The design and implementation of a continuous system simulator

This item was submitted to Loughborough University's Institutional Repository by the/an author.

Additional Information:


Metadata Record: https://dspace.lboro.ac.uk/2134/28108

Publisher: © M.J. Morse

Rights: This work is made available according to the conditions of the Creative Commons Attribution-NonCommercial-NoDerivatives 2.5 Generic (CC BY-NC-ND 2.5) licence. Full details of this licence are available at: http://creativecommons.org/licenses/by-nc-nd/2.5/

Please cite the published version.
This item was submitted to Loughborough University as a PhD thesis by the author and is made available in the Institutional Repository (https://dspace.lboro.ac.uk/) under the following Creative Commons Licence conditions.

For the full text of this licence, please go to: http://creativecommons.org/licenses/by-nc-nd/2.5/
The design and implementation of
a continuous system simulator.

by

M.J. Morse.

A Doctoral Thesis.

Submitted in partial fulfilment of the requirements for the award of Doctor of Philosophy of the Loughborough University of Technology.

# Table of Contents

Abstract.

Acknowledgements.

## 1.0 Introduction and Overview.

1.1 Introduction ................................................................. 1.1

1.2 Simulation: Application and Definition ................................. 1.1

1.2.1 Simulation design ......................................................... 1.2

1.2.2 Continuous system simulation ......................................... 1.3

1.2.3 Discrete system simulation ............................................. 1.4

1.3 The numerical solution of differential equations ......................... 1.5

1.4 Continuous system simulation languages ................................. 1.5

1.5 Parallel processing and simulation .......................................... 1.6

1.6 Simulator and thesis overview ............................................. 1.7

1.6.1 The software simulator ................................................... 1.9

1.6.2 Hardware and Interface considerations ............................... 1.10

1.7 References ........................................................................ 1.11

## 2.0 The interconnection of parallel processing elements.

2.1 Introduction ........................................................................ 2.1

2.2 Classification of parallel processors .......................................... 2.2

2.3 Bit-serial, word-parallel data transmission ............................... 2.3

2.4 Software switch design ....................................................... 2.4

2.4.1 The output router .......................................................... 2.4

2.4.2 The global switch ......................................................... 2.6

2.4.3 The input router .......................................................... 2.6
3.0 Transfer Function Simulation - the Data Processor.

3.1 Introduction ................................................................. 3.1

3.2 The relationship between continuous and discrete functions ...................... 3.2

3.2.1 The delta representation .................................................. 3.3

3.3 Incorporating state variable initial values ................................................. 3.5

3.4 The data processor .............................................................................. 3.9

3.5 Model testing ...................................................................................... 3.11

3.5.1 Errors in digital simulations ......................................................... 3.12

3.5.2 The SI tests .................................................................................. 3.14

3.5.2.1 The SI test results ................................................................. 3.15

3.5.3 The transfer function tests ............................................................ 3.19

3.5.4 Simulation of a third-order transfer function ....................................... 3.23

3.5.5 The effect of changing word and step length ....................................... 3.30

3.5.5.1 Reducing the step length to 0.00005S ....................................... 3.30

3.5.5.2 Increasing the step length to 0.0005S ........................................... 3.31

3.5.5.3 Increasing the mantissa length to 48 bits .................................... 3.31
3.5.4 Reducing the mantissa length to 32 bits ............................................ 3.31

3.6 Conclusion ............................................................................................ 3.32

3.7 References .............................................................................................. 3.32

4.0 Other processing elements.

4.1 Introduction ............................................................................................ 4.1

4.2 Simulation of a chemical process - the multiplier .................................. 4.1

4.2.1 The multiplier ..................................................................................... 4.3

4.2.2 Process simulation .............................................................................. 4.3

4.3 The third-order system revisited - logical processing elements ............ 4.5

4.3.1 The timer ............................................................................................ 4.7

4.3.2 The step-delay unit .......................................................................... 4.9

4.3.3 System simulation ............................................................................. 4.9

4.4 Nyquist diagrams for simulated systems .............................................. 4.11

4.4.1 The waveform generators ............................................................... 4.14

4.4.2 The step generator .......................................................................... 4.15

4.4.3 The event counter .......................................................................... 4.15

4.4.4 The sampler ..................................................................................... 4.16

4.4.5 The system simulation .................................................................... 4.17

4.5 Simulation of an aircraft arresting gear system .................................... 4.20

4.5.1 The non-linear function generator ................................................ 4.24

4.5.2 The comparator/function generator .............................................. 4.25

4.5.3 User-defined processing elements ............................................... 4.25

4.6 Conclusion ............................................................................................. 4.26

4.7 References ............................................................................................. 4.26
5.0 A transputer-based simulator.

5.1 Introduction ................................................................. 5.1

5.2 The transputer ............................................................... 5.1

5.3 Occam channels and transputer links .............................. 5.3

5.3.1 The link adaptors ....................................................... 5.4

5.4 Interconnecting processing elements using link adaptors ...... 5.6

5.5 Performance estimation .................................................. 5.7

5.5.1 The transmit period .................................................... 5.7

5.5.2 The routing process ................................................... 5.12

5.5.3 The execute process ................................................... 5.14

5.5.4 Overall performance .................................................. 5.15

5.6 Simulator design .......................................................... 5.17

5.6.1 The T800 external memory interface ......................... 5.17

5.6.2 Connecting the link adaptors ...................................... 5.18

5.6.3 Configuring the global switch ..................................... 5.20

5.6.4 Results collection ...................................................... 5.20

5.6.5 Downloading PE code .............................................. 5.22

5.6.6 Processor synchronisation ........................................ 5.23

5.6.7 Simulator expansion .................................................. 5.24

5.6.8 Analogue input-output .............................................. 5.25

5.7 Conclusion ...................................................................... 5.26

5.8 References .................................................................... 5.26

Appendix - execute process sheet 1 of 3 ............................. 5.28
execute process sheet 2 of 3 .............................................. 5.29
execute process sheet 3 of 3 .............................................. 5.30

6.0 The simulator-user interface.

6.1 Introduction .................................................................... 6.1

6.2 The software system interface ....................................... 6.1
7.0 Future system developments and conclusion.

7.1 Introduction ................................................................. 7.1
7.2 The software simulator ........................................................ 7.2
7.3 The switching system and system expansion ........................................... 7.4
   7.3.1 Switch-based expansion schemes ........................................ 7.5
   7.3.2 Bus-based expansion schemes ........................................ 7.5
   7.3.3 The switching system in general ........................................ 7.6
7.4 Processor architecture .......................................................... 7.7
   7.4.1 PE terminal configuration ................................................ 7.7
   7.4.2 Multiple-role PEs .......................................................... 7.7
7.5 Process implementation ........................................................... 7.8
   7.5.1 The data processor .............................................................. 7.8
   7.5.2 The other processing elements ............................................ 7.10
7.6 Transputer implementation ......................................................... 7.10
7.7 Other applications ................................................................. 7.11
7.8 Conclusion .............................................................................. 7.12
7.9 References ............................................................................. 7.13

Statement.
Abstract

Depending on the scale of the problem, continuous system simulation is usually carried out on large computer systems or in personal computers running continuous system simulation languages; there is little in between. This thesis describes the development of an inexpensive parallel-processing simulator in which outwardly identical processing elements are configured to digitally simulate continuous system transfer functions and the other components needed to model the computing and control functions of physical systems.

The simulator is peripheral to and controlled from a host computer which holds the configuration code for all the defined processing elements. Before a simulation begins, the processing elements are interconnected through a software-controllable distributed switch and downloaded with the appropriate configuration code. Connected processing elements then exchange computed values during the transmit period (TP) and compute new values during the execute period (EP). A design aim is that the step length should be 100 microseconds so that combined duration of TP and EP should be less than that period.

The design of the switch and the processing elements based on transputer-family products is described. Also discussed is the exhaustive testing programme which was carried out with a software simulation of the simulator, particularly with regard to the data processor which simulates transfer functions. A discussion of the trends in man-machine interface design and artificial intelligence research which impact the simulator is also included and the thesis ends with an assessment of the work and some suggestions for possible future development.
Acknowledgements.

This project would never have reached a conclusion had it not been for Dr. William (Bill) Forsythe. I would have given up long ago had he not persuaded me (several times) that “there was not much left to do”, or had he not given so readily of his time. Thanks Bill!

Also to John McKay, my ex-Head of School, who did everything that he could to smooth the way, I am very grateful.

And finally to my wife, Lesley, who understood, but sees the completion of this project as the end of whatever it was that made me work late.

Thank you all!
1.1 Introduction.

The purpose of this first Chapter is to present an overview of the research programme and to define some terms which are used throughout the thesis. The other Chapters explain how particular aspects of the programme have been, or will be, implemented. A great deal of hardware and software which has been developed at various stages during the project is referred to but not discussed and reference to some of the background literature research has been omitted. It goes without saying that the work which is reported would not have been possible without this background material.

1.2 Simulation: Application and Definition.

Until quite recently, simulation was an activity embraced by the enthusiast but viewed with suspicion by a wider audience. It is a statement of the obvious that all simulations simulate something, but too often, inadequate planning, design, implementation or testing has produced simulations which were eventually seen to behave quite differently to the simuland; hence the reason for scepticism. It has been said that simulation tops the extensive list of the ways in which computers have been misused [1], and, by some, it is considered to be marginally useful, probably not cost effective and at best, of academic interest only [2].

This somewhat jaundiced view is much less prevalent today because simulation has matured as a scientific discipline and because better simulation design aids are available. Indeed, there is much evidence of increasing interest in system simulation, the main application areas being:

- In the study of system behaviour before systems are constructed and commissioned. Accurate simulations indicate whether the system will work, how well it will work and
whether any potentially unstable operating conditions can be innocently induced. There is also the opportunity to try "what if ... " options.

- In providing realistic training experience where training on the simuland is impractical, hazardous or expensive.
- To predict with some degree of certainty, phenomena before they occur so that, hopefully, contingency plans can be enacted to mitigate any harmful effects of the phenomena.

The majority of military applications span the first two categories, particularly aircraft, missile and spacecraft simulations and trainers. Also included are simulations of industrial processes in the nuclear, chemical, transport and similar fields. In the third category there are the established ecological and economic simulations which are used to predict the progress of less well-defined processes.

The simulator developed out of this research programme is concerned mostly with the first type of application and partly with the third. In its unexpanded form its training use is probably limited to educational experience of simulation but if expanded, it could be used in the construction of much larger training simulators.

As systems have become more complex and the cost of physically modelling them more expensive, there has been a corresponding increase in the sophistication of computing hardware and software. This has made computer simulation not only technically feasible, but in many instances the only practicable way of evaluating design ideas and operational practices. One of the most concise definitions of simulation is that it is 'the use of a model to perform experiments to predict the probable behaviour of a system or situation under study' [3]. When the models are set up in computers, the practice is called computer simulation and experiments are carried out on the computerised model. However, it is very common for the terms, simulation, computer simulation and modelling to be used interchangeably - a practice which will be adopted throughout this thesis.

1.2.1 Simulation Design.

Simulation experiment design is an iterative loop beginning with the planning stage, during which the aims of the simulation are stated, and then going on to model definition, computer-
sation, model testing, results analysis and then back to redefinition of the aims or the model. There are two aspects of testing which are usually referred to as validation and verification. Verification is determining that the process of committing the model to software (and/or hardware) is accurate; there are no bugs. Validation is checking that the (computerised) model behaves as the system being modelled is expected to behave. Both are essential steps to producing a reliable simulation. The eventual aim of course, is to use the simulation to study the problem it was designed to solve. If iterations of the design loop are carefully planned, the risk of inaccurate or misleading results is minimised.

The progress of this research led to the design of a simulator rather than a simulation but the same iterative design procedure was adopted. For each new simulator component, simulations which used that component, and had known outcomes, were run and the operation, or sometimes definition, of the component was altered until the required (or achievable) predictable performance was obtained.

The simulator is to be used to, inexpensively, simulate continuous systems with real-time operation as an option. This concentration on continuous systems occasioned research into the distinction between continuous and discrete system simulation and the existing ways of simulating continuous systems, with or without the real-time option.

1.2.2 Continuous system simulation.

Continuous systems are described by sets of one or more differential and algebraic equations. If these are relatively simple, then any of the mathematical procedures for solving them can be used without the need for simulation. The solution is an expression which may be evaluated for different operating conditions using a computer program, but other than that, there is no role for the computer to play.

Simulation is needed if,

1. the system equations have no analytical solution, or

2. the solutions are too difficult to calculate, or
they do not easily demonstrate system behaviour, or

If the equations are non-linear in some respect, or

perhaps for a variety of other reasons.

Analogue computers are specifically designed to solve differential equations so that one way of studying continuous systems would be to patch together analogue computing elements and observe system behaviour on an oscilloscope. Alternatively, and more likely, a digital computer would be used.

Discrete (or digital) simulation of continuous systems amounts to representing continuous changes as a series of discrete steps. At each step, new values for all system and state variables are calculated and then the simulation time is advanced by the step length, h. This process continues until the program eventually terminates. The step length is usually chosen to match the dynamic behaviour of the system being modelled; too large an h will mean system changes not being registered and too small an h will mean longer than necessary simulation times and errors introduced because of the limitation of finite word lengths.

Time is an important consideration when discussing simulation and simulators. When simulations are run in conventional, sequential computers, reducing the step length or increasing the scale of the problem both increase the simulation time because more arithmetic has to be done. The various continuous system simulation languages and the many algorithms for numerically solving differential equations do not address the problem of real-time simulation; that is almost always addressed by employing additional, or more powerful, hardware.

1.2.3 Discrete system simulation.

Whereas continuous systems can be simulated using analogue or digital simulators, discrete systems are always simulated digitally. Discrete systems are systems in which a sequence of (discrete) events cause the system to change state, but between events the system is static. Models (simulations) of the system state, and the events which change it, are founded in queueing theory, probability theory and statistics. Discrete system simulation seems to be the
majority interest in the simulation community but it is probably not relevant here. As yet, the use of this simulator to study discrete systems has not been considered.

1.3 The numerical solution of differential equations.

Rather obviously, the solution of differential equations involves integration. Numerical integration is an approximation process in which a solution, or value, at time \((t + h)\) is calculated from the value at \(t\) and some function which combines the step length with function values and derivatives. The expression has the form,

\[
y(k + 1) = y(k) + F(h,y)
\]

Where \(y(k + 1)\) is the \((k + 1)^{th}\) sample output,

\(y(k)\) is the \(k^{th}\) output, and

\(F(h,y)\) is a function of \(h\), and a sequence of sample values and/or their derivatives.

The literature, of which [4], [5] and [6] are a small sample, describes the derivation and properties of the many forms of \(F(h,y)\) which differentiate one numerical integration technique from another. What all the methods have in common is that \(n^{th}\) order equations are solved by reducing them to \(n\), first-order equations. The more common algorithms have been coded and are usually available as procedures in software libraries or are otherwise obtainable [7] [8]. For engineering applications, the fourth-order Runge-Kutta method (RK4) is probably the most widely known and used [5].

1.4 Continuous system simulation languages.

A catalogue of simulation software is published every year in the October edition of Simulation, the Journal of The Society for Computer Simulation. The 1988 list contained 190 entries, of which 22 related exclusively to continuous system simulation on personal computers (PCs) or workstations. A further 6 entries described products which could be used for continuous or discrete (or combined) system simulation and the rest were concerned with discrete system simulation only, or (a few) with continuous system simulation on large computer systems.
Continuous system simulation languages are either equation-based or block-diagram based. Equation-based languages, such as ACSL and SYSL, have program statements which effectively describe analogue computer devices. For instance, the statement,

\[ X = \text{INTGRL} (1.0, \text{XDOT}) \]

specifies an integrator with input XDOT, output X and an initial value of 1.0. Other program statements define XDOT and the analogue and logical computing functions which complete and control the simulation. Connections between computing units (lines of code) are implied by the use of variable names and not by the order in which statements are listed. It is the responsibility of the language compiler to correctly order the execution steps.

In block-diagram languages, such as SIMBOL and TUTSIM, systems are described as connected blocks on a terminal screen. Transfer functions are assigned to the blocks and the simulation is constructed much as a control engineer would construct a system diagram on paper. In both equation-based and block-diagram based languages, the user can change the default settings of execution options, the integration method and the step length. There are also commands to specify the format and frequency of output and when, or under what circumstances, a simulation should terminate. The two approaches to software simulation are quite different however, although one product, EASE + ACSL from EXPERT-EASE systems, takes a block diagram input and delivers ACSL program statements to the compile/run environment. This is in recognition of the fact that most users see graphical input as more acceptable than listing equations.

Simulation languages in PC-type environments are quite useful for gauging approximate system response but, as previously discussed, they are not intended for time-critical applications. Real-time operation means that simulators must respond to input changes as the simuland would respond; which in turn means efficient analogue interfaces and very fast, or parallel, hardware and software.

1.5 Parallel processing and simulation.

Sequential processing of most digital computers is not well suited to modelling concurrent events. [9]. Continuous systems are essentially concurrent since gradual, but continuous,
changes of state occur everywhere in the system all of the time. A more satisfactory way of modelling continuous systems is therefore to use parallel processing elements with each element representing one of the blocks of a block-diagram language representation [10]. By restricting the processing load on each block, the simulation can run with a small step length and thus appear to act as a pseudo-analogue machine.

Of course, if sufficient computing power is available then the blocks need not be hardware so long as the system can switch between tasks sufficiently rapidly. The advantage of parallel operation, even if achieved with large digital computers with fast context-switching software, is that simulations run faster even if real-time operation is not the aim. There are several examples [11][12][13] of very large and expensive computers being used in this way.

For smaller scale simulation problems the computational burden on a single controlling host can be eased by distributed parallelism using a larger number of less powerful machines [14][15]. For somewhat smaller scaled problems, digital implementations of analogue computer components [16] and digital differential analysers [17] have been used in the past, but if there are any modern equivalents, they are a closely guarded secret.

1.6 Simulator and thesis overview.

This research concerns a digital (hardware) implementation of a block-diagram based continuous simulation language. It is a parallel processing system in which outwardly identical processing elements (blocks) are connected together by a distributed connection system comprising a simple global switch and a local switching function at the input and output of each processing element. This arrangement is non-blocking since any device output can be connected to any free device input. It is also a one-to-many facility with a means of reducing the electrical load on any output should multiple connections beyond its fan-out capability be needed. The connection pattern is established at the start of a simulation and maintained for its duration. Dynamic re-arrangement is not, at the moment, an option.

As presently conceived, the basic simulator comprises 16 processing elements each with 8 input and 8 output pins. The inputs and outputs are routed through the local switching system
and then connected together by the global switch. The processing step length is divided into an execute period (EP) and a transmit period (TP). It is during TP that connected elements exchange numerical and logical data before the next EP. In EP, all elements compute a new (step) output value according to their function. A design aim is that the step length should be 100 microseconds so that the combined duration of EP and TP should be less than that. Chapter 2 describes the implementation of the switching system and several of the alternatives which were considered.

Processing element type is determined by the execution (EP) code which is downloaded to every processor from the host computer at system initialisation. When a new kind of processing element is first defined, the verified and validated execution code is stored in the host file system under a suitable name. When a processing element of that type is required, the named code block is recalled from the file system and installed in a processing element.

The software description of processors classifies them as being either analogue or logical. Analogue processors, such as data processors and multipliers for instance, deal with numerical quantities relating to simulated system functions. Logical processors, such as timers and counters, deal with logic levels which control and regulate the operation of the analogue processors. The most important of the analogue processors is the data processor which digitally simulates a continuous system transfer function. The development of this device and the validation procedures which were used to investigate its operating limits are discussed in Chapter 3.

Chapter 4 is concerned with the definition of other processing elements, both analogue and digital. The technique used to identify what additional elements might be needed in a fully operational simulator was to consider a specific simulation and then define general-purpose processing elements to suit that application. Since the outcome of the experiment is known, the new processing element operation and specification can be refined by exercising the iteration loop discussed above. It should be noted that the number of different processing elements which could be defined is limited only by the ingenuity of the designer and the capacity of the storage system. The list of elements discussed in Chapter 4 is by no means exhaustive.
1.6.1 The software simulator.

The aim of this project has always been to produce a hardware simulator for continuous system simulation. The original development effort [18] resulted in the production of a data processor [19] made from AMD bit-slice components, but this was limited in that it operated only as an integrator simulating the transfer function $1/s$. During the second stage of the development, the specification of the data processor was changed so that it could simulate higher order transfer functions. Questions remained however about the maximum order of the transfer function, the word length, the step length, the input/output configuration and several other important parameters. As a result, hardware design specifications were changed more frequently than they could be completed.

In order to make progress and allow some degree of flexibility, a software simulation of the simulator was produced. This is written in Turbo Pascal and runs on an IBM-compatible PC, which is the most likely host machine in a parallel processing system. All the simulation experiments reported in this thesis were carried out using this software simulator.

The software comprises a main (executive) program, a declarations file and four procedure files which deal with different aspects of the simulator. The four procedure files are:

- **cons.pas** - This is a collection of procedures which prepare and maintain the global connection matrix. During TP, reference is made to this matrix to see which outputs are passed to which inputs. In hardware the connection matrix is the global switch.

- **bits.pas** - this contains the simulation and processing element initialisation code as well as the code blocks which identify the operating modes of the processing elements. In a functional (hardware) simulator the initialisation code would run in the host. In response to user input it would retrieve appropriate code blocks from disk, initialise the parameters associated with that block and then transfer it to the intended processing element. In the software simulator a processor is a variant record and a simulation is an array of such records.
• **fpilib.pas** - This contains all the floating point routines which were used when investigating the effect of varying the mantissa length. The word format conforms with the IEEE P754 [20] standard and the main routines are the usual arithmetic procedures. When word length is not an issue, this library is usually omitted and the default word format (40-bit mantissa, 8-bit exponent) used. The software simulator runs significantly faster without the floating point routines.

• **prints.pas** - There are no restrictions on the number of processing element output pins which can be nominated for results display. The correspondence between pins and the signals they carry is, of course, assumed. A set of co-ordinates (processor number/pin number) are constructed from user input and passed to this group of procedures at the print interval. Values read from the selected output pins are then printed on the screen. In the software simulator the print format is fixed.

The declaration file is called decs.pas and contains the software description of all the processing elements and all global variables. It is included in every compilation. Two versions of the package were produced, one which used fpilib.pas and one which did not. In one case numeric quantities were defined as a floating point number thus,

```pascal
fpnumber = record
    mant : mantissa;
    exp : exponent
end;
```

and in the other they were declared as type real. In Turbo Pascal real numbers have a 40-bit mantissa and an 8-bit exponent and are manipulated by the numerical procedures provided with the compiler. The data types 'mantissa' and 'exponent' are arrays of bits and the procedures in fpilib.pas manipulate variables of this type when arithmetic results are calculated.

1.6.2 Hardware and interface considerations.

Research into possible hardware implementations of processing elements concentrated on adapting transputer hardware so that the connection requirement could be satisfied, and testing to ensure that the target step length could be achieved. The global switch was also de-
signed around transputer parts. Chapter 5 describes the progress of this research and the definition of a simulator which meets the performance specification and can be operated in the intended manner.

Chapter 6 describes the user interface with the software simulator and some work which was carried out to produce a more acceptable user interface with the hardware simulator. To judge by the volume of published papers on the subject, there is a great deal of work going on into interface design for simulators, and computer systems generally. The view is often expressed that if systems are to be acceptable to non-expert users, they must be easy to use and provide a wide range of easily-identified user options. How that view is affecting the modern approach to simulation and simulator design is also discussed in Chapter 6.

Finally, Chapter 7 is a critical resume and appraisal of the research programme, considering its achievements and what additional development work needs to be done.

1.7 References


2.0 The Interconnection of Parallel Processing Elements.

2.1 Introduction.

Parallel processing is defined [1] as 'an efficient form of information processing which emphasises concurrent events in the computing process.... [It] demands concurrent execution of many programs in the computer.'

The use of the word efficient in this definition is an indirect comment on the difficulty of organising parallel processors. Partitioning a computing task into a number of parallel processes, synchronising those processes and allowing them to communicate effectively [2] are just some of the issues which determine parallel processing efficiency. In theory, a parallel computer with n processors is potentially n-times faster than a single processor working on the same problem. In practice, it is seldom possible to organise parallel programs so that communication and computation can overlap and so the speed-up ratio is always less than optimum.

A parallel processor which does not have the problems listed above is, of course, the analogue computer. The fact that analogue computers have largely disappeared from the computing scene is because they have other, inherent, operating difficulties which are characteristic of analogue circuits. If these could have been overcome, the analogue computer would have been (and conceptually is) an ideal parallel processing system for simulating continuous systems. Problem partitioning is obvious and dictated by the nature of the processing elements (PEs), communication is continuous and concurrent with PE operation, and interconnections are point-to-point so that communication bottlenecks do not exist.

Most current parallel processing activity seems to be concerned with using digital PEs to solve 'large', computationally-intensive problems [3]. Typical application areas are complex signal processing, artificial intelligence and similarly scaled scientific and engineering problems. Apart from the digital implementation of an analogue computer introduced by Membrain
Dynamics in the mid-1970s [4], there has been relatively little commercial or industrial interest in building digital representations of analogue computers. Real-time continuous system simulation has been tackled with large (and expensive) multi-processors such as the Applied Dynamics AD-10/AD-100 computers. These have a limited number of dedicated processing blocks (five in the case of the AD-10) and are programmed in a high-level language with full large-system support facilities. They have little in common with the ethos of analogue computing!

The aim of this research is to define a number of digital processing elements which have a much closer affinity with analogue computing components. The system is to be used for the analysis and simulation of continuous systems and a very important aspect of the study concerns the design and implementation of the interconnection network.

2.2 Classification of parallel processors.

According to Flynn [5], computer systems may be classified as either,

- Single Instruction - single data stream (SISD)
- Single Instruction - multiple data streams (SIMD)
- Multiple Instruction - single data stream (MISD)
- Multiple Instruction - multiple data streams (MIMD)

The classification therefore depends on the distribution of the instructions and data throughout the computer system. SISD computers have the conventional Von-Neumann architecture and are parallel only in as much as certain operations, such as instruction fetch and execute, may overlap. Array processors are examples of SIMD computers; a number of processing elements synchronously perform the same function on different data under the control of a host computer. For instance, in image processing systems many processors operate in the same way on different pixel data to emphasise or obscure some aspect of the image. Matrix multiplication, in which several processors compute the individual matrix element products, is another example.
In MISD computers, the same data are passed to every processing element, but each acts on it differently. Evidently [1], there are no examples of MISD parallel processors. In contrast, there are said to be very many examples of MIMD systems. Also called multi-processor systems, MIMD systems comprise several differently programmed processors each working on a different block of data. At some point in the processing cycle they communicate computed results through a global interconnection network. The ratio of data processing to data communication times is known as the granularity of the parallel program. Fine-grained (tightly coupled) programs spend relatively more time communicating than course-grained (loosely coupled) ones. Most MIMD computers are said to be loosely coupled.

Although there are other ways of classifying parallel processors [6][7], the system proposed here does not fit neatly into any category. It most closely matches the definition of a MIMD machine since every processor can be configured differently and therefore executes different code on different data, but there is a high degree of processor interaction which means a fine-grained problem subdivision. As noted earlier, this is not characteristic of MIMD system generally. Alternatively, it might be considered as an elaborate, non-linear, pipeline processor in which processors act differently on data passed between them along the pipe. At each step of the computation, PEs synchronously transmit one or more computed results. Connected PEs accept these synchronised arrivals and process them to generate the next output. A processing cycle occupying \( h \) seconds is therefore divided into an execute period (EP) and a transmit period (TP). The aim is to minimise \( h \) (without compromising accuracy) so that the system more closely approximates an analogue system, to minimise TP by providing a minimum delay connection network, and maximise the work done in EP so that the fewest number of PEs are used in a particular simulation.

2.3 Bit-serial, word-parallel, data transmission.

An interconnection network commonly used in multi-processor systems is the universal, parallel bus. All processors, memory modules and other system resources are connected to the bus and all data transfers between units take place over it. Bus arbitration logic resolves any conflict if two or more sources wish to transmit at the same time. As far as this application is concerned, a common bus is totally unsuitable since all PEs always transmit at the same time and so multiple conflicts must always occur. An interconnecting ring is only marginally more suitable, and using multiple rings, as proposed in some packet switches [8], is unlikely to be significantly better.
If each processing element has n inputs and n outputs and there are p processors in the simulator, the requirement is for a non-blocking interconnection system having p*n inputs and p*n outputs. Ideally, each input/output would be a port capable of handling all bits of a word, in parallel. In practice, word-parallel switching is not possible because it makes such a high demand on I/O bandwidth in the switching network. In VLSI chip design, which has much in common with the design of simulator PEs, the problem has been avoided by adopting bit-serial techniques, and algorithms and processing structures have been developed to meet the need. The literature [9] refers to semi bit-serial systems which use 'global' bit-serial data flow but parallel data flow within a computing device. This digital simulator is therefore bit-serial. During TP, each transmitted result is a serial bit stream which is passed through the interconnection network to the input of selected PEs. The arrivals are collated and then processed in parallel during the following EP. The interconnection system therefore deals with serial, not parallel, data streams.

2.4 Software switch design.

To test design ideas for the eventual hardware simulator, a software model was written. This included a switching arrangement which was intended to imitate the operation of a physical switch and included several elements which could be realised in hardware. These elements are described in the following paragraphs.

2.4.1 The output router.

The design of a switch is largely dictated by the nature of the PEs which it interconnects. All processing elements have 8 inputs and 8 outputs and each output can be connected through the switch to one or more inputs. The output signals need not be identical and in the case of a data processor, which simulates continuous system transfer functions, several different outputs were defined. It is conceivable that a data processor may be required to produce more than 8 different outputs. Similarly, other types of processor, including those yet to be defined, may have fewer than 8 different outputs. When the switch was being designed therefore, it was not possible or desirable to state which signals would be allocated to which device outputs or how many processors outputs there would be. To allow this degree of flexibility, the distinction between the processor outputs and the device outputs was more sharply drawn.
Processor output signals are assigned to particular processor output pins. There may be more or less processor outputs than the 8 device outputs, but between the two there is an output switch, or router, which is programmed to connect selected processor outputs to particular device outputs. These device outputs are interconnected by the global switch with the setting of the local output router determining the number (to a maximum of 8), nature and distribution of active outputs. Figure 2.1 shows this arrangement.

![Diagram of processor output signals and output router](image)

Figure 2.1 The location and function of the output router.

The router distributes the processor outputs across the device output pins behaving as an \( m \times 8 \) crosspoint switch allowing one-to-many connections between processor and device outputs. In the software simulator \( m \) is set at 8 but this an arbitrary limit which can be changed without significantly affecting the design.

An added benefit of providing an output router is that when particular processor signals have to connect with many inputs, the same signal can be routed to several device pins and the buffering effect of the router effectively increases the fan-out of the signal. Another benefit is that distributing the switching function in this way makes the design of the global switch easier.
2.4.2 The global switch.

The global switch comprises 8 identical switch planes. Plane 1 interconnects any device output pin 1 with pin 1 at one or more device inputs. Plane 2 does the same for pin 2, and so on to 8. If the basic simulator has only 16 processors therefore, the global switch comprises 8, 16-by-16 one-to-many connection matrices. If provision is made to cascade simulators when 16 PEs are insufficient for a particular application, a minimum requirement would be two extra output signals from each simulator - one 'analogue' (value) bit stream and one control signal which could interconnect through a bus (4 bus lines per simulator) or through another switch. These extra signals would, most conveniently, be assigned to particular bit planes in the switch structure.

Although a number of possible hardware configurations have been briefly examined, the design of an efficient expansion arrangement has been left for further study.

2.4.3 The input router.

The outputs of the global switch are hard-wired to the appropriate device inputs of every unit. Thus, with the combination of output router and planar global switch, an arrangement can be found which connects any processor output to any device input. The configuration software traps any invalid connection request which, if granted, would result in two outputs being connected together at the same input. However, an additional requirement is that certain processor inputs should have special significance. In particular, processor inputs 7 and 8 are logical (control) inputs which set the operating mode of the PE according to the truth table shown below.

<table>
<thead>
<tr>
<th>Processor Input</th>
<th>Mode Selected</th>
</tr>
</thead>
<tbody>
<tr>
<td>7  8</td>
<td>COMPUTE</td>
</tr>
<tr>
<td>0  0</td>
<td>HOLD</td>
</tr>
<tr>
<td>1  1</td>
<td>RESET</td>
</tr>
</tbody>
</table>

In COMPUTE mode the PE is active and executes its down-loaded configuration code during
In HOLD mode execution is suspended and the PE outputs are maintained at values which were registered at the end of the computational step immediately before this mode was selected; and in RESET mode the PE outputs are reset to some defined starting state and held there.

It is possible that an active PE may generate more than one control signal which, for instance, switches other PEs between COMPUTE and HOLD modes. If such was the requirement, pin 8 on the controlled device could be maintained at logic level 0 with the level at pin 7 exercising the control. However, this would mean that the controlling PE would need to route more than one control signal to the device pin 7 which is clearly not possible. Similar cases can be cited in which, because some processor pins have special significance, the necessary connections cannot be made. In short, the switching system is partially blocking.

To overcome this problem PEs are provided with input routers which are positioned between the device input pins and the processor input pins. The arrangement is the same as shown in Figure 2.1, but now relating to the PE input. It allows any device input to be routed to any processor input thereby overcoming the blocking feature of the previous system.

2.4.4 Router implementation.

At first sight it may appear that additional switching stages in the PEs are an over-provision which could have been avoided if a less restrictive global switching strategy had been adopted. In fact the input and output routers need be nothing more than look-up tables which are filled in when the connection pattern is established. Serial bit streams concurrently arrive at the device input registers and, at the start of an EP, the numbers or logic levels are read in parallel and in order, by referring to the input look-up table. At the end of EP, the computed results could be placed in a particular device output register with reference to the output look-up table. The router switching is therefore logical rather than physical, although, as will be discussed, potentially faster hardware implementation is possible.

In software, the input/output circuitry of every processor is defined as follows:-
Inroute : array [1..pinsperunit] of 1..pinsperunit;

Invalue : array [1..pinsperunit] of datatype (float or real);

outvalue : array [1..pinsperunit] of datatype (float or real);

outroute : array [1..pinsperunit] of 1..pinsperunit;

'pinsperunit' is a program constant currently set at 8. Figure 2.2 shows how these data structures are related.

Figure 2.2 The I/O circuitry of every processing element, regardless of type.

On the output side of a PE the expression defining the switching of 'outvalues' to selected device output pins is,

\[ o_{val} := \text{outvalue} \left[ \text{outroute} \left[ \text{pinno} \right] \right] \]

If pinno is 1 and stage 1 of the output router contains the number 3 (the first entry in the lookup table is 3), processor output 3 is routed to device output pin 1. The distribution of pro-
cessor outputs (outvalues) will be known so that the routing action is defined.

At the input, the corresponding expression is,

\[
\text{Invalue [inroute [pinno]]} := \text{ival};
\]

If now pinno is 3 and stage 3 of the input router contains the number 7, it means that the arrival on device pin 3 is routed through to processor input pin 7. The default settings of both routers is straight-through.

2.4.5 Global switch implementation.

In the global switch all connections are referenced to an output. Any output can connect with a maximum of 'maxunits' (16) inputs but if a simulation uses only, say, 4 PEs, then only 4 connections - one to each of the PEs - are allowed. Connections are specified by setting the contents of the global array variable 'matrix'. An entry in 'matrix' is interpreted as,

\[
\text{matrix [x,y,z]} = \text{destination unit number}
\]

Where x is the source PE number.

\[
y \text{ is the device output pin at that source.}
\]

and z is the connection number.

If, for instance, matrix [1,2,3] contains 4, it means that the third connection to device output 2 on PE1 is taken to device input pin 2 on PE4; the first two connections to the same output are to other inputs on different PEs. Zero array elements mean no connection at that output.
During system initialisation the user is prompted to enter x, y and the destination PE number for each connection. Invalid entries are trapped and correct re-entry is invited or the entry can be cancelled. When all the connections have been made, signified by entering RET to the source unit prompt, the global connection matrix is displayed and, once again, users may make any necessary changes. Once accepted, the matrix is used during TP to pass device outputs to the proper inputs. Router settings are declared when individual PEs are configured.

As explained before, the routines which set-up the global switching matrix are contained in a file called cons.pas which is linked the other simulator software.

2.5 Hardware interconnection systems.

2.5.1. Design issues.

The classification of switching networks [10] takes account of their operating mode, control strategy, switching method and network topology. The operating mode of a switch refers to the way in which connection paths are established through the switch fabric. A switch may operate synchronously or asynchronously. By implication, synchronous operation means that data transfers between PEs will also be synchronous and there is the possibility of the existing connection pattern being broken down and a new pattern established before the next synchronous event. This is an operating scenario most closely associated with SIMD array processors. Asynchronous connection establishment is most applicable in multiprocessor (MIMD) systems in which PEs may, for example, access data in common memory at intervals unrelated to the activity of any other PE.

The technique used in this digital simulator is synchronous, although the term refers to the data transmission through the switch rather than connection establishment. The contradiction is that, as previously discussed, it is best described as a MIMD machine.

Control of a switch may be centralised or distributed. Although the switching function itself is distributed, the control strategy used in this simulator is centralised because all switching elements are set by user instructions issued through the host computer. Distributed control implies a degree of autonomy at switch nodes which is more characteristic of routing/switching
strategies used in wide area computer networks.

Connections between PEs in the digital simulator are established at the start of a simulation and maintained for its duration. Each connection is dedicated to just one device input-output pairing. In telecommunications networks, this type of connection would be called circuit-switched. It contrasts with packet-switched connections in which the same physical medium is shared by a number of calls. Call identification is logical rather than physical but packet switched data are always likely to be delayed when packets associated with different calls pass over the same link. Since queueing delays cannot be tolerated in this application, circuit switched connections are mandatory. Circuit and packet switching are referred to as different switching methods.

Network topologies are either static or dynamic. Static networks have fixed links between processors with no possibility of reconfiguring the network. Intervening PEs store-and-forward data which is to be passed between PEs which are not directly connected. Dynamic switching networks can be reconfigured by setting individual elements of the switch fabric so that any processor can connect with any other. A dynamic reconfigurable switching network is clearly needed in this application.

2.5.2. Hardware switching systems.

Much of the published material [11][12] on interconnection networks is concerned with switches which are other than synchronous, centrally controlled, circuit switched and dynamic. If the additional constraint of non-blocking is added, the selection is even smaller. The most obvious switch configuration which satisfies all the requirements is a crossbar switch. (Figure 2.3). For a regular $N^*N$ matrix there are $N^2$ switch points and one-to-many connections are allowed. As $N$ becomes large however, the crossbar switch is said to be too costly [13], with $N=128$ being a realistic practical limit with current integrated circuit technology [14].

In this proposal the overall size of the global switch, without extension, is 128-by-128, which is at the limit. However, because of its planar construction, it would be constructed from a number of smaller crossbar switches which are readily obtainable. For instance, MITEL [15]
Figure 2.3. A 4-by-4 crossbar switch. One-to-many connections are possible but the control system should prevent two or more ins being connected to the same out.

produce a range of switches which are described as 'analog' but can be used for digital switching [16]. Each switch point introduces a delay of 10nS (typical) and has a 3dB frequency response to 45MHz. In the open state, switch points are said to provide a 'high' degree of isolation. Each of the crossbar switch points can be separately addressed allowing any combination of I/O connections.

A similar but less expansive range is supplied by GE/RCA [17] and, as discussed in 2.6.2.2, Inmos produce a switch which is particularly suitable for this application. There is thus a ready supply of small crossbar switches which can be used to build the global switch.

2.5.2.1. Multi-stage switches.

For larger sized switches, multi-stage networks can provide a cheaper alternative than a complete crossbar switch. The best known analysis of this type of switch is due to Clos [18] who considered multi-stage non-blocking arrays with less than $N^2$ switch points. All multi-stage Clos networks have an odd number of stages. Each stage consists of a number of smaller crossbar switches with identical input and output stages and a variable number of intermediate stages. The general form of a three-stage Clos network is given in Figure 2.4.

Clos showed that if $m = 2n-1$ then the network is non-blocking. Using this relationship with
Figure 2.4. A three-stage Clos network.

\[N = n^2 = 128, \; n = 8 \text{ and } m = 15,\] the required number of switch points is 7680 compared with the 16384 for a single-stage crossbar network. If the size of the network is doubled and \(n = r = 16\) and \(m = 31\), the required number of switch points is 23808 compared with 65536 - a substantial reduction. However, there is little mention in the literature of how to control such a switch or how it could be adapted to allow one-to-many connections. If it were necessary, further work would be needed to resolve these issues.

Control and implementation would also be a problem with the Benes network [19] which was similarly concerned with the need to find economical (point-to-point) switches for telecommunication traffic. The Benes switch, and many similar switches, are constructed from a number of 2x2 crossbar switches which can be set in any of the four states shown in Figure 2.5.

Figure 2.5 The possible mappings of ins to outs for a 2-by-2 switch.

For a digital simulator of the proposed scale (16 PEs with 8 inputs and 8 outputs) the compli-
cation of attempting to minimise the number of switch points is probably not worthwhile. The global switch is most easily constructed from the crossbar switches which are already commercially available and if hardware rather than software implementation of the output router was thought to be necessary, one additional 8x8 crossbar switch on each processor board could easily be provided.

The input router is different in character to the other two switches in that the required connection pattern is one-to-one rather than one-to-many. A crossbar switch could equally well be used here of course, but the different requirement does open up other possibilities. For example, there is a class of related switch architectures called banyan networks, which includes baseline networks, multi-stage shuffle exchanges and several others [20]. These have been extensively studied in connection with parallel processing and are now being suggested as possibilities for third-generation packet switches in data communications networks [21],[22],[23]. The need here is for an inexpensive, but fast, hardware switch which replaces the bus-based and software switches of earlier generations. In the main the connection pattern is one-to-one although buffered broadcast (one-to-many) algorithms have been proposed [24]. They are constructed from identical 2x2 switching elements but, for one-to-one connections, the broadcast modes are not used [25]. Figure 2.6 shows a multi-stage shuffle network (also called an omega switch) which is typical of the class.

Figure 2.6 A multi-stage shuffle network comprising 2-by-2 switches. Inputs are switched according to header bits carried with the data packets.
For an \(N \times N\) banyan network there are \((N/2)\log_2 N\) elements arranged in \(\log_2 N\) stages. Data streams are routed through the switch by a \(\log_2 N\)-bit header with the \(i^{th}\)-bit controlling the \(i^{th}\) stage. The header is a binary representation of the output line number. Synchronous arrivals on each of the two inputs are routed to an output provided that both do not require the same output. A 0 header bit connects the bit stream to the upper output and a 1 connects it with the lower output. It can be demonstrated that the appropriate header will route a bit stream to the correct output whichever input line it arrives on.

A collision occurs if at any stage both inputs require the same output line. In data communications applications the usual procedure to overcome this problem is to buffer (queue) one of the inputs until the required line is free. However, the situation can be avoided by preceding the router by a Batcher [25] bitonic sorting network which sorts the arrivals in such a way that route congestion is avoided. (Figure 2.7). Each element of the Batcher networks simply compares serial bit streams and switches them according to the direction of the arrowhead. If the arrow points up the arrival with the highest header address leaves the top output and the other is switched to the lower output. If the arrow points down the switching action is reversed. Collisions cannot occur at the output stage since only one-to-one connections are allowed.

![Figure 2.7 A Batcher bitonic sort network and a banyan switch.](image)
The combination of Batcher sorting networks and banyan routing networks have been used in a number of fast packet switches [21, 23] and could well be used in a hardware input router. Lee [27] has reported on the design of a non-blocking, one-to-many switch of this type although the technique, using a copy network followed by a point-to-point switch, is probably too elaborate for a simulator of this dimension. However, increases in scale and advances in design and integrated circuit technology would mean that multistage switches could provide an economical basis for the design of all the simulator switching functions.

2.6 Processor implementation and its influence on switch design.

2.6.1 Bit-slice implementation.

The original processing element [28],[29] was built from AMD 2900-series bit-slice components and was the equivalent of the Membrane MBD 24 integrator. Hardware was also designed and built to test that downloading of execution code was achievable [30], but the device had limited I/O capability and no provision for I/O switching. Bit-slice implementation, albeit using more modern components [31], is still likely and is not expected to present any major problems when switching functions are included. Figure 2.8 shows a design assuming that the input/output routers are hardware components.

If the routers are both crossbar switches, then control data written into the switch memory at system initialisation sets each crosspoint for the duration of a simulation. Initialisation of a Batcher-banyan input router involves a training sequence which, as before, establishes circuit-switched connections through the matrix. The routing headers containing the output line number are connected at the proper inputs and then flow through the switch setting individual elements as they go. Assuming a 1-bit delay in each switching stage, the transmit period will be extended by 6 bit times to allow the bit stream to clear the switch and reach the serial in-parallel out input register.
NB. Processor comprises control memory, microcontroller and ALU. ALU word length = n
2.6.2 Transputer Implementation.

The transputer is designed specifically for parallel processing. It first became available in late 1985 and the latest device in the product line, the T800, is a full 32-bit reduced instruction set computer (RISC) chip with 4kbytes of on-chip RAM, a 64-bit floating point arithmetic unit and configurable memory interface. The hardware feature which most obviously highlights the parallel processing application of the transputer is the four high-speed links which are used to connect transputers in a processor array. The standard link speed is 10Mbps but operation at 5 and 20Mbps is also possible.

The other facility which is particularly relevant to the simulator design is the floating-point arithmetic unit. This provides single and double-length operations conforming with the ANSI-IEEE 754-1984 floating point arithmetic standard. On the face of it, the transputer is the ideal device from which to build processing elements.

2.6.2.1 Interconnecting transputer links.

There are two major problems with transputer Implementation of processing elements,

- four links are clearly insufficient. The minimum requirement is 16; and
- all transputer links are bi-directional and there is a mandatory link protocol to be observed.

A twisted-pair connection between link interfaces on different transputers allows data to be passed in either direction across the interface. However, each byte must be separately acknowledged before the next byte will be accepted. This protocol is the means of synchronising data transfer between communicating processes. It is illustrated in Figure 2.9.
In the T800, acknowledgements can overlap forward transmission which speeds-up the data transfer phase, but the insistence on a bidirectional connection is a complication in this application. It means that although data transmission is always simplex (from an output to an input, never the reverse), a return path must be provided just for the acknowledgement. The required size of the connection matrix is therefore doubled even though only half the connections carry computational data.

Fortunately, as described in Chapter 5, there are ways of overcoming both problems. Of more immediate interest is the fact that the transputer product line includes a link switch, the C004 [32], which can be cascaded to build interconnecting switches of any size.

2.6.2.2 The C004 Link Switch.

Figure 2.10 shows the internal organisation of the C004. The device is a 32-by-32 programmable crossbar switch designed to interconnect transputer links operating at the standard link speeds of 10 or 20Mbps. It introduces a worst-case delay of 2-bit times between any input and a connected output and is organised as a set of 32, 32-to-1 multiplexers with each multiplexer controlled by a 6-bit latch. Five of the 6 bits determine which of the 32 inputs are...
Figure 2.10 The C004 transputer link switch.
connected through to the output and the 6th bit enables or disables that output. The latches are set by sending configuration data on the configuration link from the host transputer. In Figure 2.10 the switch configuration link is identified as ConfigLinkIn and ConfigLinkOut.

A configuration message is one, two or three bytes long depending on the required action. Table 2.1 lists the possibilities.

<table>
<thead>
<tr>
<th>Config Message.</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0] [input][output]</td>
<td>Connect [input] to [output].</td>
</tr>
<tr>
<td>[1] [link1][link2]</td>
<td>Connect [link1] to [link2] by connecting the input of [link1] to the output of [link2] and the input of [link2] to the output of [link1].</td>
</tr>
<tr>
<td>[2] [output]</td>
<td>Enquires which input is connected to this [output].</td>
</tr>
<tr>
<td>[3]</td>
<td>A command which marks the end of a configuration sequence. Data can then be passed through the switch.</td>
</tr>
<tr>
<td>[5] [output]</td>
<td>Disconnect this [output].</td>
</tr>
<tr>
<td>[6] [link1][link2]</td>
<td>Disconnect the output of [link1] and the output of [link2].</td>
</tr>
</tbody>
</table>

Table 2.1 The C004 configuration options.

The distinction between messages [0] and [1] is particularly interesting. The C004 data sheet includes the statement that the device supports the Inmos link protocol, which implies that all connection must be bi-directional and one-to-one. One-to-many connections would mean that several destinations will attempt to send acknowledgements to the same source, which is a clear violation of the protocol. Thus the normal (Inmos) configuration message is [1]. How-
ever, the definition of message [0] means that a single switch input can be connected to a single switch output and therefore one-to-many connections are possible, provided that the multiple acknowledgement problem can be overcome. Inmos confirm that this is a non-standard and un-documented, but valid, way to use the C004 and Chapter 5 describes how this option is used to advantage in the simulator design.

The global switch in a simulator based on T800 transputers is exactly as described in section 2.4.2. Figure 2.11 shows how an array of C004 switches would be controlled by the host computer to make the required connections.

![Figure 2.11 The Host-global switch Interface.](image)

The master (control) switch is used to connect the host transputer to each configuration link in turn. Making connections through each C004 is therefore a two-step process which can be described as follows:-
repeat

configure the master switch so that the host transputer link is connected to a particular configuration link.

send configuration messages which relate to this plane of the global switch

until all planes configured.

The host transputer can then be connected to one of the 'other function' links as described in Chapter 5.

The configuration procedure is quite slow, but speed is not critical during system initialisation. It is completed before a simulation starts and, once all the connections have been made, they remain intact for its duration. The switch delay is either 50 or 100ns depending on the link speed.

The routers are most conveniently implemented in software because the single-height Eurocard processor boards are likely to be heavily populated with the additional-link logic. The router code occupies a minimum of 256 bytes of memory, for input and output routers combined, and the execution time is 5.6 microseconds at input and output when the T800 is operating at 20MHz with 64-bit numbers.

2.7 Conclusion.

The need for an efficient connection system and a range of available options have been discussed in this chapter. Since a connection system cannot be separated from the devices it connects, the conclusion is that transputers and transputer link switching devices should be used. Transputer technology has matured and stabilised significantly since its introduction. There is now available a range of development aids which are needed to build and test systems and the Occam programming language now includes features which are essential for this application.
Research suggests that the input and output routers can be adequately provided in software and the global switch array by C004s. The design aim is a step length of 100 microseconds and with such an arrangement, the TP and routing processes are completed in an acceptably short time. Larger and faster simulators may make greater use of hardware and incorporate design options which reduce the number of switching devices, but these are unnecessary at the moment.

2.8 References.


3. Transfer Function Simulation - the Data Processor.

3.1 Introduction

A system is a combination of elements, or subsystems, intended to act together to achieve an objective [1]. There is no indication of size when systems are discussed in the abstract. They may be very small and intricate, or very large or have no association with size at all. What they have in common however, are techniques for analysis and simulation. The 'systems approach' is to consider each system element as a black box with a defined input-output relationship. The boxes are connected together as the system topology dictates and the overall behaviour of the system is studied by studying the interaction of subsystems.

A common way of describing systems or their elements is to construct a mathematical model. For a linear continuous system, such a model is a set of algebraic and/or ordinary differential equations (ODEs). Traditionally, these would have been solved on an analogue computer, but as analogue computers have declined in popularity, so digital computers, programmed to implement one of the established numerical integration techniques [2][3], have become more popular. However, the problem with describing systems using sets of equations, as distinct from transfer functions, is that the graphical view of interconnected boxes is lost and with it the 'feel' for how the system behaves.

One of the solution techniques for ODEs begins by converting them to algebraic relationships using the Laplace transform. After the initial transformation, solution is reduced to algebraic manipulation followed by inverse transformation. If the initial conditions of the state variables are assumed to be zero, the system equations can be rearranged to show the ratio between outputs and inputs when a forcing function is applied; in other words, as transfer functions. Representing systems as interconnected boxes and boxes by their transfer functions ties in nicely with the notion of parallel processing in which boxes are processes assigned to parallel processors. There are ways of accommodating initial conditions, should that be necessary, but in any event, the transfer function representation of a continuous system or subsystem is certainly the more appropriate view in this application.
3.2 The relationship between continuous and discrete functions.

If a continuous system is modelled on a digital computer there is clearly a need to establish some relationship between \( s \), the complex frequency, and an equivalent discrete (numerical) function of \( z \). Several transforms have been proposed and described in the literature \([4][5][6][7][8]\). It appears however that the best accepted is the bilinear transform, or Tustin's method \([4]\), in which a continuous function is transformed into its digital equivalent by making the substitution,

\[
\frac{1}{s} \Rightarrow \frac{h (1 + z^{-1})}{2 (1 - z^{-1})} \quad (3.1)
\]

In this expression \( h \) is the time between successive samples of the continuous signal, or function, and \( z^{-1} \) is a unit delay. If \( f(nh) \) is the sampled version of some function \( f(t) \), then the \( z \)-transform of the function, \( F(z) \), is given as \([9]\),

\[
Z\{f(nh)\} = F(z) = \sum_{n=0}^{\infty} f(nh) \cdot z^n
\]

The \( z \)-transform and and its implications for sampled data control systems analysis and design are well documented in the literature \([10]\).

An equally simple algebraic transform which was used in the original Membrand digital-analogue computer \([11]\), is the first-order predictor relationship,

\[
\frac{1}{s} \Rightarrow \frac{h (3 - z^{-1})}{2 (1 - z^{-1})} \quad (3.2)
\]

A continuous transfer function, \( G(s) \), is transformed into \( G(z) \) by substituting for all \( s \) and then
collecting terms. For a general \( n^{th} \) order transfer function, the expressions are,

\[
G(s) = \frac{n_0 + n_1 s + n_2 s^2 \ldots + n_{n-1} s^{n-1} + n_n s^n}{m_0 + m_1 s + m_2 s^2 \ldots + m_{n-1} s^{n-1} + s^n}
\] (3.3)

and,

\[
G(z) = \frac{a_0 + a_1 z^{-1} + a_2 z^{-2} \ldots + a_n z^{-n}}{1 + b_1 z^{-1} + b_2 z^{-2} \ldots + b_n z^{-n}}
\] (3.4)

Assuming zero initial conditions, these expressions are represented diagrammatically as shown in Figure 3.1.

The cascaded unit delays, shown as \( z^{-1} \) boxes in Figure 3.1b, act as a shift register so that at each step, the state variable values (the X's) are shifted up and a new output calculated by executing a series of multiplications and additions.

3.2.1 The delta representation.

One problem with transforms which use the \( z \) operator [12] is that they are prone to coefficient sensitivity. This means that the least significant bits of the transfer function coefficients have a disproportionate effect on the overall accuracy and numerical stability of the simulation. The delta (\( \delta \)) operator [13] was first discussed some time ago but little used. It is known to reduce coefficient sensitivity and to scale the state variables so that they are of similar size to the input. Both of these effects have a beneficial influence on discrete system simulations.

The technique is to replace each \( z \) by \( \delta + 1 \) so that,
Figure 3.1 Continuous (a) and discrete (b) transfer function diagrams.
Diagrammatically this relationship appears as shown in Figure 3.2.

\[
\begin{align*}
\delta^{-1} &= \frac{1}{1 + \delta^{-1}} \\
\delta^{-1} &= \frac{z^{-1}}{1 - z^{-1}}
\end{align*}
\]

The effect of the substitution is to replace each shifting stage by a shift-and-add stage. At each (computational) step the current values of the state variables are shifted through the delays but instead of overwriting the previous value, they add to them. The form of the digital transfer function is the same as equation 3.4 although, of course, the coefficients are different.

3.3 Incorporating state variable initial values.

System equations written in state variable form are a series of first-order equations of the form,

\[ x' = Ax + Bu \]

Where \( x' \) is the first derivative of \( x \), the state variable, \( u \) is the input vector and \( A \) and \( B \) are matrices defining the system. For an \( n^{th} \)-order system \( A \) is an \( n \)-by-\( n \) matrix and on an analogue computer, \( n \) integrators would be used to solve the \( n \) equations. The initial values of the
state variables are easily included as integrator initial values.

If an $n^{th}$ order system is to be modelled in one digital processing element, each of the $n$ state variables can be considered as a constant input associated with an appropriate transfer function. Application of the principle of superposition gives the composite output due to all inputs.

Where real time, or even fast, execution is the goal, computing several component outputs at each step is clearly undesirable. There is a need to calculate initial values for the state variables in the discrete system, given the corresponding values in the continuous system.

![Figure 3.3 A continuous system showing state variable initial values.](image)

In Figure 3.3 the integrator initial values are shown as separate inputs. Each of them produces a free (unforced) component of the output. The forced component is the result of applying an input at $u(s)$. When initial values are included on discrete system diagrams, the unit
delays are assumed to have a storage capability and initial values, \( X_1(0) \ldots X_n(0) \), are written into stores. Since the systems are linear, the output of the \( p^{th} \) shifting stage, \( X_p(z) \), is related to the forcing function, \( U(z) \), by the transfer function,

\[
G_p(z) = \frac{X_p(z)}{U(z)} = \frac{z^p}{1 + b_1 z^{-1} + b_2 z^{-2} \ldots + b_n z^{-n}}
\]

(3.5)

The continuous equivalent of \( G_p(z) \) is,

\[
G_p(s) = \frac{X_p(s)}{U(s)} = \frac{r_0 + r_1 s + r_2 s^2 + \ldots + r_n s^n}{m_0 + m_1 s + m_2 s^2 + \ldots + s^n}
\]

(3.6)

The denominators of \( G_p(z) \) and \( G_p(s) \) are of course unchanged when compared with the full system transfer function; the feedback components are the same, only the feed-forward components are different. The state variables, \( x_0 \ldots x_{n-1} \), shown on Figure 3.3 are therefore the same as the state variables relating to equation 3.6. Hence at \( t = 0 \), \( X_p(z) \), is given by,

\[
X_p(0) = r_0 x_0(0) + r_1 x_1(0) + r_2 x_2(0) + \ldots + r_{n-1} x_{n-1}(0) + r_n \left( u(0) - m_0 x_0(0) - m_1 x_1(0) - \ldots - m_{n-1} x_{n-1}(0) \right)
\]

Where \( X_p(0) \) is the initial value of the state variable at the output of the \( p^{th} \) shifting stage in the discrete system and \( x_0(0) \ldots x_{n-1}(0) \) are the initial values of the continuous system state variables. This expression can be written more concisely as,

\[
X_p(0) = \begin{bmatrix} 1 & -r_n \end{bmatrix} \begin{bmatrix} r_0 & r_1 & r_2 & \ldots & r_{n-1} & r_n \\ m_0 & m_1 & m_2 & \ldots & m_{n-1} & 0 \end{bmatrix} \begin{bmatrix} x_0(0) \\ x_1(0) \\ x_2(0) \\ \vdots \\ x_{n-1}(0) \\ u(0) \end{bmatrix}
\]

(3.7)
A set of r-coefficients is calculated for each of the n values of p, where n is the order of the system, and then used in equation 3.8 to calculate the initial values of the discrete state variables.

Calculation of the r-coefficients involves substituting for s in equation 3.6 and then equating numerator coefficients of powers of z with those of equation 3.5. Making the substitution using the bilinear transform reduces the numerator of equation 3.6 to,

\[
\sum_{n=0}^{\infty} n h^n z^{-l} \left(1 - z^{-1}\right)^l \left(1 + z^{-1}\right)^{-l} = \sum_{n=0}^{\infty} m h^n z^{-l}
\]

(3.8)

For example, if n = 2 (a second-order system) and the expression is expanded, the r-coefficients for determining \(X_1(0)\) are given by the relationships,

\[
ro h^2 + 2r_1 h + 4r_2 = 0
\]

\[
\frac{2ro h^2 - 8r_2}{m_0 h^2 + 2m_1 h + 4} = 1
\]

\[
ro h^2 - 2r_1 h + 4r_2 = 0
\]

And for determining \(X_2(0)\), the relationships are,

\[
ro h^2 + 2r_1 h + 4r_2 = 0
\]

\[
2ro h^2 - 8r_2 = 0
\]

\[
\frac{ro h^2 - 2r_1 h + 4r_2}{m_0 h^2 + 2m_1 h + 4} = 1
\]
Mechanising this process of calculating the \( a/b \) coefficients and the discrete system state variables given the \( n/m \) coefficients and the continuous system state variables is a fairly straightforward programming task once the technique and the conversion equations have been established. Using a different s-to-z transform necessitates re-working the equations but the technique is always the same.

The above analysis assumes that the continuous system circuit diagram is in companion form [14]. Should it be necessary, standard techniques are available to convert inappropriate circuit arrangements. Alternatively, it has been found that elementary circuit analysis of the continuous system diagram is often sufficient to obtain proper values of \( x_n(0) \) from which discrete system initial values can be found.

3.4 The data processor.

The simulation of discrete representations of continuous system transfer functions is assigned to a data processor. The data processor is one of a range of processors which are outwardly identical but distinguished by different program code stored in the processor memory. In simulating the operations implied by equation 3.4, the data processor for a third-order transfer function, can be pictured as shown in Figure 3.4.

Fixed parameters, such as the \( a/b \) coefficients and state variable initial values, are calculated as described earlier and downloaded with the configuration code. Following configuration, the mode selected is always RESET. The initial values of the state variables and sigma are loaded into the model and the processor executes one step of the execution code. This shifts values through the shifting stages - thereby lodging the proper initial values at the proper stage outputs - and then calculates a new value for \( X_0(0) \) before calculating the initial value of the output. The only addition to this sequence of events when the mode is COMPUTE is that the weighted sum of the inputs is computed and assigned to the variable sigma before shifting occurs. In HOLD mode, shifting is inhibited and the output is held at the value calculated in the previous step.

Figure 3.4 shows a number of different processor (as opposed to device) outputs which will
Figure 3.4 The data processor.
be generated at each step. As previously explained, these will be routed to particular device output pins as required.

3.5 Model Testing.

A realistic test [3] for the data processor is to consider the equations,

\[ y' = Ay \]

Where \( y_0 = \begin{bmatrix} 2 \\ 1 \\ 2 \end{bmatrix} \)

and \( A \) has the form \( \begin{bmatrix} -a & -b & 0 \\ 0 & -c & 0 \\ 0 & +d & -e \end{bmatrix} \)

Three sets of test data were used as follows,

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set 1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Set 2</td>
<td>0.5</td>
<td>9.5</td>
<td>10</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>Set 3</td>
<td>0.1</td>
<td>49.9</td>
<td>50</td>
<td>70</td>
<td>120</td>
</tr>
</tbody>
</table>

The first set is considered to be benign, the second moderately difficult and the third particularly stiff [3]. The equations have exponential solutions of the form,

\[ y_1 = e^{at} + e^{ct} \]
\[ y_2 = e^{ct} \]
\[ y_3 = e^{ct} + e^{et} \]
so that direct comparison of simulated and true values is possible. The literature considers
only the stiff set of equations, set 3, and tabulates the percentage deviation of the simulated
from the true values for a range of step lengths and a differing number of steps. Percentage
deivation is defined as,

\[
\frac{\text{true output} - \text{measured output}}{\text{true output}} \times 100\%
\]

and the integration algorithm used is one of the series of 5th-order Butcher formulae [15]
which were developed from the original Runge-Kutta equations. They are considerably more
computationally demanding than the transforms proposed here and could not reasonably be
used in a real-time hardware simulator. However, the equations are a useful benchmark.

Two series of tests were made. The first modelled each term in the system equations using
first-order data processors, and the second series modelled each equation in a single data
processor. The tests are referred to as SI (single integrator) and TF (transfer function) respec-
tively. In both series the step length was taken as 0.0001s and the word length was the 40-
bits (32-bit mantissa and 8-bit exponent) of the host (Turbo) Pascal compiler. Outputs were
recorded every 0.02s and each test was allowed to run for a simulated time of 0.5s. Figure
3.5 shows the different test arrangements.

3.5.1 Errors in Digital Simulations

The sources and significance of errors in the digital simulation of continuous systems have
been studied at length elsewhere [2][3][16]. In this study, errors are said to be either algo-
rithmic or arithmetic. Algorithmic errors arise because the s to z plane transform is an approxi-
mation and must always introduce an error. The nature of the signal or system being
simulated has an effect on algorithmic error and, in the absence of any other error source, it
can be reduced by reducing the step length.
Figure 3.5 Simulation of the three equations using (a) separate integrators, and (b) simulated transfer functions. In (a) the multiplying factors are values of $k$.

Arithmetic error is a consequence of finite word lengths. When floating point numbers are added or subtracted, the exponents of the two numbers must be the same. If this is not the case the mantissa of the smaller number is shifted down one place and its exponent in-
cremented until the two exponents are equal. The mantissae are then added or subtracted and the result normalised - another shifting operation. If the numbers are substantially different, then discarding the least significant bits of the smaller number during the initial shifting process is one component of the arithmetic error. If the numbers have the same order of magnitude and the operation is subtract, the final normalisation has the same effect. Increasing the mantissa length reduces the numerical significance of the discarded bits and therefore reduces the error, but it is never eliminated.

Another source of arithmetic error arises in multiplication. The 2n-bit product of two n-bit numbers must eventually be rounded off. The way in which numbers are held and the point in the arithmetic procedure at which rounding and normalisation takes place has a critical effect on the accumulated arithmetic error. In the main however, the adoption of sensible arithmetic procedures can minimise this error source.

3.5.2. The SI tests

The aims of these tests were as follows:

1. to verify the configuration of a data processor;

2. to compare transforms and determine, for this set of equations, whether one is any better than the others;

3. to verify the process of initialising state variables described above; and

4. to provide a yardstick against which to measure the TF test results.

To check 4, the data processor is viewed either as a first-order system with an initialised state variable, or as an Integrator with an initial value register (IVR). With the latter view, the initial value of the output is written into the IVR at system initialisation and the final operation in calculating a step output is to add the contents of the IVR to the sum of the a*X products. With the former view, an IVR is not provided but the state variable is set to an initial value such that the first computed output is the correct output initial value.
Five tests were made using the circuit of Figure 3.5.a.

Test 1: Using the bilinear transform with the IVR data processor model.

Test 2: Using the predictor transform with the IVR data processor model.

Test 3: Using the bilinear transform with an initialised state variable.

Test 4: Using the predictor transform with an initialised state variable.

Test 5: Using the delta form of the bilinear transform with an initialised state variable.

Comparing test 1 results with those of test 2, and test 3 results with test 4 would show whether there is anything to be gained by selecting the transform. Tests 1 and 3 and 2 and 4 will show if the initial value technique is valid for first-order equations and, finally, test 5 will show, for these test conditions, whether the delta representation has any advantages.

3.5.2.1 The SI test results.

Figure 3.6 shows the test 1 and test 2 results for the first data set, set 1. Since the transform used in tests 2 and 4 is predictive, the calculated value is at time $t + h$ rather than at time $t$. Other than that, there are no differences in the way results from tests 1 and 2 are gathered.

The trend illustrated in the figure is consistent throughout these tests; the predictor transform produces an approximate constant deviation and the bilinear transform produces a changing deviation. In test 1, the bilinear transform test, it is clear that the largest deviations correspond with the largest per-step change of the output function, where per-step change is defined as,

$$\frac{\text{previous value} - \text{present value}}{\text{present value}}$$
Figure 3.6. Test 1 and test 2 results for data set 1.

For the single exponential, $y_2$, the per-step change is of course constant and, apart from one test, the $y_2$ deviation is seen to increase linearly over the simulation period. When the $y_1$ and $y_3$ deviations are considered the situation is complicated by the fact that both are combinations of two exponential terms. The per-step change in $y_3$ is obviously greater than that of $y_1$ and consequently its deviation is greater. For both however, there is relatively little variation in the per-step change over the simulation period, which accounts for the curves being shallow. In both cases the dominant term is the one with the smallest exponential coefficient, $a$ or $d$.

The predictor transform tests show an initial deviation which is always larger than the bilinear transform result but which remains substantially constant throughout the test. It suggests that the initial error is attributable to the transform, or the way it is applied, but thereafter there is no accumulation of error as the approximation procedure is continued. Obtaining starting values for a predictive operation is always problematical and no provision has been made to provide any here. This may well account for the initial error but whether this is an observation which could be made for a wider range of test conditions has not been investigated.
On the whole, the results from test 1 and 2 are inconclusive. It is not possible to state that one transform is better than the other because the results they produce are so different in character. What the results do illustrate however is the unacceptability of the IVR model of a data processor. Figure 3.7 shows some of the test 1 and test 2 results for the third set of exponential coefficients.

Figure 3.7 Showing the numerical instability with data set 3 when IVRs are used.

The function $y_3$ is dominated by the $y_2$ component and consequently the $y_2$ and $y_3$ deviations are very similar. The numerical instability which is illustrated in the figure arises because the $a^*X$ products are added to the contents of the initial value register to generate a step output. Since the $y_2/y_3$ outputs get progressively and rapidly less, and the initial value is always positive, the sum of $a^*X$ products must be negative and rapidly converge towards the magnitude of the initial value. As discussed above, adding numbers of similar magnitude but opposite sign means that the difference between them is registered in the least significant bits which is the cause of arithmetic error. It has been demonstrated in another series of tests that increasing the mantissa length delays the onset of numerical instability and reducing the mantissa length advances it. However, the susceptibility of this model to this kind of instability is reason enough not to consider it further.
Tests 3 and 4 produce almost identical results to tests 1 and 2 so that, as before, it is not possible to say anything definite about the relative merits of the two transforms. A more definite conclusion is that the effect of Initialising state variables in the manner described is the same as Initialising an Initial value register, with the important difference that that the numerical instability which was previously evident in $y_2$ and $y_3$ with the third set of coefficients is no longer a problem. Figure 3.8 illustrates the stabilisation of the model.

![Graph illustrating stabilisation](image)

**Figure 3.8.** Data set 3 results with Initialised state variables. The magnitude of the deviation is the same but the numerical instability has been removed.

Test 5 produced results which were almost identical with tests 1 and 3. Whatever the merits of the delta form of the bilinear transform, they were not demonstrated under these test conditions.
3.5.3 The transfer function tests.

Three different tests were carried out with the circuit of Figure 3.5b. They correspond with the previous tests 3, 4 and 5 and used the bilinear transform (test 6), the predictor transform (test 7) and delta form of the bilinear transform (test 8). With zero initial conditions, the transfer functions derived from the differential equations are,

\[ G_1(s) = \frac{-b}{(s+a)(s+c)} \]
\[ G_2(s) = \frac{1}{s+c} \]
\[ G_3(s) = \frac{d}{(s+c)(s+e)} \]

When provision is made for state variable initial values the continuous system block diagrams are as shown in Figure 3.9. The initial values are calculated to be:

For \( y_1 \):
\[ x_0(0) = -\frac{y_1(0)}{b} = -\frac{2}{b} \]
\[ x_1(0) = \frac{(b-y_2(0) + a y_1(0))}{b} = \frac{(b + 2a)}{b} \]

For \( y_2 \):
\[ x_0(0) = y_2(0) = 1 \]

For \( y_3 \):
\[ x_0(0) = \frac{y_3(0)}{d} = \frac{2}{d} \]
\[ x_1(0) = \frac{(d y_2(0) - c y_3(0))}{d} = \frac{(d - 2e)}{d} \]

When these and the system transfer functions are translated into discrete equivalents and assigned to data processors, the most obvious difference in the test results is the overall reduc-
Figure 3.9 The three equations simulated as separate transfer functions.
In percentage deviation compared with the SI tests. The other noticeable effect is how poorly the predictor transform compares with the other two. Table 3.1 shows the first (after 0.02S) and last (after 0.5S) measured deviation for each set of equations and each transform. Graphical presentation is not appropriate in this case.

<table>
<thead>
<tr>
<th></th>
<th>$y_1$</th>
<th></th>
<th>$y_2$</th>
<th></th>
<th>$y_3$</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$t=0.02s$</td>
<td>$t=0.5s$</td>
<td>$t=0.02s$</td>
<td>$t=0.5s$</td>
<td>$t=0.02s$</td>
<td>$t=0.5s$</td>
</tr>
<tr>
<td>Set 1</td>
<td>Bilinear</td>
<td>5.41x10^{-6}</td>
<td>3.39x10^{-3}</td>
<td>9.47x10^{-10}</td>
<td>-2.49x10^{-8}</td>
<td>-1.32x10^{-5}</td>
</tr>
<tr>
<td></td>
<td>Predictor</td>
<td>-1.55x10^{-2}</td>
<td>-5.04x10^{-2}</td>
<td>-2.08x10^{-2}</td>
<td>-4.00x10^{-2}</td>
<td>-2.62x10^{-2}</td>
</tr>
<tr>
<td></td>
<td>Delta</td>
<td>-8.53x10^{9}</td>
<td>2.77x10^{7}</td>
<td>-1.89x10^{9}</td>
<td>-4.45x10^{9}</td>
<td>-6.69x10^{10}</td>
</tr>
<tr>
<td>Set 2</td>
<td>Bilinear</td>
<td>-9.03x10^{-6}</td>
<td>-2.68x10^{-3}</td>
<td>1.71x10^{-6}</td>
<td>-4.26x10^{-5}</td>
<td>-8.11x10^{-6}</td>
</tr>
<tr>
<td></td>
<td>Predictor</td>
<td>-5.71x10^{-2}</td>
<td>-1.24x10^{-2}</td>
<td>-1.20x10^{-1}</td>
<td>-6.01x10^{-2}</td>
<td>-1.92x10^{-1}</td>
</tr>
<tr>
<td></td>
<td>Delta</td>
<td>7.14x10^{-7}</td>
<td>2.53x10^{-7}</td>
<td>1.65x10^{6}</td>
<td>4.13x10^{6}</td>
<td>6.88x10^{6}</td>
</tr>
<tr>
<td>Set 3</td>
<td>Bilinear</td>
<td>-1.69x10^{-6}</td>
<td>-2.12x10^{-3}</td>
<td>2.08x10^{-4}</td>
<td>-5.21x10^{-3}</td>
<td>7.48x10^{-4}</td>
</tr>
<tr>
<td></td>
<td>Predictor</td>
<td>-2.70x10^{-1}</td>
<td>-1.74x10^{-3}</td>
<td>1.00x10^{0}</td>
<td>-1.38x10^{4}</td>
<td>-1.62x10^{0}</td>
</tr>
<tr>
<td></td>
<td>Delta</td>
<td>5.61x10^{-5}</td>
<td>-2.81x10^{-7}</td>
<td>2.08x10^{-4}</td>
<td>-5.21x10^{3}</td>
<td>7.37x10^{-4}</td>
</tr>
</tbody>
</table>

Table 3.1 A summary of the transfer function test results.

For the first time the results show evidence of arithmetic errors. The bilinear transform results for example, show that with data set 1, the smallest deviations were obtained from the first-order simulation generating $y_2$, even though $y_1$, a second-order output, has the smaller per-
step change. This is because \( y_1 \) and \( y_3 \) are combinations of simple exponential terms and thus the numerator of the equivalent transfer function, \( G(s) \), is independent of \( s \). In these circumstances it can be shown that the second-order \( a \)-coefficients are of the order of \( h^2 \) (as \( h \) tends to 0, \( a_0 \) and \( a_2 \) tend to \( noh^2/4 \) and \( a_1 \) tends to \( noh^2/2 \)) and the state variable initial values are of the order of \( 1/h^2 \). If \( h \) is small the \( a \)'s are very small and the \( X \)'s are very large. The sum of the \( a^*X \) products, which is always less than 2, is the step output.

In addition, the \( b_1 \) and \( b_2 \) coefficients tend, respectively, towards -2 and +1 as \( h \) decreases. Since \( X_1 \) and \( X_2 \) are approximately equal, \( X_0 \) is therefore calculated by adding numbers of similar order but opposite sign - another opportunity for arithmetic error to be introduced. Increasing the step length reduces the arithmetic error but algorithmic error then becomes a problem.

It is reasonable to assume that algorithmic errors are the same as in the SI tests since the functions and the step length are the same. The fact that the TF test errors are less than before must indicate that the algorithmic error is a secondary effect. The SI tests involved more arithmetic operations and, in the cases of tests 1 and 2, an inefficient way of incorporating initial values. There was therefore an inherent, background, error which was related to the calculation method and against which the algorithmic component of the total could be seen. In these transfer function tests there are fewer arithmetic operations and a seemingly more efficient way of incorporating initial values. The background error component has therefore been removed (or least reduced) and the effect of the smaller coefficients, combined with finite mantissa length, is most noticeable. Overall however, there has been reduction in the deviation.

The problem with the predictor transform is that the coefficients and the state variable initial values are of similar order to the bilinear transform data, but the summation of the \( a^*X \) products is, in fact, subtractive since \( a_0 \) and \( a_1 \) always have opposite signs. Since this potential for error cannot reasonably be avoided, the predictor transform is probably not a good choice when working with this combination of word and step length.

The interesting consequence of using the delta form of the bilinear transform is that the coefficients and state variable initial values are not of the same order. For instance, again considering one of the second-order equations, the \( d_1 \) coefficient, which corresponds with \( b_1 \) in previous expressions, is in the order of \( m_1h \) and \( d_2 \) in the order of \( m_0h^2 \). The state variable In-
Initial values tend towards $1/h$ and $1/h^2$ and, for a step length of 100 microseconds, are substantially different. With just one term dominating therefore, it is not surprising that arithmetic errors are reduced.

In the main, the results from test 8 support this contention. The first set for instance, shows that the $y_1$ and $y_3$ results using the delta operator are significantly better than those using the bilinear transform. Coefficient sensitivity has been reduced. For the other two sets of equations the results are quite similar. This is because the per-step changes are now much greater than before and therefore there is evidence of algorithmic error, and secondly the dominance of one term over all others is diminished. For example, the difference between the state variable initial values is 15 times less for set 3 than for set 1.

Even though the use of the delta operator is mostly beneficial and never detrimental, there are several reasons for retaining an interest in the bilinear transform. The main one is that the bilinear transform implementation involves a simple shift rather than shift-and-add. Shifting is a much faster operation than addition and extra time may be crucial when estimating the minimum step length that can be used. Another reason is that if the mantissa and/or step length could be changed, this may reduce the advantage of the delta-operator. Developing both holds out the prospect of a choice of transform with accuracy traded for speed, should the trade be necessary.

### 3.5.4 Simulation of a third-order transfer function.

The simulation of high-order transfer functions in a single processing element is not sensible if the accuracy of the model will not support the order of the transfer function. The previous tests showed that coefficient sensitivity can be a problem in second-order systems simulated using the bilinear transform, and the problem will very likely get worse as the system order increases. Progressing to a third-order system was intended to test this likelihood.

The circuit diagram and the transfer function of the third-order system used in this test is shown in Figure 3.10. The input is a unit step is applied at $t = 0$. 

3-23
Figure 3.10: The third-order system test circuit.

Function

\[
\frac{y(t)}{u(t)} = \frac{10 + 20s}{20 + 12s + 3s^2 + s^3}
\]

\(u(t)\) is a unit step applied at \(t=0^+\)
It can be shown that the step response output, \( x(t) \), is given by the expression,

\[
\begin{align*}
x(t) &= 1/2 + 1.05e^{-2t} - (2.74)^{0.5}e^{-0.51t}\sin((39/4)^{0.5}t + \alpha) \\
\end{align*}
\]

where the phase angle \( \alpha \) is

\[
\alpha = \tan^{-1} \left( \frac{0.55 \times 39^{0.5}}{9.75} \right)
\]

Figure 3.11 shows this function in the period \( t = 0 \) to \( t = 10\)S with ordinate values plotted at 0.1S intervals. When systems with this sort of step response are simulated, algorithmic errors are likely during the initial transient period. Reducing the step length reduces the algorithmic error but increases the arithmetic error because the coefficients sensitivity gets worse.

![Figure 3.11. The third-order system response](image-url)
Figure 3.12 shows the simulation diagram with parameter values relating to a bilinear transformation. Processor 1 is programmed to generate the unit step and processor 2 to simulate the transfer function.

For a step length of 0.0001S the resulting output is shown in Figure 3.13. The initial value is correct and the first ordinate value approximately correct but thereafter, the simulated output bears little relationship to the calculated response. Increasing the step length to 0.0005S and 0.01S verified that the technique was correct because the output for both these cases was

---

**Table**

<table>
<thead>
<tr>
<th></th>
<th>h = 0.01</th>
<th>h = 0.0005</th>
<th>h = 0.0001</th>
</tr>
</thead>
<tbody>
<tr>
<td>( b_1 )</td>
<td>-2.9692554682</td>
<td>-2.9994981257</td>
<td>-2.9996999250</td>
</tr>
<tr>
<td>( b_2 )</td>
<td>2.9397223980</td>
<td>2.9969992528</td>
<td>2.9993999700</td>
</tr>
<tr>
<td>( b_3 )</td>
<td>-9.7044723124x10^{-1}</td>
<td>-9.9950112466x10^{-1}</td>
<td>9.9970004500x10^{-1}</td>
</tr>
<tr>
<td>( a_0 )</td>
<td>4.936952876x10^{-4}</td>
<td>1.2492183989x10^{-6}</td>
<td>4.9993749438x10^{-8}</td>
</tr>
<tr>
<td>( a_1 )</td>
<td>4.9615754910x10^{-4}</td>
<td>1.2495306645x10^{-6}</td>
<td>4.9996249063x10^{-8}</td>
</tr>
<tr>
<td>( a_2 )</td>
<td>-4.887705807x10^{-4}</td>
<td>-1.2485938678x10^{-6}</td>
<td>-4.9988750188x10^{-8}</td>
</tr>
<tr>
<td>( a_3 )</td>
<td>-4.9123290842x10^{-4}</td>
<td>-1.2489061333x10^{-6}</td>
<td>-4.9991249813x10^{-8}</td>
</tr>
<tr>
<td>( X_1 )</td>
<td>4.0773293232x10^{+4}</td>
<td>3.2030046029x10^{+8}</td>
<td>4.0007502300x10^{+10}</td>
</tr>
<tr>
<td>( X_2 )</td>
<td>4.0468674561x10^{+4}</td>
<td>3.2018037017x10^{+8}</td>
<td>4.0004501850x10^{+10}</td>
</tr>
<tr>
<td>( X_3 )</td>
<td>4.0128576145x10^{+4}</td>
<td>3.2005957958x10^{+8}</td>
<td>4.0001497899x10^{+10}</td>
</tr>
</tbody>
</table>
Figure 3.13 The simulated third-order transfer function with \( h=0.0001s \).

reasonable. It is interesting to note that during the initial transient phase there was little to
choose between the 0.0005S and 0.01S simulations. When the initial transients had subsided
however the 0.01S simulation gave deviations which were approximately two orders of magni-
tude smaller. This predictable result was again caused by coefficient sensitivity which is par-
ticularly noticeable with small step lengths.

In an attempt to isolate the problem when the third-order system is simulated with a step
length of 0.0001S, the simulation program statements which are executed when a new step
value is calculated were written as a separate program. In order to track the calculation and to
ensure that inadvertent parameter changes were not being made, the a/b coefficients, the
state variables and the a^X products were displayed at every printout (0.1S) interval. The pro-
gram output is compared with simulation output in table 3.2.
Simulation/Program Parameters

\begin{align*}
a_0 &= 4.9993749438E-08 \\
b_1 &= -2.9995999250E+00 \\
a_1 &= 4.9996249063E-08 \\
b_2 &= 2.9993999700E+00 \\
a_2 &= -4.9988750188E-08 \\
b_3 &= -9.9970004500E-01 \\
a_3 &= -4.9991249813E-08
\end{align*}

Runtime measurements.

\begin{align*}
\text{Program O/P} & \quad \text{Simulation O/P} \\
\text{t = 0.0 seconds} \\
X_0 &= 4.0010499250E+10 & 4.0010499250E+10 \\
X_1 &= 4.0007502300E+10 & 4.0007502300E+10 \\
X_2 &= 4.0004501850E+10 & 4.0004501850E+10 \\
X_3 &= 4.0001497899E+10 & 4.0001497899E+10 \\
a_0 X_0 &= 2.0002748744E+03 & 2.0002748744E+03 \\
a_1 X_1 &= 2.0002250494E+03 & 2.0002250494E+03 \\
a_2 X_2 &= -1.9997750494E+03 & -1.9997750494E+03 \\
a_3 X_3 &= -1.9997248744E+03 & -1.9997248744E+03 \\
O/P &= 9.9999999627E-01 & 1.0000000205E+00 \\
\text{t = 0.1 seconds} \\
X_0 &= 4.1391355911E+10 & 4.1528422895E+10 \\
X_1 &= 4.1391448348E+10 & 4.1528115253E+10 \\
X_2 &= 4.1391538123E+10 & 4.1527805705E+10 \\
X_3 &= 4.1391625233E+10 & 4.1527494249E+10 \\
a_0 X_0 &= 2.0693090763E+03 & 2.0761615687E+03 \\
a_1 X_1 &= 2.0694171607E+03 & 2.0762499933E+03 \\
a_2 X_2 &= -2.0691112591E+03 & -2.0759231052E+03 \\
a_3 X_3 &= -2.0692190772E+03 & -2.0760113391E+03 \\
O/P &= 3.9590075612E-01 & 4.7711769119E-01 \\
\text{t = 0.2 seconds} \\
X_0 &= 4.0113401078E+10 & 4.1123088078E+10 \\
X_1 &= 4.0115713043E+10 & 4.1123975797E+10 \\
X_2 &= 4.0118023237E+10 & 4.1124862985E+10 \\
X_3 &= 4.0120316622E+10 & 4.1125749640E+10 \\
a_0 X_0 &= 2.0054193226E+03 & 2.0558973615E+03 \\
a_1 X_1 &= 2.0056351806E+03 & 2.0560445364E+03 \\
a_2 X_2 &= -2.0054498416E+03 & -2.0557805023E+03 \\
a_3 X_3 &= -2.0056655227E+03 & -2.0559276240E+03 \\
O/P &= -6.0861084610E-02 & 2.3377160728E-01
\end{align*}

Table 3.2 A comparison of the third-order system response when the system is simulated and when the response is calculated in separate program. The step length is 0.0001S. The numbers graphically illustrate coefficient sensitivity.
At $t=0$ the simulator and the program produce the same values for $X_0$ and the $a^X$ products. Interestingly though, when the $a^X$ products are added the sum is different. The only difference in the two procedures is that parameters values are calculated in the program but assigned in the simulation. The decimal representation of the floating point bit patterns may be the same but it cannot be assumed that the bit patterns themselves are the same. For a mantissa of 40 bits, the resolution is to 11 decimal places ($2^{39} = 5.5 \times 10^{11}$) so that if bit patterns are only converted to 10 decimal places the last few bits are not used but remain to be included in any subsequent floating point calculation. Where coefficient sensitivity is a problem the differences at the least significant end of the mantissa are magnified and can (and do) produce differences in the decimal equivalents.

As the simulation proceeds the program output is a reasonable approximation to the expected output and the simulated output is as shown in Figure 3.13. Further confirmation of the causes of this instability was obtained by dispensing with the simulator and modifying the program so that it computed two outputs; the first was produced using parameters which were calculated in the program and the second from assigned parameters. Exactly the same results were obtained.

As might be expected, applying the delta form of the bilinear transform greatly improves the situation. The results compared with the bilinear transform results at selected times through the simulation period are given in Table 3.3.

<table>
<thead>
<tr>
<th>h</th>
<th>time</th>
<th>percentage deviation</th>
<th>delta</th>
<th>bilinear</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01</td>
<td>0.1</td>
<td>2.0465252611E+00</td>
<td>2.0465207063E+00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.0</td>
<td>-8.9360449650E-01</td>
<td>-8.9360859693E-01</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.0</td>
<td>1.0694731498E+00</td>
<td>1.0694524945E+00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3.0</td>
<td>-3.6013925718E-01</td>
<td>-3.6013911756E-01</td>
<td></td>
</tr>
<tr>
<td>0.0005</td>
<td>0.1</td>
<td>1.0678271157E-01</td>
<td>-2.9453483803E-01</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.0</td>
<td>-4.6709418436E-02</td>
<td>-2.3113863938E+00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.0</td>
<td>5.8707192680E-02</td>
<td>-5.6549840761E+00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3.0</td>
<td>-2.0380728831E-02</td>
<td>-5.1779319843E-01</td>
<td></td>
</tr>
<tr>
<td>0.0001</td>
<td>0.1</td>
<td>2.0009400843E-02</td>
<td>-8.7528992667E-03</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.0</td>
<td>-8.7528992667E-03</td>
<td>1.1019757755E-02</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.0</td>
<td>1.1019757755E-02</td>
<td>-3.8291309241E-03</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.3 A comparison between the delta and un-modified forms of the bilinear transforms.
With a step length of 0.01S the delta and bilinear transforms give almost the same results. When the step length is reduced to 0.0005S the delta representation is noticeably better than the bilinear transform and there can be no comparison when the step length is further reduced to 0.0001S. When considering the delta transform results it is clear that at longer step lengths the algorithmic error is greater than the arithmetic error at the shorter step lengths, which accounts for the improvement as the step length is reduced.

3.5.5 The effect of changing the word and step length.

In the software simulator changes of step length simply mean that the model coefficients and state variable initial values are different. Changing the word length however is much more involved. If the effect of changes is to be studied a separate package of floating point routines must be provided. All variables previously declared as type real are now re-defined as type fpnumber, where fpnumber is a data structure defined in Chapter 1. The mantissa and exponent length are (changeable) program constants and the new arithmetic routines are bundled together into a library called fplib. For all tests the system studied was the previous set of equations for $y_1$, $y_2$ and $y_3$ using the transfer function simulation (TF) test circuits shown in Figure 3.9.

3.5.5.1 Reducing the step length to 0.00005S

Reducing the step length should improve the algorithmic error but make the arithmetic error worse. The results from this test showed that $y_1$ and $y_3$ deviation were greater than before, when the step length was 0.0001S, with the smallest increase corresponding to the stiffest equations. This is to be expected since the $a$ coefficients are proportional to $\pi h^2$ and for the three sets of equations, $n_0$ in the expression for $y_1$ is 1, 9.5 or 49.9. The stiffest equations therefore have the largest $a$-coefficients and therefore the smallest error. The effect on $y_2$ deviation is marginal in set 1 but there is a reduction of between 4 and 5 times in sets 2 and 3. Since $y_2$ is a first-order output it might be expected that arithmetic errors are less severe and that reducing the step length reduces the algorithmic error by more than the increase in arithmetic error. The larger the exponential coefficient the greater the improvement.
3.5.5.2 Increasing the step length to 0.0005S

The effect of this change is to reduce deviations in \( y_1 \) and \( y_3 \) but increase them in \( y_2 \). This predictable effect was observed across all three sets of equations.

3.5.5.3 Increasing the mantissa length to 48 bits.

The effect of this change should be to reduce the deviations which are caused by arithmetic error \( (y_1 \text{ and } y_3) \) and have little effect where the error is mostly algorithmic. In actual fact there is little or no change in any of the results. The probable reason for this is that increasing the simulator word length has no effect on the host which still works with a 40-bit mantissa. The Pascal routine which converts a 48-bit pattern representing an fpnumber to a decimal number cannot hold the result in anything greater than 40 bits, whatever the simulator word length. The only way that increasing the simulator word length could have an effect would be if the normalisation procedure caused bits which were previously discarded to be shifted up into bit position 40 and above. Since the \( a \times X \) products are all comparable in magnitude this did not happen and thus the benefit of the additional byte is minimal.

3.5.5.4 Reducing the mantissa length to 32 bits.

As might be expected, this has the effect of increasing errors in all the tests. The smallest increases were in the first-order, \( y_2 \), result where arithmetic error is less of a problem. It is worth noting however that the results from this configuration of data processor were significantly better than those obtained when separate, 40-bit, digital integrators were simulated. This is important in relation to hardware implementation since there are already a range of 32-bit arithmetic support chips on the market. The availability of 40 and 48 bit chips is less certain.
3.6 Conclusion.

Apart from verifying the data processor configuration, the main aim of these experiments was to investigate different transforms, to quantify the effect of simulating $n^{th}$ order transfer functions in a single processing element and to study the effect of step and word length changes. What the tests have shown is that the word length of the system cannot usefully be extended beyond that of the host system. 32-bit mantissae give satisfactory results but, in general, the longer the word length the better. It has also been shown that, with this particular set of equations, results are better when the step length is increased to 0.0005S from 0.0001S. The only problem with such a five-fold increase is that it is likely to reduce the upper limit of the real-time response of the hardware system by the same amount. Overall, it is expected that a data processor with a step length of 0.0001S and a word length of 32 bits will deliver acceptable performance. If the architecture of processing elements provides word lengths of greater than 32 bits, then so much the better, but the extra cost and effort of supplying hardware to increase word length beyond 32 bits is probably not justified.

The tests have also shown that the delta form of the bilinear transform becomes progressively more effective as the order of the system increases. The predictor transform is the least satisfactory and the bilinear transform susceptible to coefficient sensitivity. The maximum order of transfer function which can be simulated is probably three, with higher order functions assigned to more than one processor.

3.7 References.


4. Other processing elements.

4.1 Introduction.

Data processors by themselves will not be sufficient for the majority of simulations. They need to be supported by a variety of other analogue processing elements to operate on numerical quantities, and logical processors to generate the logic levels which control a simulation. In this chapter several new processing elements which are needed in particular simulations, will be defined. In principle there is no limit to the number of different PEs which can be proposed. In practice the limit is imposed by the capacity of the host storage system. The elements discussed here are considered to be representative of the types which are expected to be available in a fully functional simulator.

4.2 Simulation of a Chemical Process - the multiplier.

A particular chemical process is defined by the equations,

\[ y_1' = -0.04 \cdot y_1 + 10^4 \cdot y_2 y_3 \]
\[ y_2' = 0.04 \cdot y_1 - 10^4 \cdot y_2 y_3 - 3 \times 10^7 \cdot y_2^2 \]
\[ y_3' = 3 \times 10^7 \cdot y_2^2 \]

The exact solution of this set of equations is unknown but as the reaction proceeds, the relationship between the variables \( y_1, y_2 \) and \( y_3 \) is always,

\[ y_1 + y_2 + y_3 = 1.0 \]
The initial and final conditions are,

\[
\begin{align*}
\text{At } t = \text{zero}; & \quad y_1 = 1.0; \quad y_2 = y_3 = 0.0 \\
\text{At } t = \text{infinity}; & \quad y_1 = y_2 = 0.0; \quad y_3 = 1.0
\end{align*}
\]

Apart from highlighting the need for a multiplier and a squarer, the stiffness and non-linearity of the equations are a severe test for hardware [1] and software simulators. An unscaled analogue computer solution is shown in Figure 4.1. The squaring circuit is a second multiplier with the operand connected to two separate inputs and the integrators are assumed to have an input summer and a means of incorporating initial values.

Figure 4.1 An analogue computer solution to the chemical reaction problem.
4.2.1 The multiplier.

Apart from the value and router arrays at input and output, the only storage needed for the multiplier is one word, mlimit, for the comparator limit function. In all analogue-type processors, including the data processor, one of the initialised parameters is a threshold limit, in this case mlimit. The analogue output, f(u), is compared with this limit and logical levels are set accordingly. In the multiplier a greater-than or equal to (\(\geq\)) and a less-than (\(<\)) indication is given. The numbers to be multiplied are routed to processor inputs 1 and 2 and the product is delivered at processor output 1. The other outputs are defined in the figure.

\[
\text{Figure 4.2. The multiplier. } x \text{ and } y \text{ must be routed to processor inputs 1 and 2.}
\]

When the device is reset the product outputs go to 0.0, the \(\geq\) and \(<\) outputs are logically 0 and 1 respectively and the sign bit is set to logical 0.

4.2.2 Process Simulation.

The connection diagram for this simulation is shown in Figure 4.3. Each of the data processors (PE1, 2 and 3) is configured for transfer function simulation and the input multipliers are
provided by setting the k-coefficients. (Refer to figure 3.4). The bilinear transform is used with a step length of 100 microseconds and all the real numbers have a 40-bit mantissa. The simulation results are summarised in table 4.1.

Figure 4.3 Digital simulation of the chemical reaction problem.

The hardware simulator which was tested with same set of equations was said to 'completely lose' the simulation after about 12 seconds. This is clearly not the case with this simulator, and, if the checksum is any measure of performance, there is seen to be a marginal, but
consistent improvement as the simulation goes on. No tests were made with the delta form of the bilinear transform since the point of the exercise was to establish the operation of a multiplier and not compare transforms.

### 4.3 The third-order system revisited - logical processing elements.

The problems of simulating a third-order system in a single PE were recorded in Chapter 3. As part of the debugging process the system was simulated as three, first-order systems connected as shown in Figure 4.4. The aim was to verify that some measure of agreement between theoretical and measured responses could be obtained. In Figure 4.4, PE4 is configured as a step function generator which drives the simulation, and PE8 as a summing amplifier. The coefficient and state variable initial value assignments, based on the bilinear transform, for all these ‘analogue’ processors are given in the figure.

<table>
<thead>
<tr>
<th>Time</th>
<th>y₁</th>
<th>y₂</th>
<th>y₃</th>
<th>y₁ +y₂ +y₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>9.9999999999E-01</td>
<td>0.0</td>
<td>0.0</td>
<td>9.9999999999E-01</td>
</tr>
<tr>
<td>0.5</td>
<td>9.8179036788E-01</td>
<td>3.3280904296E-05</td>
<td>1.8174419797E-02</td>
<td>9.999986559E-01</td>
</tr>
<tr>
<td>1.0</td>
<td>9.6645749381E-01</td>
<td>3.0746120423E-05</td>
<td>3.505888595E-02</td>
<td>9.999981253E-01</td>
</tr>
<tr>
<td>2.0</td>
<td>9.410641361E-01</td>
<td>2.7017584032E-05</td>
<td>5.8364784715E-02</td>
<td>9.9999821950E-01</td>
</tr>
<tr>
<td>4.0</td>
<td>9.05515531E-01</td>
<td>2.2404486410E-05</td>
<td>9.440781480E-02</td>
<td>9.999983128E-01</td>
</tr>
<tr>
<td>6.0</td>
<td>8.792650574E-01</td>
<td>1.959468812E-05</td>
<td>1.071323601E-02</td>
<td>9.999984255E-01</td>
</tr>
<tr>
<td>8.0</td>
<td>8.585456280E-01</td>
<td>1.7663576142E-05</td>
<td>1.414353276E-01</td>
<td>9.9999848913E-01</td>
</tr>
<tr>
<td>10.0</td>
<td>8.413668519E-01</td>
<td>1.6233733802E-05</td>
<td>1.568156182E-01</td>
<td>9.9999853718E-01</td>
</tr>
<tr>
<td>12.0</td>
<td>8.265525726E-01</td>
<td>1.5120387825E-05</td>
<td>1.733281975E-01</td>
<td>9.9999857521E-01</td>
</tr>
<tr>
<td>16.0</td>
<td>8.022729345E-01</td>
<td>1.347597931E-05</td>
<td>1.9771238263E-01</td>
<td>9.9999865188E-01</td>
</tr>
<tr>
<td>20.0</td>
<td>7.8241658741E-01</td>
<td>1.2299176017E-05</td>
<td>2.176682932E-01</td>
<td>9.9999871590E-01</td>
</tr>
<tr>
<td>24.0</td>
<td>7.651187760E-01</td>
<td>1.140232923E-05</td>
<td>2.343754860E-01</td>
<td>9.9999876608E-01</td>
</tr>
<tr>
<td>28.0</td>
<td>7.510003161E-01</td>
<td>1.0688340041E-05</td>
<td>2.498758699E-01</td>
<td>9.9999880704E-01</td>
</tr>
<tr>
<td>32.0</td>
<td>7.360273295E-01</td>
<td>1.0101982474E-05</td>
<td>2.6193600605E-01</td>
<td>9.9999894066E-01</td>
</tr>
<tr>
<td>40.0</td>
<td>7.152513957E-01</td>
<td>0.9185486792E-06</td>
<td>2.641645818E-01</td>
<td>9.9999898323E-01</td>
</tr>
<tr>
<td>48.0</td>
<td>6.971269183E-01</td>
<td>8.4927214292E-06</td>
<td>3.0286351890E-01</td>
<td>9.999990325E-01</td>
</tr>
<tr>
<td>56.0</td>
<td>6.805498151E-01</td>
<td>7.9440593717E-06</td>
<td>3.194103071E-01</td>
<td>9.9999905562E-01</td>
</tr>
<tr>
<td>64.0</td>
<td>6.667066035E-01</td>
<td>7.4947950713E-06</td>
<td>3.3332081860E-01</td>
<td>9.9999907375E-01</td>
</tr>
</tbody>
</table>

Table 4.1 The chemical reaction test results.
Figure 4.4: Simulation of a 3rd-order system using 1st-order data processors.

<table>
<thead>
<tr>
<th>PE4</th>
<th>PE5</th>
<th>PE6</th>
<th>PE7</th>
<th>PE8</th>
</tr>
</thead>
<tbody>
<tr>
<td>b1 = -1</td>
<td>b1 = -1</td>
<td>b1 = -1</td>
<td>b1 = -1</td>
<td>b1 = 0</td>
</tr>
<tr>
<td>a0 = 1</td>
<td>a0 = a1 = h/2</td>
<td>a0 = a1 = h/2</td>
<td>a0 = a1 = h/2</td>
<td>a0 = 1</td>
</tr>
<tr>
<td>a1 = 0</td>
<td>X1 = x(0)/h - ( \sum(0) )/2</td>
<td>X1 = x(0)/h - ( \sum(0) )/2</td>
<td>X1 = x(0)/h - ( \sum(0) )/2</td>
<td>a1 = 0</td>
</tr>
<tr>
<td>X1 = 1</td>
<td></td>
<td></td>
<td></td>
<td>X1 = 0</td>
</tr>
</tbody>
</table>

x(0) = 0.4

\( \sum(0) \)

x(0) = 0.6

\( \sum(0) \)

x(0) = 0.8

\( \sum(0) \)

0.5
To remove any possibility of sequencing errors, processors 5 to 8 were programmed to compute a new output every fourth step. The active, compute (C), cycles of each processor were staggered as follows:

<table>
<thead>
<tr>
<th>Processor -&gt;</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step -&gt; 1</td>
<td>C</td>
<td>H</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td>2</td>
<td>H</td>
<td>C</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td>3</td>
<td>H</td>
<td>H</td>
<td>C</td>
<td>H</td>
</tr>
<tr>
<td>4</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>C</td>
</tr>
<tr>
<td>5</td>
<td>C</td>
<td>H</td>
<td>H</td>
<td>H</td>
</tr>
</tbody>
</table>

'H' indicates that a processor is in HOLD mode and 'C' that it is computing a new output. The simulation therefore takes four times longer than normal to execute, but all the processors are now in step with one another. Two additional logical processors, a timer and a step-delay unit, were needed to generate this pattern of mode control signals.

4.3.1 The Timer.

Timing is implemented by counting compute steps. With a step length of 100 microseconds.
and a step counter of \( n \) bits, the maximum period which can be timed-out in a single unit is \((2^n - 1) \times 10^4\) seconds. To keep track of the operation, the timer, which is shown in Figure 4.5, maintains two registers, an initial value, or \( I_{\text{value}} \), register, and a step count, or \( t_{\text{count}} \), register. When the timer is initialised or reset, a number (the timing interval divided by 100 microseconds) is written into the \( I_{\text{value}} \) register and the \( t_{\text{count}} \) register is cleared. At each \( \text{COMPUTE} \) step thereafter, the \( t_{\text{count}} \) contents are incremented and compared with \( I_{\text{value}} \). When the two are the same the required timing period has elapsed.

In \( \text{COMPUTE} \) mode counting can be inhibited by controlling the enable input at processor Input 1. The initial state of the enable pin is programmable and this selected (initial) level is restored whenever the device is reset. The internal state of the timer is shown by processor outputs 1 and 2. These are designated A and B and when the timer is reset they are set low and high respectively. In \( \text{COMPUTE} \) mode they remain in this state so long as the timing period has not elapsed. \((t_{\text{count}} < I_{\text{value}})\). When \( t_{\text{count}} \) reaches \( I_{\text{value}} \), the outputs change to high and low and remains there until the device is reset. A and B are therefore true and complemented logical outputs. By manipulating the enable and mode control inputs, a number of different digital waveforms can be generated at A and B.

The only other facility which the timer provides is access to the contents of \( t_{\text{count}} \) at processor output 3. Currently however, no other processor is able to use this information.

Figure 4.6 The step delay generator.
4.3.2 The step delay unit.

This device is neither analogue or logical since it treats both kinds of inputs alike. When operating in COMPUTE mode all arrivals at processor inputs 1 to 6, whether they be analogue values or logical levels, are delayed by one step period and then passed across to the output. As shown in Figure 4.6, there are therefore six distinct delay paths in each unit and no extra storage is needed to provide the function.

In RESET mode all inputs and outputs are set to zero and in HOLD mode the delaying action is suspended.

4.3.3 System simulation.

Figure 4.7 shows how two timers and one half of a step-delay unit are used to generate the re
Figure 4.8 Connection diagram for third-order system simulation.
quired COMPUTE/HOLD signals. Processor pin 8 on data processors 5 to 8 are maintained low and the waveforms shown in Figure 4.7 are connected to pin 7. The data processors are therefore in HOLD mode for 75% of the simulation time and in COMPUTE for the remaining 25%. The phasing of the modes is as discussed earlier.

The full connection diagram for this application is shown in Figure 4.8. Over a range of step-lengths, the simulation results were always of the correct form. The smallest step length (h = 0.0001s) gave the best results and it was possible to adequately account for all differences between the theoretical and measured values. Although the production of staggered excitation waveforms is probably unnecessary, certainly at the shortest step lengths, both the timer and the step-delay unit would be needed in an operational simulator.

4.4 Nyquist diagrams for simulated systems.

There are several different ways in which Control Engineers demonstrate the frequency re

![Nyquist Diagram]

Figure 4.9 Technique for producing Nyquist plots.
response of a system. One way is to produce a Nyquist plot in which the system frequency re-
sponse, $G(j\omega)$, is plotted as a vector when $\omega$, the input frequency, is varied between 0 and in-
finitiy. The axes of the plot are the real and imaginary parts of $G(j\omega)$. The system under test is
assumed to be excited by a sinusoid, $A \cos(\omega t)$, and the output at some value of $\omega$ is $B \cos(\omega t + \alpha)$. Figure 4.9 shows the path of the $G(j\omega)$ vector assuming $A = 1$.

Since,

$$B \cos(\omega t + \alpha) = B(\cos(\omega t) \cos(\alpha) - \sin(\omega t) \sin(\alpha))$$

then at $\omega t = \pi/2$, the system output is,

$$- B \sin(\alpha) \quad (=-y)$$

and at $\omega t = \pi$ it is,

$$- B \cos(\alpha) \quad (=-x)$$

The problem is therefore to capture the system output at $\pi/2$ and $\pi$ and plot one value against
the other. The only other thing to note is that measurements should only be made under
steady state conditions. There must be time for the transient effect [2] of changing the system
input to subside before the measurement process is started.

The test system input is generated using a cosine function generator. It is connected to the
system being tested and the low-to-high change at its sb (sign bit) output is used as the $\pi/2$
 sample timing signal. The sb output of a sinewave function generator, which operates in paral-
lel with the cosine generator, gives a low-to-high change at the $\omega t = \pi$.

The angular frequency of both waveform generators is read as a number routed to processor
pin 1. This is supplied by a step generator which is initialised to provide a default (initial) value
for $\omega$. The step generator also holds an output multiplying factor. Each active edge at the step
generator's control input caused the current output to be multiplied by the multiplying factor.
During TP this new value is passed to the connected waveform generators and in the sub-
sequent EP, read as a new value of $\omega$. An active edge is timed to occur at the end of every
measurement cycle.
The measurement delay, during which transients decay, is achieved by counting out a number of cycles of the sinusoidal generator before the phase detection process begins. The requirement is therefore for an event counter, where an event is defined as a low-to-high transition at the counter input. In this case, the low-to-high transition coincides with $\omega t = 0$. The count input is therefore the not sb output of the sinewave generator. Experiment showed that a five cycle delay was adequate at all the measurement frequencies used with the test system.

The only other new device which was used in this application was a sampler which captures the system output at the appropriate times. The signal being sampled is routed to processor pin 1 and the low-to-high change at any of the control inputs causes the sampled input to be held at the corresponding output.

The explanatory diagram of Figure 4.10 shows how these new devices are connected.

![Explanatory Diagram](image)

**Figure 4.10** Schematic diagram of the Nyquist plot test system.

The system under test is assumed to occupy one processing element and the delay generator, which is connected in parallel with it, is used to delay sampling by one step period. The READ signal is an event which is recognised in the host program as indicating that a measurement cycle has been completed. The values are then read from the sampler output, the
counter is reset and step generator triggered to supply a new angular frequency. The following sections describe the new devices in more detail.

4.4.1 The waveform generators.

Sinusoidal and cosinusoidal waveforms are produced by processing elements which simulate a second order system with zero damping. They are data processors in which the coefficients and state variable initial values are dependent only on \( \omega \) and \( h \). When waveform generators are specified, the user supplies an initial value for \( \omega \) and a value for \( h \). From these the model parameters are calculated. A change in \( \omega \) is indicated by the input on processor pin 1 being different from the internally held value of \( \omega \). If a change occurs, the model parameters are recalculated and the cycle begins again. At RESET, the default (initialised) value of \( \omega \) is used in the calculation.

![Diagram of waveform generator](image)

Figure 4.11 The waveform generator.

Figure 4.11 shows that the waveform is delivered at processor pin 1, the sign bit at processor pin 2 and the complemented sign bit at pin 3. All other outputs are, currently, undefined.
4.4.2 The step generator.

The step generator (Figure 4.12) has an initial (output) value register, \( I \), a multiplier register, \( n \), and storage for a Boolean variable called \( \text{last}_\text{ip} \). The control input is pin 1 and in COMPUTE mode, a low-to-high change at this pin causes a new output, which is \( n \)-times the previous output, to be generated. This produces a geometrical progression of numbers, starting with \( I \), with each advance triggered by a control input change of the correct polarity. \( \text{last}_\text{ip} \) records the previous control input level. In every COMPUTE cycle it is compared with the current level so that active edges can be detected.

![Figure 4.12 The step generator.](image)

In RESET mode the output is restored to its initial value and \( \text{last}_\text{ip} \) is set high. In HOLD mode any changes at the control input are ignored and the current output is maintained. The only output so far defined for this device is \( f(u) \) on processor pin 1.

4.4.3 The event counter.

This is very similar to the timer except that it counts events - changes at processor input 1 - rather than step periods. It has an end_count register, which is similar to the timer's initial
value register, and a current_count register which is similar to the timer’s count register. The contents of current_count are incremented each time an event occurs and end_count indicates how many events need to be counted. If current_count is less than end_count processor output 1 and 2 are low and high respectively. When the two counts are equal the outputs change state and counting is inhibited until the counter is first RESET and then switched to COMPUTE.

Figure 4.13 The event counter.

Low-to-high changes at the input (event) pin are detected by comparing previous (last_level) and present levels, as in the step generator. Figure 4.13 shows all the counter pin assignments.

4.4.4 The sampler.

This is a device (Figure 4.14) which has one analogue input, five control (sample) inputs and the usual mode controls. The analogue input is routed to processor input 1 and and output 1 is a copy, one step delayed, of the input. When a low-to-high change occurs on one of the sample controls the analogue input is sampled and the output corresponding to the active control retains the current sample amplitude. In effect the device behaves as a multiple track-
The technique for detecting active edges is as described before. When RESET all outputs are taken low and in HOLD mode operation is suspended.

4.4.5 The system simulation.

The connection diagram for this application is shown in Figure 4.15. Where no connections are made to the mode control inputs it is assumed that the elements have been correctly initialised and that they operate in COMPUTE mode for the duration of the test.

Two series of measurements were made. The first considered the first order system,

\[
G(s) = \frac{1}{s + 1}
\]

The angular frequency, \( \omega \), was varied between 8 and 0.125 rad/s and two different step lengths, \( h = 0.01 \) and \( 0.0001 \) s, were used. In the second series the test circuit was a second order system having the transfer function,
Figure 4.15. Connection diagram for the Nyquist plot experiments.
G(s) = \frac{\omega_n^2}{s^2 + 2\omega_d s + \omega_n^2}

In this case \(\omega_n\) was 10 rads/sec and the three tests were made with the damping factor, \(d\), set at 1, 1.5 and 0.5.

The correctness of the test arrangement was verified by comparing the calculated response with the simulator output. Since measured values are captured at zero crossings of sine and cosine waves, the accuracy of the measurement depends upon how clearly these crossings are defined. For a sinusoid of unit amplitude and angular frequency \(\omega\), the maximum change of amplitude between steps is \(\omega h\) volts. This occurs at the zero crossing point so that the maximum allowable difference between the calculated and measured value is also \(\omega h\) volts. In almost every case the differences were significantly less than that. The limit was exceeded however when the first order system was tested with \(\omega\) set at 0.5, 0.25 and 0.125 rads/sec. and the step length was 0.0001s. Table 4.2 records this result.

<table>
<thead>
<tr>
<th>Omega</th>
<th>Measured Real</th>
<th>Measured Imaginary</th>
<th>Deviation Real</th>
<th>Deviation Imaginary</th>
<th>Allowable</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Real</td>
<td>Imaginary</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0.01567916</td>
<td>-0.1228239</td>
<td>0.00029</td>
<td>0.00025</td>
<td>0.0008</td>
</tr>
<tr>
<td>4</td>
<td>0.9416452</td>
<td>-0.2344139</td>
<td>0.0000235</td>
<td>0.0000217</td>
<td>0.0004</td>
</tr>
<tr>
<td>2</td>
<td>0.8002113</td>
<td>-0.3998065</td>
<td>0.000051</td>
<td>0.000022</td>
<td>0.0002</td>
</tr>
<tr>
<td>1</td>
<td>0.5007978</td>
<td>-0.4999498</td>
<td>0.000079</td>
<td>0.0000502</td>
<td>0.0001</td>
</tr>
<tr>
<td>0.5</td>
<td>0.20005107</td>
<td>-0.3999780</td>
<td>0.00021</td>
<td>0.000294</td>
<td>0.00005</td>
</tr>
<tr>
<td>0.25</td>
<td>0.05884672</td>
<td>-0.2352723</td>
<td>0.000469</td>
<td>0.00088</td>
<td>0.000025</td>
</tr>
<tr>
<td>0.125</td>
<td>0.9848240</td>
<td>-0.1219957</td>
<td>0.000209</td>
<td>0.00108</td>
<td>0.0000125</td>
</tr>
</tbody>
</table>

Table 4.2 Nyquist plot for a first-order system using the bilinear transform and a step length of 0.0001s.

Because the software simulator is sequential rather parallel, and because a measurement delay which allows transients to subside is included, the measurement time for \(\omega = 8\) rads/sec. is approximately 30 minutes. As \(\omega\) is reduced by a factor of two, the measurement time increases by the same amount. When \(\omega\) is 0.125 rads/sec., where the difference was the grea-
test, the measurement time is approximately 32 hours! It was suspected that the combination of small step length (and the consequent effect of that on coefficient values) and excessively long run times, was producing an accumulation of errors which eventually exceeded the limit. To check this assertion, the whole series of measurements was repeated using the delta form of the bilinear transform. It is known that this reduces coefficient sensitivity, although the run times cannot be reduced. The results from these tests, listed in Table 4.3, show that the differences are now within the allowable limit.

<table>
<thead>
<tr>
<th>Omega</th>
<th>Measured Real</th>
<th>Measured Imaginary</th>
<th>Deviation Real</th>
<th>Deviation Imaginary</th>
<th>Allowable</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0.01559834</td>
<td>-0.1228213</td>
<td>0.0002137</td>
<td>0.0002555</td>
<td>0.0008</td>
</tr>
<tr>
<td>4</td>
<td>0.05865565</td>
<td>-0.2352700</td>
<td>0.000321</td>
<td>0.000024</td>
<td>0.0004</td>
</tr>
<tr>
<td>2</td>
<td>0.2000323</td>
<td>-0.3999846</td>
<td>0.000323</td>
<td>0.0000154</td>
<td>0.0002</td>
</tr>
<tr>
<td>1</td>
<td>0.5000403</td>
<td>-0.4999615</td>
<td>0.000403</td>
<td>0.0000385</td>
<td>0.0001</td>
</tr>
<tr>
<td>0.5</td>
<td>0.8000122</td>
<td>-0.3999785</td>
<td>0.0000122</td>
<td>0.0000215</td>
<td>0.00005</td>
</tr>
<tr>
<td>0.25</td>
<td>0.9411777</td>
<td>-0.2352926</td>
<td>0.0000013</td>
<td>0.0000015</td>
<td>0.000025</td>
</tr>
<tr>
<td>0.125</td>
<td>0.9846160</td>
<td>-0.1230759</td>
<td>0.0000007</td>
<td>0.000001</td>
<td>0.0000125</td>
</tr>
</tbody>
</table>

Table 4.3 Nyquist plot for a first-order system using the delta form of the bilinear transform and a step length of 0.0001s.

When the second order system was tested, the measurement frequencies were higher and excessive differences were not therefore observed. The Nyquist plots were predictably those of an over-damped, critically damped and under-damped second order system.

4.5 Simulation of an aircraft arresting gear system.

This is an application which has been used to demonstrate the capabilities of two well-known simulation packages, ACSL [3] and SYSL [4]. The system being simulated is designed to halt an aircraft which is landing on the deck of an aircraft carrier or a runway which is too short to allow unaided deceleration. The system diagram is given in Figure 4.16.
Figure 4.16  Aircraft arresting gear system.
A hook at the rear of the aircraft catches a restraining cable which pulls against 'water squeezer' energy absorbers. The purpose of the simulation is to determine how the aircraft approach speed and weight are related to the permissible tensile forces in the cable and the allowable piston travel. The system equations are as follows,

\[
\begin{align*}
\ddot{y}_3 \quad \frac{m_3}{dt^2} &= f_{K2} - f_0 \\
\ddot{y}_2 \quad \frac{m_2}{dt^2} &= 2f_{K1} - f_{K2} \\
\ddot{y}_1 \quad \frac{m_1}{dt^2} &= -2f_{K1} \sin(a)
\end{align*}
\]

The cable tension are given as,

\[
\begin{align*}
f_{K1} &= k_1(y_1 - 2y_2) \quad \text{if } y_1 > 2y_2 \\
&= 0 \quad \text{otherwise} \\
f_{K2} &= k_2(y_2 - y_3) \quad \text{if } y_2 > y_3 \\
&= 0 \quad \text{otherwise}
\end{align*}
\]

The drag force from the water squeezer is given by,

\[
f_0 = F(y_3) \left[ \frac{dy_3}{dt} \right]^2
\]

From the geometry of the system, the following relationships hold,

\[
\begin{align*}
y_1 &= (x^2 + h^2)^{0.5} - h \\
\sin(a) &= \frac{x}{h + y_1} = \frac{x}{(x^2 + h^2)^{0.5}}
\end{align*}
\]

A first attempt at a simulation diagram is given in Figure 4.17. It illustrates the need for a non-linear function generator to produce \(F(y_3)\), a divider, a square root function and a combined comparator/function generator. In this case the square root is part of a user-defined 'special' processing element. In the example simulations, all weights, including that of the aircraft, the spring constants \(k_1\) and \(k_2\), and the dimension \(h\), are all supplied as program constants. The only system input is the aircraft's initial velocity, \(dx/dt\). As far as this study is concerned, the
Figure 4.17 Simulation of the aircraft arresting gear system.
relevant aspects of the example are the new processing elements.

4.5.1 The non-linear function generator.

Functions like sine waves, ramps, steps and saw-tooth waveforms can be produced with existing processing elements. Irregular functions are produced by non-linear function generators. When a non-linear function generator is initialised, values for \( x \) and \( f(x) \), however determined, are written into two arrays and, as each pair is entered, a count of the number of points is incremented. Currently the array sizes are limited to 100 elements and it is assumed that the values of \( x \) increase as the array is scanned from its lower bound.

The COMPUTE code for the device is an interpolation procedure. The value of \( x \) is read from processor pin 1 and the appropriate \( f(x) \) is delivered at output pin 1. Lagrangian [5] interpolation was used to test the operation of the function generator. The relevant formula is,

\[
f(x) = L_n(x) = \sum_{k=0}^{n} \frac{k(x)}{k(x_k)} \cdot f_k
\]

Where

\[
l_0(x) = (x-x_1) \cdots (x-x_n)
\]

\[
l_k(x) = (x-x_0) \cdots (x-x_{k-1})(x-x_{k+1}) \cdots (x-x_n)
\]

\[
l_n(x) = (x-x_0) \cdots (x-x_{n-1})
\]

Acceptable values were obtained when the COMPUTE mode code was tested with several known functions, although no attempt was made to measure computation time. It may be necessary to accept less accurate, but more speedily computed, results in a working simulator. This would mean working with interpolation polynomials of order less than \( n \) (with \( n + 1 \) points defined) but this is not thought likely to be a major problem. Interpolation (and extrapolation if needed) are standard computational procedures.

4-24
4.5.2 The comparator/function (c/fg) generator.

This device is supplied with a constant multiplier, \( k \), and an arithmetic operator, given the legend \(< \text{op}>\), which can be \('\,'+', '-', '*', or '.'\). Two signals are routed to processor inputs 1 and 2 and each input has associated with it a programmable multiplier \( m_1 \) and \( m_2 \). With the following relationships,

\[
A = m_1 \times \text{Invalue}[1] \\
B = m_2 \times \text{Invalue}[2]
\]

then,

\[
\text{output} = \begin{cases} 
  k(A < \text{op} > B) & \text{if } A > B \\
  0 & \text{otherwise}
\end{cases}
\]

The application of such a processing element in this example is obvious. The function \( f_{x1} \) for example would be produced by setting \( m_1 = 1; m_2 = 2 \) and \(< \text{op} > = '+'\).

4.5.3 User-defined processing elements.

There are precedents for specialised computer suppliers to provide facilities which allow users to add to the basic instruction set of the machine and in the scientific community, the ability to tune a computer for particular kinds of application is often valued. One user-defined element is shown in Figure 4.17. It generates the variable \( y_1 \) and the denominator in the expression for \( \sin (a) \). No internal storage is needed in this particular device, although that is not thought to be a necessary attribute of all user-defined PEs.

It is envisaged that the PE definition would take place in a separate editing session and that converting it to a code block on disk will be similar to compilation. What is needed is an acceptable software-user interface and guidelines about estimating execution time and maximum code length. It is a facility that not all users will require but a worthwhile provision nonetheless.
4.6 Conclusion

The concern of this Chapter has been to define a number of different processing elements which can be used with the data processors described in chapter 3. The definition code and data structures for all the different kinds of processors are stored on disk and selected code blocks are retrieved when a processing element of a particular sort is required. The relevant PE parameters are initialised by the user before the code and data are downloaded to a target processing element. This follows the interconnection phase.

Since it is not possible to predict all the different processor types which might be needed, a facility will be provided for users to define special units which are useful in particular kinds of application. This will reduce the constraints imposed by having only a limited number of system-defined processing elements. The next Chapter discusses the hardware design of a processing element based on transputers and Occam.

4.7. References.


[3] Advanced Continuous Simulation Language (ACSL). Distributed by: Mitchell and Gauthier Associates. 73, Junction Square Dr., Concord, MA 01742. USA.

5. A Transputer-based Simulator.

5.1 Introduction.

The possibility of using a transputer as the main component in a processing element was discussed in Chapter 2. Despite what at first appears to be its major drawbacks, there are also considerable benefits to be had from using transputers. This chapter describes the design of a processing element, and other simulator sub-systems, based on the transputer, and the steps which were taken to compensate for the transputer deficiencies.

5.2 The Transputer.

The Inmos view of a transputer [1] is that it is a 'building block for concurrent processing systems, with Occam as the associated design formalism.' Occam [2][3] is the high-level language which was developed in tandem with the transputer and is said to be the first language specifically designed for programming concurrent distributed systems. Application programs written in Occam are described as a collection of processes which communicate through channels. The hardware implementation of a channel is a transputer link so that processes can be assigned to different processors and communicate over links or they may execute on the same processor and communicate over software channels.

Each of the three processors in the transputer family has four links, various amounts of on-chip RAM and a configurable external memory interface. The T800, shown in Figure 5.1, also has a floating point arithmetic unit which operates concurrently with and is controlled by the CPU. Double (64-bit) and single-length floating point operations are ten-times faster in hardware compared with software execution in the T414. The T800 is a 32-bit reduced instruction set computer on a chip, with a clock rate which is adjustable in steps between 20Mhz and 30MHz and the link speeds of 5, 10 or 20Mbps. For the performance estimates which follow, the minimum configuration is considered to be a 20MHz processor with 10Mbps links and the maximum configuration is a 30MHz processor with 20Mbps links.
Figure 5.1 The T800 transputer.
5.3 Occam channels and transputer links.

An Occam process can be thought of as a black box which has an internal state and communicates with other processes using channels. Simple processes can be combined into more complex structures but at the lowest level there are only three basic (primitive) processes, namely the assignment process, the input process and the output process.

An assignment statement simply assigns a value to a named variable and is no different to assignment statements in any other language. The input process reads a value from an Occam channel and assigns it to a variable, and an output process sends a value over a channel to another input process. The syntax of these primitive processes is,

\[
\begin{align*}
\text{newval} & : = \text{lastvalue} & \quad \text{-- assign lastvalue to newval} \\
\text{chanin} & \ ? \ \text{fred} & \quad \text{-- read from chanin to fred} \\
\text{chanout} & \ ! \ \text{fred} & \quad \text{-- send value of fred over chanout}
\end{align*}
\]

Communication over a channel can only go ahead if the two processes at either end of it are ready. When the channel is a hardware link, the indication that both processes are ready is that the first byte (of what may be a string of bytes) is sent and acknowledged. If the acknowledgement does not arrive (the input process is not ready) or the first byte of an expected string is not in the input buffer (the output process is not ready), then the sending or receiving processes are suspended until the two are in synchronism. Each byte of a message must be separately acknowledged before the next byte can be sent.

Input and output data transfers over transputer links involve,

1. Initialising a pointer to a message buffer,
2. Initialising a channel address (link number),
3. Initialising a byte count, and then
executing an input-message or output-message instruction.

All these operations, which are transparent to the user, configure the link interfaces as DMA-like channels and the transfer then goes ahead with no further involvement on the part of the CPU.

5.3.1 The link adaptors.

Although the multiplexed address and data buses of the transputer are brought out to terminal pins, the intention is that they should be used to connect external memory, not memory-mapped peripherals. The normal way of connecting a non-transputer peripheral device is to use a link adaptor. These have a standard link interface on one side and a conventional parallel interface on the other. The link interface is connected to a transputer link and the adaptor converts the serial link data into parallel data streams which are compatible with conventional microprocessor and peripheral sub-system architectures. Figure 5.2 shows this interface and the more important link adaptor terminal signals.

![Figure 5.2. The Link Adaptor Terminal Signals.](image)

An unconventional orientation of the link adaptor is to reverse the connections so that the parallel interface is memory mapped into the transputer address space. With proper programm-
ing, a write to a link adaptor output port will be the equivalent of a channel output process and a read will be the equivalent of a channel input process. Every added link adaptor increases the number of links by one. However, a significant difference between the link adaptor links and the normal links is that the adaptor links do not have a DMA capability. The provision of buffers and byte counts has to be made in the user program which accesses them.

There are two different link adaptors in the transputer product line, the CO11 and the CO12. The CO11 can operate in two modes with mode selection made by asserting a logic level on the SeparateIQ pin. In mode 1 the device has separate byte-wide I (input) and Q (output) ports, each with its own set of handshake control signals. In mode 2 the parallel interface is a single byte-wide bi-directional port. The CO12 only has a bi-directional peripheral port. Consequently the CO12 is physically smaller than the CO11 and, since its bus structure is compatible with the transputer data bus, it is a suitable choice for this application.

Inside each link adaptor there are four registers which can be accessed by asserting a bit pattern on the two register select (RS) inputs. The relevant lines of the access truth table are,

<table>
<thead>
<tr>
<th>RnotW</th>
<th>RS1</th>
<th>RS0</th>
<th>Register and Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Read from data port - input from the link.</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Write to data port - output to the link</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Read the Input port status register</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Read the Output port status register</td>
</tr>
</tbody>
</table>

Only the least significant bit, bit 0, in either status register is of interest when the link adaptors are used as described. Bit 1 is designated 'Interrupt Enable' but the Interrupt refers to the activity of a peripheral processor when the link adaptor is used conventionally and is ready for service. It is not relevant here. All other bits in the status registers are undefined.

In the output status register, bit 0 is designated 'output available'. When the link adaptor is configured to provide an extra link, the significance of this bit being set is that a serial trans-
mission has been completed and an acknowledgement has been received. A byte can now be sent, in parallel, from the transputer to the link adaptor output data port which causes the 'output available' bit to become false and serial data transmission on LinkOut to begin again. When this is completed and the acknowledgement has been received on LinkIn, 'output available' will again be set.

In the input status register, bit 0 is called 'data present'. When set it indicates that a byte has been received on LinkIn and can be read, in parallel, by the transputer. The action of reading automatically triggers the transmission of an acknowledgement and clears 'data available'.

5.4 Interconnecting processing elements using link adaptors.

Since the transputer is supplied with four links, the minimum requirement is for an additional 12 link adaptor links so that PEs with 8 inputs and 8 outputs can be built. However, other connections have to be made to a PE and the complication of having a mix of normal links and link adaptor links connected through the global switch is best avoided. It is for these reasons that all 16 of the interconnection links are link adaptor links and the normal transputer links are reserved for control and expansion purposes. Assuming therefore that the down-loaded processing code can be held in the on-chip RAM, a PE board contains a T800 transputer, an array of CO012 link adaptors, a minimal amount of port address decoding logic and some miscellaneous components. The detailed board design is considered in section 5.6.

The printed circuit board tracking is likely to be quite regular, so that a single size Eurocard (100x160mm) should easily be able to accommodate all the components. If external memory is needed however, this will not only occupy board area but also adversely affect the speed and cost of a processing element. The design aim is therefore to compact all the PE code so that it fits into the on-chip RAM, and to configure the hardware and software so that the slowest (busiest) PE allows the simulator to operate with a step length of 100 microseconds. PE performance is therefore measured in terms of code size and speed.
5.5 Performance estimation.

For the purposes of estimating code size and execution speed, the activities of a processing element are divided into transmission, routing, and execution. Each of these activity phases takes time to complete and each involves the execution of program which is stored in memory.

5.5.1 The transmit period.

At the beginning of every computation step, numerical and logical values computed in the previous step, are transmitted between connected processors. Since the connecting links are all provided by link adaptors, the sending and reading of values should be preceded by a check of appropriate bits in the link adaptor status registers. The output-byte process for instance might be written as,

```
begin
  read the output status byte
  while D0 in byte ('output available' status bit) is not true
    read the output status byte
  endwhile
  send value to output data port (ie. the link adaptor)
end
```

The link adaptor specification guarantees that link transmission will begin within 2 bit times of the byte being written to the data port. A further 11 bit times are needed to send the byte and the delay in the global switch accounts for another 2 bit times. Assuming therefore that the acknowledgement (ACK) and data transmission overlap, the link adaptor output port is ready to accept a new byte after 13 bit times - the switch delay does not add to that. The test at the start of output-byte is to see that sufficient time has elapsed since the last byte transfer. After 15 bit times, the transmitted byte will have arrived at its destination.

Even if, as in this case, the return path for the acknowledgement is not provided and the ACK is anyway ignored, the minimum time needed to send a 64-bit floating point number through
each link adaptor port would be 104 (8x13) bit times; and this assumes that the other end of the link is perfectly synchronised with the source and not itself attempting to send, rather than read, data. Since all sources are also destinations (processing elements have inputs and outputs and arrivals and departures all occur in the same TP), there is clearly a synchronisation problem and a need to organise the transmit period program so that it runs as fast as possible.

When the first byte of a word is sent through a link adaptor port, bit 0 in the status register must always be true since the port has previously been idle. A test at this point is therefore unnecessary. However, if an n-byte number is sent through one port before the next port in sequence is serviced, the test will be needed for second and subsequent bytes. It would be unnecessary in every case if it could be guaranteed that sufficient time had elapsed since the previous byte transmission for all destinations to have cleared their input buffers. This implies that a byte is sent through a port and then the processor is otherwise engaged until it is time to send the next byte. Since there are other output ports to service, the time between successive byte transfers at one port could be spent in sending a byte through all the other output ports. The obvious way to organise the transmit code is therefore,

```plaintext
for n := 1 to the no_of_bytes per value
  for p :=1 to 8 (the number of inputs and outputs)
    send byte [n] on output link [p]
  end send phase
  for p :=1 to 8
    read byte [n] from input link [p]
  end read phase
end transmit
```

At the beginning of TP, the first bytes of every output are sent and received; then the second bytes, and so on until complete numbers have been transmitted. The destinations then have to construct complete numbers from a dispersed sequence of bytes. Since the link adaptors will accept bytes at the rate they are passed from the source transputer, the loop execution time needs to be at least 13 bit times so that writes to the same output port do not overlap previous write operations. This is not a problem at the transmit end of the link, but it may at the destination. Unfortunately, the existence of routers means that the order in which outputs are sent is not necessarily the order in which inputs will be serviced. In other words, the last byte sent may be the first byte read. If this were the case, the read would start before the trans-
mission was completed, with obvious consequences. To overcome this problem, the read-byte process could be written as,

```
begin
  read the input status byte
  while D0 in the status byte ('data present' bit) is not true
    read the input status byte
  endwhile
  read byte from input data port
end
```

A read from the data port is conditional on the 'data present' bit in the status register being true. In most cases of course, the test will always be true and including it before every read operation is a considerable overhead. It is only inconsiderate router settings which makes the test necessary at any time. A less elegant but faster technique would be to dispense with the test and just wait for 15 bit times before changing from send phase to read phase. After that time the last byte sent must have arrived and so all bytes can be read with safety.

Listing 1, shown over, is one of several Occam program fragments used to test ideas about implementing routing and the transmit period. The aim of these tests was to verify, as far as possible, that the strategy was correct, and to check on memory occupancy and execution speed. Additional code which was needed for testing was stripped away and the compiled fragments were disassembled into transputer mnemonics. Some minor program changes were made, for instance, CHAN OF BYTE was changed to PORT OF BYTE and the port address PLACED AT the proper physical addresses, but essentially the disassembled program was the same as that executed on a single T414.

The disassembler output showed the minimum code space used in any version of transmit was 93 bytes. However, minimal memory occupancy is often allied with slow speed because, although program loops compact the code, the loops are executed several times and the compiler must add and execute end-of-loop test instructions.

Since, in this case, the loop control variable is a constant, a faster but larger program results
Listing 1  The transmit period send and read procedures.

from expanding the loops into a list of assignment statements. When such a program is compiled and then disassembled the transputer (machine) code sequence to send one byte to an output port address is seen to be,

```
pfix; ldl; ldc; stnl;
```

The first two mnemonics mean load the byte to send; the next means load the port address, and the next, store the byte at the port address. The function codes and their operands are 1-byte instructions so that the total memory occupied by the expanded loop which sends one byte of a value through all 8 output ports is 32 bytes. A maximum of 8 bytes of memory are used to store the dummy instructions which idle the process between sending and reading.
On the input side the disassembled code which reads a byte from a port address and stores it in memory is,

```
Idc; Idnl; pfix; st
```

The first instruction sets a pointer to the link adaptor base address, the second reads a byte and the final two instructions store the byte. As before, each instruction occupies one byte of storage so that the sequence which reads a byte across all input ports accounts for 32 bytes of program memory. The total memory requirement for the transmit code is therefore,

For 64-bit numbers - \( 8 \times (32 + 8 + 32) = 576 \text{ bytes} \)

For 32-bit numbers - \( 4 \times (32 + 8 + 32) = 288 \text{ bytes} \)

There is no additional requirement for temporary storage space beyond that which is needed for the routers and processor specific data structures. The RETYPES Occam command does not use extra memory, it simply changes the view of a data structure.

The T800 instruction set summary shows that the send and read instruction sequences both execute in 6 machine cycles. However, because the ports are considered to be the same as external memory locations, the stnl and ldnl instructions times must be increased by 50% from 2 to 3 machine cycles [4]. This increases the total execution time to 7 machine cycles per byte and thus the total transmit period lasts for a time \( T \), where \( T \) is given by,

\[ T = (112 \text{ machine cycles} + 15 \text{ bit times}) \times \text{number of bytes/value} \]

The duration of a machine cycle obviously depends on the processor clock speed and the time to transmit a bit depends on the link speed. For a 20MHz transputer one processor cycle lasts for 50nS. Thus the time \( T \), in microseconds, for various processor and link speed options are,
### 5.5.2 The routing process.

Once an array of bytes has arrived on every link the data structure is re-typed and treated as the input values which will be processed in the next execute period. Similarly, as the execute period comes to an end, output values are generated which, after routing, are divided into bytes and transmitted to the link adaptor output ports. Thus at Input, the routing process, which properly precedes re-typing, can be described as shown in listing 2.

```plaintext
PROC route (VAL [8]INT arrivals,) -- arrivals at each port
    VAL [8]INT inroute, -- being routed
    [8]INT invalues) -- to the correct invalue
SEQ
    invalues[inroute[0]] := arrivals[0]
    invalues[inroute[1]] := arrivals[1]
    :    :    :
    invalues[inroute[7]] := arrivals[7]
```

Listing 2. The input routing process.

Individual bytes arrive on input links and are placed in appropriate input registers with reference to the router settings. Again the loop has been expanded into a list of assignment statements which when compiled occupied 64 bytes of code space. An additional 64 bytes needed for temporary storage. There is an identical requirement at the output so that, in total, routing 64-bit numbers takes 256 bytes and routing 32-bit numbers takes 128 bytes.
The disassembled listing of the routeing process comprised the following transputer code sequence:

```
Idl; Idnl; Idl; wsub; Idnl; Idl; stnl;
```

Executing this sequence accounted for 14 machine cycles with all loads and stores referencing internal memory. The routeing times, in microseconds, are therefore,

<table>
<thead>
<tr>
<th>Processor Speed</th>
<th>20MHz</th>
<th>30MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>32-bit 64-bit</td>
<td>32-bit 64-bit</td>
</tr>
<tr>
<td>2.8</td>
<td>5.6</td>
<td>1.87</td>
</tr>
</tbody>
</table>

Routeing times are clearly independent of the link speed.

For testing, the transmit and routeing code was integrated into the same program. A null PE, in which the inputs were passed directly to the outputs, and input/output router settings were specified and trial (real number) values were written into output values. The program was then executed and the correct arrivals were noted for different router settings and different repetitions with the same settings. Since routeing could, potentially, be carried out in hardware, the program was then dissected so that separate performance estimates for routeing and transmit code sequences could be made. Listings 1 and 2 are therefore program fragments taken from a much larger program.

To check that the link adaptor proposals were correct, two T414 boards, each with an extra link adaptor link, were connected together and identical transmit processes were assigned to each. Extra code was needed because only one, rather than eight, extra links was available, but the acceptability of the send, read and delay strategies across link adaptor links was verified.
5.5.3 The execute process.

Of all the processors in the simulator, the data structure which defines a data processor is the largest and the tasks assigned to the data processor are the most computationally intensive. All processors have invalues and outvalues which are 8-element arrays of 64 or 32 bit numbers, and all have input and output routers which are 8-element arrays of integer. Assuming that the router integer arrays are defined in Occam as type INT16, the minimum memory requirement for all processors is therefore either 160 or 96 bytes depending on the word length. The data processor, which has already been defined, has the following additional parameters,

```occam
data : (ks : array[1..6] of real;
    sigma : real;
    init_sigma : real;
    order : integer;
    Xs : array[0..maxorder] of real;
    init_Xs : array[1..maxorder] of real;
    as (or cs) : array[0..maxorder] of real;
    bs (or ds) : array[1..maxorder] of real;
    limit : real
    );
```

If the maximum order of transfer function which will be simulated is limited to 3, the additional static memory requirement of the data processor is either 176 or 90 bytes. Initialisation of these parameters is carried out in the host so that no program memory is needed to hold initialisation code.

The instruction sequence used to model a data processor using the bilinear transform in the software simulator was translated into Occam as shown in listing 3 at the end of the chapter. As before, the compiled program was disassembled into transputer code and estimates of memory occupancy and execution speed were made. When 64-bit numbers were specified the execute period code occupied 686 bytes. With 32-bit numbers this was reduced to 530 bytes. There was also a need for 11 and 7 bytes, respectively, of additional data storage so that total memory space figures were 697 and 537 bytes. When the delta form of the execution code was disassembled the corresponding figures were 709 and 553 bytes respectively. The addition part of the shift-and-add accounts for the increase.
Estimation of execution time is more difficult in this case because the program includes several conditional statements and is much longer than the other fragments. In several instances, maximum and typical execution times are given for particular instructions and alternative paths through conditional statements are not of equal length. Estimating the longest path through the program involves consideration of the mode and the longest path through each of the conditional statements. Although a data processor in HOLD mode is very lightly loaded, RESET and COMPUTE mode programs are of similar duration as shown below,

<table>
<thead>
<tr>
<th>Machine Cycles to execute</th>
<th>BILINEAR</th>
<th>DELTA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RESET</td>
<td>COMPUTE</td>
</tr>
<tr>
<td>32-bit numbers</td>
<td>664</td>
<td>888</td>
</tr>
<tr>
<td>64-bit numbers</td>
<td>838</td>
<td>1126</td>
</tr>
</tbody>
</table>

The extra floating point additions add 55 and 47 machine cycles respectively when the execute code is manipulating 64 and 32 bit numbers. Since execution time is proportional to the number of machine cycles and the processor speed, the worst case consideration is a data processor in COMPUTE mode.

5.5.4 Overall performance

The two criteria used to judge the suitability of the transputer as the basis for a processing element design were that it should be able to accommodate all the program and data structures within the on-chip memory and that it should achieve a worst-case step length of 100 microseconds. As far as memory is concerned, the T800 has 4kbytes of on-chip memory starting at address #80000000. The lower part of this space is reserved, so that user memory begins at the labelled address MemStart (#80000070) and extends up to #80000FFF. This is 3984 bytes and the total data processor code and data structures must be contained within this allocation. The addition of all the separate components of memory occupancy gives the table shown over.

Even with 64-bit numbers, less than half of the available memory space is taken up, although this is a minimum estimate. No allowance has been made for program code which sends
back simulation results to the host or for any additional storage which may be needed to service expansion simulators. However, even when all other possible uses of memory are considered, it seems unlikely that the available space will be a problem.

<table>
<thead>
<tr>
<th></th>
<th>BILINEAR</th>
<th>DELTA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>64-bit</td>
<td>32-bit</td>
</tr>
<tr>
<td>Transmit</td>
<td>576</td>
<td>288</td>
</tr>
<tr>
<td>Route</td>
<td>256</td>
<td>128</td>
</tr>
<tr>
<td>Static</td>
<td>336</td>
<td>186</td>
</tr>
<tr>
<td>Execute</td>
<td>697</td>
<td>537</td>
</tr>
<tr>
<td>Total memory size</td>
<td>1865</td>
<td>1139</td>
</tr>
</tbody>
</table>

The total cycle time (EP + TP) is dependent on the processor clock speed, the program length and the link speed. By combining the estimates from all the separate program fragments, the following overall estimate of minimum step length can be made. In the following table all estimates are in microseconds.

<table>
<thead>
<tr>
<th>Link Speed</th>
<th>Processor Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>20MHz</td>
</tr>
<tr>
<td></td>
<td>32-bit</td>
</tr>
<tr>
<td>10Mbps</td>
<td></td>
</tr>
<tr>
<td>BILINEAR</td>
<td></td>
</tr>
<tr>
<td>20Mbps</td>
<td></td>
</tr>
<tr>
<td>10Mbps</td>
<td></td>
</tr>
<tr>
<td>DELTA</td>
<td></td>
</tr>
<tr>
<td>20Mbps</td>
<td></td>
</tr>
</tbody>
</table>

If 32-bit numbers are acceptable then any combination of transform, processor or link speeds will deliver the required performance. If 64-bit numbers are used then the highest speed processor will be needed. It should be noted however that there is some uncertainty about execu-
tion times. Although worst case times were taken when these were given, the T800 data sheet [4] warns that, because of concurrent operation within the chip, it is not possible to be precise. Furthermore, the extra cycles which must be added when reference is made to external (off-chip) devices are obtained from chip simulations and should only be treated as 'reasonable estimates'. Exact times can only be measured on an operational system. However, the margins in hand are quite generous and it is reasonable to conclude that the target step length of 100 microseconds is achievable.

5.6 Simulator Design.

The design of a transputer-based PE involves interfacing a T800 with an array of CO12 link adaptors and providing some logic to indicate the end of a processing step. The simulator comprises sixteen of these boards and additional logic to set up the global switch and configure each board. This section considers each of these requirements separately.

5.6.1 The T800 external memory interface.

The 32-bit wide external memory interface of the T800 is a multiplexed address and data bus together with a set of control signals. Word addresses are aligned on four-byte boundaries and carried on the terminal pins MemAD31-2, with MemAD31 being the most significant address and data bit. Two other lines, MemnotWrD0 and MemnotRfD1, carry the two least significant bits of data when 32-bit words are passed over the bus. Read and write operations always refer to word addresses.

An external memory reference cycle time is divided into six T-states. Addresses are generated, and must be latched, during T1 and T2, and the remaining states are used to set up and pass data over the bus. The minimum duration of a state is half of the processor clock period. For interfacing external devices there are five strobe outputs, notMemSO-S4, four of which can be configured to suit the timing requirements of the connected device. NotMemS0 is a fixed format strobe and always starts in T2 and ends with T5. The others can be controlled by using the MemConfig pin. There are 13 different strobe configurations which are predefined.
and selectable by connecting particular address/data lines to the MemConfig input. Following
Reset, the transputer takes each of these lines low in turn and at each step reads from Mem-
Config. If a low is read during this (internal) configuration phase, the identification of the con-
ected address/data bus line indicates what duration and alignment the strobe signals should
have when external devices are referenced. In this application the most appropriate sequence
is obtained by connecting MemConfig to MemnotWrDO [4]. This also positions the read/write
control signal, notMemWr, at the most appropriate point of a write-to-peripheral cycle.

5.6.2 Connecting the link adaptors.

Because the link adaptor status registers are not checked before an access is made to a data
port, only 16 addresses are occupied by the input/output links. Output ports, or links, are allo-
cated addresses #00000000 to #00000007 and input ports are positioned at the following 8
addresses. Internal memory address references all have MemAD31 set high so that any ad

Figure 5.3 The link adaptor connections.
dress reference which has MemAD31 set low must be to the peripheral logic on a processor board. The bit pattern on the lines MemAD5-2, with MemAD6 and MemAD31 set low, indicates which link is active and, if MemAD6 is high, the ready latch is set. Ready indicates that the processor has completed its computational step. Its significance beyond that is explained in 5.6.6. Figure 5.3 shows how connections are made to a link adaptor and Figure 5.4 how the local control signals might be generated.

![Diagram of link adaptor and control signals]

Figure 5.4. Generation of the local control signals.

A timing requirement is that the link adaptor controls RS0, RS1 and RnotW should be established before notCS is taken low. Since the output links only use the write data register and the input links only use the read data register, this requirement is easily satisfied by strapping RS1 to the ground rail and RS0 to Vcc or Grd depending on the direction of transfer. With the chosen strobe pattern, notMemS3 and notMemWr are correctly aligned with notCs and RnotW and, in a write operation, with the data bus contents. The required logic is minimal and could conveniently be contained in a programmable logic array.
5.6.3 Configuring the global switch.

Transputers, including the one on the host transputer board, have four links. One of the links of the host transputer connects the peripheral transputer system to the PC bus through a link adaptor. The remaining three links are made available at the back of the host. They are used to configure the global switch, to gather simulation results and to provide a boot link for downloading executable code to the processing elements. Figure 5.5 shows how they are connected to make a fully functional simulator.

The construction of the global switch from COO4 link switches was discussed in Chapter 2. Each plane of the global switch has 16 inputs and 16 outputs which is half the capacity of a COO4. Thus the global switch can be made from four COO4s and configured through the four configuration links. One additional COO4, the RC switch, is used to route configuration (C) messages from the host to the global switch. This extra switch also needs to be configured so that one of the available host links is permanently committed to carrying RC configuration messages. In Figure 5.5 this link is labelled ConfigRC.

Another host link, the RC link, is used to carry the global switch configuration messages. These are routed to the appropriate plane of the global switch array by sending messages over ConfigRC. In total, the configuration connections take up four inputs and four outputs on the RC switch with the RCLink accounting for another I/O pair.

5.6.4 Results collection.

When the global switch has been configured the RC link is used to pass simulation results (R) back to the host. A minimum of one (normal) link from each PE is connected to the RC switch and, again, the ConfigRC messages indicate which PE output will be monitored. By changing the RC switch settings, it will be possible to track and display several PE parameters during the course of a simulation. Although the design of a user interface has not been studied in any detail, some of the issues involved are discussed in the next chapter.
Figure 5.5: The transputer-based simulator.
Of course, the output primitives which transmit results to the host at the end of every execute phase occupy memory and take time to complete. However, neither addition is expected to impact too greatly on the total figures for memory occupancy and execution time. There is some debate as to the precise meaning of some statements on the T800 data sheet but it appears that the 'out' instruction occupies one byte of memory and, for a single word transfer, executes in 23 machine cycles. Extra space and time will also have to be allowed for selecting which output is to be monitored.

Another concern will be to ensure that every output process, whether or not it is connected to the host, completes successfully. A sequential (results) output process which begins but which is not acknowledged because the link is not connected, will lock-up the processor with predictable consequences.

The links which return results from the simulator account for a 16 inputs and 16 outputs on the RC switch so that there is ample space to expand this facility should this be necessary.

5.6.5 Downloading PE code.

The process of setting up a simulation involves specifying a connection pattern - which subsequently becomes a series of global switch configuration messages - and specifying and initialising the number and type of processor which have been connected. Each processor type executes a different program so that specifying a processor by type amounts to identifying which program code block must be retrieved from host storage and passed to that processor. Initialisation involves assigning values to the processor parameters - a host program function.

All transputer chips have an input labelled BootFromRom which is connected to the positive supply rail, Vcc, or ground. Following a Reset, and after the memory strobe patterns have been decided, the transputer reads the BootFromRom input. If it is high (Vcc) the transputer starts to execute code stored in external ROM; if it is low, the transputer waits for the first piece of data to arrive on any of its links. If this is program code it is stored in memory at MemStart and execution begins when all the code has been loaded and a start signal has been detected.
The Transputer Development System (TDS) [5] includes a number of utilities which are used to load transputer networks and check that the distributed code will execute properly. Configuration statements are added to single-processor Occam programs to PLACE processes with processors, identify the processor type (T2, T4 or T8) and map (PLACE) software channels onto physical transputer links. The procedures for configuring and loading transputer networks are explained in the TDS Reference Manual and Inmos Technical Notes. In this application the bootstrap links connect the 16 PEs in a daisy chain and code is shifted through the chain into the intended PE. The system utilities manage the whole operation.

To make absolutely sure that the proposed method of retrieving code blocks and sending them to nominated processors in an array was sensible, some work was initiated [7] to test the strategy. A small four-node network was used for the test although the connections between nodes had to be made manually. This work verified that this aspect of simulator configuration would not be a problem.

5.6.6 Processor Synchronisation.

The first action in every computational step, including the first, is for a processor to signal that it is ready to proceed. This it does by writing to port address #00000020 (32) to set the READY flip-flop. On every processor board the READY flip-flop outputs are AND-ed to the EventReq pin on the host transputer so that when all PEs are ready, EventReq is taken high. This is acknowledged by the host when it executes the input process,

\[ \text{event ? ANY} \]

Acknowledgement means that the EventAck output from the host goes high which resets all the ready flip-flops. The PEs now begin TP and the host begins to time out to the start of the next step period. Assuming suitable variable declarations, the Occam statements which implement this method of synchronising activities in the simulator are,
VAL looptime IS 100:
WHILE TRUE
SEQ
  event ? any
  clock ? timenow
  clock ? AFTER timenow PLUS looptime

When the statements which read results are added, the host execution code is,

  event ? ANY -- wait for all PEs to get code
  clock ? timenow -- start the clock
  WHILE TRUE
  SEQ
  .. .read results if any (at the start of TP)
    clock ? AFTER timenow PLUS looptime
    event ? ANY -- step time elapsed
    clock ? timenow -- reset the clock

Results should be read by the end of TP since the value may change during EP. At the start of the results collection sequence the host sends a byte (of ANY) on the RClink. Only the connected PE will receive that byte although the others may expect it. To prevent lock-up, the results collection statements are enclosed within an ALT(ernate) construct which either reads from the results link or falls (ie. timesout). The fall statement ensures that no processor waits indefinitely and also ensures swift delivery of the desired data to the host.

5.6.7 Simulator Expansion.

Only two of the PE transputer links, the boot link and the results collection link, have an assigned role. The other two links could be connected with other simulators through a master switch and so expand the scale of the simulations which can be set up. The expansion potential exists but the detailed problems remain to be identified and solved.
5.6.8 Analogue Input - Output.

A real-time simulator responds to stimuli (events) at the same rate and in the same way as the system it is simulating. Communicating inputs and responses between the simulator and its environment means that an analogue input-output (I/O) interface must be provided. The input interface converts analogue inputs to floating point outputs during EP and transmits them to other PEs during TP. On the output side, floating point numbers arrive during TP and are converted to integer and then to an equivalent analogue voltage during EP. The minimum provision on the I/O board would be two or four analogue input channels and an equal number of output channels connected to a T800 through some interface circuitry.

There are several different ways in which such a board could be configured. For instance, the data acquisition/distribution system could be connected to the parallel side of link adaptors and exchange values with the T800 over its normal links. Connections to the simulator could then be made by providing link adaptor links. Alternatively, the data acquisition system could be memory-mapped into the transputer's address space with the normal transputer links connecting the I/O board with the rest of the simulator. Yet another possibility is to memory map the data acquisition system and use link adaptor links. In this case the links are either connected to the existing global switch, suitably expanded, or, less likely, through another COO4 to the spare (normal) link on each PE. There is likely to be a problem in combining inputs arriving over the normal links with those arriving from the global switch.

Completing digital-analogue and analogue-digital conversions within the allowed time should be easy enough and the T800 tasks of controlling the analogue interface and converting between integer and floating point numbers are minimal. The problems have already be confronted elsewhere [8], although the application and the proposed implementation technique are quite different.

Using four link adaptors (for the 8 channels) and expanding four planes of the global switch by one additional I/O connection (or adding to all planes with each I/O channel connected to two planes) adds nothing to the storage space taken up in a PE and nothing to the execution time. The T800 on the interface board is programmed to remain in step with all the other PE transputers so that analogue input/outputs which terminate on the interface board are indistinguishable from any other serial bit stream. The cost is a larger global switch.
The I/O interface, although important, is one of the easier system functions to provide. If extra hardware development is to be avoided then there is a range of analogue I/O boards for transputer systems already available [6]. These connect with the normal transputer links and provided that using them does not increase the execution time too greatly, they may be a cheaper alternative than designing special-purpose I/O boards.

5.7 Conclusion

The hardware study has indicated that a T800-30, working with double-length numbers, could be used as the basis of a processing element design using either of the preferred transforms. However, as stated earlier, performance estimates are not precise and additional program statements, such as those used for synchronisation and results collection, have not been considered. It is unlikely that the program size limit will be exceeded but speed estimates are less certain. In the event of a prototype unit failing to meet the target there are a number of options which could be investigated. Limiting the maximum order of transfer function simulations to two, rather than three, would give a worthwhile reduction in execution time; redefining some, or all, of the real parameters as single rather double length number would do the same. Alternatively of course, the 100 microseconds target could be relaxed. The likelihood is that none of these measures will be necessary but certainty will only follow construction and successful testing.

5.8 References


PROC execute64 ()

-- an occam representation of the data processor execute code. All the code contributes
-- to memory occupancy, the longest code sequence determines execution time.

-- declarations for 64-bit reals.

[8]REAL64 invalues:
[8]INT16 lroute:
[6]REAL64 ks:
REAL64 sigma:
REAL64 initsigma:
INT16 order:
[4]REAL64 Xs:
[3]REAL64 initXs:
[4]REAL64 as:
[3]REAL64 bs:
REAL64 limit:
[8]INT16 outroute:
[8]REAL64 outvalues:

-- local variables; additional memory here is 11 bytes

BOOL compute, hold, reset:
REAL64 funcx:

VAL one IS 1.0(REAL64):
VAL zero IS 0.0(REAL64):

-- main program starts here;

SEQ
hold := (invalues[6]=invalues[7])
IF
reset
SEQ
sigma := initsigma
Xs[0] := initXs[0]
Xs[1] := initXs[1]
Xs[3] := zero

Listing 5.3 The execute process. Sheet 1 of 3.
compute
SEQ
sigma := zero
sigma := sigma + (ks[0] * invalues[0])
sigma := sigma + (ks[1] * invalues[1])
sigma := sigma + (ks[2] * invalues[2])
sigma := sigma + (ks[3] * invalues[3])
sigma := sigma + (ks[4] * invalues[4])
sigma := sigma + (ks[5] * invalues[5])
TRUE
SKIP
IF
compute OR reset
SEQ
outvalues[3] := sigma
IF
sigma = zero
SEQ
outvalues[4] := sigma
TRUE
SEQ
outvalues[4] := -sigma
outvalues[6] := one
-- compute a new output

Xs[1] := Xs[0]

sigma := sigma - (bs[0] * Xs[1])
sigma := sigma - (bs[1] * Xs[2])
sigma := sigma - (bs[2] * Xs[3])
Xs[0] := sigma

funcx := zero
funcx := funcx + (as[0] * Xs[0])
funcx := funcx + (as[1] * Xs[1])
funcx := funcx + (as[2] * Xs[2])
funcx := funcx + (as[3] * Xs[3])
outvalues[1] := funcx
IF
  funor = zero
  SEQ
  TRUE
  SEQ
  outvalues[5] := on8
IF
  funor = limit
  SEQ
  outvalues[6] := one
  outvalues[7] := zero
  TRUE
  SEQ
  outvalues[7] := one

hold
SKIP

... raise the ready flag.
6. The simulator-user interface.

6.1 Introduction.

One of the most actively investigated aspects of simulation is the user's interface with the simulator. This is indicated by the volume of published material on interactive graphical and animated display systems for simulators and the use of artificial intelligence (expert systems) for model construction and control. A key aim of this work is to make simulations easier to build and use by protecting inexpert users from unnecessary or inappropriate details of the model. At the same time, more experienced users should be able to probe more deeply if they desire [1].

This chapter describes how the software simulator interfaces with its user and some work which was initiated to provide graphical input. It also gives some indication of the trends in simulator design and how this may impact on the proposed hardware system.

6.2 The software system interface.

The software simulator which was used to evaluate and clarify the issues surrounding the construction of its hardware counterpart has a fairly inflexible interface. Before it can be used all aspects of the simulation such as the nature and number of PEs, their interconnection, PE parameters etc., have to be known. This information is made known to the simulator when the user responds to specific questions and, except when connection data is being entered, there is no opportunity to change individual replies or backtrack to earlier questions.

The user first defines the number of PEs in the simulation (unitcnt), up a maximum of 16 (max-units). The global connections are then established by responding to the sequence of questions:-
Unit No?
Pin No?
Destination Unit No?

Entries are checked to ensure that the 'Unit No' is not greater than 'unitcnt' and that the same destination (input) is not connected to different outputs. The maximum number of connections which can be made to any output is also 'unitcnt' and unallowed connections can be cancelled or changed. When all the connections have been declared the complete connection matrix is displayed and the user is invited to accept it or change individual entries. Once it has been accepted however it is unalterable.

Again responding to specific questions, the user next declares the simulation parameters for each PE, including the router settings. The final selection is of which PE outputs carry useful results and whether elapsed time or an event is to trigger results collection. In the former case the interval between successive printouts is specified as a list of numbers with the letter 'c' (for continue) between each pair of numbers. The list terminates in 'e'. If an event triggers results collection then the PE and pin number where the event will occur has to be declared.

When the simulation runs, results are printed on the screen as a list of numbers with no provision for any alternative presentation. For the inexperienced user, the interface is somewhat bleak and it is usually easier to prepare datafiles in an editor and use the operating system redirection facility to run the simulator from the datafile. The format of the command line is,

```
simulator < datafile > resultsfile (or screen)
```

Gross errors in the datafile are signalled by a runtime error message which halts the simulation and provides some help in finding the mistake. For more subtle errors, such as misplaced connections or incorrectly initialised variables, there are no obvious debugging aids. A hardware system which was made generally available would need a much more user-friendly interface.
6.3 Investigation of a graphical input facility.

As previously explained, initialisation of a simulation involves defining every aspect of the model. Time is not a critical factor here since the initialisation software runs in the host and responds to user input, which is fragmentary. The more important issue is to make this aspect of simulation as user-friendly as possible by providing as much support as is needed.

The ideal interface would allow the user to position processing elements on a graphics screen, draw line connections, fill in routing table entries and associate attributes with each processor. The system would issue helpful prompts if inappropriate or incomplete data were entered, and pan and zoom facilities would also be provided so that all components of the simulation could be viewed in detail and in context. At the end of this process, the topology and content of the diagram would be translated into physical connections between processors with the correct operating code and data downloaded to each. Configuration data would also connect selected outputs to a results gathering switch and results would be displayed as the simulation runs.

A first step towards this ideal [2] involved the use of a commercial drawing package, AutoCad, to prepare network connection diagrams. The devices which were connected were transputers, with the normal four links, and the switch which connected them was a C004. The maximum number of network nodes was four, being limited by the available B001 transputer boards, and the host computer was an Olivetti M24.

With AutoCad it is possible to assign attributes to nodes and to links and to edit diagrams on screen. The procedure used in the project was to draw the diagrams and then write all the information it contained to an ASCII text file. An Occam program then accessed this file and configured the C004 and the simulation parameters. The work was intended to demonstrate the feasibility of producing a usable interface between a commercial drawing package and a network of transputers and, to that extent, it was successful. However, the interconnected hardware was considerably less extensive than this proposal and the connection requirements considerably less arduous.
Even for this rudimentary facility however, the development effort was considerable and probably unnecessary. Similar, and more elegant, facilities are becoming available in commercial software packages such as the Windows File Server (WFS) from Nexis Technology [3]. WFS runs under Microsoft windows and is said to act as a versatile front end to transputer programs. It is aimed at image processing applications but has many features which would be needed in an ideal simulator interface. With an extensive library of graphical functions it provides the means by which configuration and results 'information can be readily exchanged between applications (running in the host) and transputer systems'. Although there has been no opportunity to test the claims of this package, it seems to satisfy the aims of [2] with considerable less effort and considerably more user support.

Of course, tailoring software would be needed to attach WFS to the simulator, but future research in this direction is likely to amount, mostly, to evaluation of this and similar products. Producing new, dedicated graphics software for the simulator is not thought to be a sensible strategy.

6.4 Expert systems and simulation.

The techniques for building expert systems [4] have aroused considerable interest in the simulation community. Simulation and expert system design are both iterative processes in which successive refinements of a prototype are made after analysis of test (debugging) results. The combination of the two is an attempt to make the expert knowledge of experienced simulation designers accessible to inexperts who have the task of building simulations. It is not just a question of building an expert system around traditional simulation software and hardware; the proposals seem to suggest a new approach to simulation.

Expert systems are one aspect of Artificial Intelligence (AI) which is concerned with making intelligent computer systems that 'exhibit the characteristics we associate with intelligence in human behaviour'. This is a somewhat lofty attribute to attach to expert systems since they are computer programs which attempt to duplicate expertise without regard to how a human expert would arrive at a decision. The aim, it would seem, is not to understand thought processes, only to consistently duplicate their results. Expert knowledge is condensed into a series of rules which are used to draw inferences and suggest, or implement, actions which
will deal with a problem.

The skill in designing an expert system is demonstrated in the representation and use of the knowledge that the expert possesses. Knowledge is usually organised into a database which contains facts about objects and events, a knowledge base which contains information (rules) about courses of action, and a control or inference structure (or engine) which defines how the rules should be applied in a particular situation.

It has been pointed out [5] that the main differences between an expert simulation system and a simulation based on a traditional technique or language is in the way the model is constructed. The steps involved in building a 'traditional' simulation were discussed in Chapter 1. Essentially a model is proposed and then tested; the results are then analysed and (usually) the model is changed so that the test results more closely match what was expected. The expertise is in choosing an appropriate model in the first place and then interpreting its behaviour. In contrast, an expert simulation system user 'declares the knowledge about the system, defines the goals and lets the computer find the solution' [5].

When contemplating this approach to simulation and the simulator which is being proposed, the inescapable conclusion is that formidable problems remain to be solved before the two can be brought together. One kind of expert system interface would require the user to 'declare the knowledge' by entering sets of differential and algebraic equations, and 'define goals' by specifying acceptable performance indicators. The expert simulation system might then be expected to translate this information into a network of processing elements and experiment with different stimuli and operating parameters before finding and reporting a suitable configuration.

An alternative, which might be favoured by control engineers, would allow users to describe a system in block diagram (transfer function) form and have the expert simulation system translate the diagram into a network of connected PEs. Transfer functions would be scaled where necessary, parameters would be initialised and all the interconnection commands and settings would be issued. Test sequences could then be run to see how the simulated system behaved in comparison with expected behaviour, or the expected behaviour could be declared in some way and the simulation might then be expected to adjust its own parameters in some way to obtain an optimal fit. In short, the expert system would be expected to do everything
which currently an experienced user has to do. These are not trivial tasks but automating them would be the aim.

Currently the main application area for expert simulation systems seems to be in connection with manufacturing systems. The concerns are factory layout, operating procedures, scheduling etc. One of the few examples of a continuous control system simulation using an expert system [6] is a model of aerial combat involving an ‘evader’ and a ‘pursuer’. The authors’ aim is to develop ‘completely automated tools for the model design phase in the simulation process’. Theirs is a software project but it may be possible to adapt their design technique and use it with hardware.

On a broader front however, there is no doubt that simulation and expert system design are now seen as connected activities. Simulation techniques are going through a fundamental change, although quite slowly and more obviously in connection with discrete systems. If expert systems could be used to build simulations from parallel processing elements, and if the model description methods made that a sensible approach, then user unfriendliness would not be a problem. The only other concern would be how to display simulation results in the best, most easily assimilated, way.

6.5 Graphical interfaces and animation.

Presenting simulation results as lists of numbers is not ideal. For some time now, static graphics such as plots, histograms and bar charts have been used to make simulation results more understandable. With real-time computer animation now available, animated graphical outputs are being used not only for results presentation but in every stage of model development. In [7] for instance, the authors describe an approach to animation design and demonstrate its use in a case study concerning modifications to a manufacturing plant. Their contention is that animation makes ‘lively and accessible what would otherwise be a dry and somewhat obscure presentation of figures and tables’. These sentiments are certainly shared by the designers of control system simulations, but the movement of mechanical handling robots and machines on a factory floor is more suitable for animation than the alteration of traditional control system variables. The benefits [8] of animation are mostly self evident but the applicability less obvious.
The T800 is particularly well suited for graphics support and animation. It has microcoded block move instructions which can be used to build graphics display systems [9][10]. The performance of these systems is related to the number of transputers used, but there is already available a useful library of software and hardware design information. Using it in connection with this project however, is firmly rooted in the realms of future work. The intention would be to overlay the display system onto the network topology diagram and track process variables which are identified by position rather than by name. Alternatively, and probably easier, the results would be displayed in a separate window with obvious indicators as to which variable was being displayed. WFS, mentioned earlier, already offers such a facility.

Graphics and animated displays for simulation are an obvious and very desirable addition to the basic hardware and software. Interaction [11][12], in which parameters are altered during the simulation run, is a step beyond that and, for discrete system simulation, is already available. The aim of all these developments is to remove any mystique which may still surround computer simulation systems and thereby make their use more widespread.

6.6 Conclusion

Apart from the work with AutoCad and the scant user/simulator dialogue which is used with the software simulator, relatively little development effort has gone into improving the user's view of the simulator. Most attention has been paid to specifying the processing elements, the interconnection system and how these may be built in hardware. The next stage of this development will be correct this omission, but one aim of the foregoing discussion was to show that other researchers have already tackled (and are tackling) the problem. In many cases the product of research programmes has been turned into a commercial, marketable, product. There seems little that is new to be done, except to pull together the relevant strands of what is already available.

All the literature states the case for a well-designed user interface which covers all phases of experiment design and use. The challenge in this case will be to economically adapt techniques which have been primarily developed for discrete system simulations. Cost will also be a factor. A display system which is more expensive than the simulator it serves is not really sensible, but, equally, an unacceptable interface will make a simulator unacceptable, however
Inexpensive it may be. There is a balance to be struck and further investigation will be intended to show where that is.

6.7 References


[10] High performance graphics with the IMS T800. Technical Note 37. Inmos Ltd.


7. Future system developments and conclusions.

7.1 Introduction.

There have been significant technological changes during the period of this project. Ten years ago, the demand for the highest possible speed was always met by bit slice components. Parallel processing was in its infancy and there were few if any devices which were obviously engineered for the task and little sign of any emerging. Consequently the first (and only) Mark-I device was built from bit-slice components with little thought for how emerging parallel processing techniques might impact the system design.

Conceivably the system described in earlier Chapters could be made in the same way. High speed, a prime requirement in real time applications, is still the hallmark of bit slice systems and customised instruction sets, written as microcode sequences, and much tighter control over most aspects of the hardware design (for example memory size and arithmetic word length) are still attractive attributes, but there is a price to be paid. Bit slice hardware has a commendably regular structure, but writing, installing and testing microcode is an experience to be avoided if possible. There are a few development aids to be had, and for users wishing to use the 'specials' facility described in Chapter 4, the preparation of suitable code is not trivial.

The other disadvantages are that, for a system which relies heavily on efficient communications, bit slice product lines provide very little support for high speed serial communications, and the arithmetic unit would probably need to be boosted by the addition of specialised floating point chips. The modern, and wider, counterpart of the AMD 2901/2903 arithmetic slices appears to be the Am29501 [1] which is 8 rather 4 bits wide, but still has no floating point instructions. AMD application notes show fixed-length arithmetic units which are very fast, but comprise separate integer/logic, floating-point and multiplier chips. More chips means bigger systems, more opportunity for system failure and more expense.
The introduction of the transputer, in particular the T800, was the technological innovation which this project needed. It is an ideal component where processes and processors need to communicate at high speed and fast floating operations are called for. Engineering decisions are sometimes difficult but in this case there would need to be compelling reasons for building the system using anything other than transputer-family parts. The approximate cost of a processing element comprising a T800 and 16 C012s is £303. To this must be added the cost of a half-height Eurocard and the small amount of 'glue' logic. When the cost of the link switches is also considered, the total estimated component cost of a 16-processor simulator is less than £7000.

The following review of the project is therefore based on the assumption that transputers and transputer-family products will be used to build the system. The intention is to re-consider most aspects of the proposed simulator and try to establish whether, in hindsight, things could have been done differently or better, and where originality could be claimed.

7.2 The software simulator

The software simulator was written because it was thought easier to experiment with software options rather than hardware ones. Since little money was available for hardware development, another important consideration was that writing software does not incur capital cost, only time! Having written it and used it to specify hardware, the main need for the software simulator is finished but there are several reasons why it might usefully be retained and developed further.

The simulator itself is about 2600 lines of Pascal code. This does not include all the known processor definitions or the additional input-checking statements which would make it more robust. The other point to make is that it is not particularly easy to use. In most cases significant editing of the source is needed before it can be applied to a new problem. This clearly detracts from its appeal as a standalone product.

It was written in Pascal because it was also intended to be a hardware description document. Pascal is said to be a self-documenting language so that reading the code should indicate
how the hardware will work, albeit in sequential rather than parallel mode. This is a very useful facility when testing new hardware descriptions. It is a relatively easy task to add specifications to `decs.pas` and `bits.pas` and run trial simulations. The subsequent conversion of Pascal to Occam is also not difficult as shown by the Occam fragments listed in Chapter 5; Occam and Pascal are very similar. There may be more difficulty in linking the new description with the simulator file system, but a step-by-step user guide could easily be provided.

The software simulator could also be used to check out simulation ideas before the hardware system is employed. It is assumed that the hardware resource will be less readily available than software running on the ubiquitous PC. The problem is likely to be execution speed. Pascal was originally conceived as a teaching language [2]. It was meant to include all that was needed to make possible good Software Engineering practice but it was not built for speed. Admittedly the floating point package can now be removed because further experiments with variable number formats are not needed, but, depending on the problem scale, the system is still likely to execute quite slowly. It may be sensible to contemplate re-writing the simulator in C, with its lower-level syntax and bit manipulation instructions, or some other language which would make it run faster. As a product it is still not likely to challenge existing software simulation systems, but when supplied with the hardware it would undoubtedly be useful because of the close affinity which exists between the software and hardware on which the simulation will eventually run.

One other software concern which needs attention is the pre-processing software which is used to calculate all the parameters of the hardware elements. At the moment, this pre-processor is run as a separate exercise before the simulator is started. The results it produces are entered into the simulator during the initialization phase but it is obviously more sensible to include the pre-processing software as a part of the simulator itself. Typical calculations are of the a and b (or c and d) coefficients, state variable Initial values, printout intervals and the like. Software for calculating model parameters for a number of different transforms (the Taylor series [3] expansion method as well as the bilinear, predictor and delta forms) is available and even if the software simulator is not developed further at least some of this will need to be integrated with the host software in the hardware system. It total, the separate Pascal modules probably amount to something over 4000 lines of code, which is not large but not insignificant either.
There is merit in being able to handle a moderately-sized software project, but any originality is in the concept rather than the software implementation. One of the main aims of Software Engineering is to positively discourage originality (or inventiveness) during the code construction stage. In other words, there are few instances of clever uses of regular data structures. The package was thoroughly tested and can be demonstrated to work. It is moderately well structured and easy to read, particularly in respect of PE descriptions. Although deficient in some, relatively minor, respects, it is extremely important because without it the project would not have progressed.

7.3 The switching system and system expansion.

Simulator expansion simply means increasing the number of PEs beyond the current limit of 16. There was no profound engineering reason for choosing 16; indeed, if transputer technology and the 32-by-32 C004 link switch are used to build the system, it would seem sensible to expand the basic simulator to 32 processors. The global switch is then a linear array of eight C004s instead of 4, but all else remains the same.

An expansion of this scale is not likely to present any major difficulty although the synchronisation mechanism using the READY flag may need to be looked at again. In theory the approach is sound; in practice, engineering experience indicates that problems are proportional to size and the potential difficulties of buffering (fan-out), extended line lengths and noise will need closer attention. There are however several examples of parallel systems using many more than 32 processors [4][5] although in most of these cases processes running in different processors are synchronised by the link protocol and not imposed by external hardware. Parallelism in this system is by virtue of having multiple processors and not by having PAR (parallel) constructs in the processor code.

Expansion beyond 32 (or 16) processors can be approached in at least three ways,

- Provide an expansion switch and use some of the existing switching capability of the global switch for expansion purpose;
- Use the remaining (‘proper’) transputer link which is not used for results collection or downloading;
• Provide an expansion bus.

None of these ideas has been tested but it is interesting to consider whether, and how, they might be achieved.

7.3.1 Switch-based expansion schemes.

With this technique, a lesser than maximum number of PEs (16 or 32) are connected through the global switch to allow for expansion. For instance, an 8-plane, 16-by-16 global switch connecting only 15 PEs would provide one connection (in and out) per-plane to be combined in an expansion switch. If this expansion switch were also 16-by-16, then two, 15-PE simulators could be connected together. It is fairly easy to see how the proposition could be scaled-up.

A more restrictive arrangement would nominate one processor per simulator as the interface (between simulators) processor and make a connection from one (designated) bit of that processor to the expansion switch. All inter-simulator connections would need to be made through the designated I/O pin. Ideally of course, there would be at least two (one control, one data) signals passing between simulators which introduces even more topological restrictions. As previously stated, the most sensible arrangement for an expansion switch has not yet been properly investigated so that integration of multiple simulators is very definitely a topic for further investigation.

Using unassigned transputer links for extension purposes would also involve additional (or different) switching, but this is the most problematical of the extension methods. Since transputer links are fundamentally different to link adaptor links, it is most sensible to use one type for control and monitoring and the other for PE connection. Mixing the two is fraught with difficulty and is not, at this stage, recommended.

7.3.2 Bus-based expansion schemes.

In the larger and faster packet switches which are being proposed for some new communications equipment, high speed buses are being researched. These will be driven by very fast (100 MHz) serial chip sets intended for use in building network nodes. It is possible that this
research could have application here if the high-speed bus were used to connect between shelves in a multi-shelf rack. Each shelf would house one simulator and shelves would be linked from the global switches to the bus. However, even with such high speeds there may still be a problem because inter-shelf communications cannot be spread over the whole computational cycle time. An idea worthy of investigation in that respect is to modify the rigid TP/EP timing cycle so that PEs are allowed to transit when ready. Since the execution time of all processors is different this would mean that bus transfers would be less deterministic than before and, so long as all the arrivals occurred before the end of the scheduled TP, the system would operate properly. Conventional bus arbitration logic would settle any remaining conflicts.

7.3.3 The switching system in general.

When viewed in the light of the number and nature of the processing elements it connects, the switching system is considered to be a novel and efficient way of getting the job done. A non-restrictive, minimum delay arrangement of manageable size (and therefore cost) was the aim and this has been achieved using the planar global switch with input/output routers. In connection with the output router it was said in Chapter 2 (2.4.1) that any number (within reason) of processor outputs, \( m \), could be defined but the device outputs would always be limited to \( n = 8 \) (or possibly a smaller number; see 7.4.1). With software implementation, this simply means that more storage is needed to hold processor output values. Since storage space is not likely to be at a premium however, this should not present any particular difficulty.

If the output router is a one-to-many crossbar switch, an irregular \( m \)-by-\( n \) matrix would be needed, in which the outputs \( n + 1 \) to \( m \) are not used. If \( m \) is less than \( n \), a regular \( n \)-by-\( n \) matrix, in which inputs \( m + 1 \) to \( n \) are not used, is the answer. Having unused router capacity is a disadvantage, but not greater than providing a larger global switch than is absolutely necessary.

At PE input the most straightforward way of providing a hardware router is using another crossbar switch rather than the Batcher-banyan network discussed in Chapter 2. In any event, it has been shown that routeing can be achieved quite adequately in software and, however provided, is an original solution to a particular switching problem.
7.4 Processor Architecture.

7.4.1 PE Terminal Configuration.

The terminal configuration of processors is another issue which bears re-consideration. For the majority of applications 8 inputs and 8 outputs could be viewed as a quite costly over-provision. On input, the two control signals are always needed unless some basic re-definition of PE operation is proposed. However, a device with two, or perhaps three, non-control inputs, and a corresponding number of outputs, will, in most cases, be sufficient. The significance of this reduction is that the global switch is smaller (four or five planes as opposed to eight) and the transmit/route code is faster. In the few cases where a greater number of inputs and outputs is needed, different inputs could be assigned to identical processors operating in parallel (input expansion) or the same inputs could be connected to identical processors, again connected in parallel, and the two output routers set differently (output expansion). In the case of input expansion the separate outputs would need to be combined and in both cases a greater number of PEs would be used. In the majority of cases however, simulations would occupy the same number of smaller, and therefore cheaper, processors. The significance of faster code is obvious.

7.4.2 Multiple-role PEs.

Inspection of the code sequences for different PEs clearly shows that most of them are much less heavily loaded than the data processor. This is particularly true of some of the logical processors which spend the majority of EP waiting for TP to begin. It may be worth considering composite units which provide more than one function. For instance, a timer may also serve as a counter in the same simulation. There is sufficient memory available to hold the code for both functions and both could easily be completed within the 100 microsecond step length.

Just two of the implications of operating in this way are that the different functions of composite units must always operate in the same mode and that units with the full 8 inputs and outputs are needed. It might be argued however that a device serving as two devices and having 16 terminal connections is more efficient than two devices acting individually and having a combination of 16 terminal signals.
The technique could be extended to several other devices. For instance multiple multipliers are sensible and waveform (sine/cosine, for example) generators which generate more than one type of output could be possible. The only PE to which the principle could not be applied is the data processor; each data processor occupies one processing element no matter what order of transfer function it is simulating.

7.5 Process Implementation.

In every step, the greatest computational load falls on a data processor simulating an $n^{th}$ order transfer function. In reviewing PE implementation therefore it is sensible to consider the data processor separately.

7.5.1 The data processor.

The questions which were asked during the design and testing of a data processor were,

- which is the best s-to-z transform?
- which is the best mantissa length?
- what is the maximum order of transfer function which can be simulated?

The answers were conditioned by the effect that changes (in mantissa length or transform for example) had on the errors (or deviations) and, as reported in Chapter 3, exhaustive testing was carried out. The conclusions are that, from the evidence of these tests, the delta form of the bilinear transform is the most appropriate. It takes a little longer to compute than others and it occupies a little more memory, but neither increase is excessive. The crucial point is that its error performance is never worse than the bilinear transform and is often much better. Whether, because of the improvement, it might be possible to simulate $4^{th}$ order transfer functions using it has not been investigated. Further work may supply an answer although execution time and, to a lesser extent, code size are equally likely to be concerns.

Investigations of mantissa length in relation to arithmetic errors provided a very interesting demonstration of the phenomenon. Although more tests, with different example simulations,
could have been made, it is claimed that the conclusions reached from those which were
done are generally true and, although the testing technique is straightforward enough, the con-
struction of the tests and the interpretation of of results are evidence of a professional and
scientific approach to this type of investigation. In the event of course, the adoption of the
T800, with its 64-bit arithmetic capability, provided a complete answer. Realistically however,
it was always felt that customised word lengths were never really sensible and that once the
limits had been established, it would be a question of choosing a device which could supply
(or better) the required performance.

No testing has been done with 64-bit numbers (11-bit exponent and a 53-bit signed mantissa)
although increasing the mantissa length can do nothing but good. The mantissa length of the
host machine will always limit the resolution of numerical data, but the extra bits should signifi-
cantly delay the onset of arithmetic errors. Assuming that the speed and size constraints can
be met, it would be interesting to see a 4th order transfer function simulation using the delta
form of the transform and 64-bit numbers. This remains however another assignment for fu-
ture work.

The step length requirement, its effect on coefficient sensitivity and arithmetic errors are all
linked of course. In addition, the step length also sets the upper frequency of the real-time re-
sponse. Crosbie [6] defines real-time simulation as ‘the execution of a model contained totally
within the digital computer in such a way that the execution of the model is synchronised to
system time’. He talks of values being updated at ‘communication points’ which should be
close together for smoothness and far apart if the computing time is to be reduced. Like
others however, he offers no help in estimating the system bandwidth. Application of the
sampling theorem gives the very best estimate but it is unrealistic. A more reasonable upper
limit might be obtained by specifying that a sample taken in one time slot should be pro-
cessed and delivered to the output in the next. If processing involves passage through several
PEs this means that the time between communication (or sample) points should be several
times greater than the step length. A figure of between 5 and 10 is said to be reasonable
which would set the bandwidth of this simulator at approximately 1kHz. Crosbie’s reading of
the situation is that the major users of real-time simulations are suppliers of training simula-
tors where the emphasis is on speed (system response) rather than accuracy. In other words
the simplest, most rapidly computed integration algorithms are chosen. Since users will nor-
mally graduate from trainers to the real thing, this acceptance of inaccuracy is quite worrying.
The structure of the data processor is much the same as that of a recursive digital filter. The additions which have been made are mostly obvious but it would be argued that the way in which the well-known model is used and, certainly, the method of adding state variable initial values are novel. Simulating transfer functions is standard analogue computing practice but other instances of it being done in one hardware block have not been discovered.

7.5.2 The other processing elements.

The number of different processing elements and what each of them should do, has not been decided. It is tempting to group functions together (much as described in 7.4.2) so that the maximum amount of work is completed in every PE in every cycle. The disadvantage of doing that is that some of the generality of the processing element is lost and the consequence would be that a larger number of different processors would have to be defined. In any case, getting the most out of hardware is only necessary if the maximum available simulator capacity is being used while individual PEs are being under utilised. Another aim of future work would be to look more closely at typical applications and estimate simulation size and individual PE requirements.

7.6 Transputer implementation.

The next generation of transputer, codenamed the H1, is already under development and will be available early in 1991 [7]. It will be code compatible with the T800 but have five times the performance of the current 20MHz device. It will also have a larger and faster internal RAM but the same external memory interface, and faster links. Although not explicitly stated anywhere, it is reasonable to assume that the other members of the family, the link adaptors and switches, will be similarly upgraded. The significance of this is that the current simulator design is protected. The enhancements planned for the H1 will only mean that the potential scope of the PEs is proportionately enhanced so that it will probably be possible to extend the current operating limits. This stability of design is important when investment in hardware and software is contemplated.

The use of the transputer as the active device was a predictable provision. As mentioned in 7.1, bit slice components, or other conventional processors, are increasing unlikely to be com-
petitive. Design ideas were tested on two transputer networks but, for financial reasons, it was never possible to build a complete PE and check that the performance estimates were correct. With suitable backing, an early item of further work will be to correct that omission and build a small prototype system. It should then be possible to compare the various alternatives which have been discussed, for example, alternative ways of collecting results and different expansion schemes.

The novelty of the transputer system design is in the unconventional use of link adaptors and the undocumented use of link switches. There is no evidence in the literature of other groups using transputers in the manner described and little evidence of continuous system simulation being considered as a suitable transputer application area. It can fairly be claimed that some originality has been demonstrated in the choice and application of the transputer.

7.7 Other applications.

The configuration of a transputer based simulator has much in common with a general purpose parallel processor so it is reasonable to consider whether the simulator has wider application beyond the present view. The main problem in proposing a change, or duplication, of role is in the switching system. In the simulator, the connection links carry 64 (or 32) bit floating point numbers which are transmitted as a dissociated sequence of bytes. The link protocol is not used and the fact that the link adaptor links do not have DMA capability means that it is not easy to change the established way of using them. The communication software, which constrains all transfers to take place within the transmit period, would also need to be changed so that communicating processes can communicate when ready. There is clearly a need to look again at this aspect of the simulator.

Nonetheless the hardware is in place and despite the fact that changes may be necessary, there is evidence of a shift towards hardware, as opposed to software, simulation which could be met by this system. This shift is prompted by the increases in complexity of the simuland which have outstripped the increases in the available computing power. With complex structures such as telecommunications networks for instance [8], satisfactory performance can only be achieved with hardware. Typical simulation studies are concerned with assessing network routing strategies and protocols and even aspects of network node operation. For local
and metropolitan (LAN and MAN) simulations, it is not difficult to see how this system could be configured to allow simplified models of network nodes built.

The software installed in each node (each PE) would of course be very different for discrete system simulations and include procedures for generating Poisson or other distributions and for organising and maintaining queues. Even so, the simulation of a discrete system seems no more difficult than of a continuous one, and although the detailed investigative work has not been done, implementation certainly looks a possibility. It just might be that the present configuration is an over-provision for the majority of discrete systems simulations.

Much the same arguments apply to the use of the system as a general parallel processing tool. The PE hardware has very general application but attention would be needed to the switching system (perhaps restoring the handshaking connections) and to the software which accesses it. The investment may however be worthwhile.

7.8 Conclusion.

It was established at the beginning of phase-2 of this project that there was a gap in the market between the PC-based software systems and the larger (dedicated) simulators which could be neatly filled by this proposal. The question has been kept under review ever since and had it appeared that the exercise was becoming wholly academic, the chances of this final conclusion being reached would have been minimal. At this point, most of the difficult questions about implementation have been answered - the need now is for support to build a prototype.

The concept of outwardly identical devices being configured to act as particular kinds of processing elements by sending to them different blocks of execution code, is not totally original because MIMD machine are known to work on much the same principle. When combined with the flexibility of the proposed switching system however, the result is unlike any known previous proposal. The way in which the data processors have developed, with added functionality and, most particularly, state variable initial value assignments, is claimed to be a considerable enhancement on digital filter structures. Further, the specification of other pro-
cessing elements in hardware, although a direct consequence of the overall system philosophy, is new and, as previously stated, there is no evidence that implementation using transputers and transputer link devices is being considered elsewhere. Finally, the testing technique will have ensured that all the performance estimates are soundly based.

As a simulation product it is unique and as a research assignment it has been intellectually and professionally challenging.

7.9 References.


Statement.

Of the matters covered in this thesis the following is stated:-

- The development of the local and global switches and the adoption of a particular switching philosophy is my responsibility.

- The proposal to produce a software simulation of the simulator, and it subsequent production, were my responsibilities.

- The basic configuration of a data processor is a development of the recursive digital filter and is not original. The means of incorporating state variable initial values was developed from an idea of Dr. Forsythe's. Most of what remains in Chapter 3 about data processors, and all the testing strategy, is my responsibility.

- The idea of identical re-configurable processing elements has emerged over several years of discussion between Dr. Forsythe and myself - I am not sure now where it originated. However, the details of operation, pinout, device types and almost everything else about them are my responsibility.

- The proposal to use transputers was my idea. I gratefully acknowledge the help of Mark Bowler in setting up some of the tests on his transputer equipment, and the two project students, Steve Wallis and Peter Benjamin, who verified some aspects of the proposed design for me. The design itself is my responsibility.

- The proposals regarding the user interfaces (Chapter 6) and the changes and alternative operating procedures suggested in Chapter 7 are my ideas.

M.J. Morse.

December 1989.