of the devices will be activated. Only the CPU may output to the address bus. It is therefore uni-directional, with one source and one or more destinations.

The data bus comprises eight wires and is used to communicate bytes of data (op codes, operands etc) between the CPU and memory. It may be written to by either the CPU or any other components to which it is connected, depending on whether the CPU is reading or writing data. Therefore the data bus is bi-directional, having either one source (the CPU) and many destinations or one of many sources (the currently addressed device) and one destination (the CPU).

The control bus has a less rigidly defined structure than the data and address buses. The number of wires, their direction and purpose vary widely between microprocessors. The individual control lines are used to synchronise the

operation of the components in the system. In 6800-based systems, the most frequently used control lines are READ/WRITE (R/W), VALID MEMORY ADDRESS (VMA), INTERRUPT REQUEST (IRQ) and Phase 1 and Phase 2 clock lines (E1 and E2). The R/W and VMA lines carry values from the CPU to each component in the system, whereas IRQ, E1 and E2 carry signals to the CPU from particular components. So each wire in the control bus is uni-directional, but some carry values to the CPU and others carry values from the CPU.

Now, the properties of occam channels must be considered. Occam channels provide a uni-directional communications link between exactly two processes. A channel has only one source and one destination. Clearly then, they are not a sufficiently powerful mechanism to simulate the address, data and control buses. This simulation must be done by building a generalised bus process, which accepts data through a channel and passes it on to the desired destination. Such a process should be able to simulate a bus of any width, connected to any number of memory and input-output devices.

5.3.2 Address Bus

The address bus will always input a 16-bit address from the CPU and output it to each device. Figure 5.2 shows this situation as a set of occam processes. It represents five occam processes executing in parallel. The following segment of code describes the diagram in occam:-

```
-- number of devices connected to the bus
VAL no.devices  IS 3:
-- channels to connect processes
CHAN OF ANY CPU.to.address.bus:
[no.devices]CHAN OF ANY address.bus.to.devices:
PAR                    -- execute in parallel
  CPU ( CPU.to.address.bus )
  ROM ( address.bus.to.devices[0] )
  RAM ( address.bus.to.devices[1] )
  PIA ( address.bus.to,devices[2] )
  CPU.to.devices.bus( CPU.to.address.bus,
                      address.bus.to.devices,
                      no.devices )
```

An address is generated by the CPU and passed to the bus process by the channel 'CPU.to.address.bus'. The bus process 'CPU.to.devices.bus' then relays the address to each device in the system via the individual elements of the channel vector 'address.bus.to.devices'. Each device process is allocated one channel from the vector, down which it will expect to receive an address from the process 'CPU.to.devices.bus'.

The 'CPU.to.devices.bus' process then completely satisfies the requirements of the address bus simulation. It also satisfies one of the requirements of the data bus, when the CPU is writing data to memory. The process 'CPU.to.devices.bus' is shown below:-

```
PROC CPU.to.devices.bus([]CHAN OF ANY data.in,
                                    data.out,
                        VAL INT no.devices        )
-- data.in is the channel from the CPU.
-- data.out is the array of channels used to send
-- data to the components on the bus.
  INT numb :
  WHILE TRUE
    SEQ
      data.in ? numb    -- wait for data from CPU
      -- send data to each component on the bus
      SEQ i = 0 FOR no.devices
        data.out[i] ! numb :
```

5.3.3 Data Bus

Now consider the second requirement of the data bus process. It must be able to input from one particular device and pass the message on to the CPU process. This occurs during a CPU read cycle, in which the CPU is expecting to receive data from the addressed device. The occam code below shows the interconnections required between processes.

```
VAL  no.devices  IS 3:
CHAN OF ANY data.bus.to.CPU:
[no.devices]CHAN OF ANY devices.to.data.bus:
PAR
  CPU ( data.bus.to.CPU )
  ROM ( devices.to.data.bus[0] )
  RAM ( devices.to.data.bus[1] )
  PIA ( devices.to.data.bus[2] )
  devices.to.CPU.bus( data.bus.to.CPU
                      devices.to.data.bus,
                      no.devices )
```

The major difference here from the previous example is due to the fact that the bus process 'devices.to.CPU.bus' does not know which device wishes to send data to the CPU. It must therefore wait for an input on any one of the device channels, not just one pre-determined input channel as in the previous example. This situation is easily modelled in occam by using the ALT construct. ALT allows a process to wait for an input on any one of several channels simultaneously. If inputs are received at exactly the same moment, one of them is chosen at random and executed. The process 'devices.to.CPU.bus' is given below:-

```
PROC devices.to.CPU.bus ([]CHAN OF ANY data.out,
                                       data.in,
                         VAL INT no.devices    )
  -- data.out is used to send data to the CPU.
  -- data.in is the array of channels used to
  -- receive data from the components on the bus.
  INT data.item:
  WHILE TRUE
    ALT i =  0 FOR no.devices
      data.in[i] ? data.item   -- wait for input
        data.out ! data.item  :-- send data to CPU
```

The two bus processes defined above individually satisfy the two requirements of the data bus. Still they are not sufficient in this form to completely model a bi-directional bus. To do this they must be brought together and instanced within an occam PAR statement.

Finally all it is necessary to do is instance the data and address bus processes in parallel with the device simulations. The system is 'wired up' by allocating channels to the bus and device processes in the manner

already explained. Each device process will have one channel for each bus process it communicates with. The program code below shows an example system comprising a CPU and three components, communicating via an address and data bus.

```
VAL  no.devices IS  3:
CHAN OF ANY CPU.to.address.bus, data.bus.to.CPU
            CPU.to.data.bus:
[no.devices]CHAN OF ANY address.bus.to.devices,
                        devices.to.data.bus,
                        data.bus.to.devices  :
PAR
  -- address bus simulation
  CPU.to.devices.bus(CPU.to.address.bus,
                     address.bus.to.devices,
                     no.devices)

  PAR          -- data bus simulation
    CPU.to.devices.bus(CPU.to.data.bus,
                       data.bus.to.devices,
                       no.devices)
    devices.to.CPU.bus(data.bus.to.CPU,
                       devices.to.data.bus,
                       no.devices)

  CPU(CPU.to.address.bus, CPU.to.data.bus,
      data.bus.to.CPU )

  ROM(address.bus.to.devices[0],
      data.bus.to.devices[0],
      devices.to.data.bus[0])

  RAM(address.bus.to.devices[1],
      data.bus.to.devices[1],
      devices.to.data.bus[1])

  PIA(address.bus.to.devices[2],
      data.bus.to.devices[2],
      devices.to.data.bus[2])
```

5.3.4 Control Bus

Due to the irregular nature of the control bus in most microprocessor systems, no attempt was made to define a strict simulation structure. Instead, each device is merely allocated a channel for each control line which it uses. Control signals are then simply fed out to memory from the CPU or multiplexed to the CPU from memory by a simple

'control.bus' process. This process operates in a manner identical to the bus simulation processes described above.

5.4 Implementation on a Single Transputer

Initially, simulations of a number of the basic components in the 6800 family were constructed. These include the 6800 and 6802 microprocessors, various memory devices, parallel and serial input-output controllers and an analogue-digital converter. Individual component simulations were then brought together to form simulations of example MCSs, with components communicating via a bus system identical to the one described above. The CPU register values were displayed on a monitor, enabling the simulations to be fully debugged and tested. No attempts were made at this stage to optimise the performance of the simulations. In fact, the use of the monitor to display the state of the components caused a severe reduction in performance. A monitor screen is a slow output device, and causes the processor to wait while a character is written to the screen.

When all the simulations had been thoroughly tested, it was decided to carry out several experiments, in order to measure the performance of simulations on a single transputer. This would provide a useful reference point for comparison with the performance of equivalent simulations on networks of transputers. In order to achieve this, a sample MCS simulation was constructed comprising a 6800 microprocessor and clock, ROM, RAM and a PIA. All monitor output was removed from the component models (except start and finish messages), so that the observed execution times would not contain delays due to the screen. Varying assembler programs were then simulated using this configuration, and the execution times recorded. The transputer utilised to perform these experiments was an IMS T414 running at 20 MHz, which resided on the B004 board. Overall an average of approximately four hundred (6800) cycles per second were simulated.

As extra memory and I-O components were added, the performance of the simulation deteriorated. This occurred due to the additional demand placed on the processor by the inclusion of more concurrent processes. On a single transputer, the processor must be shared between the processes in the simulation. Consequently, as more processes are added, the overall performance of the transputer decreases. However, this problem should not occur when a network of transputers is used as the target architecture for the simulation. In this situation, as more components are added, more transputers can be incorporated into the network. This increases the computational power available for simulation, and compensates for the additional workload.

5.5 Implementation on a Transputer Network

5.5.1 The Problem

Consider an example simulation of a MCS comprising a CPU, two 128-byte RAMs, one 1K-byte EPROM and one PIA. If this simulation is executed on a single processor (transputer or otherwise), there is no need to make any changes to the system bus simulation. The channels which enable communication between devices are implemented as memory locations. Consequently no severe restrictions are placed on their number. However problems are encountered if the simulation is mapped on to a network of transputers and each transputer simulates one component. Transputers have only four external communications links, each of which implements a pair of occam channels. Therefore one transputer can be linked to a maximum of four other transputers. This restriction makes the above scheme impossible to implement, as a situation such as the one shown in Figure 5.3 will arise, in which one component cannot be interfaced.

Possibly the simplest solution to the problem is achieved by configuring the transputers in a ring-type network as in

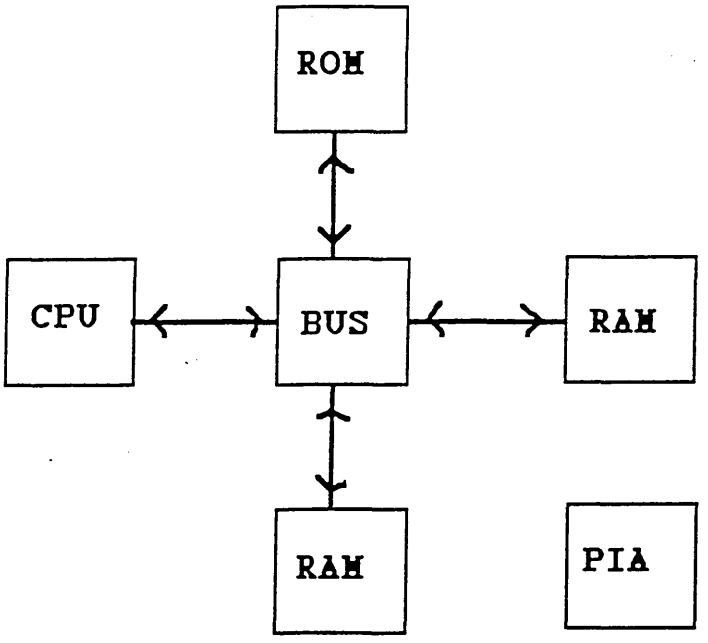


Figure 5.3 Configuration Problems

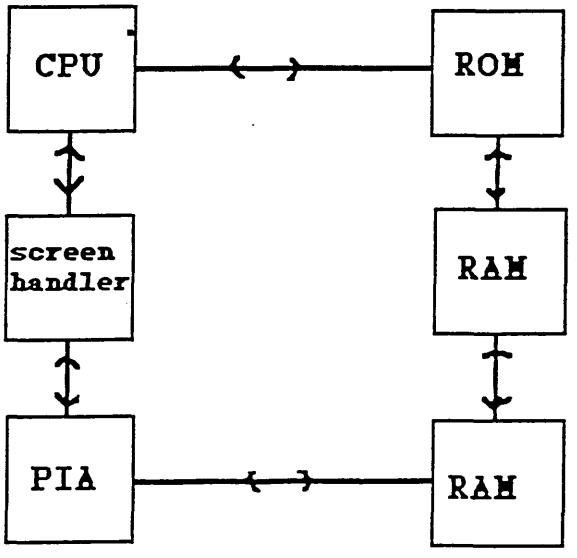


Figure 5.4 Ring Network for Simulations

Figure 5.4. A modified version of the system bus simulation is placed on each transputer. The function of the bus is to input messages from specified links, interpret the messages and pass them on to the next processor in the ring. Each bus process also needs to send and receive messages to and from the component simulation residing on the same transputer, and must be able to pass messages back to the CPU process.

The revised system bus simulation was implemented initially on a single transputer. Using this approach, it was possible to verify the modified simulation, without any concern for the eventual configuration of the processes. The ability to write and test programs intended for transputer networks on a single-processor host development system is one of the most significant features of the occam/transputer pairing. Programs may be written with only the logical behaviour in mind. As soon as they function correctly, the constituent processes can be distributed on to a network simply by adding configuration information. There is no need to alter the program logic.

5.5.2 Experimental Strategy

The behaviour of multi-transputer networks is extremely complex, and often counter-intuitive. The addition of more transputers does not always yield a proportional improvement in system performance. In fact it can lead to performance degradation[5.22]. It is possible to reach a situation where transputers in the network may not have enough work to perform. This can lead to the system becoming communication-bound. The speed of communication between processors is considerably slower than internal communications. It is sometimes possible therefore for one transputer to execute two communicating concurrent processes faster than an equivalent two transputer system, with one process residing on each processor.

This trade-off between processor work-load and external communications is one of the main areas of difficulty regarding the performance of simulations. To achieve maximum execution speed the external communications must be brought down to a minimum. In MCS simulations, the majority of memory accesses made by the CPU simulation are to read-only memory, in which the controlling code resides. It therefore makes sense to locate the ROM simulations next to the CPU in the ring network. This ensures the CPU simulation receives the data as early as possible. Another strategy would be to place the ROM simulation on the same transputer as the CPU simulation. The ROM simulation is a relatively simple process, and may execute in less time than it takes to perform external communications. By the same token, it is sensible to place the simulation of the least accessed component at the end of ring network. It will take longer for data to be read or written from the last component in the ring, as the data must be passed through the simulations in between.

5.5.3 Experiments Performed

Five T212 20 MHz 16-bit transputers, each with 56K external memory[5.23] were available for executing simulations. For the initial experiments, the component simulation processes were configured as in Figure 5.5. This configuration was chosen merely as a convenient starting point for the experiments, given the number of transputers available. Several example 6800 assembler programs were simulated on this network, and timings were taken using the transputer's real-time clock facilities. A summary of results is given below in Table 5.1.

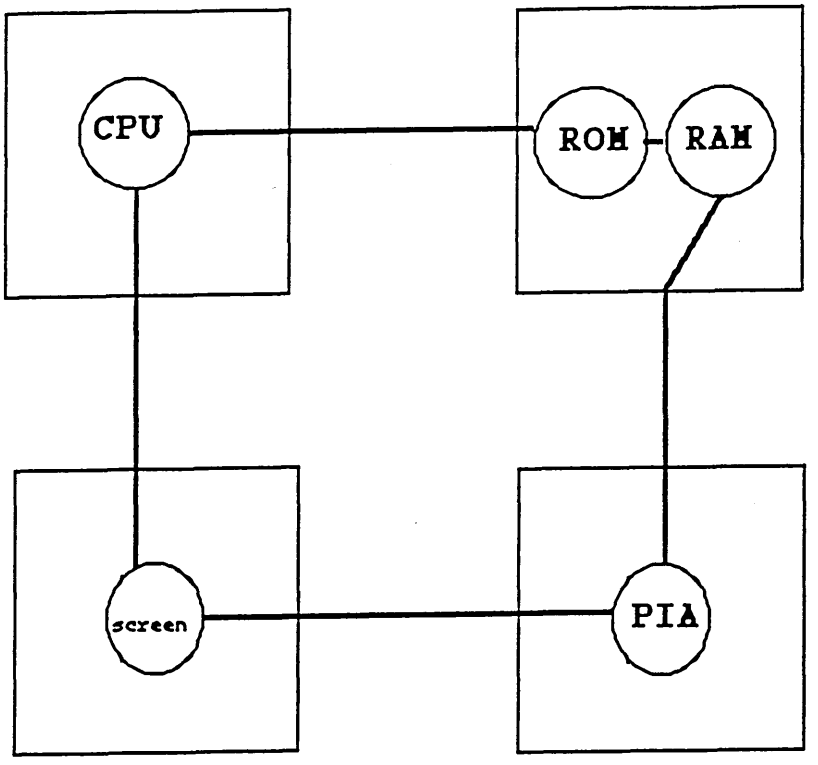


Figure 5.5 An Example Configuration

TABLE 5.1 Summary of example simulation performances.

	6800 cycles	one transputer	five transputers
SIMULATION 1	2030	2.9secs	0.641secs
SIMULATION 2	683	1.1secs	0.183secs

Simulation 1 involved mainly accesses to ROM and RAM.
Simulation 2 involved accesses to ROM, RAM and PIA.

The initial results proved that the performance of simulations could be significantly improved by placing the component simulations on a suitably organised transputer network. The application of five transputers gave roughly a five-fold speed increase. Further experiments included separating the 6800 CPU simulation into two processes (the 'control unit' and the 'ALU'), and placing one of these processes (the 'control unit')on the same transputer as the ROM simulation. This brought some improvement in performance, but the results were not as impressive as expected. The combination of these two enhancements gave approximately a twenty percent improvement in execution speeds over the figures given for five transputers in Table 5.1. This gave a peak performance level of just under six thousand (6800) cycles per second.

5.5.4 Discussion

From the results obtained so far, it would appear that there is little chance of obtaining near real-time simulations. Although there is certainly room for improvement in performance, it is unlikely that a sufficiently large increase could be made. Still, in many cases it could be feasible to use the simulation to control the desired physical resource. This would give further insights into the problem, and sometimes, in less time-critical situations, it may even fulfil the system requirements.

An examination of the behaviour of simulations has highlighted the reason for this unexpectedly low level of performance. Microprocessor systems do undoubtedly operate in a highly parallel fashion, which can be exploited by an appropriate parallel processor architecture. However the parallelism at component level may be separated in to two distinct areas, which do not overlap. For example, when the CPU simulation wishes to fetch data from memory, it generates an address, outputs this address on the address bus, and then waits until data is available on the data bus. During this time, before data is available, the CPU can usually do no useful processing. During this same period, the component simulations attached to the address bus are all busy receiving the address and control information from the CPU. Each component, in parallel, decodes the address, and if selected, performs a read operation from the addressed word of memory. This data is then placed on the data bus. The memory and I-O simulations have now completed their task for this clock cycle, and wait in an idle state for the next cycle to begin. (I-O devices may generate interrupts at any time during a clock cycle, and therefore do not totally conform to this description of their behaviour.) As the memory components fall idle the CPU simulation is awakened by the arrival of data from the data bus. It can now process this data, possibly performing the required tasks in parallel. No useful work is done by the memory components (except perhaps generating interrupts) while the CPU processes data.

The conclusion that can be drawn from this description of the simulation's behaviour is that much of the potential processing power of the transputer network is lost as processes wait for data. It seems that the only strategy that may help in this case is to attempt to overlap some of the CPU simulation processing with reading and writing data to memory. This means generating addresses on the address bus simulation as early as possible during a clock cycle. The CPU simulation could then carry out any outstanding

processing while it awaits the arrival of data from memory. This situation is especially relevant when instruction operands are fetched from memory. Immediately the instruction code is received, the address of the first operand may be output on the address bus simulation. The memory component simulations can then deal with this address, and simultaneously the CPU simulation can decode the instruction operation code.

Further performance improvements may be achieved by optimising the inter-process communication. Information passed between component simulations is currently in the format of a sequence of messages, comprising either one byte or one integer. Such message sequences could be packaged into arrays and passed in a single communication between processes. For external communications, this method would lead to more efficient use of the transputer's link interfaces. The link interfaces operate independently of the transputer processor, only interrupting the processor each time a message has been received. Longer messages mean that the processor need only be interrupted once, when a whole message is received, instead of several times, when each constituent part of a message is received[5.24].

Finally different transputer architectures were considered as a basis for running simulations. The ring network was originally used mainly for its simplicity and ease of implementation. A ring is a very flexible architecture, and allows processors to be added without any change to its basic message passing protocol. The positioning of processors in the ring is unimportant, and no processor needs to know explicitly the position of any other. There is also no need to know how many component simulations are on each transputer. An alternative to the ring, which is often used as the basis for transputer networks, is a tree. A slight variation on the tree structure could be used as shown in Figure 5.6. The advantage of the Figure 5.6 is that the access time to each component from the CPU is more even than a ring network. However the average access time

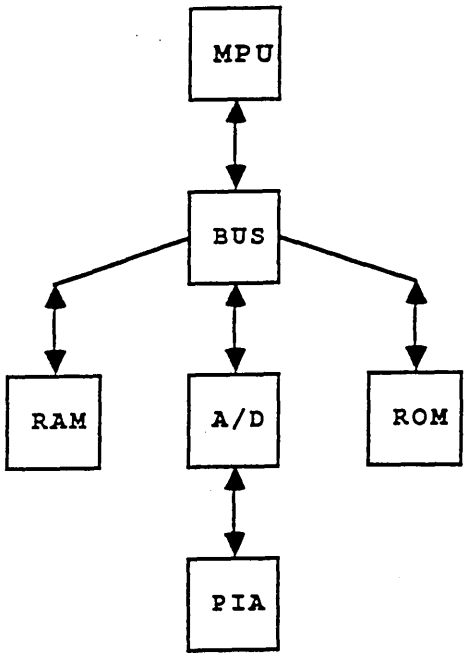


Figure 5.6 A Tree Network

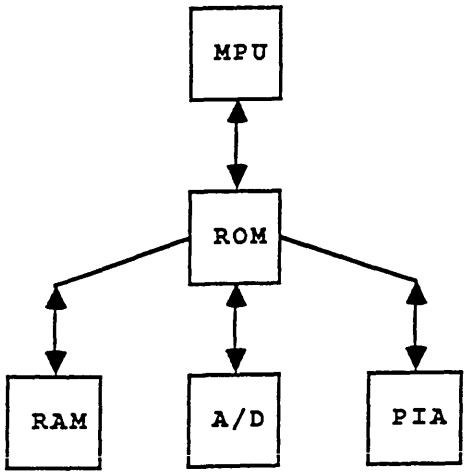


Figure 5.7 An Improved Tree Network

may be worse than for a ring. In Figure 5.6, reading or writing data involves mostly two and occasionally three external communications. In a suitably configured ring, access to the ROM simulation only ever involves one external communication. As the majority of CPU read cycles will be from ROM, a ring network may prove to be more efficient. Figure 5.6 is also less flexible than a ring in terms of process configuration, as the bus processor has no spare links to allow for expansion.

Figure 5.7, in which the bus process is placed on the same transputer as the CPU simulation, provides a better solution. It gives the same access time to ROM as the ring network, and a constant access time to the other three component simulations. It is still however less flexible than a ring. The ROM process would have to know exactly how many processors it is connected to, and the message passing mechanism would have to know how many component simulations reside on each processor. The expansion of the network to accommodate more processors would also cause complications to the message passing strategy.

Figure 5.8 is effectively the same as Figure 5.7, but it offers a constant access time of one external communication to each component simulation. Once again though, the addition of more component simulations and more processors would greatly complicate the software which controls the routing of messages.

So it appears that although each of the these networks has different advantages and disadvantages, no single network has any significant advantage over a simple ring. Figure 5.8 is certainly worth considering as an alternative. Still it is far less adaptable than a ring, and would require a considerably more complex bus simulation process.

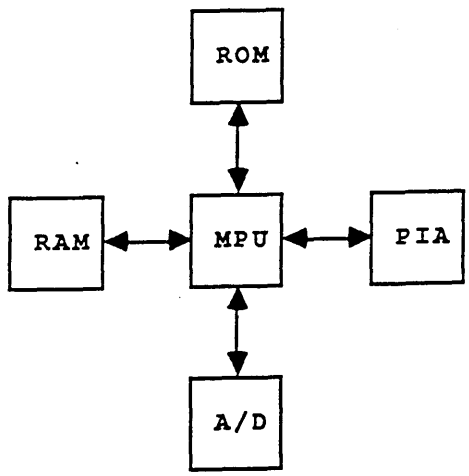


Figure 5.8 A General Network

5.6 Conclusions

The feasibility of constructing a fast simulation facility for real-time microprocessor control systems has been discussed. Occam has been used to describe the behaviour of a set of LSI components, and to model the behaviour of a system bus. The individual component simulations are generalised, providing a standard interface to the bus system. This enables any valid microprocessor system configuration to be easily simulated.

The use of occam has provided two major advantages over previous approaches. Firstly, occam has proved to be an appropriate language for simulating microprocessor systems. The model of concurrency in occam corresponds closely to the actual operation of hardware devices, thus allowing the behaviour of components to be expressed naturally. Second, the close relationship between occam and the transputer enables simulations to be implemented on transputer networks. This approach may yield a high level of performance, possibly making it feasible to test the simulation in a realistic physical environment. The initial results have not proved as encouraging as expected, but they have shown that the parallelism inherent in microprocessor systems can be exploited by appropriate tools to increase the performance of simulations. However more work needs to be carried out in order to discover the maximum attainable performance level of simulations.

References

5.1 Gorton,I.,Kerridge,J. and Jervis,B.:'Simulating Microprocessor Systems using Occam and a Network of Transputers', to be published, IEE Proceedings-E Computers and Digital Techniques, January 1989

5.2 Zaks,R. : 'Microprocessors', ch.4, pp 163-204, SYBEX Inc, USA, 1977

5.3 Piloty,R. :'The Conlan Project: Concepts, Implementations and Applications',IEEE Computer, Feb 1985, vol 18, no 4, pp 81-92

5.4 Dietmeyer,D. :'Logic Design of Digital Systems', Allyn and Bacon, 1971

5.5 Muehlemann,K. :'Software Model of the M6800 Microprocessor', Euromicro Newsletter, France, vol 3, no 2, April 1977, pp 76-79

5.6 Armstrong,J.R. and Woodruff,G. :'Simulation Techniques for Microprocessors', Procs 14th Design Automation Conf, June 1977

5.7 Flake,F.P.,Moorby,P.R. and Musgrave,G.: 'HILO MARK 2 Hardware Description Language', Procs IFIP 5th International Conf. on CHDLs, Kaiserslautern, West Germany, Sept 1981

5.8 Hudson,C. :'Techniques for Developing and Testing Microprocessor Systems', Software and Microsystems, vol 4, no 4, August 1985, pp 85-94

5.9 INMOS Ltd.: 'occam 2 Reference Manual', Prentice-Hall, 1988

5.10 May,D. :'Occam (Hardware Description Language', Procs IEE Colloq on Software Tools for Hardware Design, London 1983, Digest 98, pp 5/1-5

5.11 May,D. and Keane,C. :'Compiling occam into Silicon', in Communicating Process Architecture, INMOS Ltd., 1986

5.12 Collis,G.V. and Kappos,E.J. :'Occam as a Hardware Description Language', Software Engineering Journal, vol 2, no 6, November 1987, pp 213-219

5.13 INMOS Ltd., :Transputer Reference Manual', Prentice-Hall International, 1988

5.14 Gorton,I. :'A Distributed Architecture for Simulating Microprocessor Systems', Procs. Conf. 7th Occam User Group, Grenoble, September 1987

5.15 Collis,G.V. and Edwards,M.D. : 'Automatic Hardware Synthesis from a Behavioural Description Language: occam', Microprocessing and Microprogramming, 1986, vol 18(1-5), pp 243-250

5.16 Dowsing,R.D. : 'Simulating hardware structures in occam', Software and Microsystems, August 1985, vol 4, no 4, pp 77-84

5.17 Motorola INC. : '8-bit Microprocessors Data Manual 1985', 1985

5.18 Streitmatter,G.A. and Fiore,V.:'Microprocessors - Theory and Application', ch.12, pp 219-253, Reston Publishing Company, Virginia, 1979

5.19 Barbacci,M.R. :'Instruction Set Processor Specifications(ISPS): The Notation and its Applications',IEEE Trans. on Computers, vol c-30, no 1, Jan 1981, pp 24-39

5.20 Waxman,R. :'Hardware Description Languages for Computer Design and Test', IEEE Computer, April 1986, vol 19, no 2, pp 90-97

5.21 INTEL Corporation : ' INTEL 8080 Microcomputer Systems', Santa Clara, California, 1975

5.22 Gorton,I. :'Configuring Occam Programs for Transputer Networks', Computing, March 24th, 1988 pp 33-35

5.23 Gorton,I. :'Developing Transputer Networks using the IMS B006 Evaluation Board', Computing, March 17th, 1988, pp 24-25

5.24 Newport,J.R. : 'An Introduction to occam and the Development of Parallel Software', Software Engineering Journal, July 1986

6. Evaluation

6.1 Introduction

In the control examples considered so far, the arrival of a single input on a channel has been sufficient to generate a value on a single corresponding output channel. However, many control applications do not operate in such a regular, simplistic manner[6.1]. By considering several examples, it will be shown how the behavioural specification language (BSL) can be used to describe control systems with various characteristics and features. The high-level behavioural simulation and the macro-assembly language are also assessed as methods of representing behavioural specifications at different intermediate levels of design description. Finally, the advantages of using occam as a hardware description language for constructing component and system simulations are considered.

6.2 Applicability of the Behavioural Specification Language

6.2.1 Data Stream Applications

Many control applications require that a number of readings are taken from the environment before a value can be produced on an output channel. Multiple inputs are referred to as data streams[6.2]. A simple example of a data stream application is a system which takes a number of readings from an input channel and outputs their average. The number of values in each data stream varies; it is always indicated by the first value in the data stream. The BSL for this application is given in Figure 6.1.

When the arrival of a value is detected on the input channel, it is passed to the service routine as a parameter, along with the name of the channel on which the input occurred. The service routine can now input the remainder of the data stream itself, as the correct input

```
CHAN IN stream:
CHAN OUT average:

PROC calc.mean : UNINTERRUPTABLE
                 (BYTE len, CHAN IN stream)
  BYTE value:
  INT mean, sum:
  SEQ
    sum := 0
    SEQ i = 0 FOR len
      SEQ
        stream ? value
        sum := sum + INT (value)
    mean := sum / INT (len)
    RESULT( BYTE(mean) )
:

POLL

  BYTE len:
  stream ? len
    average ! calc.mean( len, stream )
:
```

Figure 6.1 Averaging a Single Data Stream

```
CHAN IN  stream1,  stream2,  stream3:
CHAN OUT average1, average2, average3:

...  PROC calc.mean

POLL
  BYTE len:
  stream1 ? len
    average1 ! calc.mean ( len, stream1 )

  BYTE len:
  stream2 ? len
    average2 ! calc.mean ( len, stream2 )

  BYTE len:
  stream3 ? len
    average3 ! calc.mean ( len, stream3 )
:
```

Figure 6.2 Averaging Multiple Data Streams

channel is accessible to it. When the average of the data stream has been calculated, it is returned from the service routine and subsequently output by a RESULT statement.

The extension of this application to perform the same averaging function on several different pairs of input and output channels is straightforward . The service routine itself does not require any modifications. Only the channel declarations and control section need to be altered, as shown in Figure 6.2.

Thus, by making the input channel a parameter to the service routine 'calc.mean', the routine has become generalised, enabling it to handle data streams from different input channels. Note that in this example, the routine is defined as UNINTERRUPTABLE. This ensures that as soon as the number of values in a new data stream is received on a particular channel, the remainder of that stream is input and processed immediately.

6.2.2 Discrete-State Controllers

A discrete-state system is one for which at every instant of time the state of the system is defined by the values of a set of variables, each of which can only be defined to be in one of two conditions, namely on or off[6.1]. An example specification of a discrete-state control system is given below:-

An engine control system has two-state input variables of rpm, temperature and load. The two-state outputs are fuel-feed, air-feed and spark advance. the outputs are required to be high under the following conditions:-

Fuel feed: when the rpm is low and the load is high
Air feed: when the temperature is low and the rpm is high
Spark advance: when the temperature and load are high

Each input should be sampled at one second intervals.

Figure 6.3 gives the behavioural specification for this example. As each of the system variables can only ever have

```
CHAN IN  rpm, temp, load:
CHAN OUT fuel.feed, air.feed, spark:

PROC fuel.control: INTERRUPTABLE
                   ( BOOL rpm, load )
  SEQ
    IF
      ( NOT(rpm) AND (load) )
        RESULT(TRUE)
      TRUE
        RESULT(FALSE)
:

PROC air.control: INTERRUPTABLE
                  ( BOOL temp, rpm )
  SEQ
    IF
      ( NOT(temp) AND (rpm) )
        RESULT(TRUE)
      TRUE
        RESULT(FALSE)
:

PROC spark.control: INTERRUPTABLE
                    ( BOOL load, temp )
  SEQ
    IF
      ( (temp) AND (load) )
        RESULT(TRUE)
      TRUE
        RESULT(FALSE)
:

BOOL temp.val, rpm.val, load.val:

WHILE TRUE

  INTERRUPT

    temp ? temp.val    SAMPLE 1 second
      air-feed ! air.control( temp.val,
                              rpm.val )
    rpm ? rpm.val      SAMPLE 1 second
      fuel-feed ! fuel.control ( rpm.val,
                                 load.val )
    load ? load.val    SAMPLE 1 second
      spark ! spark.control (load.val,
                             temp.val )
:
```

Figure 6.3 BSL Description for the Engine Controller

one of two values, they are most naturally defined by Boolean variables, with 'TRUE' representing 'on' and 'FALSE' representing 'off'. The three input variables are declared globally, because the value of each is required in more than one service routine. Thus, for example, the most recent temperature reading is used to calculate the output state on both the air-feed and spark output channels. The values of all the global variables which are needed to produce an output signal by a particular routine are passed as parameters. In many applications, there will be a mixture of shared and private input values. Although such a mixture can be catered for by declaring all the input variables globally, it would be better to use a mixture of global and local data, as this would more accurately portray the structure of the application.

6.2.3 Proportional Mode Controllers

A proportional mode algorithm gives a high degree of accuracy of control, based on the difference between the current state of the environment and its desired state[6.1]. The value of the desired or ideal state of the environment is known as the set point, and the variation from the set point of the current value is known as the error. In this mode, the output of the controller is simply proportional to the error itself. Thus if the error is considerable (i.e. the current input value deviates greatly from the set point), the magnitude of the corrective action taken is proportionally greater.

The equation which describes the proportional control mode is given below:-

```
d.out = ( kp * error ) + zero.output

where
d.out = output value
kp    = constant defining the proportional gain
error = current.value - set.point
zero.output = output when error is zero
```

The following is a requirements specification for an application which is to use the proportional mode algorithm. It is adapted from an example in [6.1].

A microprocessor system is to be used to control the temperature in a system. Input from the A/D converter is in the range #00 to #7F. The controlled output is a continuous heater. The heater is turned off when a value of #00 is output, and is full on when #7F is output. When the temperature value exceeds #72, the heater is to be turned off, and an alarm signal generated by sending #FF to the alarm output port. The set point is #37, and the proportional gain is #03. The zero error heater setting has been found to be #52. The output value should be updated every second.

The above application is an example of a reverse acting proportional mode system. This is because a temperature reading above the set point must result in the heater setting being reduced. The reverse action is catered for in the behavioural specification in Figure 6.4. by effectively reversing the sign of the error value. This specification demonstrates that the description of such control algorithms in the BSL presents no particular difficulties. The BSL retains all the expressive capabilities for algorithm description normally associated with a high-level language, while augmenting these with constructs which allow both abstract and explicit specification of the system's behaviour.

Figure 6.4 also illustrates how the BSL can describe more than one output resulting from a single input value. In the service routine, if the temperature reading is greater than the emergency value, two output signals are generated, one to turn off the heater (the RESULT statement), and another to set off the alarm. The latter is actually a simple occam output statement. Thus, by passing output channels as parameters, an output statement can be utilised to enable a service routine to send values to output channels other than the one associated with its RESULT statement. Conversely, the BSL can describe applications in which the results of processing values from multiple input channels

```
CHAN IN  temperature:
CHAN OUT heater, alarm:

PROC temp.control : INTERRUPTABLE
                ( BYTE value, CHAN OUT alarm )
  VAL set.point IS  #37 (BYTE) :
  VAL emergency IS  #72 (INT)  :
  VAL zero.error IS #52 (BYTE) :
  VAL prop.gain IS  #3  (INT)  :
  VAL heater.off IS #00 (BYTE) :
  VAL ring.bell IS  #FF (BYTE) :
  INT error, output:
  SEQ
    error := INT (value - set.point)
    IF
      error > emergency
        SEQ
          alarm ! ring.bell
          RESULT(heater.off)
      error = 0
        RESULT(zero.error)
      error < 0     -- temperature too low
        SEQ         -- increase heater setting
          error := - (error)
          error := error * prop.gain
          output:= error + INT (zero.error)
          RESULT( BYTE(output) )
      error > 0     -- temperature too high
        SEQ         -- decrease heater setting
          error := error * prop.gain
          output:= INT (zero.error) - error
          RESULT( BYTE(output) )
:

WHILE TRUE

  INTERRUPT

    BYTE value:
    temperature ? value SAMPLE 1 second
      heater ! temp.control( value, alarm)

:
```

Figure 6.4 An Example Proportional Mode Control Algorithm

can be sent to a single output channel. An example of this arrangement is shown below:-

```
CHAN IN in1, in2:
CHAN OUT out1:
WHILE TRUE
  POLL
    BYTE v:
    in1 ? v
      out1 ! transform (v)
    BYTE v:
    in2 ? v
      out1 ! alter (v)
```

6.2.4 A Problem

Consider a very simple terminal driver. The driver accepts characters from the keyboard, and stores them in a buffer. Only when a carriage return character is received are the characters in the buffer sent to the screen. When the buffer contents have been output, the driver merely waits for the start of the next line of characters from the keyboard. Figure 6.5 gives a behavioural specification for this problem.

Although the specification in Figure 6.5 is complete and does correctly describe the required behaviour of the terminal driver, this example exposes a slight semantic inadequacy in the language. The language definition states that a service routine must return a single, simple typed value via a RESULT statement. The value contained in the RESULT statement is sent to the output channel associated with that service routine. In the case of the terminal driver, the result of executing the service routine is a line of text, which is composed of a number of individual values (characters).A line of text is not a single, simple typed value, and consequently cannot be returned in a RESULT statement.

Figure 6.5 overcomes this problem by passing the output channel as a parameter to the service routine. The characters in the line are then output directly to the

```
CHAN IN keyboard:
CHAN OUT screen:

PROC terminal.driver: UNINTERRUPTABLE
                    (BYTE ch, CHAN IN key,
                              CHAN OUT scr )
  VAL C.R IS 9 (BYTE):
  [80]BYTE buffer:
  INT line.len:
  SEQ
    buffer[0] := ch
    line.len := 1

    -- input rest of line

    WHILE (ch <> C.R)
      SEQ
        key ? ch
        buffer[line.len] := ch
        line.len := line.len + 1

    -- output characters

    SEQ i = 0 FOR (line.len - 1)
      scr ! buffer[i]

    -- output carriage return

    RESULT( C.R )
:

POLL

  BYTE first.char:
  keyboard ? first.char
    screen ! terminal.driver( first.char )
:
```

Figure 6.5 A Terminal Driver Specification

screen channel using an output statement. The requirement for a RESULT statement in the service routine is satisfied by including such a statement to return the carriage return character, which is always the last one in the line.

A much clearer, concise and semantically consistent solution to this problem is to allow a RESULT statement to return variable length arrays as well as values of simple variable types. The complete line of text could then logically be regarded as the result of executing the service routine. The amended statement would thus become:-

```
RESULT( line.len, buffer )
```

From this statement, the BSL compiler would generate the necessary SEQ loop, in macro-assembler, to output the line of characters. Modifying the language in this manner would expand the expressive power of the language, and make it semantically more secure.

6.2.5 Evaluation

The above examples have demonstrated the flexibility and generality of the BSL features and constructs by describing a number of behavioural characteristics, which are common in control systems. Specifically, these are:-

1. The capability of service routines to access both private and shared input data.
2. The capability of service routines to initiate outputs to multiple output channels.
3. The capability of service routines to process values from multiple input channels.
4. The capability to generalise service routines through the use of channel parameters.
5. The ability of service routines to process data streams.
6. The ability to naturally describe the features of discrete-state control systems.
7. the ability to express mathematically-based control strategies of arbitrary complexity.

Therefore it seems that the BSL is well-suited to describe the behaviour of a range of applications within the broad spectrum of control systems. However, as the mathematical complexity of the control algorithms increases, the BSL would suffer from omissions such as extended arithmetic data types and mathematical function libraries. Such libraries though are available in the occam language, making their inclusion in the BSL a simple task. Still, in its current form, the BSL should be capable of describing the behaviour of all but the most intricate of control applications, and should easily be sufficient for systems of moderate complexity, such as single-board embedded controllers.

6.3 Evaluation of the Behavioural Simulation

The behavioural simulation which is generated directly from a behavioural specification provides a prototype implementation of the desired control system. Its main purpose is to facilitate the precise testing of the control algorithms in the specification. It also presents the designer with the opportunity to explore the implications of the interrupt, polling and priority strategies which have been selected for the application.

An automatically generated, interactive user interface allows the designer to initiate inputs to service routines, and to attempt to interrupt the currently active service routine. By presenting a range of typical input values to the service routines, the control algorithms can be verified, and if necessary, modified, with the minimum of cost and effort. The designer may also simulate a variety of different input sequences, in order to observe the system's behaviour in a number of likely (or possibly unlikely) scenarios.

Thus this implementation-independent level of simulation constitutes an important opportunity for the verification of a behavioural specification at a very early stage of the

system design. However, it should be noted that no attempt is made to simulate the behaviour of the physical environment. This is performed by the designer, who must choose the sequence of input values for the simulation to act upon. Consequently, this level of simulation can only be used to check that the designer has correctly encoded the chosen control strategy into the specification. It cannot be used to ensure that the actual algorithms that have been selected are correct with respect to the control of the environment. This depends upon certain other factors such as the rate of change of the controlled variable and the selected sampling interval. The realm of control engineering provides theories and methodologies which enable the selection of control strategies[6.3]; these however are beyond the scope of this work. Therefore it is important to remember that the simulation can only be used to check the correctness of particular implementation of a control strategy. It provides no guarantee that this strategy is actually appropriate to control the physical environment under consideration.

6.4 Evaluation of the Macro-Assembly Language

The purpose of the macro-assembly language is to provide an intermediate design representation into which behavioural specifications can be transformed. This intermediate design representation should then serve as a basis for the generation of assembly language for a given microprocessor configuration. The macro-assembly language satisfies these criteria due to the following attributes:-

1. Abstraction

The data types and control structures of the BSL compile on a one-to-one basis into the macro-assembler. They are thus retained at a sufficiently abstract level, which does not rely on the features of any particular microprocessor to implement. In fact, only complex BSL calculation and expression evaluation statements need to be decomposed and represented by multiple macro-assembler operations.

Consequently, the translation process essentially involves the simplification of the more sophisticated features of the BSL, such as parameter passing and local data access. It also reorders the specification into a format which is more convenient for generating microprocessor assembly code. Therefore the macro-assembler code for any particular example bears many similarities to the BSL from which it has been produced. This is illustrated by Figure 6.6, which shows the macro-assembler which results from compiling the proportional mode control example in Figure 6.4. The two examples are similar in size, and the control and data structures of the specification are still readily apparent in the macro-assembler. The target assembler code which can be generated from the control structures is generally efficient, though it is heavily processor-dependent, as it relies upon the collection of compare and branch instructions available.

2. Operations

While the data and control structures remain abstractly defined in the macro-assembly language, the data manipulation operations are represented in a fashion more resembling actual microprocessor instructions. Features which are common to most commercial processors have an equivalent macro operation, which may be implemented by one or more microprocessor assembler instructions. Thus substituting the sequence of instructions which represents an operation is a relatively straightforward task. As mentioned in Chapter 4, because no assumptions regarding register usage are made, a degree of inefficiency is introduced into the code which implements the operations. This though could be eliminated by processor-specific optimisation tools.

The macro-assembly language can be viewed as essentially defining the functionality of an abstract microprocessor, without specifying the architecture which is to implement that functionality. The ability of existing microprocessor architectures to implement this required functionality

```
CHAN temperature
CHAN heater
CHAN alarm
INT error
INT output
BYTE value
INT itemp
BYTE btemp
WHILE(1=1)
  INTON
  WAIT
  INPUT temperature, value
  CALL temp.control
ENDWHILE
END

temp.control:
  INTON
  B-COPY btemp, value
  B-SUB  btemp, 37
  BTI    error, btemp
  IF
    error,>,0072
      OUTPUT alarm, FF
      OUTPUT heater, 00
    error,=,0
      OUTPUT heater, 52
    error,<,0
      I-COM error
      I-COPY itemp, error
      I-ADD itemp, 0052
      I-COPY output, itemp
      ITB   btemp, output
      OUTPUT heater, btemp
    error,>,0
      I-MUL error, 0003
      BTI   itemp, 52
      I-SUB itemp, error
      I-COPY output, itemp
      ITB   btemp, output
      OUTPUT heater, btemp
  ENDIF
  RETURN
```

Figure 6.6 Macro-Assembler Code to Represent the BSL for the Proportional Mode Control Example

varies greatly. This project has concentrated on defining the macro operations using 8-bit architectures. However, due to the requirements for 16- and 32-bit macro operations, 8-bit processors produce some complex and cumbersome implementations. Consequently, it is probable that the functionality of the macro-assembly language would be best implemented by 16- and 32-bit processors such as the Motorola 68000 or 68020[6.4,6.5]. These processors have instructions which can manipulate 8-, 16- or 32-bit words, which would make the implementation of the macro operations considerably easier.

The macro-assembly language therefore seems to possess sufficient generality to make it an appropriate and convenient intermediate design representation for microprocessor system behaviour. It inherits the data and control abstraction from the BSL which enable it to be processor-independent, while containing a set of data manipulation operations which closely mimic those usually found in microprocessor instruction sets. This combination facilitates ease of translation at all levels, and gives a representation from which alternative design proposals can be generated and evaluated.

6.5 Evaluation of the Microprocessor Simulation Facility

The purpose of the component level simulation facility is to provide a means of testing the integration of the software with the selected hardware configuration for a desired system. In order to achieve this, the hardware simulations should model exactly the behaviour of the hardware components themselves. Further, it should be possible to simulate any given configuration of components which is valid. This requires that the individual component simulations are totally generalised and usable in different configurations, precisely the same as the hardware components they model.

The use of occam as a hardware simulation language proved to be important in satisfying the above requirements. By simulating a component as a parameterised, concurrent occam process, it is possible to model a component as a black box, the behaviour of which is completely defined by the value of the signals it receives on its input channels. Occam channels then form the basis of a simulation of a microprocessor system bus, which models the communication paths between components. Finally, the individual component models which comprise a simulation can be executed concurrently, with the underlying execution model of occam automatically providing the required scheduling and synchronisation of components. This would not be so if a conventional sequential language had been used: in that case, the programmer would have to write a scheduler to control the order of execution of components[6.6].

The exploitation of occam's parallel execution facilities also afforded the opportunity to increase the speed of simulation by distributing the component simulations on to a multi-transputer network. Experiments showed that a near linear speed-up in the execution rate of simulations could be achieved by the addition of up to five transputers, giving a peak execution rate of just below six thousand Motorola 6800 instructions per second. It is worthwhile comparing this value with the performance of a microprocessor simulator constructed in Prolog[6.7]. This simulated a simple, hypothetical 32-bit processor which could execute 28 instructions. When the simulation was executed (on a single 5 MIPS processor), average simulation rates of approximately eleven instructions per second were observed. Although this comparison is subject to certain discrepancies, such as the relative complexities of the simulations, the vast difference between the simulation speeds does serve to illustrate the high rate of performance that can be accomplished when the inherent parallelism of hardware systems is exploited.

It is hoped that, in order to test proposed designs, simulations could be connected, via a standardised interface board, to the actual physical environment to be controlled. This could certainly be achieved in applications which are not of a strictly time-critical nature. In the meantime however, a more general testing mechanism has been defined, which allows the designer to supply files of test data to be incorporated into simulations[6.8]. Test values can also be input into the simulation via a user interface. The user interface displays the state of the simulation's registers together with input and output ports of the various components on a monitor. It further allows the designer to single-step (one simulated machine instruction at a time) through the execution of a simulation, in order to examine the operations in detail.

The hardware simulation facility therefore forms the final stage of verification for a given design. Through extensive testing of the component level simulation it should be possible to ensure that the hardware and software designs are indeed compatible, and that the control algorithm is suitable to the needs of the application. When the designer is eventually satisfied that all the major design faults have been eradicated, the implementation of the system may proceed.

6.6 Summary

This chapter has assessed the extent to which the tools and techniques developed in this project are applicable to the problem of specifying and simulating the behaviour of microprocessor control systems. A summary of the conclusions of this assessment is presented below:-

1. The BSL possesses the descriptive power necessary to adequately and naturally capture the crucial behavioural aspects of a wide range of microprocessor control applications.

2. The automatic production of a test environment, which facilitates the validation of behavioural specifications is a vital stage of the development process. It enables the designer to ensure that the behavioural specification has been correctly encoded, before any implementation issues are considered.

3. The macro-assembly language is an appropriate notation for representing behavioural specifications at a lower level of abstraction, and from which microprocessor assembly language code may be easily generated.

4. The component level simulation provides a fast and accurate environment for testing the compatibility of the hardware and software designs. It also offers the potential for judging the correctness of the control strategies employed. This is all achieved without any application-specific hardware construction.

References

6.1 Johnson,C.D.:'Microprocessor-based Process Control', ch.1, pp 1-32, Prentice-Hall, INC., New Jersey 1984

6.2 Welsh,P.H.:'Managing Hard Real-Time Demands on Transputers',Procs 7th Occam User Group Conference, Grenoble, 14-16 Sept 1987

6.3 Marshall,S.A.:'Introduction to Control Theory', Macmillan, 1978

6.4 Kane,G., Hawkins,D., and Leventhal,L. :'68000 Assembly Language Programming', Osborne/ McGraw-Hill, USA, 1981

6.5 Motorola INC. :'MC68020 32-Bit Microprocessor User's Manual', Second Edition, Prentice-Hall, Inc., New Jersey, 1985

6.6 Armstrong,J.R. and Woodruff,G. :'Simulation Techniques for Microprocessors', Procs 14th Design Automation Conf, June 1977

6.7 Pashtan,A. :'A Prolog Implementation of an Instruction-Level Processor Simulator', Software-Practice and Experience, vol 17, no 5, May 1987, pp 309-318

6.8 Everson,U. :'A Test Harness for a Designer's Workbench', MSc Dissertation, Dept. of Computer Studies, Sheffield City Polytechnic, Sept 1985

7. Future Work

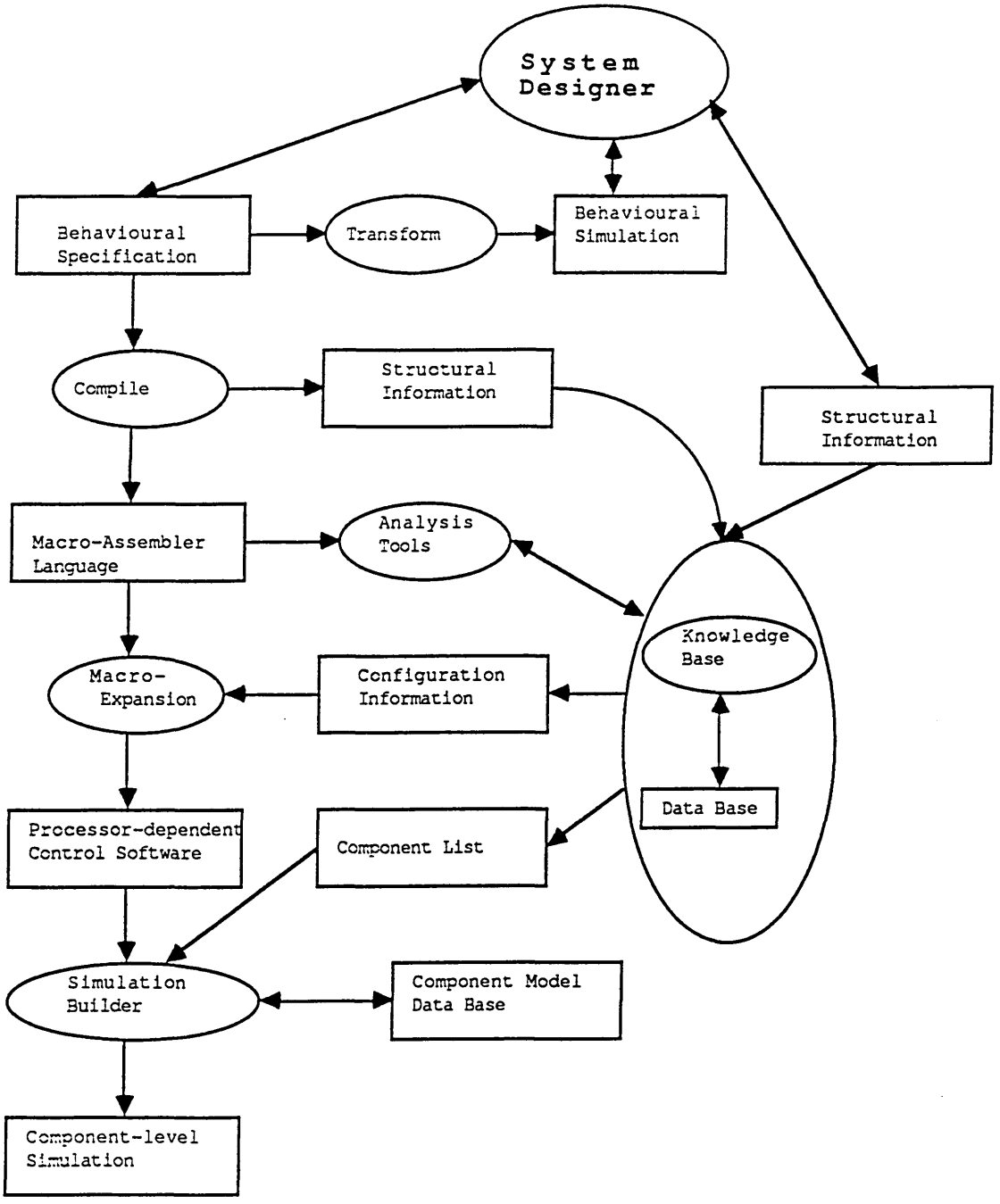
7.1 Integrating and Interfacing Behavioural and Structural Design Tools

7.1.1 Introduction

The work described in this thesis constitutes the behavioural specification and simulation phase of a sophisticated design aid for minimum configuration computer control systems. However, while the complementary structural design phase of the system is under construction, the tools currently implemented may be used in isolation, effectively forming a software specification system for control systems. In such circumstances, the structural aspects of the design must be performed by the system designer and supplied interactively to the software specification system. Eventually though, these tools will need to be integrated with the knowledge-based structural design subsystem. Therefore this section presents a description of the required interaction between the individual design tools. It also suggests a number of design evaluation and exploration strategies which can be applied at the macro-assembly level of design representation.

7.1.2 Design System Operation

Figure 7.1 illustrates the architecture of the system. The transformation of a behavioural specification of an application into occam constitutes the starting point of the design system's operation. The behavioural simulation is executed, and possibly modified, until the designer is satisfied that it responds as intended to the test data supplied. At this stage, the automated design process begins with the compilation of the behavioural specification. As well as producing a macro-assembler level representation of the specification, the compiler extracts certain structural information regarding each of the input



Key: = design information or representation

= design tool or process

Figure 7.1 System Architecture

and output channels in a specification. More specifically this information comprises:-

```
channel name       -- character string
channel direction  -- IN or OUT
channel type       -- usually BOOL or BYTE
access method      -- POLL, INTERRUPT or, for
                      output channels undefined.
priority level     -- integer value, or zero if
                      no priority specified or
                      channel is output.
sampling period    -- time period and units as in
                      specification. Zero if
                      undefined or output channel.
constraints        -- name of associated service
                      routine and maximum allowable
                      time period for processing
                      inputs. Undefined for output
                      channels.
```

Thus a record structure for each channel in the specification can be passed to the knowledge-based structural design tools. The arrival of this information constitutes the starting point of the structural design process. Using this information as a basis, the structural design tools can decide precisely what further details are required concerning each channel before a design may be sensibly attempted. These extra details, which cannot be extracted from the behavioural specification, must be obtained interactively from the system designer. For example, the following characteristics about an input channel may be supplied to the structural design process by the BSL compiler:-

```
CHAN NAME     -  in1
CHAN DIR      -  IN
CHAN TYPE     -  BYTE
ACCESS        -  INTERRUPT
PRIORITY      -  0
SAMPLING      -  UNDEFINED
SERVICE       -  p1  UNDEFINED
ROUTINE
```

The above channel characteristics are not sufficient to enable the selection of a suitable input-output interface to implement this channel. Although this information conveys the fact that the channel accepts eight-bit values,

the source of the value could be a serial, parallel or analogue interface. (In this example however the latter is unlikely, as the channel is interrupt driven and no sampling period is specified. As analogue signals vary continuously, they cannot generally be used to generate interrupts in the same manner as digital signals. Still, a situation could arise in which the channel is to receive values from an analogue interface, and the designer has erred in neglecting to give a sampling period. Such circumstances should be detected and catered for by the structural design tools.) Therefore the precise characteristics of each channel in the specification must be obtained through some kind of goal-driven 'question and answer' session with the designer[7.1]. Further constraints such as cost, power consumption and size can also be acquired interactively at this stage.

When sufficient structural details have been gathered about the application, the structural design may commence. This essentially consists of selecting a compatible set of components from the data base, which can implement the required functionality. This task will be performed using some kind of heuristic search strategy, most likely based upon production systems[7.2].

An important sub-task of the hardware design process is the selection of read-only (ROM) and read-write (RAM) memory components. In other purely structural design systems[7.1], the designer is asked to estimate the amount of ROM and RAM required by a particular application. This however is not necessary when the behavioural and structural design tools are integrated as in Figure 7.1. In this case, the memory sizing information may be automatically extracted from the macro-assembly language representation of the application. Analysis tools could examine the macro-assembler code and estimate the amount of ROM and RAM required. One method of estimating the ROM size is to produce a table giving the average code size needed to implement each macro operation. This could then be used to calculate an estimate of the

total amount of ROM an application needs. The creation of individual tables for each microprocessor in the system's data base would make this process more precise and hence more reliable. Estimating the amount of RAM required by an application is much simpler. This can be done by calculating the number of bytes which the variables in the macro-assembly code will occupy. It should be remembered though that the RAM is also needed to implement the microprocessor's interrupt and subroutine stack. The structural design tools must always allow for this when deciding on the amount of RAM to include.

There does exist the potential for further analysis tools at the macro-assembler level of the design process. By far the most important of these would be a performance estimation tool, which could estimate execution times for individual service routines. This could be performed in a similar manner to estimating ROM size, with the creation of a table of average execution times for each macro operation. An estimation of the total execution time for a particular service routine could then be arrived at through totalling up the average execution times for each of the operations in that routine. Again, the estimate could be made more accurate through the utilisation of processor-specific execution time tables. These would enable the structural design process to request estimates of the execution times of service routines for each of the microprocessors it is considering. The performance estimation tool would then calculate the required estimates and pass back the results, leaving their interpretation to the structural design tools. The performance estimation tool would be a complex utility, as it would have to incorporate strategies for dealing with iterative and conditional constructs. However, ensuring that a processor is capable of satisfying the performance constraints of an application is one of the most important, and potentially one of the most difficult tasks that the design system has to perform. The performance estimation tool suggests one method of solving the problem, and further highlights the

benefits accrued from adopting a fully integrated approach to microprocessor system design.

The structural design process is complete when a compatible set of components has been selected, and the address map and input-output port configurations have been finalised. This information can then be fed into the macro expansion routine, which generates the software required to control the application. It is envisaged that any software optimisation would also take place at this stage. The resulting software, together with the component list, can finally be made available to the simulation builder. This extracts the required component models from the component model data base, and generates the occam code to instantiate the processes in parallel. This can be compiled and executed to give a component level simulation of the complete system design.

In order to evaluate different solutions which are reached, it is expected that a full implementation of the design system would allow a high degree of interaction between individual design tools. This would allow potential solutions to be designed, simulated and evaluated both by the designer and the system itself. The system should also be able to offer explanations of its own design decisions, and allow the designer to intervene in the design process should modifications be necessary. One way of achieving the full integration and cooperation of the individual design tools that would be necessary to achieve such complex behaviour is to implement a supervisory function. This would control the order of the execution of design tools and be responsible for the handling of messages between design processes. The control of the execution of design tools is in itself a significant problem: many of the issues involved and potential solutions to these problems are approached in [7.3].

7.2 Designing Transputer-based Control Systems

The Inmos transputer was originally conceived and designed as a processor for use predominantly in embedded control systems[7.4]. Although the processing power of the transputer has in reality led to many different practical uses, it remains an excellent processor for constructing real-time control systems[7.5]. One of the major advantages that the transputer possesses in this respect is that it incorporates a microprocessor, serial input-output links and RAM on to the same area of silicon. This greatly simplifies the hardware design necessary to implement an application's behaviour. It is possible for the transputer to be utilised across the whole spectrum of control applications. However currently the high cost of the transputer prohibits its use in applications which do not necessarily require its full processing capabilities.

One of the problems encountered in developing control software for transputers is the use of parallelism. In order to implement multiple interrupts and priorities of service routines, communicating concurrent processes must be defined[7.5]. This is a level of software complexity with which most system designers are unfamiliar, and therefore it presents an obstacle in the adoption of transputers. Further the use of parallelism can lead to subtle synchronisation errors which may be difficult to detect in system testing.

However the behavioural specification language defined in this thesis may provide a solution to these problems. The BSL is sufficiently abstract to enable a very wide choice of implementation strategies to be adopted, including the concurrent approach of the transputer. In fact, the behavioural simulation which is directly generated from the BSL can be regarded as a concurrent occam implementation of the required system. Thus the BSL description can be transformed automatically into an occam representation of an application, which incorporates all the necessary

parallelism to implement the application's behaviour. This approach alleviates the need for the designer to be concerned with the difficulties associated with parallel activity, and should ensure that the parallelism introduced cannot deadlock.

The use of transputers to implement control systems would also have an influence on the architecture of the design system defined in section 7.1. The macro-assembly language would no longer be needed, as an existing occam compiler could translate the occam code into transputer assembly language. The compiler could would also produce exact information concerning the size of the code and workspace areas required. The structural design tools would also be simplified. The selection of input-output components would be the same as for any other processor. However, once selected, their connection to the transputer via standard Inmos link adapter chips[7.6] is trivial. Further, many of the hardware/software trade-offs associated with other processors do not have to be performed when using transputers, because the transputer has timing and priority facilities on-chip.

Therefore the BSL may be an appropriate notation to describe the behaviour of transputer-based control systems. If the behavioural specification techniques were integrated with a suitable set of structural design tools, the process of designing transputer-based control systems could be largely automated. Due to the presence of many on-board hardware facilities, it is expected that the construction of such a set of structural design tools would be considerably simpler for transputers than for other competitive microprocessors.

7.3 Summary

The overall operation of an automated design environment has been explained. Specific emphasis has been placed on the interaction between the behavioural and structural

design tools. A number of design analysis and evaluation tools have been proposed which can aid in the structural design process by analysing an intermediate behavioural representation. These highlight the benefits that can be gained by integrating the different design functions into a single design system.

Finally the possibility of automatically designing transputer control systems from behavioural specifications is considered. Areas where the use of transputers would simplify both the behavioural and structural design processes are explained. This aspect of the project is still in the early stages of investigation. However, because of the potentially major design simplifications offered by the use of transputer technology, it could indeed prove to be a most profitable line of research and development.

References

7.1 Bowen,J.A. and Smith,M.F.:'Expert Systems for the Analysis and Design of Microprocessor Applications', Journal of Microcomputer Applications, (1983)6, pp 155-161

7.2 Hayes-Roth,F. :'Rule-based Systems', CACM, vol 28, no 9, September 1985, pp 921-932

7.3 Bushnell,M.L. and Director,S.W. :'ULYSSES - a Knowledge-based VLSI Design Environment', Artificial Intelligence, vol 2, no 1, Jan 1987, pp 33-41

7.4 May,D. :'Occam :Hardware Description Language', Procs IEE Colloq on Software Tools for Hardware Design, London 1983, Digest 98, pp 5/1-5

7.5 Welsh,P.H.:'Managing Hard Real-Time Demands on Transputers',Procs 7th Occam User Group Conference, Grenoble, 14-16 Sept 1987

7.6 Inmos Ltd., :'Transputer Reference Manual', Prentice-Hall Int. UK Ltd., 1988

8. Conclusions

The work described in this thesis represents an attempt to rectify the major difficulties which exist in the development of microprocessor-based control systems. The architecture of an integrated Computer-Aided Design (CAD) system for minimum configuration control systems has been presented. The problems associated with constructing the necessary behavioural specification, synthesis and simulation facilities for the CAD system have been considered, and a collection of design representations and synthesis techniques have been proposed as solutions.

The notation developed for describing the behaviour of control systems enables the systems designer to completely and naturally capture the behavioural aspects of an application. These behavioural aspects are expressed explicitly in an abstract, implementation-independent manner. This creates a large number of possible design and implementation alternatives, ranging at the extremes from a wholly software controlled microprocessor system, to the fabrication of an application-specific integrated circuit. This notation can therefore be regarded as providing an abstract behavioural specification of the required application.

The use of an existing high-level language as a basis for the behavioural specification language has two major advantages. Firstly, it creates a notation which contains many constructs and facilities which are already familiar to microprocessor system designers. Second, and more importantly, it facilitates the utilisation of the available software tools for the base language. With the assistance of a pre-processor to convert the additional specification language constructs into the base language, the specification can be compiled and executed to automatically give a behavioural simulation and test environment for an application.

The synthesis technique developed involves the compilation of behavioural specifications into an intermediate level of design representation. The intermediate design representation is of a sufficiently abstract level so as not to be committed to any specific microprocessor implementation. However, it is also at a suitably low level of abstraction to facilitate the automatic generation of assembler code for existing microprocessors, by means of a relatively simple macro-expansion process. Further, the intermediate design representation should prove useful in the automatic provision of certain structural information that is required by the structural design tools.

The generalised component simulation facility provides an effective method of testing the complete design for a given application, before any hardware construction takes place. The use of occam as a hardware description language for simulating microprocessor systems has provided two major advantages over other approaches. Firstly, the model of concurrency in occam corresponds closely to the actual operation of hardware devices, thus allowing the behaviour of components to be easily modelled. Second, the close relationship between occam and the transputer enables simulations to be implemented on transputer networks. This approach yields a relatively high level of performance, which offers the potential to test simulations in a realistic physical environment.

Thus the integration of the behavioural specification, synthesis and simulation techniques forms a powerful design environment for generating microprocessor control software automatically from an abstract behavioural specification. In combination with a suitable structural design system, prototype designs for control applications could be quickly generated and evaluated, until a design that satisfies the application's requirements is encountered. However, even in the absence of such a structural design system, the behavioural design tools still afford significant

advantages over other existing development languages and techniques for control applications.

Appendix A

Occam and Transputers

An occam program consists of a collection of concurrent processes communicating via point to point communication channels. Each process performs a number of actions. An action may be a set of sequential processes performed one after the other, or a set of parallel processes performed at the same time as one another. Since processes themselves are constructed using other processes, some of which may be executed in parallel, a process may contain any amount of internal concurrency.

All occam processes are built from three primitive processes :-

1. assignment	variable := expression
2. output	c ! e
	output expression e to channel c
3. input	c ? v
	input variable v from channel c

Output is denoted by the symbol ! and input by the symbol ? .

These primitive processes are combined to form constructs.

1. SEQ represents a sequential construct.

```
e.g.  SEQ
        a := b + c
        d := e + f
        comms ! a
        comms ! b
```

2. PAR represents a parallel construct.

```
e.g.  PAR
        a := b + c
        d := e + f
```

3. IF represents a conditional construct.

```
e.g.  IF
        count = timeout
          controller ! device.failed
        count < timeout
          device ! read
```

4. ALT represents an alternative construct.

```
e.g.  ALT
        in1 ? x
          x:= x + 1
        in2 ? y
          y:= y + 1
```

The alternative construct is used when a process needs to input from any one of several other concurrent processes. An input is performed from the channel which is first used for output by another process. The inputting process waits until another process is ready to communicate with it.

Constructs themselves are processes, and may be used as components of another construct:-

```
e.g.    SEQ
          PAR
            a := a + b
            d := e + f
          comms ! a
          comms ! b
```

Conventional sequential programs can be written using variables and assignments, combined in sequential and conditional constructs. All variables, channels and expressions are typed, and strong matching rules are

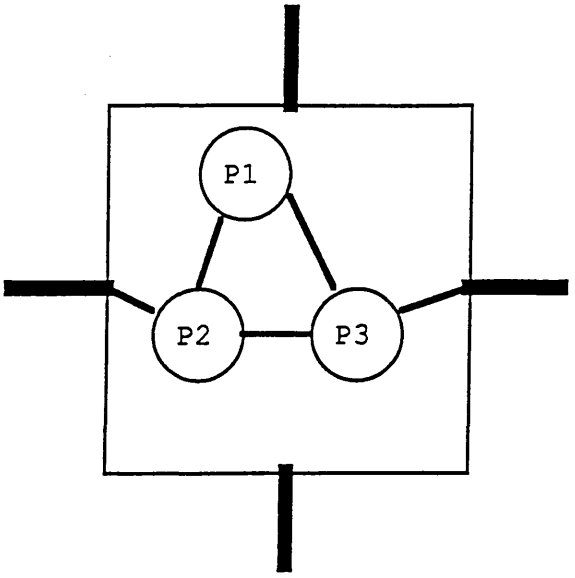
enforced. A WHILE loop can be used to express iterative programs. Concurrent programs must use channels, together with input and output operations to enable communication between processes. These are combined using parallel and alternative constructs. Each occam channel provides a uni-directional communications path between two concurrent processes. Communication only takes place when both the inputting and outputting processes are ready. The value to be output is copied from the outputting process to the inputting process, and both processes then proceed. Thus communication is synchronised and unbuffered, similar to the handshake method of communication used in digital systems.

The transputer is a single-chip microcomputer comprising a 10 MIPS processor, local memory, an external memory interface and four communication links. These links provide fast point-to-point connections between transputers, enabling one transputer to communicate directly with a maximum of four others. The communications links and the processor may all operate concurrently, allowing processing to continue while data is being transferred on all of the links.

There are three main variations of the transputer currently available. These are the (IMS) T212, the T414 and the T800. The T212 is the 16-bit member of the transputer family, and the T414 and T800 are 32-bit processors. The T800, the most recent addition to the range, is essentially a T414 with extra memory and an on-chip floating point unit.

The transputer was designed to efficiently implement occam processes. The occam concepts of concurrency and communication are implemented by the transputer. This means occam programs may execute on a single transputer, with processor time shared between concurrent processes, or on a network of transputers, in which each transputer executes

one or more processes. Communication channels between processes on a single transputer are implemented by memory locations. Communication between processes on different transputers is implemented directly by transputer links. Therefore the same occam program may be implemented on a variety of different transputer configurations. An example of this is given in figure A1.



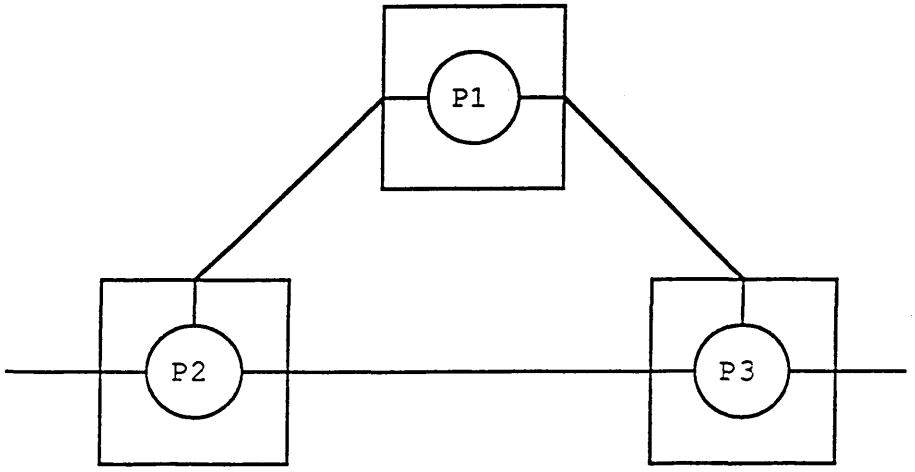


Figure A1 **Distributing Processes on to Transputers**

Appendix B

Behavioural Specification Language Definition

```
<BSL Specification> ::= <title> <specification section>

<title> ::= TITLE <text> :

<specification section> ::= <channel declarations>
                            <routine declarations>
                            <control section> :

<channel declarations> ::= CHAN <direction>
                           <channel id list>  :
                           { <channel declarations> }

<direction> ::= IN | OUT

<channel id list> ::= <channel id> { , <channel id> }

<channel id> ::= <valid variable name>

<Service routine declarations> ::= PROC <routine id> :
                                   <type> <formal parameter
                                          list>
                                   <time constraint>
                                   <routine body> :
                                   {<Service routine
                                          declarations>}

<routine id> ::= <valid variable name>

<type> ::= INTERRUPTABLE | UNINTERRUPTABLE

<formal parameter list> ::= () |
                            ( <typed variable list> ) |
                            (  <typed variable list>
                            { , <typed variable list>} )

<time constraint> ::= <empty> | <integer value> <units>

<units> ::= SECS | MSECS

<routine body> ::= <local declarations> <block>

<local declarations> ::= <empty> | <constant definitions> |
                         <variable declarations>

<constant definitions> ::= VAL <constant name>  IS
                               <constant value>
                               <constant type>  :
                           { <constant definitions> }

<constant name> ::= <valid variable name>
```

```
<constant value> ::= <integer value> | <byte value> |
                     <real value> | <Boolean value>

<constant type>  ::= BOOL | BYTE | INT | REAL

<variable declarations> ::= <array definition>
                            <variable type>
                            <variable list> :
                            { <variable declarations> }

<typed variable list> ::= <variable type> <variable list>

<control section> ::= <global variable declarations>
                      <execution condition>
                      <input structure>

<global variable declarations> ::= <array definition>
                                   <variable type>
                                   <variable list> :
                                   { <global variable
                                      declarations> }


<array definition> ::= <empty> | [ <integer value> ]

<variable type> ::= BOOL | BYTE | INT | REAL
                    | CHAN <direction>

<variable list> ::= <valid variable name>
                    { , <valid variable name> }

<execution condition> ::= WHILE <Boolean condition>

<input structure> ::= <control statement>
                      <event statement>
                      { <event statement> }
                      { <input structure> }

<control statement>::= INTERRUPT | PRI INTERRUPT | POLL

<event statement> ::= <input variable declaration>
                      <input statement>
                      <output action>

<input variable declaration> ::= <empty> |
                                 <variable type>
                                 <valid variable name>:

<input statement> ::= <channel id>     ?
                      <valid variable name>
                      <sample statement>

<sample statement> ::= <empty> | SAMPLE <interval>

<interval> ::= <empty> | <integer value> <units>
```

```
<output action> ::= <assignment> | <output statement>

<output statement> ::= <channel id>    !
                       <routine id>
                       <actual parameter list>

<actual parameter list> ::= ( <variable list> )

<block> ::= <contructor> <statement>

<statement> ::= <constructor> | <expression>

<contructor> ::= <sequence> | <iteration> |
                 <selection>

<sequence> ::= SEQ | SEQ <replicator> <body>

<replicator> ::= <name> = <base> FOR <count>

<name> ::= <valid variable name>

<base> ::= <integer value>

<count> ::= <integer value>

<body>  ::= <expression> | <block>

<iteration> ::= WHILE  <condition>  <body>

<condition> ::=   <conditional expression>  |
                  ( <conditional expression>
                    { <logical operator>
                      <conditional expression> } )

<conditional expression> ::= ( <valid variable name>
                               <comparison operator>
                               <comparison value>   )
<comparison operator> ::=   <>  | = | < | > | <= | >=

<comparison value>  ::= <valid variable name> |
                        <constant value>

<logical operator>  ::= AND | OR

<selection> ::= IF <condition> <body>
                  { <condition> <body> }

<expression> ::= <input expression> | <output expression> |
                 <assignment> | <result statement>

<input expression> ::= <channel name>   ?
                       <valid variable name>

<output expression> ::= <channel name>  ! <output value>

<output value> ::= <valid variable name> | <constant value>
```

```
<assignment> ::= <target variable> := <assignment
                                       expression>

<target variable> ::= <valid variable name>

<assignment expression> ::= <simple assignment> |
                            <complex assignment>

<simple assignment> ::= <inline type>
                        <valid variable name>
                        | <constant value>

<inline type> ::= <empty> | INT | BYTE | REAL

<complex assignment> ::= <simple assignment>
                         <assignment operator>
                         { <complex assignment> }
                         <simple assignment>

<Assignment operator> ::= + | - | * | / | REM | /\ | \/ |
                          >> | << | ~

<result statement ::= RESULT ( <valid variable name |
                               byte value >          )

<byte value>    ::= integer value

<integer value> ::= <digit> { <digit> }  |
                    #<hex digit> { <hex digit> }

<real value>    ::= <integer value>  .
                    <digit> { <digit> }

<Boolean value> ::= TRUE | FALSE

<digit> ::=  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<hex digit> ::= <digit> | A | B | C | D | E | F

<valid variable name> ::= <identifier> | <identifier>
                                         <subscript>

<identifier> ::= <letter> { <remainder> }

<subscript> ::= [ <integer value> ]

<remainder> ::= <letter> | . | <digit>

<letter> ::= A | a | B | b | C | c | D | d | E | e | F |
             f | G | g | H | h | I | i | J | j | K | k |
             L | l | M | m | N | n | O | o | P | p | Q |
             q | R | r | S | s | T | t | U | u | V | v |
             W | w | X | x | Y | Z | z
```

Appendix C

6800 Implementations of Macro Operations

Introduction

This appendix gives a complete definition in Motorola 6800 assembler, of each of the operations and control structures which form the macro-assembly language defined in this project.

In order to keep the definitions concise, macro operations which have already been given are used in certain of the more complex operation definitions. This applies mainly in the definition of INT and REAL operations, which are often described by the two separate applications of the equivalent BYTE and INT operation respectively. Also, for clarities sake, labels of more than six characters are used in definitions. This does not strictly adhere to the rules of 6800 assembler, which does not allow labels of more than six characters.

Note this appendix does not contain definitions for the arithmetic and logic operations which may have a constant value as their second operand. e.g.

```
I-ADD op1, op2  or
I-ADD op1, 10
```

The only difference between the two operation definitions is the use of the immediate addressing mode as opposed to the direct or extended addressing modes which are used below.

Assignment Operations

1. B-ASSIGN op1, FF op1 := FF

```
B-ASSIGN  LDA A #$FF    load hex constant
          STA A op1     store in memory
```

2. I-ASSIGN op1, FFFF

```
I-ASSIGN  LDX  #$FFFF  load index reg
          STX  op1     store in memory
```

3. R-ASSIGN op1, FFFFEEEE

```
R-ASSIGN LDX  #$FFFF   load high order bytes
         STX  op1      store in memory
         LDX  #$EEEE   load low order bytes
         STX  op1+2    store in memory
```

4. B-COPY op1, op2 op1 := op2

```
B-COPY   LDA A op2
         STA A op1
```

5. I-COPY op1, op2

```
I-COPY   LDX op2
         STX op1
```

6. R-COPY op1, op2

```
R-COPY   LDX op2
         STX op1
         LDX op2+2
         STX op1+2
```

Input-Output Operations

1. INPUT channel, operand

```
INPUT    LDA A channel  input from port
         STA A operand  store value
```

2. OUTPUT channel, operand

```
OUTPUT   LDA A operand  load value
         STA A channel  output to port
```

Arithmetic Operations

1. B-ADD op1, op2 op1 := op1 + op2

```
B-ADD    LDA A op1
         ADD A op2
         STA A op1
```

2. I-ADD op1, op2

```
I-ADD    LDA A op1+1
         ADD A op2+1
         STA A op1+1
         LDA A op1
         ADC A op2
         STA A op1
```

(Could also be defined as follows:-

```
I-ADD    B-ADD op1+1, op2+1
         LDA A op1
         ADC A op2
         STA A op1
```

This briefer notation will be used where possible from this point onwards for INT and REAL operations)

3. R-ADD op1, op2

```
R-ADD    I-ADD op1+2, op2+2
         LDA A op1+1
         ADC A op2+1
         STA A op1+1
         LDA A op1
         ADC A op2
         STA A op1
```

4. B-SUB op1, op2 op1 := op1 - op2

```
B-SUB    LDA A op1
         SUB A op2
         STA A op1
```

5. I-SUB op1, op2

```
I-SUB    B-SUB op1+1, op2+1
         LDA A op1
         SBC A op2
         STA A op1
```

6. R-SUB op1, op2

```
R-SUB    I-SUB op1+2, op2+2
         LDA A op1+1
         SBC A op2+1
         STA A op1+1
         LDA A op1
         SBC A op2
         STA A op1
```

7. B-MUL op1, op2 op1 := op1 * op2

```
B-MUL    LDA A op1
         LDA B op2
         DEC B
         BEQ ENDOP     IF op1 * 1 , do nothing
MLOOP    ADD A op1
         DEC B
         BNZ MLOOP
ENDOP    STA A op1
```

8. I-MUL op1, op2

```
I-MUL    LDX X op2
         DEX
         BEQ ENDOP
MLOOP    I-ADD op1, op1
         DEX
         BNZ MLOOP
ENDOP    NOP
```

9. R-MUL op1, op2

```
R-MUL    R-COPY rtemp, op2
         LDA A #$00
         STA A rtemp+3       set decimal places to zero
         R-SUB rtemp, 1
         R-NE  rtemp, 00, ENDOP        if rtemp = 0
MLOOP    R-ADD op1, op1                   goto endop
         R-SUB rtemp, 1
         R-EQ  rtemp, 00, MLOOP
ENDOP    NOP
```

(Notes -- For simplicities sake, this implementation only multiplies op1 by the integer portion of op2. Consequently some precision is lost.)

10. B-DIV op1, op2 op1 := op1 / op2

```
B-DIV    LDA A op1
         LDA B op2
         CMP B #$00
         BEQ END      divide by zero, terminate program
         LDA B #$00   set count to zero
DLOOP    SUB A op2
         BLE RESULT
         INC B
         BRA DLOOP
RESULT   STA B op1
```

11. I-DIV op1, op2

```
I-DIV    LDX op2
         CPX #$0000
         BEQ END
         LDX #$0000
DLOOP    I-SUB op1, op2
         I-LE op1, op2, RESULT
         INX
         BRA DLOOP
RESULT   STX op1
```

(Notes -- see later for defintion of I-LE)

12. R-DIV op1, op2

```
R-DIV    R-EQ op2, 0000, END
         LDX #$0000
DLOOP    R-SUB op1, op2
         R-LE  op1, op2, RESULT
         INX
         BRA DLOOP
RESULT   STX itemp
         ITR op1, itemp
```

(Notes -- see later for definitions of R-EQ, R-LE, ITR
-- The above implementations of division operators assume that both operands are postive. However it is a simple, if labourious task to cater for negative operands and results. It merely entails checking the sign of the operands, deciding on the sign of the result, and then convert any negative operands to their equivalent positive value. The division may then be performed as above.)

13. B-REM op1, op2 op1 := op1 REM op2

```
B-REM   LDA A op1
        CMP A op2
        BLT RESULT
RLOOP   SUB A op2
        CMP A op2
        BLT RESULT
        BRA RLOOP
RESULT  STA A op1
```

14. I-REM op1, op2

```
I-REM   LDX op1
        CPX op2
        BLT RESULT
RLOOP   I-SUB op1, op2
        I-LT  op1, op2, RESULT
        BRA RLOOP
RESULT  NOP             result already in op1
```

15. B-INC op1 op1 := op1 + 1

```
B-INC   INC op1
```

16. I-INC op1

```
I-INC   LDX op1
        INX
        STX op1
```

17. B-DEC op1 op1 := op1 - 1

```
B-DEC   DEC op1
```

18. I-DEC op1

```
I-DEC   LDX op1
        DEX
        STX op1
```

Logical Operations

1. B-AND op1, op2 — op1 := op1 /\ op2

```
B-AND   LDA A op1
        AND A op2
        STA A op1
```

2. I-AND op1, op2

```
I-AND   LDA A op1+1
        AND A op2+1
        STA A op1+1
        LDA A op1
        AND A op2
        STA A op1
```

3. B-OR op1, op2 — op1 := op1 \/ op2

```
B-OR    LDA A op1
        ORA A op2
        STA A op1
```

4. I-OR op1, op2

```
I-OR    LDA A op1+1
        ORA A op2+1
        STA A op1+1
        LDA A op1
        ORA A op2
        STA A op1
```

5. B-NOT op1 — op1 := NOT op1 (1's complement)

```
B-NOT   COM op1
```

6. I-NOT op1

```
I-NOT   COM op1+1
        COM op1
```

7. B-NEG op1 — op1 := (NOT op1) + 1

```
B-NEG   NEG op1
```

8. I-NEG op1

```
I-NEG   I-NOT op1
        I-INC op1
```

9. B-XOR op1, op2 — op1 := op1 >< op2

```
B-XOR   LDA A op1
        EOR A op2
        STA A op1
```

```
10 I-XOR op1, op2

     I-XOR   LDA A op1+1
             EOR A op2+1
             STA A op1+1
             LDA A op1
             EOR A op2
             STA A op1

11 B-SLL op1                    op1 := op1 << 1

     B-SLL   CLC
             ROL op1

12 I-SLL op1

     I-SLL   CLC
             ROL op1+1
             ROL op1

13 B-SRL op1                    op1 := op1 >> 1

     B-SRL   CLC
             ROR op1

14 I-SRL op1

     I-SRL   CLC
             ROR op1
             ROR op1+1
```

Index Operation

```
1. B-INDEX   table, element, temp1

     B-INDEX  I-DEC element
              I-ADD element, table
              B-COPY temp1, element

2. I-INDEX   table, element, temp1

     I-INDEX   I-ASSIGN disp, 0002
               I-MUL    disp, element
               I-DEC    disp
               I-ADD    disp, table
               I-COPY   temp1, disp

3. R-INDEX   table, element, temp1

     R-INDEX   I-ASSIGN disp, 0004
               I-MUL    disp, element
               I-DEC    disp
               I-ADD    disp, table
               R-COPY   temp1, disp
```

Re-typing Operations

1. BTI op1, op2 op1 := INT op2

```
BTI     LDA A op2
        CMP A #$00
        BLT NEG
        STA A op1+1         op2 is positive
        LDA A #$00
        STA A op1
        BRA ENDOP
NEG     STA A op1+1         op2 is neagtive
        LDA A #$FF
        STA A op1
ENDOP   NOP
```

2. ITR op1, op2 op1 := REAL op2

```
ITR     LDA B #$00
        STA B op1+3         set decimal part to zero
        LDA A op2
        CMP A #$00          op2 < 0 ?
        BLT NEG
        STA B op1           op2 is positive
        LDA A op2+1         copy value
        STA A op1+2
        LDA A op2
        STA A op1+1
        BRA ENDOP
NEG     LDA A #$FF          op2 is negative
        STA A op1
        LDA A op2+1         copy value
        STA A op1+2
        LDA A op2
        STA A op1+1
ENDOP   NOP
```

3. RTI op1, op2 op1 := INT op2

```
RTI     LDA A op2+2         copy value
        STA A op1+1
        LDA A op2+1
        LDA B op2
        CMP B #$00          if op2 < 0
        BLT NEG
        BRA ENDOP
NEG     ORA A #$80          then make op1 negative
ENDOP   STA A op1
```

4. ITB op1, op2 op1 := BYTE op2

```
ITB     LDA A op2+1
        LDA B op2
        CMP B #$00          if op2 < 0
        BLT NEG
        BRA ENDOP
NEG     ORA A #$80          then make op1 negative
ENDOP   STA A op1
```

Comparison Operations

The basic algorithm for each of the comparison operations is as follows:-

```
if op1 (operator) op2 then
   SKIP
else
   goto addr
endif
```

1. B-EQ op1, op2, addr

```
B-EQ    LDA A op1
        CMP A op2
        BNE addr
```

2. I-EQ op1, op2, addr

```
I-EQ    LDX op1
        CPX op2
        BNE addr
```

3. R-EQ op1, op2, addr

```
R-EQ    I-EQ op1, op2, addr
        I-EQ op1+2, op2+2, addr
```

4. B-NE op1, op2, addr

```
B-NE    LDA A op1
        CMP A op2
        BEQ addr
```

5. I-NE op1, op2, addr

```
I-NE    LDX op1
        CPX op2
        BEQ addr
```

6. R-NE op1, op2, addr

```
R-NE    I-NE op1, op2, addr
        I-NE op1+2, op2+2, addr
```

7. B-GT op1, op2, addr

```
B-GT    LDA A op1
        CMP A op2
        BLT addr
        BEQ addr
```

8. I-GT op1, op2, addr

```
I-GT    LDX op1
        CPX op2
        BLT addr
        BEQ addr
```

9. R-GT op1, op2, addr

```
R-GT    I-GT op1, op2, addr
        I-GT op1+2, op2+2, addr
```

10 B-LT op1, op2, addr

```
B-LT    LDA A op1
        CMP A op2
        BGT addr
        BEQ addr
```

11 I-LT op1, op2, addr

```
I-LT    LDX op1
        CPX op2
        BGT addr
        BEQ addr
```

12 R-LT op1, op2, addr

```
R-LT    I-LT op1, op2, addr
        I-LT op1+2, op2+2, addr
```

13 B-GE op1, op2, addr

```
B-GE    LDA A op1
        CMP A op2
        BLT addr
```

14 I-GE op1, op2, addr

```
I-GE    LDX op1
        CPX op2
        BLT addr
```

15 R-GE op1, op2, addr

```
R-GE    I-GE op1, op2, addr
        I-GE op1+2, op2+2, addr
```

16 B-LE op1, op2, addr

```
     B-LE    LDA A op1
             CMP A op2
             BGT addr
```

17 I-LE op1, op2, addr

```
     I-LE    LDX op1
             CPX op2
             BGT addr
```

18 R-LE op1, op2, addr

```
     R-LE    I-LE op1, op2, addr
             I-LE op1+2, op2+2, addr
```

Control Structures

WHILE

General Format

```
   WHILE  condition, op1, op2, ENDWHILE

                -- loop body --
             BRA WHILE
   ENDWHILE  NOP
```

SEQ

General Format

```
   SEQ       I-LT, op1, limit, ENDSEQ
                -- loop body --
             BRA SEQ
   ENDSEQ    NOP
```

IF

General Format

```
   IF        condition1 op1, op2, P1

             -- first action --

             BRA ENDIF
   P1        condition2 op1, op2, Pn

             -- second action --

             BRA ENDIF
   Pn        conditionN op1, op2, ENDIF

             -- nth action --
   ENDIF     NOP
```

Subroutine Call

CALL R1

```
JSR R1
```

RETURN

```
RTS
```

INTON

```
CLI     -- clear interrupt
```

INTOFF

```
SEI     -- set interrupt
```

WAIT

```
WAI     -- wait for interrupt
```

POLL

This construct is device dependent, but can be given the general form of:-

```
POLL  address, mask
```

The mask (byte) is ANDed with the result of reading a value from the address (status register), until a result greater than zero is found. This information needs to be supplied by the designer, and eventually by the structural design system.

DELAY

This operation needs to be defined for each processor which has a different clock rate. The definition below assumes a 1 MHz 6800 processor, and generates a 1 millisecond delay:-

```
D1msec   LDA A #$64
DELAY    NOP
         NOP
         DEC A
         BNE DELAY
```

Other time delays can be produced by calling the above routine the required number of times.