



THE
▪
TRANSPUTER
▪
APPLICATIONS
▪
NOTEBOOK

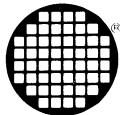
Architecture and Software

FIRST EDITION 1989



ST SGS-THOMSON
MICROELECTRONICS

inmos is a member of the SGS-THOMSON Microelectronics group



Worldwide Headquarters

INMOS Limited
1000 Aztec West
Almondsbury
Bristol BS12 4SQ
UNITED KINGDOM
Telephone (0454) 616616
Fax (0454) 617910
Telex 444723

Worldwide Business Centres

USA

INMOS Business Centre
Headquarters (USA)
SGS-THOMSON Microelectronics Inc
2225 Executive Circle
PO Box 16000
Colorado Springs
Colorado 80935-6000
Telephone (719) 630 4000
Fax (719) 630 4325
Telex Easylink 62944936

SGS-THOMSON Microelectronics Inc
Sales and Marketing Headquarters (USA)
1000 East Bell Road
Phoenix
Arizona 85022
Telephone (602) 867 6100
Fax (602) 867 6102
Telex 249976 SGSPH UR

INMOS Business Centre
SGS-THOMSON Microelectronics Inc
Five Burlington Woods Drive
Suite 201
Burlington
Massachusetts 01803
Telephone (617) 229 2550
Fax (617) 229 6010
Telex Easylink 62934544

INMOS Business Centre
SGS-THOMSON Microelectronics Inc
9861 Broken Land Parkway
Suite 320
Columbia
Maryland 21045
Telephone (301) 995 6952
Fax (301) 290 7047
Telex Easylink 62034521

INMOS Business Centre
SGS-THOMSON Microelectronics Inc
200 East Sandpointe
Suite 650
Santa Ana
California 92707
Telephone (714) 957 6018
Fax (714) 957 3281
Telex Easylink 62034531

INMOS Business Centre
SGS-THOMSON Microelectronics Inc
2620 Augustine Drive
Suite 100
Santa Clara
California 95054
Telephone (408) 727 7771
Fax (408) 727 1458
Telex Easylink 62203010

INMOS Business Centre
SGS-THOMSON Microelectronics Inc
1310 Electronics Drive
Carrollton
Texas 75006
Telephone (214) 466 8844
Fax (214) 466 7352

United Kingdom

INMOS Business Centre
SGS-THOMSON Microelectronics Ltd
Planar House
Parkway Globe Park
Marlow
Bucks SL7 1YL
Telephone (0628) 890 800
Fax (0628) 890 391
Telex 847458

France

INMOS Business Centre
SGS-THOMSON Microelectronics SA
7 Avenue Gallieni
BP 93
94253 Gentilly Cedex
Telephone (1) 47 40 75 75
FAX (1) 47 40 79 10
Telex 632570 STMHQ

West Germany

INMOS Business Centre
SGS-THOMSON Microelectronics GmbH
Bretonischer Ring 4
8011 Grasbrunn
Telephone (089) 46 00 60
Fax (089) 460 54 54
Telex 528211

EUROPE

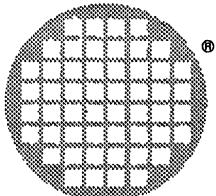
Japan

INMOS Business Centre
SGS-THOMSON Microelectronics K K
Nisseki Takanawa Building, 4th Floor
18-10 Takanawa 2-chome
Minato-ku
Tokyo 108
Telephone (03) 280 4125
Fax (03) 280 4131

Singapore

INMOS Business Centre
SGS-THOMSON Microelectronics Pte Ltd
28 Ang Mo Kio Industrial Park 2
Singapore 2056
Telephone (65) 482 14 11
Fax (65) 482 02 40
Telex RS 55201 ESGIES

ASIA PACIFIC



inmos®

TRANSPUTER APPLICATIONS NOTEBOOK

Architecture and Software

First Edition May 1989

INMOS Databook Series

Transputer Databook

Transputer Support Databook: Development and Sub-systems

Memory Databook

Graphics Databook

Digital Signal Processing Databook

Military Micro-Products Databook

Transputer Applications Notebook: Architecture and Software

Transputer Applications Notebook: Systems and Performance

Copyright ©INMOS Limited 1989

INMOS reserves the right to make changes in specifications at any time and without notice. The information furnished by INMOS in this publication is believed to be accurate; however, no responsibility is assumed for its use, nor for any infringement of patents or other rights of third parties resulting from its use. No licence is granted under any patents, trademarks or other rights of INMOS.

 , **inmos** , IMS and occam are trademarks of the INMOS Group of Companies.

INMOS is a member of the SGS-THOMSON Microelectronics Group of Companies.

INMOS document number: 72-TRN-206-00

Printed at Redwood Press Limited, Melksham, Wiltshire

Contents overview

1	INMOS	An overview.
Principles		
2	<i>Communicating processes and occam</i>	Discusses the design of occam. <i>(INMOS technical note 20)</i>
3	<i>The transputer implementation of occam</i>	Explains how the concurrent processes of occam can be implemented by the transputer. <i>(INMOS technical note 21)</i>
4	<i>Communicating process computers</i>	Discusses construction of parallel computers from collections of transputers. <i>(INMOS technical note 22)</i>
5	<i>Compiling occam into silicon</i>	Describes how communicating processes can be implemented directly on silicon. <i>(INMOS technical note 23)</i>
Practice		
6	<i>The development of occam 2</i>	Shows how occam has been extended to support numerical applications. <i>(Originally INMOS technical note 32)</i>
7	<i>IMS T800 architecture</i>	Describes the design of the IMS T800 transputer. <i>(INMOS technical note 06)</i>
8	<i>The role of occam in the design of the IMS T800</i>	Shows how the floating-point unit of the IMS T800 was verified. <i>(INMOS technical note 47)</i>
9	<i>Simpler real-time programming with the transputer</i>	Illustrates how occam and the transputer simplifies the design of real-time processing systems. <i>(INMOS technical note 51)</i>
10	<i>Long arithmetic on the computer</i>	Shows how to perform efficient multiple length arithmetic on the transputer. <i>(INMOS technical note 39)</i>
11	<i>Exploiting concurrency: a ray tracing example</i>	Describes a concurrent ray tracing program. <i>(INMOS technical note 07)</i>
12	<i>High-performance graphics with the IMS T800</i>	Discusses concurrency in computer graphics. <i>(INMOS technical note 37)</i>
13	<i>A transputer based multi-user flight simulator</i>	Describes the operation of a multi-pilot flight simulator. <i>(INMOS technical note 36)</i>
14	<i>Porting SPICE to the INMOS IMS T800 transputer</i>	Describes an example of porting existing FORTRAN applications to the transputer. <i>(INMOS technical note 52)</i>
15	<i>A transputer farm accelerator for networked computing facilities</i>	Shows a general purpose technique for using multiple transputers to accelerate conventional applications. <i>(INMOS technical note 54)</i>

Contents

Preface	xiii
1 INMOS	1
1.1 Introduction	2
1.2 Manufacturing	2
1.3 Assembly	2
1.4 Test	2
1.5 Quality and Reliability	2
1.6 Military	2
1.7 Future Developments	3
1.7.1 Research and Development	3
1.7.2 Process Developments	3
1 Principles	5
2 Communicating processes and OCCAM	7
2.1 Introduction	7
2.2 Architecture	7
2.2.1 Locality	7
2.2.2 Simulated and real concurrency	7
2.3 The OCCAM primitives	8
2.4 The parallel construct	9
2.4.1 Synchronised communication	9
2.5 The alternative construct	10
2.5.1 Output guards	11
2.6 Channels and hierarchical decomposition	12
2.7 Arrays and replicators	12
2.8 Time	13
2.9 Types and data structures	14
2.10 Implementation of occam	14
2.10.1 Compile-time allocation	15
2.11 Program development	15
2.11.1 Configuration	16
2.12 occam programs	17
2.12.1 Example: systolic arrays	18
2.12.2 Example: OCCAM compiler	18
2.13 Conclusions	20
2.14 References	20
3 The transputer implementation of OCCAM	21
3.1 Introduction	21
3.2 Architecture	21
3.3 occam	21
3.4 The transputer	22
3.4.1 Sequential processing	22
3.4.2 Instructions	23
Direct functions	23
Prefix functions	24
Indirect functions	24
3.4.3 Expression evaluation	25
Efficiency of encoding	25

3.4.4	Support for concurrency	25
Communications		27
Internal channel communication		27
External channel communication		29
Timer		30
Alternative		30
3.4.5	Inter-transputer links	30
3.5	Summary	31
3.6	References	31
 4	Communicating process computers	33
4.1	Introduction	33
4.2	Applications with special configurations	33
4.2.1	Algorithmic parallelism or dataflow decomposition	34
Example: OCCAM compiler		34
Example: solid modelling		35
4.2.2	Geometric parallelism or data structure decomposition	37
Statistical mechanics		37
4.2.3	Farming out processing	39
Example: Graphical representation of the Mandelbrot Set		39
Example: ray tracing		41
Some comments about the previous two examples		42
4.3	General-purpose structures	42
4.3.1	Routing and the communication/computation trade-off	43
4.3.2	Comparison of pipelines and processor farms	44
4.4	References	46
 5	Compiling OCCAM into silicon	47
5.1	Introduction	47
5.2	VLSI design	47
5.3	occam	47
5.4	Implementation of occam	48
5.5	The abstract micro-machine	49
5.6	The compiler output	49
5.7	Variables, expressions, assignment and SEQ	50
5.8	IF and WHILE	51
5.9	Arrays	52
5.10	Procedures	53
5.11	PAR	53
5.12	Channels and communication	53
5.13	ALT	54
5.14	Example: the prime farm	55
5.15	Example: signal processing	56
5.16	Example: simple processor	56
5.17	Conclusions	59
5.18	References	59

2 Practice	61
<hr/>	<hr/>
6 The development of OCCAM 2	63
6.1 Introduction	63
6.2 The data types of occam 2	63
6.3 Channel protocols	64
6.4 Numerical behaviour	66
6.5 Abbreviations	67
6.6 Alias checking	68
6.7 Checking the validity of parallel constructs	70
6.8 Run-time error handling in Occam 2	71
6.9 Conclusions	72
6.10 References	72
<hr/>	<hr/>
7 IMS T800 architecture	73
7.1 Introduction	73
7.2 The transputer: basic architecture and concepts	73
7.2.1 A programmable device	73
7.2.2 Processor and memory on a single chip	73
7.2.3 Serial communication between transputers	74
7.2.4 Simplified processor with microcoded scheduler	74
7.2.5 Transputer products	74
7.3 IMS T800 architecture	74
7.3.1 Instruction encoding	75
7.3.2 Floating-point instructions	76
7.3.3 Optimising use of the stack	78
7.3.4 Concurrent operation of FPU and CPU	78
7.4 Floating-point unit design	80
7.5 Floating-point performance	82
7.6 Formal methods ensure correctness and quick design	82
7.6.1 Z specification	83
7.6.2 High-level occam implementation	83
7.6.3 Low-level OCCAM implementation	84
7.6.4 Flattened low-level implementation	84
7.6.5 Microcode	85
7.6.6 Summary	85
7.7 Communication links	85
7.8 Graphics capability	86
7.8.1 Instruction description	86
7.8.2 Drawing coloured text	88
7.9 Conclusions	90
7.10 References	90
7.11 Note on occam	90
7.12 Note on the 'Livermore Loops'	92
7.13 Note on the formal specification language Z	92
<hr/>	<hr/>
8 The role of OCCAM in the design of the IMS T800	95
8.1 Introduction	95
8.2 occam	95
8.2.1 occam transformations	95
An example transformation	96
8.2.2 The OCCAM transformation system	97

8.3	Instruction development	97
8.4	An example instruction development	97
8.4.1	Preliminary definitions	98
8.4.2	The instruction specification	99
8.4.3	Refining to procedural form	99
8.4.4	Refining to OCCAM	99
8.4.5	High-level OCCAM implementation	100
8.4.6	Transformations towards microcode	101
Refining the conditionals		101
Refining the expressions		101
Introducing sequencing		102
8.4.7	Translation to microcode	103
8.4.8	Microcode assembler source	104
8.5	Current and future work	104
8.6	Conclusions	105
8.7	References	105
 9	Simpler real-time programming with the transputer	107
9.1	Introduction	107
9.2	The OCCAM programming language	107
9.2.1	OCCAM programs	107
9.2.2	Timers in OCCAM	110
9.2.3	Timer values	110
9.2.4	Modulo operators	111
9.3	Using timers	112
9.3.1	Measuring time intervals	112
9.3.2	Generating a known delay	112
9.3.3	Generating events at regular intervals	113
9.3.4	Use in ALTs	114
Interleaving-processing		114
Timeouts on channels		114
Multiple delayed inputs		115
9.4	Transputer implementation of timers	116
9.4.1	Scheduling latency	116
Low-priority processes		116
High-priority processes		117
9.4.2	Timer instructions	117
Delayed inputs		117
9.5	Some application examples	117
9.5.1	Interrupts	117
9.5.2	Polling	118
9.5.3	A real-time clock/calendar	120
9.5.4	A task scheduler	121
9.5.5	Very long delays	121
9.6	Conclusions	121
9.7	References	122
 10	Long arithmetic on the transputer	123
10.1	Introduction	123
10.2	Requirements	123
10.3	Facilities available on the transputer	123
10.4	Interface description for the OCCAM Predefines	124
10.4.1	The integer arithmetic functions	124

10.4.2	Arithmetic shifts	126
10.5	Methodology	126
10.5.1	Addition	127
10.5.2	Subtraction	128
10.5.3	Multiplication	128
10.5.4	Division	131
10.6	Shift Operations	132
10.6.1	Normalisation	133
10.7	Performance	133
10.7.1	Optimisation, using multiplication as an example	133
Simple Code		134
Using Array Abbreviations and opened loops		134
10.7.2	Performance Figures	135
10.8	Conclusions	135
10.9	References	135
10.10	The Occam Predefined Procedures	136
10.10.1	Definition of terms	136
10.10.2	The integer arithmetic functions	136
10.10.3	Arithmetic shifts	141
10.10.4	Word rotation	142
11	Exploiting concurrency: a ray tracing example	144
11.1	Introduction	144
11.2	Logical architecture	145
11.2.1	Ray tracing	145
11.2.2	Introducing concurrency	146
11.3	Physical architecture	147
11.3.1	General description	147
11.3.2	The control/display transputer	148
11.3.3	The calculating transputers	148
11.4	Maximising performance	150
11.5	Fault tolerance of the system	151
11.6	References	151
11.7	Note on the ray tracing program	152
12	High-performance graphics with the IMS T800	155
12.1	Introduction	155
12.2	Computer graphics techniques	155
12.2.1	Modelling objects	155
12.2.2	Transformation	156
The homogeneous coordinate system		156
Translation		156
Rotation		157
Concatenation		157
Perspective projection		157
12.2.3	Scan conversion	158
12.2.4	Shading	158
12.2.5	Clipping	159
12.2.6	Hidden surface removal	159
12.3	The IMS T800 transputer	159
12.3.1	Serial links	160
12.3.2	On-chip floating-point unit	161
12.3.3	2-D block move instructions	161

12.3.4	The OCCAM programming language	161
12.3.5	Meeting computer graphics requirements	161
12.4	3-D transformation on the IMS T800	162
12.5	The INMOS distributed Z-buffer	164
12.5.1	The Z-buffer algorithm	165
12.5.2	Scan conversion	165
	Scan-converting polygons	165
	Scan-converting spheres	165
	Implementation details	166
	Distributing scan conversion over multiple transputers	167
12.5.3	Architecture	168
12.5.4	Performance	169
12.6	The INMOS multi-player flight simulator	169
12.6.1	Requirements	169
12.6.2	Implementation details	169
	The distributed polygon shader	169
	Geometry system	170
	BSP-trees	170
12.6.3	Architecture	172
12.6.4	Performance	173
12.7	Conclusions	174
12.8	References	174
13	A transputer based multi-user flight simulator	175
13.1	Introduction	175
13.2	Flight simulators	175
13.3	Architecture	176
13.3.1	An overview	176
13.4	Implementation	179
13.4.1	The ring Control process	179
	The ring Controller	179
	The simulation process	180
	The main process	181
13.4.2	The Data Base manager	182
	Building the BSP tree	182
	Traversing the BSP tree	185
13.4.3	The transformation process	186
13.4.4	Clipping	186
13.4.5	Shading	187
13.4.6	The display	188
13.4.7	User interface	189
13.4.8	The hardware implementation	190
13.5	Conclusions	191
13.6	References	192
14	Porting SPICE to the INMOS IMS T800 transputer	193
14.1	Introduction	193
14.2	Background on SPICE	193
14.3	Background on transputers	194
14.3.1	Transputers	194
14.3.2	The transputer / host relationship	195
14.3.3	SPICE and the transputer	195
14.3.4	Multiple tasks on one or many transputers	195

14.4	The transputer implementation of FORTRAN	196
14.4.1	Placement of the run-time stack	196
14.4.2	Placement of the code	197
14.4.3	Use of stack space	197
14.5	Porting SPICE	197
14.5.1	Routines needing no modification	197
14.5.2	Routines that set the size of VALUE in a COMMON block	198
14.5.3	Routines often supplied in assembler	199
14.5.4	Other routines to be modified	199
14.5.5	Calculating the FORTRAN VALUE array size	200
14.5.6	Problems with long or large simulations	201
14.6	Performance information	201
14.6.1	Performance comparisons	201
14.6.2	Additional performance improvements to SPICE on a T800	201
Faster memory and shorter cycle times	201	
Optimum linkage strategy	202	
Rewriting critical routines in assembler	202	
14.7	Multiple transputer SPICE	202
14.7.1	Ways of running SPICE on multiple transputers	202
14.7.2	A multiple SPICE farm	205
14.7.3	A networked SPICE farm example	206
14.8	Summary	206
14.9	References	207
14.10	Routines for copy, zero and move	208
14.11	Changes to ROOT found by VAX DIFFERENCES	210
14.12	Changes to TITLE found by VAX DIFFERENCES	211
14.13	Rewriting routines in transputer assembler	212
 15	A transputer farm accelerator for networked computing facilities	213
15.1	Introduction	213
15.1.1	A modern trend	213
15.1.2	Resolving the loading problem	213
15.2	The systems involved	214
15.2.1	The INMOS transputer	214
15.2.2	The transputer host	215
15.2.3	The existing computing resource	215
15.2.4	The communications network	215
DECnet Introduction	215	
DECnet concepts	215	
15.2.5	How everything fits together	216
15.3	A specific implementation	216
15.3.1	Overview	216
15.3.2	System design notes	217
Requirements	217	
Overall system floorplan and development strategy	217	
Automated failure recovery and network topology implications	217	
15.3.3	PC support	219
An outline of the PC server	219	
Server extensions	220	
System operation	220	
Implementation of the new server commands	221	
15.3.4	Transputer support	225
Modifications to the application	225	

	The OCCAM harness	227
	The OCCAM multiplexers	228
15.3.5	VAX support	229
15.3.6	Operating the system	230
	Running MultiSPICE at the PC end	230
	Running MultiSPICE at the VAX end	230
15.4	Other considerations	231
15.4.1	Implementation guidelines	231
	Tools required	231
	Suitable applications	231
	Implementation strategy	231
	Timescales	232
15.4.2	Multiple task farms	232
15.4.3	Receiving work from DOS rather than DECnet	232
15.4.4	Network monitoring software	232
15.4.5	Other transputer hosts	233
15.4.6	Is it worth it? — Weighing up the pros and cons of using transputers	233
15.5	Summary and conclusions	233
15.6	References	234

Preface

The Transputer Applications Notebook — Architecture and Software is a compilation of technical notes written by INMOS technologists to explain the architectural foundation of occam and the INMOS transputer. The collection is divided into two sections which describe an approach to VLSI computer architecture based on communicating processes.

The papers were originally written as a series of individual technical notes with the intention of investigating and developing specific areas of interest or application. The publication will be of particular interest to the computer scientist, electronic engineer, mathematician and system designer. It has been published in response to the growing interest and requests for information about occam and the transputer.

The INMOS transputer is a VLSI building block for concurrent processing systems with occam as the associated design formalism. occam is an easy and natural language for the programming and specification of concurrent systems.

Information concerning the use of transputer products is available in a companion publication of technical notes, ie The Transputer Applications Databook — Systems and Performance.

In addition to transputer products, the INMOS product range includes graphics devices, digital signal processing devices and fast static RAMs. For further information concerning INMOS products please contact your local INMOS sales outlet.

The role of OCCAM in the design of the T800, presented at the 20th Workshop on Microprogramming, December 1-4, 1987, Colorado Springs, CO. To appear in ACM SIGMICRO Newsletter, Vol. 18, No. 4, 1987.

A transputer based multi-user flight simulator, parts of this chapter are published by The International Supercomputing Institute, Inc. (ISI, Inc.) in the proceedings of the 3rd International Conference of Supercomputing and, as such, are copyright of ISI, Inc.



INMOS

1.1 Introduction

INMOS is a recognised leader in the development and design of high-performance integrated circuits and is a pioneer in the field of parallel processing. The company manufactures components designed to satisfy the most demanding of current processing applications and also provide an upgrade path for future applications. Current designs and development will meet the requirements of systems in the next decade. Computing requirements essentially include high-performance, flexibility and simplicity of use. These characteristics are central to the design of all INMOS products.

INMOS has a consistent record of innovation over a wide product range and supplies components to system manufacturing companies in the United States, Europe, Japan and the Far East. As developers of the Transputer, a unique microprocessor concept with a revolutionary architecture, and the OCCAM parallel processing language, INMOS has established the standards for the future exploitation of the power of parallel processing. INMOS products include a range of transputer products in addition to a highly successful range of high-performance graphics devices, an innovative and successful range of high-performance digital signal processing (DSP) devices and a broad range of fast static RAMs, an area in which it has achieved a greater than 10% market share.

The corporate headquarters, product design team and worldwide sales and marketing management are based at Bristol, UK.

INMOS is constantly upgrading, improving and developing its product range and is committed to maintaining a global position of innovation and leadership.

1.2 Manufacturing

INMOS products are manufactured at the INMOS Newport, Duffryn facility which began operations in 1983. This is an 8000 square metre building with a 3000 square metre cleanroom operating to Class 10 environment in the work areas.

To produce high performance products, where each microchip may consist of up to 300,000 transistors, INMOS uses advanced manufacturing equipment. Wafer steppers, plasma etchers and ion implanters form the basis of fabrication.

1.3 Assembly

Sub-contractors in Korea, Taiwan, Hong Kong and the UK are used to assemble devices.

1.4 Test

The final testing of commercial products is carried out at the INMOS Newport, Coed Rhedyn facility. Military final testing takes place at Colorado Springs.

1.5 Quality and Reliability

Stringent controls of quality and reliability provide the customer with early failure rates of less than 1000 ppm and long term reliability rates of better than 100 FITs (one FIT is one failure per 1000 million hours). Requirements for military products are even more stringent.

1.6 Military

Various INMOS products are already available in military versions processed in full compliance with MIL-STD-883C. Further military programmes are currently in progress.

1.7 Future Developments**1.7.1 Research and Development**

INMOS has achieved technical success based on a position of innovation and leadership in products and process technology in conjunction with substantial research and development investment. This investment has averaged 18% of revenues since inception and it is anticipated that future investment will be increased.

1.7.2 Process Developments

One aspect of the work of the Technology Development Group at Newport is to scale the present 1.2 micron technology to 1.0 micron for products to be manufactured in 1988/89. In addition, work is in progress on the development of 0.8 micron CMOS technology.



Principles

2 Communicating processes and occam

2.1 Introduction

The occam programming language [1] enables an application to be described as a collection of processes which operate concurrently and communicate through channels. In such a description, each occam process describes the behaviour of one component of the implementation, and each channel describes a connection between components.

The design of occam allows the components and their connections to be implemented in many different ways. This allows the choice of implementation technique to be chosen to suit available technology, to optimise performance, or to minimise cost.

occam has proved useful in many application areas. It can be efficiently implemented on almost any computer and is being used for many purposes — real-time systems, compilers and editors, hardware specification and simulation.

2.2 Architecture

Many programming languages and algorithms depend on the existence of the uniformly accessible memory provided by a conventional computer. Within the computer, memory addressing is implemented by a global communications system, such as a bus. The major disadvantage of such an approach is that speed of operation is reduced as the system size increases. The reduction in speed arises both from the increased capacitance of the bus which slows down every bus cycle, and from bus contention.

The aim of occam is to remove this difficulty; to allow arbitrarily large systems to be expressed in terms of localised processing and communication. The effective use of concurrency requires new algorithms designed to exploit this locality.

The main design objective of occam was therefore to provide a language which could be directly implemented by a network of processing elements, and could directly express concurrent algorithms. In many respects, occam is intended as an assembly language for such systems; there is a one-to-one relationship between occam processes and processing elements, and between occam channels and links between processing elements.

2.2.1 Locality

Almost every operation performed by a process involves access to a variable, and so it is desirable to provide each processing element with local memory in the same VLSI device.

The speed of communication between electronic devices is optimised by the use of one directional signal wires, each connecting only two devices. This provides local communication between pairs of devices.

occam can express the locality of processing, in that each process has local variables; it can express locality of communication in that each channel connects only two processes.

2.2.2 Simulated and real concurrency

Many concurrent languages have been designed to provide simulated concurrency. This is not surprising, since until recently it has not been economically feasible to build systems with a lot of real concurrency.

Unfortunately, almost anything can be simulated by a sequential computer, and there is no guarantee that a language designed in this way will be relevant to the needs of systems with real concurrency. The choice of features in such languages has been motivated largely by the need to share one computer between many independent tasks. In contrast, the choice of features in occam has been motivated by the need to use many communicating computers to perform one single task.

An important objective in the design of occam was to use the same concurrent programming techniques both for a single computer and for a network of computers. In practice, this meant that the choice of features in occam was partly determined by the need for an efficient distributed implementation. Once this had been achieved, only simple modifications were needed to ensure an efficient implementation of concurrency on a single sequential computer. This approach to the design of occam perhaps explains some of the differences between OCCAM and other 'concurrent' languages.

2.3 The occam primitives

occam programs are built from three primitive processes:

v := e	assign expression e to variable v
c ! e	output expression e to channel c
c ? v	input variable v from channel c

The primitive processes are combined to form constructs:

SEQ	sequence
IF	conditional
PAR	parallel
ALT	alternative

A construct is itself a process, and may be used as a component of another construct.

Conventional sequential programs can be expressed with variables and assignments, combined in sequential and conditional constructs. The order of expression evaluation is unimportant, as there are no side effects and operators always yield a value.

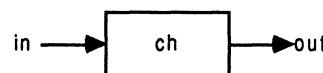
Conventional iterative programs can be written using a while loop. The absence of explicit transfers of control perhaps needs no justification in a modern programming language; in occam it also removes the need to prohibit, or define the effect of, transferring control out of a parallel component or procedure.

Concurrent programs make use of channels, inputs and outputs, combined using parallel and alternative constructs.

The definition and use of occam procedures follows ALGOL-like scope rules, with channel, variable and value parameters. The body of an occam procedure may be any process, sequential or parallel. To ensure that expression evaluation has no side effects and always terminates, occam does not include functions.

A very simple example of an occam program is the buffer process below:

```
WHILE TRUE
  VAR ch:
  SEQ
    in ? ch
    out ! ch
```



Indentation is used to indicate program structure. The buffer consists of an endless loop, first setting the variable **ch** to a value from the channel **in**, and then outputting the value of **ch** to the channel **out**. The variable **ch** is declared by **VAR ch**. The direct correspondence between the program text and the pictorial representation is important, as a picture of the processes (processors) and their connections is often a useful starting point in the design of an efficiently implementable concurrent algorithm.

2.4 The parallel construct

The components of a parallel construct may not share access to variables, and communicate only through channels. Each channel provides one-way communication between two components; one component may only output to the channel and the other may only input from it. These rules are checked by the compiler.

The parallel construct specifies that the component processes are 'executed together'. This means that the primitive components may be interleaved in any order. More formally:

$$\begin{array}{ll} \text{PAR} & \text{SEQ} \\ \text{SEQ} & = \quad x := e \\ x := e & \quad \text{PAR} \\ P & \quad P \\ Q & \quad Q \end{array}$$

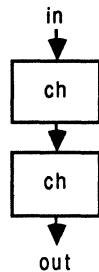
so that the initial assignments of two concurrent processes may be executed in sequence until both processes start with an input or output. If one process starts with an input on channel **c**, and the other an output on the same channel **c**, communication takes place:

$$\begin{array}{ll} \text{PAR} & \text{SEQ} \\ \text{SEQ} & = \quad x := e \\ c ! e & \quad \text{PAR} \\ P & \quad P \\ \text{SEQ} & \quad Q \\ c ? x & \\ Q & \end{array}$$

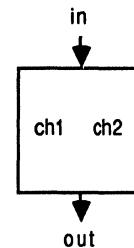
The above rule states that communication can be thought of as a distributed assignment.

Two examples of the parallel construct are shown below:

```
CHAN c:
PAR
WHILE TRUE
  VAR ch:
  SEQ
    in ? ch
    c ! ch
  WHILE TRUE
    VAR ch:
    SEQ
      c ? ch
      out ! ch
```



```
VAR ch1:
VAR ch2:
SEQ
  in ? ch1
  WHILE TRUE
    SEQ
      PAR
        in ? ch2
        out ! ch1
    PAR
      in ? ch1
      out ! ch2
```



The first consists of two concurrent versions of the previous example, joined by a channel to form a 'double buffer'. The second is perhaps a more conventional version. As 'black boxes', each with an input and an output channel, the behaviour of these two programs is identical; only their internals differ.

2.4.1 Synchronised communication

Synchronised, zero-buffered, communication greatly simplifies programming, and can be efficiently implemented. In fact, it corresponds directly to the conventions of self timed signalling[2]. Zero-buffered communication eliminates the need for message buffers and queues. Synchronised communication prevents accidental loss of data arising from programming errors. In an unsynchronised scheme, failure to acknowledge data often results in a program which is sensitive to scheduling and timing effects.

Synchronised communication requires that one process must wait for the other. However, a process which requires to continue processing whilst communicating can easily be written:

```
PAR
  c ! x
  p
```

2.5 The alternative construct

In occam programs, it is sometimes necessary for a process to input from any one of several other concurrent processes. This could have been provided by a channel 'test', which is true if the channel is ready, false otherwise. However, this is unsatisfactory because it requires a process to poll its inputs 'busily'; in some (but by no means all) cases this is inefficient.

Consequently, occam includes an alternative construct similar to that of CSP [3]. As in CSP, each component of the alternative starts with a guard — an input, possibly accompanied by a boolean expression. From an implementation point of view, the alternative has the advantage that it can be implemented either 'busily' by a channel test or by a 'non-busy' scheme. The alternative enjoys a number of useful semantic properties more fully discussed in [4,5]; in particular, the formal relationship between parallel and alternative is shown below:

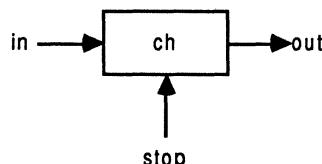
<pre>ALT c ? x PAR SEQ c ? x P SEQ d ? y Q SEQ d ? y PAR Q SEQ c ? x P</pre>	$=$	<pre>PAR c ? x p</pre>
--	-----	----------------------------

This equivalence states that if two concurrent processes are both ready to input (communicate) on different channels, then either input (communication) may be performed first.

One feature of CSP omitted from occam is the automatic failure of a guard when the process connected to the other end of the channel terminates. Although this is a convenient programming feature, it complicates the channel communication protocol, introducing the need for further kinds of message. In addition, it can be argued that many programs are clearer if termination is expressed explicitly.

A simple example of the alternative is shown below; this is a 'stoppable' buffer program:

```
WHILE going
  ALT
    in ? ch
    out ! ch
    stop ? ANY
    going := FALSE
```



in which **stop ? ANY** inputs any value from the channel **stop**, and as a result causes the loop to terminate.

2.5.1 Output guards

Output guards are a very convenient programming tool. In particular, they allow programs such as the following buffer process to be written in a natural way:

```
WHILE TRUE
  ALT
    count>0 & output ! buff [ outpointer ]
      SEQ
        outpointer := (outpointer + 1) REM max
        count := count - 1
    count<max & input ? buff [ inpointer ]
      SEQ
        inpointer := (inpointer + 1) REM max
        count := count + 1
```

It is very tempting to include output guards in a communicating process language, and attempts have been made to include output guards in occam. The major difficulty is in the distributed implementation; in a program such as:

```
PAR
  ALT
    c ! x1
    d ? x2
  ALT
    c ? y1
    d ! y2
```

what is expected to happen in the event that two identical processors both enter their alternative at exactly the same time? Clearly some asymmetry must be introduced; the easiest way to do this is to give each processor in a system a unique number. Even so, the provision of output guards greatly complicates the communications protocol. For this reason, output guards are omitted from occam, and the above buffer must be written as shown below:

```
PAR
  WHILE TRUE
    ALT
      count>0 & req ? ANY
        SEQ
          reply ! buff [ outpointer ]
          outpointer := (outpointer + 1) REM max
          count := count - 1
      count<max & input ? buff [ inpointer ]
        SEQ
          inpointer := (inpointer + 1) REM max
          count := count + 1
    WHILE TRUE
      SEQ
        req ! ANY
        reply ? ch
        output ! ch
```

On the other hand, an occam implementation with only input guards can be used to write the communications kernel for a 'higher level' version of occam with output guards. An example of an algorithm to implement output guards in CSP is given in [6]; and one for occam is given in [7].

2.6 Channels and hierarchical decomposition

An important feature of occam is the ability to successively decompose a process into concurrent component processes. This is the main reason for the use of named communication channels in occam. Once a named channel is established between two processes, neither process need have any knowledge of the internal details of the other. Indeed, the internal structure of each process can change during execution of the program.

The parallel construct, together with named channels provides for decomposition of an application into a hierarchy of communicating processes, enabling occam to be applied to large-scale applications. This technique cannot be used in languages which use process (or 'entry') names, rather than channels, for communication.

In specifying the behaviour of a process, it is important that a specification of the protocol used on the channel exists, and the best way to do this varies from program to program (or even from channel to channel!). For example, Backus-Naur Form is often suitable for describing the messages which pass between the individual processes of a linear pipeline of processes. On the other hand, for more complex interactions between processes, it is often useful to describe the interactions by an occam 'program' in which all unnecessary features are omitted. This often enables the interactions between processes to be studied independently of the data values manipulated. For example:

```
SEQ
  request ?
  WHILE TRUE
    PAR
      reply !
      request ?
```

describes a process which inputs a request, and then endlessly inputs a new request and outputs a reply, in either order. Such a process would be compatible, in some sense, with any of the following processes:

<pre>WHILE TRUE SEQ request ! WHILE TRUE reply ?</pre>	<pre>SEQ request ! WHILE TRUE SEQ request ! reply ?</pre>	<pre>SEQ request ! WHILE TRUE PAR request ! reply ?</pre>
--	---	---

More design aids are needed to assist in the specification and checking of channel protocols.

2.7 Arrays and replicators

The representation of arrays and 'for' loops in occam is unconventional. Although this has nothing to do with the concurrency features of occam, it seems to have significant advantages over alternative schemes.

To eliminate trivial programming errors, it is desirable that there is a simple relationship between an array declaration and a loop which performs some operation for every element of an array. This might lead a language designer to a choice of:

```
ARRAY a [base TO limit] ...
FOR i IN [base TO limit] ...
```

It is also useful if the number of elements in an array, or the number of iterations of a loop, is easily visible. For this reason, a better choice might be:

```
ARRAY a [base FOR count] ...
FOR i IN [base FOR count] ...
```

For the loop, this gives a further advantage: the 'empty' loop corresponds to **count=0** instead of **limit|base**. This removes the need for the unsatisfactory 'loop':

```
FOR i IN [0 TO -1]
```

Implementation can be simplified by insisting that all arrays start from 0. Finally, in **occam** the **FOR** loop is generalised, and its semantics simplified. An **occam** 'replicator' can be used with any of **SEQ**, **PAR**, **ALT** and **IF**; its meaning is defined by:

$$\begin{array}{lcl} \mathbf{x} \; n = b \; \mathbf{FOR} \; c & = & \mathbf{x} \\ \mathbf{P}(n) & & \mathbf{P}(b) \\ & & \mathbf{P}(b+1) \\ & & \vdots \\ & & \mathbf{P}(b+c-1) \end{array}$$

where **x** is one of **SEQ**, **PAR**, **ALT** and **IF**, **n** is a name and **b**, **c** expressions. This definition implicitly defines the 'control variable' **n**, and prevents it being changed by assignments within **P**.

The introduction of arrays of variables and channels does complicate the rules governing the correct use of channels and variables. Simple compile-time checks which are not too restrictive are:

No array changed by assignment (to one of its components) in any of the components of a parallel may be used in any other component

No two components of a parallel may select channels from the same array using variable subscripts

A component of a parallel which uses an array for both input and output may not select channels from the array using variable subscripts

where a variable subscript is a subscript which cannot be evaluated by the compiler.

2.8 Time

The treatment of time in **occam** directly matches the behaviour of a conventional alarm clock.

Time itself is represented in **occam** by values which cycle through all possible integer values. Of course, it would have been possible to represent time by a value large enough (say 64 bits) to remove the cyclic behaviour, but this requires the use of multiple-length arithmetic to maintain the clock and is probably not justified.

Using an alarm clock, it is possible at any time to observe the current time, or to wait until the alarm goes off. Similarly, a process must be able to read the clock at any time, or wait until a particular time. If it were possible only to read the clock, a program could only wait until a particular time 'busily'. Like the alternative construct, the 'wait until a time' operation has the advantage that it can be implemented 'busily' or 'non-busily'.

A timer is declared in the same way as a channel or variable. This gives rise to a relativistic concept of time, with different timers being used in different parts of a program. A localised timer is much easier to implement than a global timer.

A timer is read by a special 'input':

```
time ? v
```

which is always ready, and sets the variable **v** to the time. Similarly, the 'input':

```
time ? AFTER t
```

waits until time **t**.

The use of an absolute time in OCCAM instead of a delay is to simplify the construction of programs such as:

```
WHILE TRUE
  SEQ
    time ? AFTER t
    t := t + interval
    output ! bell
```

in which n rings of the bell will always take between $(n * \text{interval})$ and $n * (\text{interval} + 1)$ ticks. This would not be true of a program such as:

```
WHILE TRUE
  SEQ
    DELAY interval
    output ! bell
```

because of the time taken to ring the bell.

It is not possible, in OCCAM, for a process to implement a timer. This would require a 'timer output' such as:

```
timer ! PLUS n
```

which advances the timer by n ticks. There is no obvious reason why this could not be included in OCCAM. It would be particularly useful in constructing timers of different rates, or in writing a process to provide 'simulated time'.

2.9 Types and data structures

The OCCAM described so far makes few assumptions about data types. Any data type could be used — provided that values of that type can be assigned, input and output according to the rule:

```
PAR
  c ! x      =      y := x
  c ? y
```

To preserve this rule, and keep the implementation of communication simple, it is best for assignment not to make type conversions.

The initial version of OCCAM provides untyped variables and one-dimensional arrays. No addressing operations are provided, as this would make it impossible for the compiler to check that variables are not shared between concurrent processes.

OCCAM has been extended to include data types. The simple variable is replaced with boolean, byte and integer types, and multi-dimensional arrays are provided. Communication and assignment operate on variables of any data type, allowing arrays to be communicated and assigned.

A detailed description can be found in [8].

2.10 Implementation of OCCAM

The implementation of concurrent processes and process interaction in OCCAM is straightforward. This results from the need to implement OCCAM on the transputer using simple hardware and a small number of microcoded instructions. Conveniently, the transputer instructions used to implement OCCAM can be used as definitions of the 'kernel primitives' in other implementations of OCCAM. A discussion of the implementation of OCCAM can be found in [9]. However, some measure of the efficiency of the OCCAM primitives is provided by the performance of the INMOS transputer: about 1 microsecond/component of PAR, and 1.5 microseconds for a process communication.

Another interesting feature of OCCAM is that the process interactions directly represent hardware mechanisms, which is one reason why OCCAM is being used as a hardware description language.

2.10.1 Compile-time allocation

For run-time efficiency, the advantages of allocating processors and memory at compile-time are clear. To allow the compiler to allocate memory, some implementation restrictions are imposed. Firstly, the number of components of an array, and the number of concurrent processes created by a parallel replicator, must be known at compile-time. Secondly, no recursive procedures are allowed. The effect of these restrictions is that the compiler can establish the amount of space needed for the execution of each component of a parallel construct, and this makes the run-time overhead of the parallel construct very small.

On the other hand, there is nothing in occam itself to prevent an implementation without these restrictions, and this would be fairly straightforward for a single computer with dynamic memory allocation.

A distributed implementation of 'recursive OCCAM' might allow a tree of processors to be described by:

```
PROC tree (VALUE n, CHAN down, CHAN up)
IF
n=0
    leaf ( down, up )
n>0
    CHAN left.down, left.up
    CHAN right.down, right.up
PAR
    tree (n-1, left.down, left.up)
    tree (n-1, right.down, right.up)
    node ( down, up,
            left.down, left.up,
            right.down, right.up )
```

If the depth of the tree is known at compile-time (as it normally would be if the program is to be executed on a fixed-size processor array), the same effect can be achieved by a non-recursive program such as:

```
DEF p = TABLE [1, 2, 4, 8, 16, 32, 64, 128] :
-- depth of tree = n
CHAN down [n*(n-1)] :
CHAN up [n*(n-1)] :

PAR
PAR i = [0 FOR n - 1]
PAR j = [0 FOR p[i]]
branch ( down [p[i] + j], up [p[i] + j],
          down [p[i+1]+(j*2)], up [p[i+1]+(j*2)],
          down [p[i+1]+(j*2)+1], up [p[i+1]+(j*2)+1] )
PAR i = [0 FOR p[n]]
leaf ( down [p[n]+i], up [p[n]+i] )
```

Obviously, a preprocessor could be used to provide a correctness preserving transformation between these two programs.

If the depth of the tree above were not known, it is not clear how such a program could be mapped on to a processor array, either explicitly by the programmer or implicitly by the implementation. Fortunately, this problem can be left for the future; many applications require only simple compile time allocation of processors and memory space.

2.11 Program development

The development of programs for multiple processor systems is not trivial. One problem is that the most effective configuration is not always clear until a substantial amount of work has been done. For this reason, it is very desirable that most of the design and programming can be completed before hardware construction is started.

This problem is greatly reduced by the property of OCCAM mentioned above: the use of the same concurrent programming techniques for both a network and a single computer. A direct consequence of this is that a program ultimately intended for a network of computers can be compiled and executed efficiently by a single computer used for program development.

Another important property of OCCAM in this context is that OCCAM provides a clear notion of 'logical behaviour'; this relates to those aspects of a program not affected by real-time effects. It is guaranteed that the logical behaviour of a program is not altered by the way in which processes are mapped on to processors, or by the speed of processing and communication.

This notion of 'logical behaviour' results from the relatively abstract specification of parallel and alternative; it allows almost any scheduling system to be used to simulate concurrency. For the parallel construct, an implementation may choose the order in which the individual actions of the components are executed. If several components are ready (not waiting to communicate), the implementation may execute an arbitrary subset of them and temporarily ignore the rest. For the alternative, an implementation may select any ready component; there is no requirement to select the 'earliest', or to select randomly.

2.11.1 Configuration

The configuration of a program to meet real-time constraints is provided by annotations to the parallel and alternative constructs. For the parallel construct, the components may be placed on different processors, or may be prioritised. For the alternative construct, the components may be prioritised. A better version of the 'stoppable' buffer shown earlier would therefore be:

```
WHILE going
PRI ALT
  stop ? ANY
    going := FALSE
  in ? ch
  out ! ch
```

A prioritised alternative can easily be used to provide either a prioritised or a 'fair' multiplexor:

```
WHILE TRUE -- prioritised
PRI ALT i = 0 FOR 10
  in [i] ? ch
  out ! ch

WHILE TRUE -- 'fair'
PRI ALT i = 0 FOR 10
  in [(i+last) REM 10] ? ch
    SEQ
      out ! ch
      last := (i+1) REM 10
```

In practice, only limited use is made of prioritisation. For most applications, the scheduling of concurrent processes and the method of selecting alternatives is unimportant. This is because, assuming that the system is executing one program, the processes which are consuming all of the processing resources must eventually stop, and wait for the other processes to do something. If this is not the case, the other processes are redundant, and can be removed from the program. An implementation should not, of course, allow a processor to idle if there is something for it to do. But this property is true of any programming language!

Scheduling is important where a system executes two disjoint processes, or has to meet some externally imposed constraint. Both of these occur, for example, in an operating system which deals with disjoint users, and needs to take data from a disk at an externally imposed rate.

2.12 OCCAM programs

Despite being a fairly small language, occam supports a very wide variety of programming techniques. Most important, the programmer may choose between a concurrent algorithm or an equivalent sequential one. A final program often consists of a mixture of the two, in which the concurrent algorithm describes a network of transputers, each of which executes the sequential algorithm.

In practice, it is often best to write the concurrent algorithm first. The reason for this is that only the concurrent program provides freedom in the implementation. A pipeline of ten processes could be executed by a pipeline constructed from up to ten transputers; the number being chosen according to the performance required. It is very unlikely that a sequential program can easily be adapted to produce a concurrent program, never mind one suitable for execution by a network of transputers with no shared memory.

The following example is a concurrent searching algorithm. It uses the tree program shown earlier. The data to be searched is held in the leaf processors; the node processors are used to disperse the data to the leaves and collect the replies:

```

PROC leaf (CHAN down, up) =
    VAR data, enq:
    SEQ
        ... -- load data
        WHILE TRUE
            SEQ
                down ? enq
                up ! (enq = data)

PROC node (CHAN down, up,
           CHAN left.down, left.up,
           CHAN right.down, right.up) =
    WHILE TRUE
        VAR enq, left.found, right.found :
        SEQ
            down ? enq
            PAR
                left.down ! enq
                right.down ! enq
            PAR
                left.up ? left.found
                right.up ? right.found
            up ! left.found OR right.found

```

However, it is unlikely to be economic to store only one data item in each leaf. Although each leaf could itself execute the above algorithm using a tree of processes, this would not be very efficient. What is needed in each leaf is a conventional sequential searching algorithm operating on an array of data:

```

PROC leaf (CHAN down, up) =
    VAR enq, data [length], found:
    SEQ
        ... -- initialise data
        WHILE TRUE
            SEQ
                found := FALSE
                down ? enq
                SEQ i = [0 FOR length]
                    found := (data [i] = enq ) OR found
                up ! found :

```

It now remains to choose the number of items held in each leaf so that the time taken to disperse the enquiry and collect the response is small relative to the time taken for the search at each leaf. For example, if the time taken for a single communication is 5 microseconds, and the tree is of depth 7 (128 leaves) only 70 microseconds are spent on communication, about one-tenth of the time taken to search 1000 items.

2.12.1 Example: systolic arrays

A very large number of concurrent algorithms require only the simplest concurrency mechanisms: the parallel construct and the communication channel. These include the 'systolic array' algorithms described by Kung [10]. In fact, OCCAM enables a systolic algorithm to be written in one of two ways, illustrated by the following two versions of a simple pipeline, each element of which performs a 'compute' step. First, the traditional version:

```

VAR master [ n ]:
VAR slave [ n ]:
WHILE TRUE
SEQ
    PAR i = 0 FOR n
        compute ( master [ i ], slave [ i ] )
    PAR
        input ? master [ 0 ]
        PAR i = 0 FOR n-1
            master [ i + 1 ] := slave [ i ]
        output ! slave [ n ]
    
```

This pipeline describes a conventional synchronous array processor. The compute operations are performed in parallel, each taking data from a master register and leaving its result in a slave register. The array processor is globally synchronised; in each iteration all compute operations start and terminate together, then the data is moved along the pipeline. The initialisation of the pipeline is omitted, so the first n outputs will be rubbish.

The main problem with the above program is the use of global synchronisation, which gives rise to the same implementation difficulties as global communication; it requires that the speed of operation must be reduced as the array size increases. A more natural program in OCCAM would be:

```

CHAN c [ n + 1 ] :
PAR i = 0 FOR n
WHILE TRUE
    VAR d:
    VAR r:
    SEQ
        c [ n ] ? d
        compute ( d, r )
        c [ n + 1 ] ! r
    
```

In this program, $c[0]$ is the input channel, $c[n+1]$ the output channel. Once again, all of the compute operations are performed together. This time there is no need for initialisation, as no output can be produced until the first input has passed right through the pipeline. More important, the pipeline is self-synchronising; adjacent elements synchronise only as needed to communicate data. It seems likely that many systolic array algorithms could usefully be re-expressed and implemented in this form.

2.12.2 Example: occam compiler

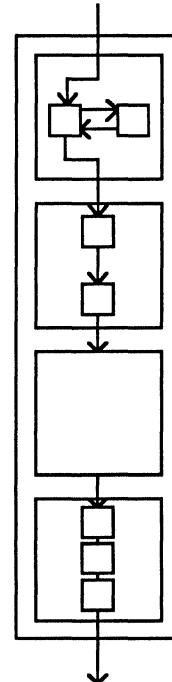
The structure of the OCCAM compiler is shown opposite. It demonstrates an important feature of the OCCAM support system; the ability to 'fold' sections of program away, leaving only a comment visible. This enables a program, or part of a program, to be viewed at the appropriate level of detail.

```
-- occam compiler
CHAN lexed.program:
CHAN parsed.program:
CHAN scoped.program:
PAR
  -- lexer
  CHAN name.text:
  CHAN name.code:
  PAR
    -- scanner
    -- nametable

  -- parser
  CHAN parsed.lines :
  PAR
    -- line parser
    -- construct parser

  -- scoper

  -- generator
  CHAN generated.constructs :
  CHAN generated.program :
  PAR
    -- construct generator
    -- line generator
    -- space allocator
```



The compiler also illustrates an important programming technique. The nametable process contains data structures which are hidden from the rest of the program. These structures are modified only as a result of messages from the lexical analyser. They are initialised prior to receipt of the first message:

```
-- nametable
SEQ
  -- initialise
  WHILE going
    -- input text of name
    -- look up name
    -- output corresponding code
  -- terminate
```

From the outside, the compiler appears to be a single-pass compiler. Internally, it is more like a multiple-pass compiler; each process performs a simple transformation on the data which flows through it. The effect of decomposing the compiler in this way was that each component process was relatively easy to write, specify and test; this meant that the component processes could be written concurrently!

2.13 Conclusions

In many application areas, concurrency can be used to provide considerable gains in performance provided that programs are structured to exploit available technology. For many application areas (especially signal processing and scientific computation) suitable algorithms already exist, but many areas remain to be explored.

Writing programs in terms of communicating processes tends to produce programs with a large number of concurrent processes, ranging in size from 1 to 1000 lines. Consequently, it is particularly important that the concurrent processing features in the language are efficiently implementable. *occam* demonstrates that this efficiency can be achieved for a widely applicable language.

In *occam* programs, the process/channel structure tends to be used as a major program structuring tool, procedures being used in the normal way within the larger concurrent processes. The process/channel structure seems to be effective for managing the construction of large programs, although more experience is needed in this area.

2.14 References

- 1 *occam Programming Manual*, Prentice-Hall International 1984.
- 2 *Introduction to VLSI Systems*, C A Mead and L A Conway, Addison Wesley 1980, Section 5.
- 3 *Communicating Sequential Processes*, C A R Hoare, Communications of the ACM Vol. 21 8 (August 1978), p. 666.
- 4 *Denotational Semantics for occam*, A W Roscoe, Presented at NSF/SERC Seminar on Concurrency, Carnegie-Mellon University, July 1984, To be published.
- 5 *The Laws of occam Programming*, A W Roscoe and C A R Hoare, Programming Research Group, Oxford University 1986.
- 6 *An Effective Implementation for the Generalised Input-Output Construct of CSP*, G N Buckley and A Silberschatz, ACM Transactions on Programming Languages and Systems Vol. 5 (April 1983), p. 224.
- 7 *A Protocol for Generalised occam*, R Bornat, Department of Computer Science, Queen Mary College, London 1984.
- 8 *occam 2 Reference Manual*, INMOS Ltd, Prentice Hall 1988.
- 9 *The Transputer Implementation of occam*, INMOS Ltd, Technical note 21.
- 10 *Lets Design Algorithms for VLSI Systems*, H T Kung in *Introduction to VLSI Systems*, C A Mead and L A Conway, Addison Wesley 1980, Section 8.3

3 The transputer implementation of occam

3.1 Introduction

VLSI technology allows a large number of identical devices to be manufactured cheaply. For this reason, it is attractive to implement an occam [1] program using a number of identical components, each programmed with the appropriate occam process. A transputer [2] is such a component.

A transputer is a single VLSI device with memory, processor and communications links for direct connection to other transputers. Concurrent systems can be constructed from a collection of transputers which operate concurrently and communicate through links.

The transputer can therefore be used as a building block for concurrent processing systems, with occam as the associated design formalism.

3.2 Architecture

An important property of VLSI technology is that communication between devices is very much slower than communication on the same device. In a computer, almost every operation that the processor performs involves the use of memory. A transputer therefore includes both processor and memory in the same integrated circuit device.

In any system constructed from integrated circuit devices, much of the physical bulk arises from connections between devices. The size of the package for an integrated circuit is determined more by the number of connection pins than by the size of the device itself. In addition, connections between devices provided by paths on a circuit board consume a considerable amount of space.

The speed of communication between electronic devices is optimised by the use of one-directional signal wires, each connecting two devices. If many devices are connected by a shared bus, electrical problems of driving the bus require that the speed is reduced. Also, additional control logic and wiring is required to control sharing of the bus.

To provide maximum speed with minimal wiring, the transputer uses point-to-point serial communication links for direct connection to other transputers.

3.3 occam

occam enables a system to be described as a collection of concurrent processes, which communicate with each other and with peripheral devices through channels. occam programs are built from three primitive processes:

```
v := e    assign expression e to variable v
c ! e    output expression e to channel c
c ? v    input from channel c to variable v
```

The primitive processes are combined to form constructs:

SEQuential	components executed one after another
PARallel	components executed together
ALTernative	component first ready is executed

A construct is itself a process, and may be used as a component of another construct.

Conventional sequential programs can be expressed with variables and assignments, combined in sequential constructs. IF and WHILE constructs are also provided.

Concurrent programs can be expressed with channels, inputs and outputs, which are combined in parallel and alternative constructs.

Each OCCAM channel provides a communication path between two concurrent processes. Communication is synchronised and takes place when both the inputting process and the outputting process are ready. The data to be output is then copied from the outputting process to the inputting process, and both processes continue.

An alternative process may be ready for input from any one of a number of channels. In this case, the input is taken from the channel which is first used for output by another process.

3.4 The transputer

A transputer system consists of a number of interconnected transputers, each executing an OCCAM process and communicating with other transputers. As a process executed by a transputer may itself consist of a number of concurrent processes the transputer has to support the OCCAM programming model internally. Within a transputer concurrent processing is implemented by sharing the processor time between the concurrent processes.

The most effective implementation of simple programs by a programmable computer is provided by a sequential processor. Consequently, the transputer processor is fairly conventional, except that additional hardware and microcode support the OCCAM model of concurrent processing.

3.4.1 Sequential processing

The design of the transputer processor exploits the availability of fast on-chip memory by having only a small number of registers; six registers are used in the execution of a sequential process. The small number of registers, together with the simplicity of the instruction set enables the processor to have relatively simple (and fast) data paths and control logic.

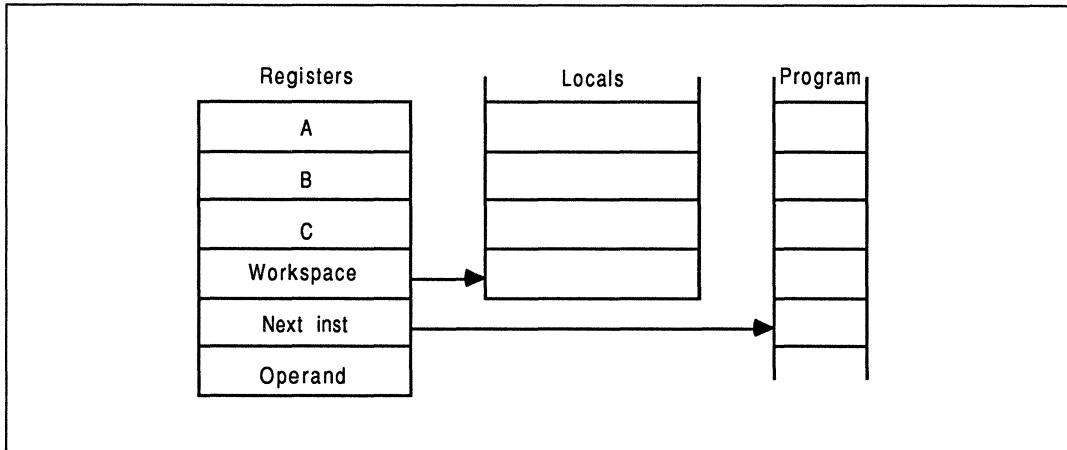
The six registers are:

The workspace pointer which points to an area of store where local variables are kept.

The instruction pointer which points to the next instruction to be executed.

The operand register which is used in the formation of instruction operands.

The A, B and C registers which form an evaluation stack, and are the sources and destinations for most arithmetic and logical operations. Loading a value into the stack pushes B into C, and A into B, before loading A. Storing a value from A, pops B into A and C into B.

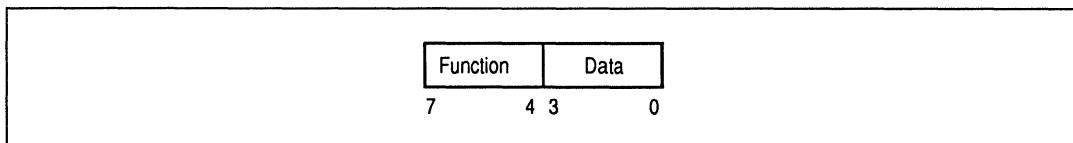


Expressions are evaluated on the evaluation stack, and instructions refer to the stack implicitly. For example, the 'add' instruction adds the top two values in the stack and places the result on the top of the stack. The use of a stack removes the need for instructions to respecify the location of their operands. Statistics gathered from a large number of programs show that three registers provide an effective balance between code compactness and implementation complexity.

No hardware mechanism is provided to detect that more than three values have been loaded on to the stack. It is easy for the compiler to ensure that this never happens.

3.4.2 Instructions

It was a design decision that the transputer should be programmed in a high-level language. The instruction set has, therefore, been designed for simple and efficient compilation. It contains a relatively small number of instructions, all with the same format, chosen to give a compact representation of the operations most frequently occurring in programs. The instruction set is independent of the processor wordlength, allowing the same microcode to be used for transputers with different wordlengths. Each instruction consists of a single byte divided into two 4-bit parts. The four most significant bits of the byte are a function code, and the four least significant bits are a data value.



Direct functions

The representation provides for sixteen functions, each with a data value ranging from 0 to 15. Thirteen of these are used to encode the most important functions performed by any computer. These include:

load constant
add constant

load local
store local
load local pointer

load non-local
store non-local

jump
conditional jump

call

The most common operations in a program are the loading of small literal values, and the loading and storing of one of a small number of variables. The 'load constant' instruction enables values between 0 and 15 to be loaded with a single byte instruction. The 'load local' and 'store local' instructions access locations in memory relative to the workspace pointer. The first sixteen locations can be accessed using a single byte instruction.

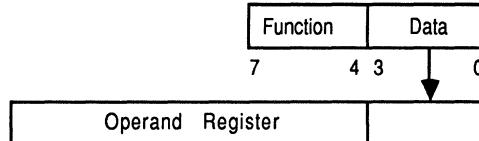
The 'load non-local' and 'store non-local' instructions behave similarly, except that they access locations in memory relative to the A register. Compact sequences of these instructions allow efficient access to data structures, and provide for simple implementations of the static links or displays used in the implementation of block-structured programming languages such as OCCAM.

Prefix functions

Two more of the function codes are used to allow the operand of any instruction to be extended in length. These are:

- prefix
- negative prefix

All instructions are executed by loading the four data bits into the least significant four bits of the operand register, which is then used as the instruction's operand. All instructions except the prefix instructions end by clearing the operand register, ready for the next instruction.



The 'prefix' instruction loads its four data bits into the operand register, and then shifts the operand register up four places. The 'negative prefix' instruction is similar, except that it complements the operand register before shifting it up. Consequently operands can be extended to any length up to the length of the operand register by a sequence of prefix instructions. In particular, operands in the range -256 to 255 can be represented using one prefix instruction.

The use of prefix instructions has certain beneficial consequences. Firstly, they are decoded and executed in the same way as every other instruction, which simplifies and speeds instruction decoding. Secondly, they simplify language compilation, by providing a completely uniform way of allowing any instruction to take an operand of any size. Thirdly, they allow operands to be represented in a form independent of the processor wordlength.

Indirect functions

The remaining function code, 'operate', causes its operand to be interpreted as an operation on the values held in the evaluation stack. This allows up to sixteen such operations to be encoded in a single byte instruction. However, the prefix instructions can be used to extend the operand of an 'operate' instruction just like any other. The instruction representation therefore provides for an indefinite number of operations.

The encoding of the indirect functions is chosen so that the most frequently occurring operations are represented without the use of a prefix instruction. These include arithmetic, logical and comparison operations such as

- add
- exclusive or
- greater than

Less frequently occurring operations have encodings which require a single prefix operation (the transputer instruction set is not large enough to require more than 512 operations to be encoded!).

3.4.3 Expression evaluation

Evaluation of expressions may require the use of temporary variables in the process workspace, but the number of these can be minimised by careful choice of the evaluation order.

Let $\text{depth}(e)$ be the number of stack locations needed for the evaluation of expression e , defined by:

```

 $\text{depth}(\text{constant}) = 1$ 
 $\text{depth}(\text{variable}) = 1$ 
 $\text{depth}(e_1 \text{ op } e_2) = \begin{cases} \text{IF } \text{depth}(e_1) > \text{depth}(e_2) \text{ THEN} \\ \quad \text{depth}(e_1) \\ \text{ELSE IF } \text{depth}(e_1) < \text{depth}(e_2) \text{ THEN} \\ \quad \text{depth}(e_2) \\ \text{ELSE } \text{depth}(e_1) + 1 \end{cases}$ 
```

Let $\text{commutes}(\text{operator})$ be *true* if the operator commutes, *false* otherwise.

Let e_1 and e_2 be expressions. The expression of $(e_1 \text{ op } e_2)$ is compiled for the three-register stack by:

```

 $\text{compile}(e_1 \text{ op } e_2) = \begin{cases} \text{IF } \text{depth}(e_2) > \text{depth}(e_1) \\ \quad \text{THEN} \\ \quad \text{IF } \text{depth}(e_1) > 2 \\ \quad \quad \text{THEN } (\text{compile}(e_2); \text{store temp}; \text{compile}(e_1); \text{load temp}; \text{op}) \\ \quad \quad \text{ELSE IF } \text{commutes}(\text{op}) \\ \quad \quad \quad \text{THEN } (\text{compile}(e_2); \text{compile}(e_1); \text{op}) \\ \quad \quad \quad \text{ELSE } (\text{compile}(e_2); \text{compile}(e_1); \text{reverse}; \text{op}) \\ \quad \text{ELSE} \\ \quad \quad \text{IF } \text{depth}(e_2) < 3 \\ \quad \quad \quad \text{THEN } (\text{compile}(e_1); \text{compile}(e_2); \text{op}) \\ \quad \quad \quad \text{ELSE } (\text{compile}(e_2); \text{store temp}; \text{compile}(e_1); \text{load temp}; \text{op}) \end{cases}$ 
```

where $(I_1; I_2; \dots; I_n)$ represents a sequence of instructions.

Efficiency of encoding

Measurements show that about 80% of executed instructions are encoded in a single byte (i.e. without the use of prefix instructions). Many of these instructions, such as 'load constant' and 'add' require just one processor cycle.

The instruction representation gives a more compact representation of high-level language programs than more conventional instruction sets. Since a program requires less store to represent it, less of the memory bandwidth is taken up with fetching instructions. Furthermore, as memory is word accessed the processor will receive several instructions for every fetch.

Short instructions also improve the effectiveness of instruction prefetch, which in turn improves processor performance. There is an extra word of prefetch buffer so that the processor rarely has to wait for an instruction fetch before proceeding. Since the buffer is short, there is little time penalty when a jump instruction causes the buffer contents to be discarded.

3.4.4 Support for concurrency

The processor provides efficient support for the occam model of concurrency and communication. It has a microcoded scheduler which enables any number of concurrent processes to be executed together, sharing the processor time. This removes the need for a software kernel. The processor does not need to support the dynamic allocation of storage as the occam compiler is able to perform the allocation of space to concurrent processes.

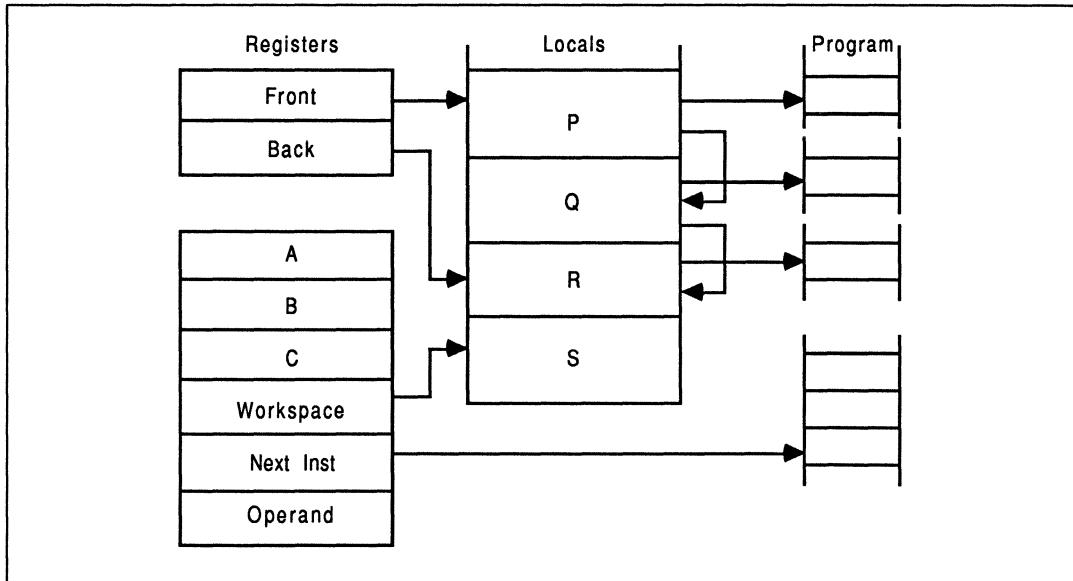
At any time, a concurrent process may be:

- | | |
|----------|---|
| active | <ul style="list-style-type: none"> - being executed - on a list waiting to be executed |
| inactive | <ul style="list-style-type: none"> - ready to input - ready to output - waiting until a specified time |

The scheduler operates in such a way that inactive processes do not consume any processor time.

The active processes waiting to be executed are held on a list. This is a linked list of process workspaces, implemented using two registers, one of which points to the first process on the list, the other to the last.

In this illustration, S is executing, and P, Q and R are active, awaiting execution:



A process is executed until it is unable to proceed because it is waiting to input or output, or waiting for the timer. Whenever a process is unable to proceed, its instruction pointer is saved in its workspace and the next process is taken from the list. Actual process switch times are very small as little state needs to be saved; it is not necessary to save the evaluation stack on rescheduling.

The processor provides a number of special operations to support the process model. These include:

```
start process
end process
```

When a parallel construct is executed, 'start process' instructions are used to create the necessary concurrent processes. A 'start process' instruction creates a new process by adding a new workspace to the end of the scheduling list, enabling the new concurrent process to be executed together with the ones already being executed.

The correct termination of a parallel construct is assured by use of the 'end process' instruction. This uses a workspace location as a counter of the components of the parallel construct which have still to terminate. The counter is initialised to the number of components before the processes are 'started'. Each component ends with an 'end process' instruction which decrements and tests the counter. For all but the last component, the counter is non-zero and the component is descheduled. For the last component, the counter is zero and the

component continues.

Communications

Communication between processes is achieved by means of channels. occam communication is point-to-point, synchronised and unbuffered. As a result, a channel needs no process queue, no message queue and no message buffer.

A channel between two processes executing on the same transputer is implemented by a single word in memory; a channel between processes executing on different transputers is implemented by point-to-point links. The processor provides a number of operations to support message passing, the most important being:

input message
output message

The 'input message' and 'output message' instructions use the address of the channel to determine whether the channel is internal or external. This means that the same instruction sequence can be used for both hard and soft channels, allowing a process to be written and compiled without knowledge of where its channels are connected.

As in the occam model, communication takes place when both the inputting and outputting processes are ready. Consequently, the process which first becomes ready must wait until the second one is also ready.

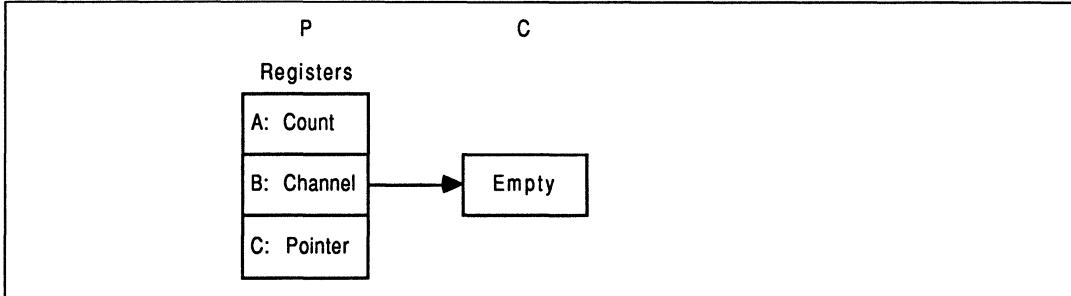
A process performs an input or output by loading the evaluation stack with a pointer to a message, the address of a channel, and a count of the number of bytes to be transferred, and then executing an 'input message' or an 'output message' instruction.

Internal channel communication

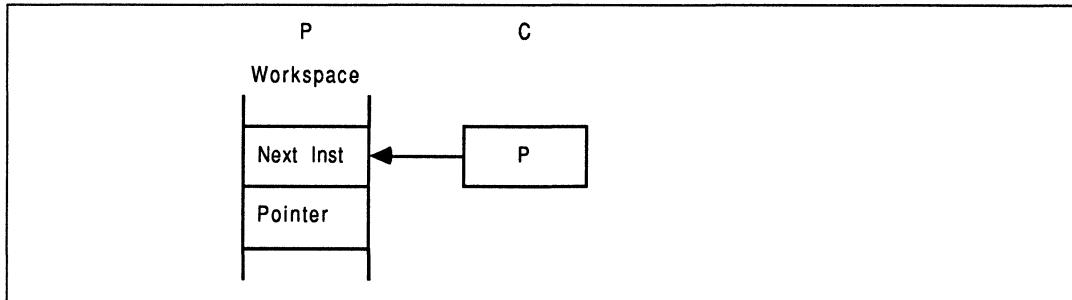
At any time, an internal channel (a single word in memory) either holds the identity of a process, or holds the special value 'empty'. The channel is initialised to 'empty' before it is used.

When a message is passed using the channel, the identity of the first process to become ready is stored in the channel, and the processor starts to execute the next process from the scheduling list. When the second process to use the channel becomes ready, the message is copied, the waiting process is added to the scheduling list, and the channel reset to its initial state. It does not matter whether the inputting or the outputting process becomes ready first.

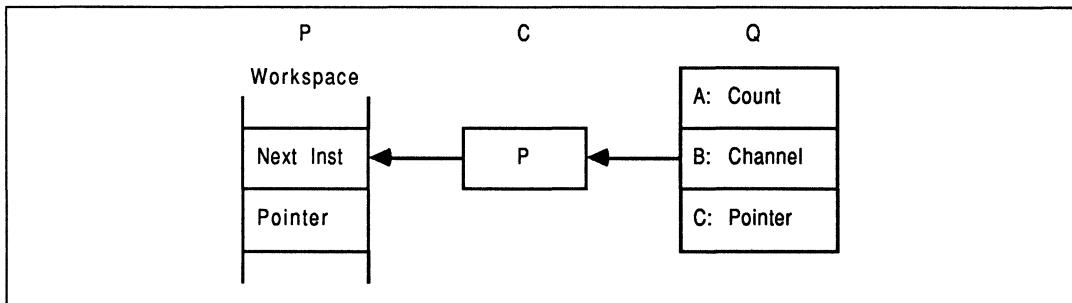
In the following illustration, a process P is about to execute an output instruction on an 'empty' channel C. The evaluation stack holds a pointer to a message, the address of channel C, and a count of the number of bytes in the message.



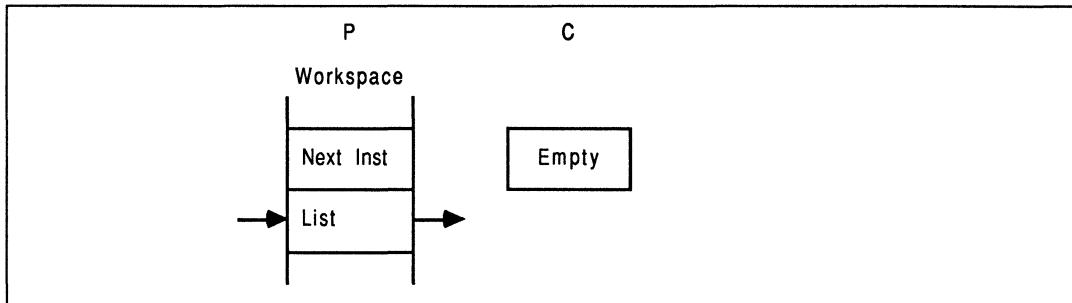
After executing the output instruction, the channel C holds the address of the workspace of P, and the address of the message to be transferred is stored in the workspace of P. P is descheduled, and the process starts to execute the next process from the scheduling list.



The channel C and the process P remain in this state until a second process, Q executes an output instruction on the channel.



The message is copied, the waiting process P is added to the scheduling list, and the channel C is reset to its initial 'empty' state.



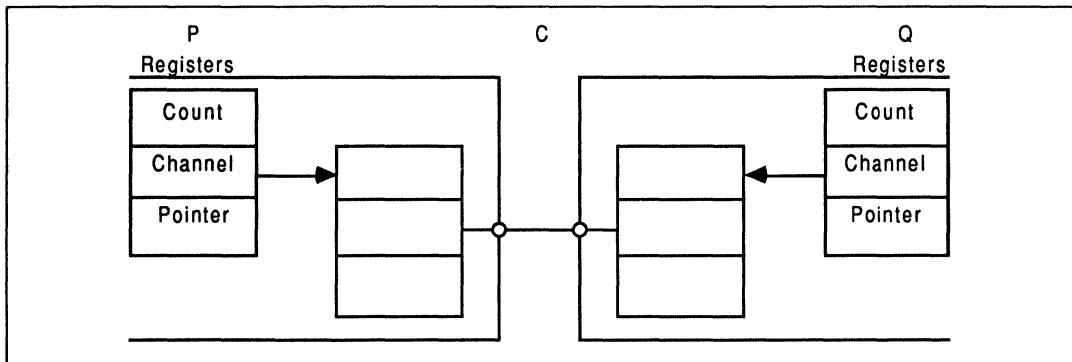
External channel communication

When a message is passed via an external channel the processor delegates to an autonomous link interface the job of transferring the message and deschedules the process. When the message has been transferred the link interface causes the processor to reschedule the waiting process. This allows the processor to continue the execution of other processes whilst the external message transfer is taking place.

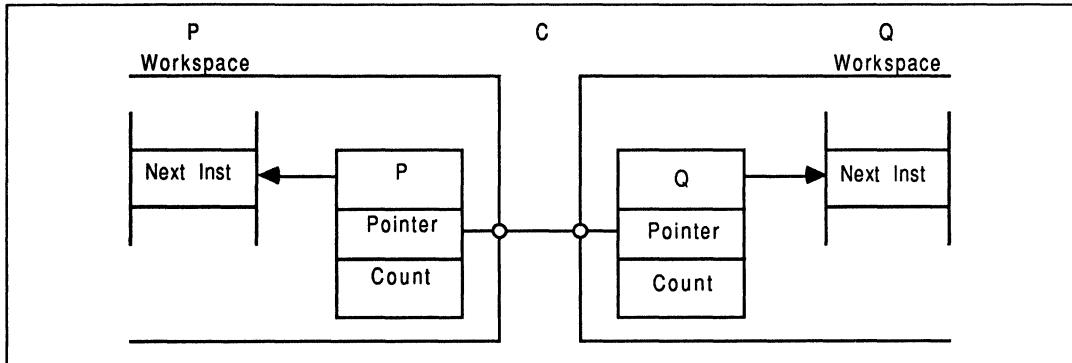
Each link interface uses three registers:

- a pointer to a process workspace
- a pointer to a message
- a count of bytes in the message

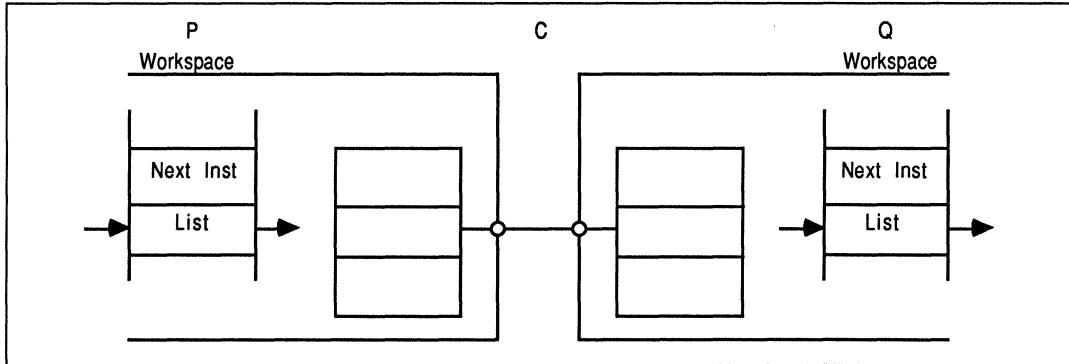
In the following illustration, processes P and Q executed by different transputers communicate using a channel C implemented by a link connecting two transputers: P outputs, and Q inputs.



When P executes its output instruction, the registers in the link interface of the transputer executing P are initialised, and P is descheduled. Similarly, when Q executes its input instruction, the registers in the link interface of the process executing Q are initialised, and Q is descheduled.



The message is now copied through the link, after which the workspaces of P and Q are returned to the corresponding scheduling lists. The protocol used on P and Q ensures that it does not matter which of P and Q first becomes ready.



Timer

The transputer has a clock which ‘ticks’ every microsecond. The current value of the processor clock can be read by executing a ‘Read timer’ instruction.

A process can arrange to perform a ‘timer input’, in which case it will become ready to execute after a specified time has been reached.

The timer input instruction requires a time to be specified. If this time is in the ‘past’ (i.e. *ClockReg AFTER SpecifiedTime*) then the instruction has no effect. If the time is in the ‘future’ (i.e. *SpecifiedTime AFTER Clockreg* or *SpecifiedTime = ClockReg*) then the process is descheduled. When the specified time is reached the process is scheduled again.

Alternative

The OCCAM alternative construct enables a process to wait for input from any one of a number of channels, or until a specific time occurs. This requires special instructions, as the normal ‘input’ instruction deschedules a process until a specific channel becomes ready, or until a specific time is reached. The instructions used are:

```
enable channel    enable timer
disable channel   disable timer
alternative wait
```

The alternative is implemented by ‘enabling’ the channel input or timer input specified in each of its components. The ‘alternative wait’ is then used to deschedule the process if none of the channel or timer inputs is ready; the process will be rescheduled when any one of them becomes ready. The channel and timer inputs are then ‘disabled’. The ‘disable’ instructions are also designed to select the component of the alternative to be executed; the first component found to be ready is executed.

3.4.5 Inter-transputer links

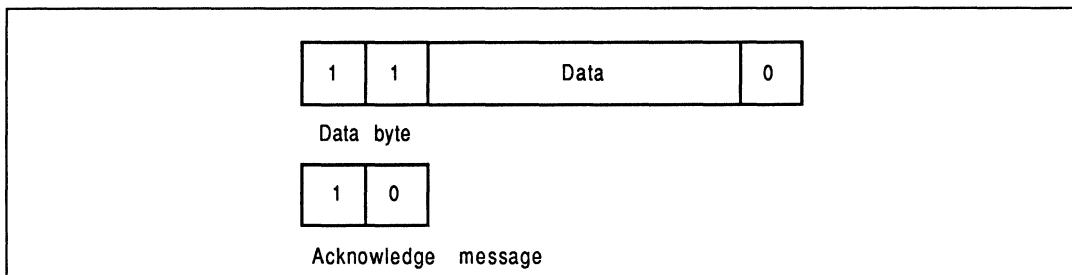
To provide synchronised communication, each message must be acknowledged. Consequently, a link requires at least one signal wire in each direction.

A link between two transputers is implemented by connecting a link interface on one transputer to a link interface on the other transputer by two one-directional signal lines, along which data is transmitted serially.

The two signal wires of the link can be used to provide two OCCAM channels, one in each direction. This requires a simple protocol. Each signal line carries data and control information.

The link protocol provides the synchronised communication of OCCAM. The use of a protocol providing for the transmission of an arbitrary sequence of bytes allows transputers of different wordlength to be connected.

Each message is transmitted as a sequence of single byte communications, requiring only the presence of a single byte buffer in the receiving transputer to ensure that no information is lost. Each byte is transmitted as a start bit followed by a one bit followed by the eight data bits followed by a stop bit. After transmitting a data byte, the sender waits until an acknowledge is received; this consists of a start bit followed by a zero bit. The acknowledge signifies both that a process was able to receive the acknowledged byte, and that the receiving link is able to receive another byte. The sending link reschedules the sending process only after the acknowledge for the final byte of the message has been received.



Data bytes and acknowledges are multiplexed down each signal line. An acknowledge is transmitted as soon as reception of a data byte starts (if there is room to buffer another one). Consequently transmission may be continuous, with no delays between data bytes.

3.5 Summary

Experience with OCCAM has shown that many applications naturally decompose into a large number of fairly simple processes. Once an application has been described in OCCAM, a variety of implementations are possible. In particular, the use of OCCAM together with the transputer enables the designer to exploit the performance and economics of VLSI technology. The concurrent processing features of OCCAM can be efficiently implemented by a small, simple and fast processor.

The transputer therefore has two important uses. Firstly it provides a new system 'building block' which enables OCCAM to be used as a design formalism. In this role, OCCAM serves both as a system description language and a programming language. Secondly, OCCAM and the transputer can be used for prototyping highly concurrent systems in which the individual processes are ultimately intended to be implemented by dedicated hardware.

3.6 References

- 1 *occam Programming Manual*, Prentice-Hall International 1984.
- 2 *The Transputer Databook*, INMOS Ltd 1989.

4 Communicating process computers

4.1 Introduction

This paper is concerned with the construction of computers based on communicating process architecture. We wish to establish that this architecture is practical and that it is feasible to build a general-purpose computer based on this architecture. We shall start by looking briefly at the technological background and the questions that this raises, then look at a number of real applications, and finally we will discuss the possible structure of a general-purpose parallel computer.

At the present level of VLSI technology we can implement in the same area of silicon the following components of a computer:

- a 10 MIPS processor
- 2 Kbytes of memory
- a 10 Mbyte/second communications system

Consequently, using the same silicon area, we can construct a single 10 MIPS processor with 4 Mbytes of memory (a conventional sequential computer) or a 10 000 MIPS computer with 2 Mbytes of memory. Both machines would require about 1000 VLSI devices, and so are quite small computers.

The problems are now to decide on the correct ratio of memory to processors and how to construct a system with many processing elements with small amounts of memory dispersed through the system, in such a way that it can be applied to practical problems. Obviously, a collection of 1000 or more processing elements must be arranged in a regular structure, and a number of different structures have been proposed. Examples are:

- pipeline
- array (1D, 2D, 3D ...)
- hypercube
- toroidal surface
- shuffle

These structures vary in three important respects:

- ability to extend
- ability to implement on silicon (in two dimensions)
- cost of non-local communication

We will return to these matters when we consider the implementation of applications on general-purpose communicating process computers. But first we will look at some applications.

4.2 Applications with special configurations

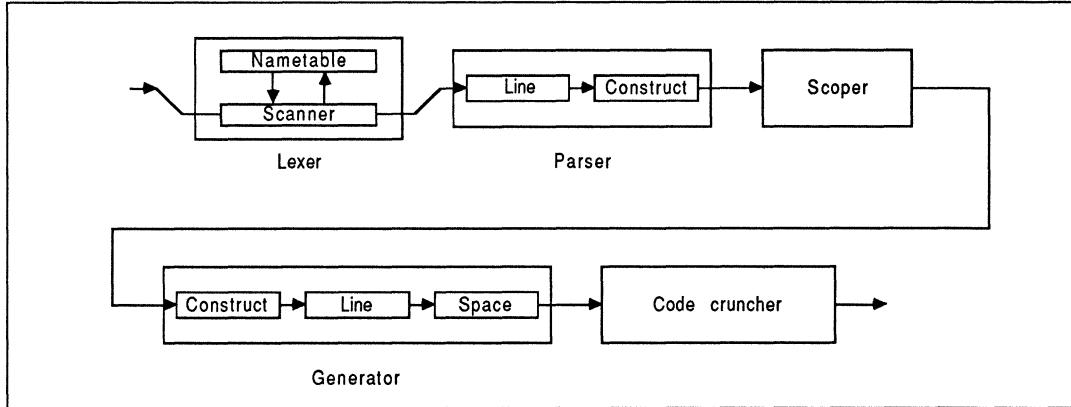
We now look at a number of applications where the concurrent implementation seems to dictate a specific processor structure. All these applications have been developed and actually run on multi-transputer systems. The examples are divided into three groups. The first group contains applications where the parallelism has been obtained by decomposing the algorithm into a number of smaller, simpler components which can be executed in parallel. The second group contains applications where the parallelism has been obtained by distributing the data to be processed between a number of processors in such a way that the geometrical structure of the data is preserved. The final group contains applications where a number of processors are used to process data farmed out by a controlling processor. Of course, these groups are not mutually exclusive, and our solid modelling application shows aspects of both algorithmic and geometric decomposition.

4.2.1 Algorithmic parallelism or dataflow decomposition

In the following two examples the algorithm used follows from a dataflow analysis of the application and the parallelism arises directly from that algorithm.

Example: occam compiler

The first example is the occam-in-occam compiler. One of the reasons for the choice of this example is to illustrate that concurrency can arise where it might not be expected. In order to write this compiler concurrently (deliberate ambiguity!) a dataflow approach was taken; the parallel decomposition of the algorithm then follows straightforwardly. The diagram below shows the structure of the compiler.



From the outside, the compiler appears to be a single-pass compiler. Internally, it is more like a multiple-pass compiler; each process performs a simple transformation on the data which flows through it. For example, the lexer process inputs a sequence of characters and outputs a sequence of tokenised lexemes. It is able to do this continuously; as soon as it has recognised a sequence of characters as a lexeme it is able to output the appropriate token.

The effect of decomposing the compiler in this way was that each component process was relatively easy to write, specify and test; this meant that the component processes could be written concurrently! Also, as the machine dependencies were restricted to the final stages of the compiler, it was possible to develop the compiler for different targets concurrently.

The occam program for the compiler is outlined below:

```
-- occam compiler
CHAN lexed.program:
CHAN parsed.program:
CHAN scoped.program:
CHAN coded.program:
PAR
-- lexer
CHAN name.text:
CHAN name.code:
PAR
... scanner
... nametable

-- parser
CHAN parsed.lines :
PAR
... line parser
... construct parser

... scoper

-- generator
CHAN generated.constructs :
CHAN generated.program :
PAR
... construct generator
... line generator
... space allocator

... code cruncher
```

The program, as shown, could be executed on a pipeline of processors. However, it is unlikely that it will offer an increase in speed which is proportional the number of processors used.

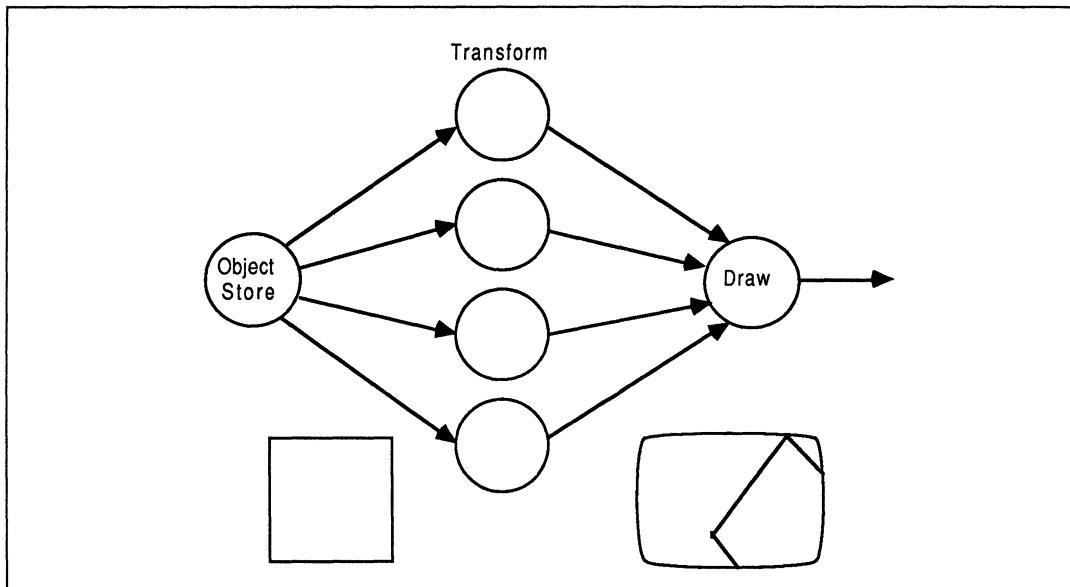
There are two important reasons for this. The first is that the throughput of a pipeline is limited by the throughput of the slowest element of the pipeline. This means that in order to have the potential for maximum multi-processor speed-up a pipeline must be 'balanced'; that is each component of the pipeline must process data at the same rate. The compiler pipeline is not balanced; measurements show that the code cruncher accounts for about 40% of the processing resource used. The second reason is that the pipeline does not contain sufficient buffering to allow each individual stage to operate as fast as possible. For example, the line parser operates on a line of lexemes at a time, whereas the lexer operates on only a lexeme at a time. This means that without a buffer inserted between the lexer and the line parser, the lexer will halt whilst the line parser transforms a line.

Example: solid modelling

Another example of an application for which the algorithm decomposes easily is solid modelling. This involves the generation of shaded images of polygonal objects in real time. This has application in the areas of computer-aided design and computer animation.

For each object the following steps are performed. First the object is translated into the 'world space' (the world space defines the spacial relationships between the objects to be modelled). The object is then transformed into the 'image space', this involves rotating and projecting the object so that it will appear in proper perspective as seen by an observer at the chosen 'viewpoint'. The image of each object must be 'clipped' to the screen and then 'drawn' into a Z-buffer which is used to resolve depth. The algorithm can be extended to provide animation by allowing the objects, the world, and the viewpoint to change for each frame.

At the top level we choose to implement the algorithm as shown in the next diagram.



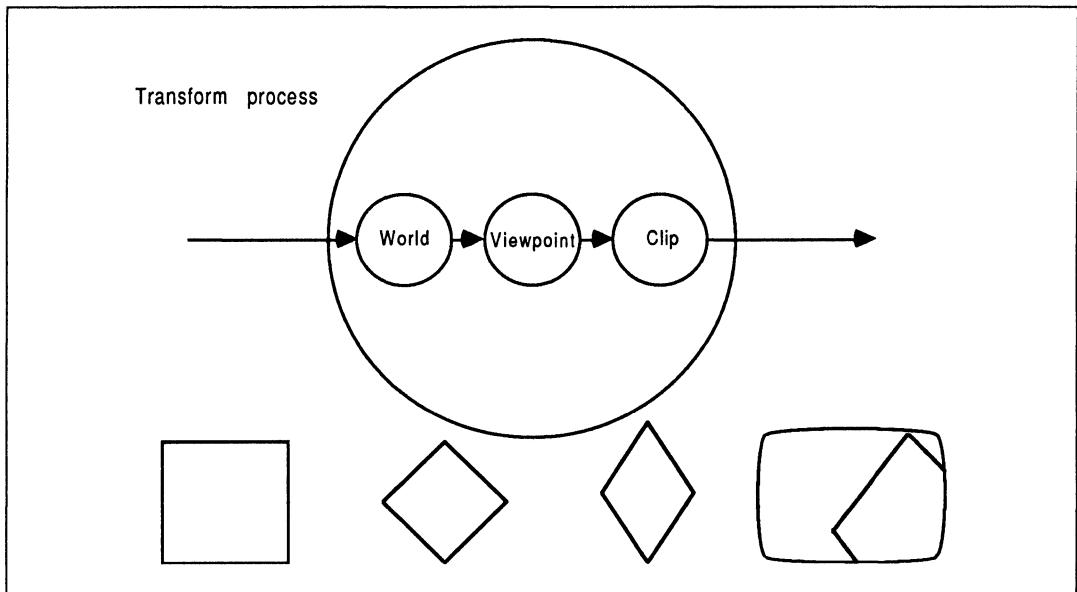
Here each object is passed to a transformer which passes the transformed object to the drawing process. We use several transformers to increase the rate at which we can draw objects. As a transformer becomes free, the object store can send it another object to transform. In this way we obtain a linear multiprocessor speed-up; n -transformers can process data at n -times the rate that one transformer can. This speed-up is predicated on the object store being able to supply objects at a great enough rate and on the drawing process being able to draw objects fast enough.

We can now turn to the implementation of the transformation process. The sequential implementation of this process could be written as:

```

WHILE active
  SEQ
    from.object.store ? object
    world(object)
    viewpoint(object)
    clip(object)
    to.drawing.process ! object
  
```

This process can be distributed over a pipeline:



the program becoming:

```

CHAN world.to.viewpoint :
CHAN viewpoint.to.clip :
PAR
... world
... viewpoint
... clip

```

The parallel processes would all have the same general form; for example, the viewpoint process would be:

```

WHILE active
SEQ
  world.to.viewpoint ? object
  viewpoint(object)
  viewpoint.to.clip ! object

```

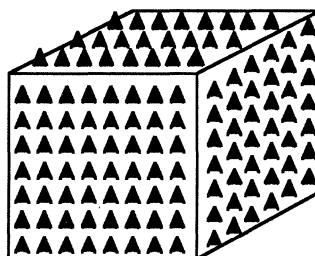
In practice these processes would be probably be more complex than the program above suggests, we would want to introduce buffering so that the whole of the transformation pipeline could be kept busy.

4.2.2 Geometric parallelism or data structure decomposition

In the example below use is made of the geometric structure of the data to distribute the application on to a number of processors.

Statistical mechanics

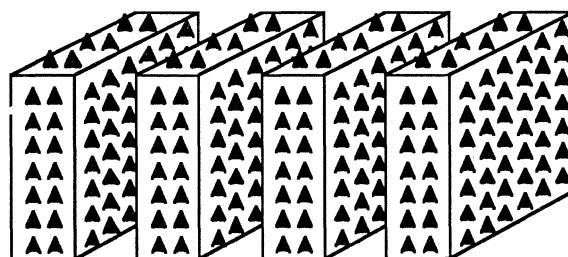
Statistical mechanics is the study of mechanical systems where the behaviour of the components is described statistically and cannot be resolved analytically. A familiar example of a statistical mechanical system is provided by the magnetic properties of iron. For this purpose iron can be modelled as a cubic lattice of small magnets.



The orientation of these magnets is known as a spin because the magnetism is related to the spins of the electrons in the iron. This model is thus called a three-dimensional spin system.

We can simulate the behaviour of iron on heating by examining what happens to the lattices over successive time steps as it is heated. During each time step there are two important influences on each small magnet. Firstly, there are thermal vibrations which will tend to move the magnet away from its current orientation. The thermal effects are described statistically, with the distribution being dependent on the temperature of the iron. The second influence will be the magnetic forces applied by the neighbours of the magnet under consideration. If we start with a magnetised lattice and raise the temperature the thermal effects will eventually overcome the magnetic forces and the lattice will become disordered and thus demagnetised.

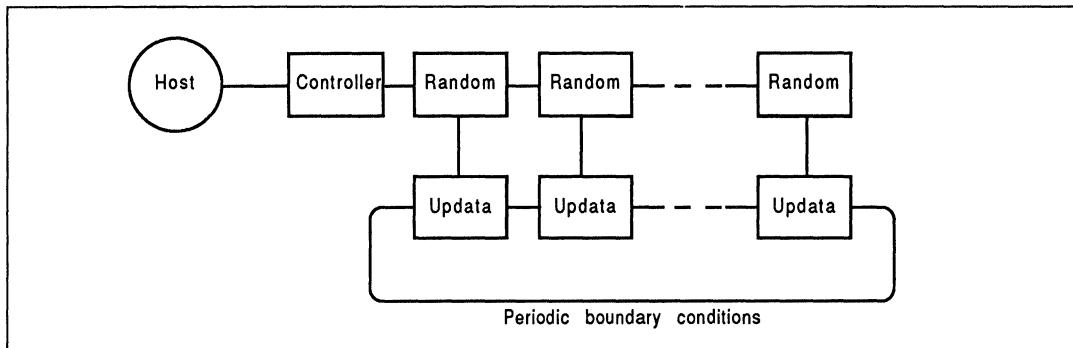
It is easy to see that a statistical mechanical system can be decomposed in terms of its natural geometrical structure. For example, the cubic lattice of iron could be split between a number of transputers, each dealing with a small portion of the problem. Each transputer can then update that part of the lattice for which it contains the data. Communication will be needed with neighbouring transputers so as to exchange information about adjacent lattice sites which are placed on different transputers.



We will now look at a practical example of a statistical mechanical simulation. This is a simulation of a generalised planar spin model (i.e. a 2-D spin system) with both 'Exchange' and 'Nematic' interactions [1] which has actually been implemented on a number of transputers. The system can be interpreted in terms of liquid crystal films; however, the major interest in the system is theoretical in that it exhibits an unusual phase structure.

The program operates on an $L \times L$ square lattice of spins with periodic boundary conditions. The spins are represented by angles which are discretised to lower the storage requirements and to allow a table look-up for fast cosine generation.

The original aim of the design was for the system to be implemented on an array of transputers without any external memory. This imposed a large constraint because the straightforward geometric decomposition of the updating process gave rise to a collection of processes each of which was too large to reside in the memory of a single transputer. The solution was to split the updating work into two parallel processes, 'random' and 'updata' each of which could fit on a transputer.



The random process generates uniformly distributed and exponentially distributed random numbers and communicates with the controller process. The updata process performs the rest of the updating algorithm, stores data (512 spins) and computes correlations. Each random/updata pair of processes implements a vertical 'strip' of the lattice. Horizontal communication is required for the interaction of the spins on the vertical edges of the strips.

In practice the 'no-external memory' requirement was relaxed. The program was run at INMOS on seventeen transputer evaluation boards (each board having an 80 nS cycle time transputer with external memory). The extra memory permitted the random and updata processes to be implemented on a single transputer, the lattice to be decomposed into sixteen 4×64 strips, and the discretisation of the spins to be increased to 128 states as the size of the cosine table could be enlarged.

The efficiency of the simulation was:

$$\frac{\text{time of program on 1 processor}}{17 \times \text{time for program on 17 processors}} \approx 80\%$$

The simulation, which took about 60 hours to run, would have taken about 3 months on a VAX 11/780.

4.2.3 Farming out processing

Example: Graphical representation of the Mandelbrot Set

The Mandelbrot set, M , is the set of complex numbers:

$$M = \{c : |M_n(c)| < \infty \quad \forall n \in \mathbb{N}\}$$

where:

$$\begin{aligned} M_0(c) &= 0 \\ M_{n+1}(c) &= M_n(c)^2 + c \end{aligned}$$

It can be shown that if $\exists n : |M_n(c)| > 2$ then $c \notin M$.

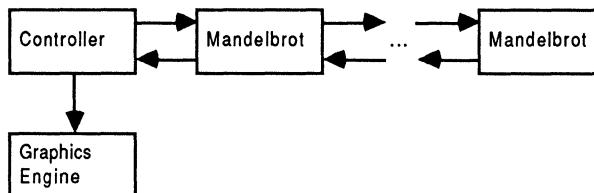
The edges of the Mandelbrot set are intricate, and, because complex numbers can be represented on a two-dimensional plane, the set can be plotted on a graphics screen with impressive results. In practice, the colour of each pixel on the screen represents whether or not the corresponding point of the complex plane is in the Mandelbrot set. If a point is not in the Mandelbrot set then the colour plotted at that point represents the number of applications of the recurrence required to determine that it is not in the set. A point will be considered to be in the set if the recurrence has been applied more than a fixed number of times (for example 1000) without the modulus becoming greater than 2.

For a given point (x, y) the following process applies the recurrence until a colour can be chosen:

```

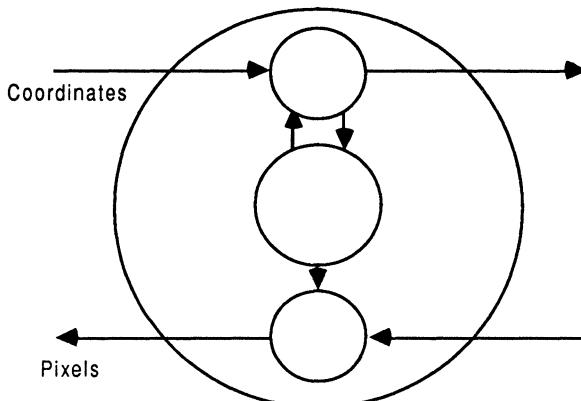
iterations := 0
z := COMPLEX(0.0, 0.0)
WHILE (iteration < 1000) AND ((MOD z) < 2)
  SEQ
    z := (z*z) + COMPLEX(x, y)
    iteration := iteration + 1
  
```

To plot a picture of the Mandelbrot set requires that we perform the above process for every pixel on the screen. However, as the computation for each pixel is independent we may perform it for many pixels in parallel. The implementation we have chosen is shown in the diagram below:



The basic idea used in this implementation is that the controller process hands out a point to each Mandelbrot process. When a Mandelbrot process has computed the colour to be displayed at that pixel it sends the information to the controller which passes the pixel to the graphics engine and hands the Mandelbrot process another pixel. This approach is very attractive because the amount of computation required varies from pixel to pixel and this implementation automatically balances the load.

As can be seen from the previous diagram, Mandelbrot processes not only compute the colour for a pixel but they also provide a means for the controller to communicate with Mandelbrot processes to which it is not directly connected. The structure of the Mandelbrot process is as shown below:



This implementation turns out to be quite effective. If there are N processors available to execute Mandelbrot processes then an upper bound on the amount of communication required for each pixel will be $10 \times N$ bytes. This is not a large amount considering that the computation for each pixel may require up to 2000 operations on floating-point complex numbers.

It turns out that in order to keep the processors busy the Mandelbrot process has to buffer an extra item of

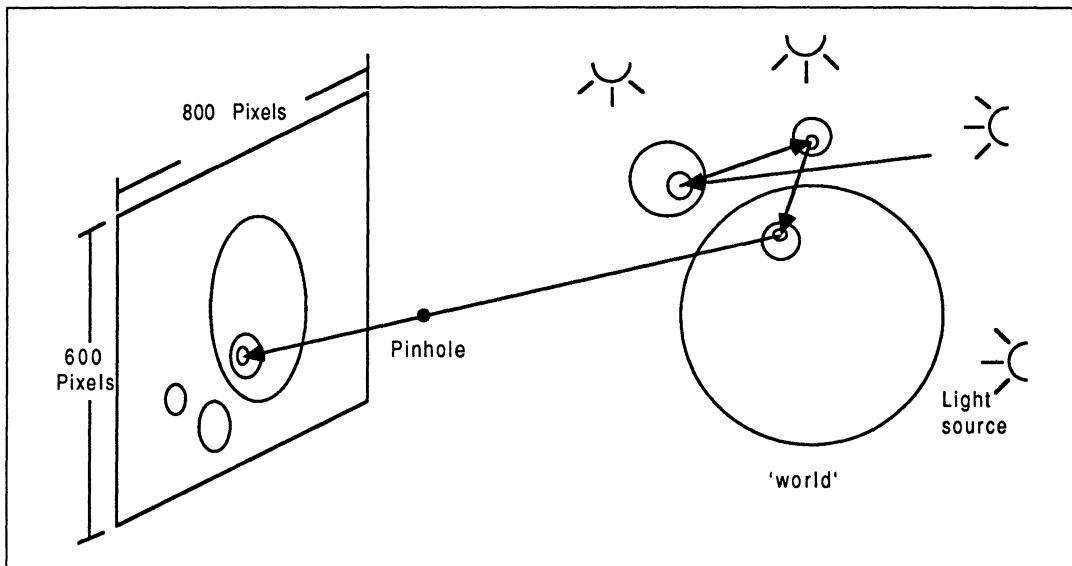
work so that when it completes the computation for a pixel it can start on its next pixel at once rather than having to wait for the controller to send it the next item of work. In the diagram above the extra work is buffered in the router, and when the Mandelbrot computer process finishes its computation it sends a message to the router requesting more work.

The algorithm sketched above can be improved upon so as to decrease the number of interactions between the controller and the Mandelbrot processes by handing each Mandelbrot process more than a single pixel as its item of work. In practice we have chosen to use a quarter of a scan line as the unit of work. We have found that the communication cost with this approach is insignificant even with tens of processors connected in the manner shown.

Example: ray tracing

Whilst the previous example of a computer graphics application may seem a little artificial and especially suited to parallel implementation this example is very real. A large amount of super-computer time is spent on this application by people such as film-makers.

The application is 'ray tracing'. This is a means of producing very high-quality, life-like computer graphics. It is capable of correctly representing reflective and refractive objects (mirrors and lenses) and light sources.



The way in which the technique works is to take a point on the screen and to produce the ray that would arrive at that point from a pinhole sitting between the screen and the objects to be drawn. The ray is then extended into the object space and intersected with each object in the space. The first object with which the ray intersects is determined. In the simplest case this object will be matt and the colour of the object is plotted at that point on the screen. If the object is reflective, the path that the ray would take after reflection is computed and the process repeated. The same general principle allows the pinhole to be replaced by a lens, giving depth-of-field effects.

It may now be seen that the basic structure of the problem is essentially the same as plotting the Mandelbrot set. For each point on the screen a colour has to be generated. The computation for each point is independent and computationally intensive. Ray tracing, can, therefore, be implemented on exactly the processor structure as was used for drawing the Mandelbrot set.

Some comments about the previous two examples

It is quite interesting that the previous two examples are implementable on exactly the same configuration of processors. It is also interesting that these configurations actually seem to have nothing to do with the application in hand.

In fact further consideration of both these algorithms will show that almost any configuration of processors will do, subject to it providing sufficient communication capability. Both these applications have two distinct parts; the first farms out work to a of number application-specific processes, the second is the application-specific process. We call this type of arrangement a processor farm.

4.3 General-purpose structures

From the last two examples we have seen that there are applications which are basically insensitive to the arrangement of processors on which they are run. Of course, there is the proviso that the arrangement of processors must provide sufficient communication capability. As we now have evidence that it might be reasonable to try and construct a general-purpose structure of processors we can return the issues raised in the introduction.

Pipelines and simple (two-dimensional) arrays can be easily implemented or extended. Arrays (and hypercubes) become progressively more difficult to implement as the dimension increases, with much space taken by connections which need to cross over. However, 1000 or so processing elements can be connected in this way.

One difficulty with the hypercube structure is that the number of links provided at each node must be at least the dimension of the hypercube. This means that a standard component (which has a fixed number of links) cannot be used to implement an arbitrarily extensible array. An alternative structure which avoids this problem is obtained by implementing each node of the hypercube with a ring of transputers – this structure is known as 'cube-connected cycles'.

The cost of non-local communication, which arises when two nodes need to communicate via intermediate nodes, varies widely. A one-dimensional array is obviously the worst. It is clearly desirable that the worst case path between two points (the 'diameter') of the network is small in relation to the number of nodes, and several structures have this property:

structure	diameter	size
hypercube	$\sim n - 1$	2^n
cube-connected cycle	$\sim (n \times 5)/2$	$n \times 2^n$
folded tree	$\sim n$	$n \times 2^{n-1}$

If such a structure is being used, for example to implement a processor farm, it may be necessary to implement a routing algorithm. It is quite easy to design a general-purpose algorithm for this purpose but for many applications an application-specific router may be better.

4.3.1 Routing and the communication/computation trade-off

An example of a routing process is shown below:

```

... declarations
SEQ
... initialisation
WHILE active
SEQ
ALT
ALT 1 = 0 FOR 4
link.in[1] ? message
SKIP
internal.in ? message
SKIP
dest := route.table[message[0]]
IF
dest = internal
internal.out ! message
TRUE
link.out[dest] ! message
... check for termination

```

The above routing process inputs a message from a link or from the process coresident on the transputer. The process examines the first word of the message to determine the destination, and looks up that destination in a route table which identifies whether it should be sent to the local process or retransmitted down a link.

More complex versions of the routing process would enable the transputer's links to operate concurrently. However, they would almost certainly impose a larger overhead on the processor's computing power, and thus might be suitable for algorithms where the required communication bandwidth is relatively high.

Normally the routing process in a transputer would be prioritised over other processes. This ensures that when a message arrives at the routing process it is inspected (and forwarded if necessary) immediately it is received. If a high-priority process were not used the message would not be examined until the routing process was executed on the round-robin.

Although a routing process has an impact on the computing power available at each node, once a data transfer has been initiated the transputer's autonomous links will transfer the data without the further intervention of the processor. This means that the processor resource used by a routing process is dependent on the number of communications rather than the quantity of data transmitted in each communication. This in turn suggests that the correct strategy is to maximise the length of message passed at one time.

On the other hand, where the length of time it takes for a message to reach its destination is critical, there are advantages in breaking data into small messages. This enables several processors to transfer the data concurrently. This is also true where it is necessary to broadcast data throughout an array. These matters have been investigated elsewhere in the literature [2].

For a given problem, it is usually possible to adjust the processing time per communication by use of a combination of parallel and sequential algorithms. At one end of the spectrum is the 'dataflow' program with many simple processes each of which inputs a message, performs a single operation and outputs it; at the other end is a sequential program which inputs a message, performs many operations, and outputs the result. One of the advantages of a communicating process language is that it combines both sequential and parallel programming techniques, and one of the uses of program transformations is to perform this kind of optimisation.

It is possible to write programs in a manner whereby the granularity of the computation is easy to adjust. For example, in the Mandelbrot set drawing program it is easy to alter the granularity from a single pixel (large potential for parallelism) to a whole screen (small amount of communication). This is useful because the communication-to-computation ratio can vary as hardware changes. For example, the introduction of the floating-point transputer will drastically reduce the computation load of a transputer which is drawing the Mandelbrot set. As a result of this, the program should be altered to increase the number of pixels computed

at a time.

Experience suggests that many numerical problems can be organised so that communication times are dominated by computing time. For example, a process which inputs two $n \times n$ arrays, and outputs the products involves $3 \times n^2$ communications but the multiplication involves n^3 operations.

4.3.2 Comparison of pipelines and processor farms

Given a general-purpose structure, such as a two-dimensional array, it is obviously possible to use a number of different techniques to implement an application. For example, a number of applications could suit either a pipeline or processor farm implementation.

The question then arises as to which implementation is preferable. There are a number of considerations here:

- 1 The throughput of a pipeline is limited by the throughput of the slowest part of the pipeline. This means that the processing time for an n -stage pipeline is $n \times \max(t(1), \dots, t(n))$ where $t(i)$ is the time taken for stage i of the pipeline, whereas the processing time for the equivalent sequential implementation (as would be used on a farm), is $\sum_{i=1}^n t(i)$, which is smaller. In addition the pipeline implementation will use some processor time passing messages from one stage to the next.
- 2 The amount of code required in each stage of the pipeline will be smaller than the amount of code needed in each processor in a farm. This could be important where memory capacity is limited. The smaller code might also run faster due to better utilisation of the transputer's on-chip memory. However, the code size consideration will only apply to heterogeneous pipelines; the code to implement all stages of a homogeneous pipeline on a single farm processor will be essentially the same size as the code to implement one stage of the pipeline.
- 3 There may be sequential dependencies in the data which would be difficult to deal with using a processor farm. For example, in a compiler, it is necessary to know which procedures have already been compiled in order to enforce scope roles. This would seem to make it difficult to transform the implementation to a farm.

We would like to give one final example of a processor farm implementation. The application we have chosen is producing the sum of all prime numbers less than a specified number. We calculate this by producing all prime numbers less than the specified number and summing them. The prime numbers are produced by successively testing the primality of odd integers. We test the primality of an integer n by dividing by primes up to \sqrt{n} .

This problem is of interest because it contains the sequential dependency that a number n cannot be tested for primality until we have tested all numbers up to \sqrt{n} . We have chosen a very simple solution to this problem for the sake of exposition.

We distribute the problem by having a number of processors running primality testers and a single controller processor. Each primality tester maintains a list of prime numbers, supplied to it by the controller process. It uses this list to determine the primality of candidates passed to it by the controller. The controller ensures that when a primality tester tests a candidate n the tester contains all primes up to \sqrt{n} .

The program for the primality tester is:

```

... initialisation
WHILE active
SEQ
  from.controller ? object.type; object.value
  IF
    object.type = candidate
    VAR candidate.is.prime :
    SEQ
      IF
        IF i = 0 FOR primes.stored
          (object.value \ primes[i]) = 0
          candidate.is.prime := FALSE
        TRUE
        candidate.is.prime := TRUE
      to.controller ! object.value; candidate.is.prime
      object.type = prime
      ... add to list of primes
      object.type = halt
      active := FALSE

```

The program for the controller is:

```

... initialisation
problem ? upper.bound; root.upper.bound
... generate primes until prime > root.upper.bound

WHILE next.candidate < upper.bound
  VAR nactive :
  SEQ
    ... hand out next batch of primes
    ... start primality testers
    nactive := ntesters
  WHILE nactive > 0
    ALT i = [0 FOR number.testers]
    from.prime.test[i] ? resolved.candidate; is.a.prime
    SEQ
      ... add into prime sum if is.a.prime
      IF
        more.candidates
        ... send next candidate
      TRUE
      nactive := nactive - 1
    ... terminate primality testers
    result ! prime.sum

```

The inner **WHILE** loop hands out a new candidate to a tester in response to the tester returning the result of its previous test. The loop terminates when there are no more candidates which can be tested using only the primes currently stored by the testers.

The outer **WHILE** loop will then cause another prime to be supplied to all the testers and the testers to be restarted. This continues until all candidates less than the upper bound have been tested.

Although this solution requires a certain number of primes to be generated sequentially the program could be altered so that just sufficient primes were generated to ensure that the testers could start operating; that is, in order to sum primes up to n , primes up to $\sqrt{\sqrt{n}}$ would be generated. At this stage the testers can start working and a further concurrent process could start generating the primes which become needed by the testers as testing continues.

It is also possible to make the controller maintain an ordered list of primes produced by the testers. The early

primes produced can than be used in the testing of larger primes.

4.4 References

- 1 *Simulation of Statistical Mechanical Systems on Transputer Arrays*,
C R Askew, D B Carpenter, J T Chalker, A J G Hey, D A Nicole and D S Pritchard
Physics Department, University of Southampton, To be published.
- 2 *Signal Processing with Transputer Arrays*, J G Harp, J B G Roberts and J S Ward,
Royal Signals and Radar Establishment, Malvern, Worcestershire,
Computer Physics Communications 1985.

5 Compiling occam into silicon

5.1 Introduction

The **occam** language [1] allows a system to be hierarchically decomposed into a collection of concurrent processes communicating via channels. An **OCCAM** program can be implemented by a single programmable microcomputer, or by a collection of programmable computers each executing an **occam** process. An **occam** process can also be implemented directly in hardware. This paper describes a compiler which translates **occam** programs into silicon layout.

5.2 VLSI design

In designing a VLSI device, it is useful to have a behavioural description of what the device does, and a hardware description of the components of the device and the way in which they are interconnected.

Hardware description languages are used in many computer-aided design systems. The hardware description of a device can be checked against the silicon layout supplied by the designer and can be used as input to simulators. The hardware description language used by INMOS allows libraries of standard checked modules to be assembled. All of these techniques combine to remove much of the risk from silicon design once the hardware description of a device has been constructed.

Behavioural description languages have been used to design sequential processors for many years. As the process of interpreting instructions in a sequential computer is (nearly) sequential, a conventional sequential programming language can be used to write the behavioural description of a processor. An advantage of using a programming language for this purpose is that the description of the device can be compiled into an efficient simulator of the device.

The behaviour of VLSI devices with many interacting components can only be expressed in a language which can express parallelism and communication. Communicating process languages are therefore beginning to be used to describe the behaviour of such devices. For example, **occam** has been used extensively for this purpose in the design of the INMOS transputer.

occam has several advantages as a behavioural description language. Firstly, the concepts of concurrency and communication in **occam** correspond closely to the behaviour of hardware devices. Secondly, as a programming language, **occam** has a very efficient implementation, and this enables fast execution of a system description as a simulation. Thirdly, **occam** has rich formal semantics [2] which facilitate program transformation and proof, and a simple interactive transformation system has been constructed. These techniques have been used to formally establish the correctness of an **occam** implementation of IEEE standard 754 floating-point arithmetic [3], a task which takes too long to be performed by experimental testing. The transformation system can also be used to optimise programs and can, for example, transform certain kinds of sequential program into an equivalent parallel program, and conversely.

The problem of ensuring that the hardware description of a device indeed implements the behavioural description in **occam** is a significant one. One possible approach is to write a compiler to compile an **occam** program into a hardware description.

5.3 occam

occam programs are built from three primitive processes:

v := e assign expression e to variable v
c ! e output expression e to channel c
c ? v input variable v from channel c

The primitive processes are combined to form constructs:

SEQ sequence
IF conditional

PAR parallel
ALT alternative

A construct is itself a process, and may be used as a component of another construct.

Conventional sequential programs can be expressed with variables and assignments, combined in sequential and conditional constructs. Conventional iterative programs can be written using a **WHILE** loop.

Concurrent programs make use of channels, inputs and outputs, combined using parallel and alternative constructs.

In hardware terms, it is useful to think of a variable as a storage register and a channel as a communication path with no storage.

Each OCCAM channel provides a communication path between two concurrent processes. Communication is synchronised and takes place when both the inputting and the outputting process are ready. The data to be output is then copied from the outputting process to the inputting process, and both processes continue.

An alternative process may be ready to input from any one of a number of channels. In this case, the input is taken from the channel which is first used for output by another process.

5.4 Implementation of OCCAM

The concepts of sequence and concurrency in OCCAM are abstract, and allow a wide variety of implementations. An OCCAM process can be implemented:

- 1 by compilation into a program for execution by a general-purpose computer such as a transputer
- 2 (1) with a fixed program held in ROM
- 3 by compilation into a special-purpose computer, with just sufficient registers, ALU operations, memory and microcode to implement the process
- 4 by compilation into 'random' logic

Similarly, the concept of communication is abstract, and allows a channel to be implemented in various ways:

- 1 store location(s) and program
- 2 (1) with microprogram instead of program
- 3 a parallel path with handshaking signals
- 4 a (more) serial version of (3), the communicating processes breaking the data into several pieces
- 5 a completely serial path

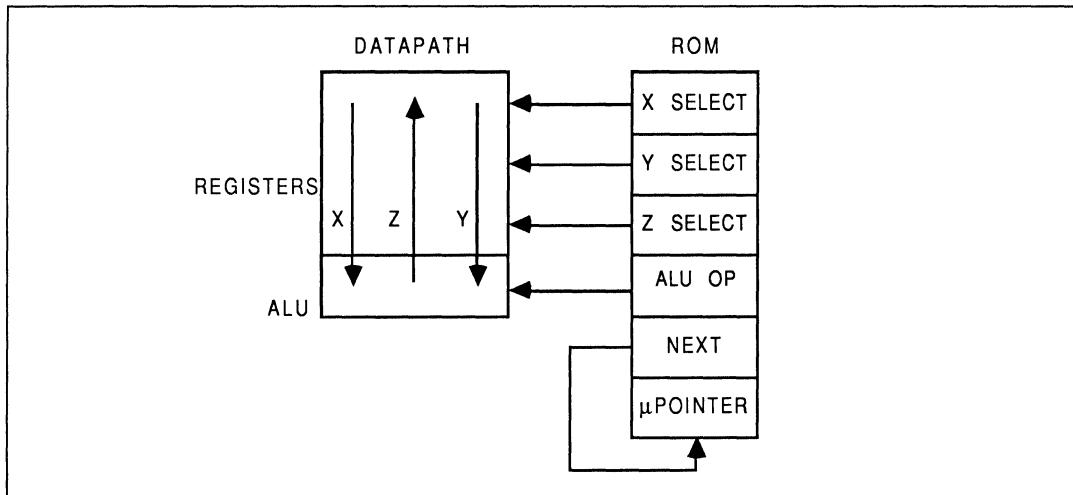
Any of the above can be implemented using any clocking scheme, ranging from a globally synchronous system to a fully self-timed system. It should be possible to mix the implementation techniques within a system, though this requires a range of different channel implementations which operate as 'adaptors' to provide communication between processes implemented in different ways.

Implementation of OCCAM processes using programmable computers and transputers has been described elsewhere [4]. Implementation of processes using self-timed circuit elements is the subject of current research e.g.[5]. This paper concentrates on the compilation of a process into a tailored datapath controlled by compiled

microcode. A set of concurrent processes may be compiled into a corresponding set of such machines, with each communication channel implemented by a simple synchronous connection between two machines.

5.5 The abstract micro-machine

Each process is compiled into a datapath controlled by horizontal microcode. The datapath contains a set of registers connected to an arithmetic logic unit by three buses. These are called the Xbus, Ybus and Zbus. Each cycle of the machine involves transferring the contents of two selected registers via the Xbus and Ybus to the arithmetic logic unit for use as operands, and transferring the result from the logic unit back to a selected register via the Zbus.



The selection of the registers and the operation to be performed by the logic unit is determined by four components of a microinstruction held in the read-only memory (ROM). The registers in the datapath are designed so that a single microinstruction can use the same register as both an operand and as the result, but this is not essential (a compiler can easily allocate registers to avoid the need for it). The microinstruction ROM is addressed by a microinstruction pointer register.

A further 'next address' component of each microinstruction gives the next value of the microinstruction register. The microinstruction pointer register is loaded from this field as each microinstruction is executed. The 'next address' field can be omitted and the microinstruction pointer register replaced by an incrementer if the process to be implemented consists only of a simple loop with no conditional behaviour.

A number of other components of the microinstruction may be needed, depending on the program being compiled. These will be described below.

5.6 The compiler output

The compiler makes extensive use of the module library used in the INMOS transputer itself. This library contains all of the hardware modules needed to construct ALUs and registers, together with special control logic for fast multiplication, division, shifts etc. It also provides for microinstruction pointer registers, control line drivers and clock generators. The microcode ROM itself can be generated and optimised automatically from the textual form of the microinstructions. The output of the compiler is therefore:

- 1 a microprogram ready for input to the ROM generator
- 2 an HDL (INMOS hardware description language) description of the datapath including the minimum number of registers and the simplest ALU which are sufficient to implement the process

- 3 an 'array' file containing information about the physical placement of the modules comprising the datapath

The output can be 'input' to the INMOS CAD system, enabling logic and circuit simulations to be performed, allowing the layout to be inspected, and ultimately enabling masks to be produced. It is, however, envisaged that the design process would be interactive, and that having inspected the result of a compilation the designer would modify the OCCAM specification (probably using correctness preserving transformations) and try again.

5.7 Variables, expressions, assignment and SEQ

Values of variables are held in registers, and expressions are evaluated as a sequence of microinstructions of the form described above.

Expressions also involve 'literal' operands. These are derived directly from a 'literal' component of the microinstruction. This need only be able to supply a single operand of each microinstruction, as any operation involving two literal operands can be performed by the compiler.

The compilation of:

```
WHILE TRUE
  P
```

where P is a sequence of assignments therefore proceeds as follows:

- 1) Identify the number of registers needed. At any point in the program, a number V of variables is in scope, and each of these must have a register allocated to it. Also, a number T of temporary registers may be needed to hold temporary values arising during the evaluation of complex expressions. The number of registers needed for P is the largest value taken by $V+T$ in P . This is a conventional compiling technique.
- 2) Identify the operations needed in the arithmetic logic unit. This depends on the expression operators used in the program being compiled. If only bit operators are used, the carry path can be omitted, and it is worthwhile only including:

- the carry path (adder)
- the shifter
- the multiply divide step control logic
- the conditional logic

if they are needed. The multiply and divide control logic require conditional selection of the next microinstruction to be executed, and this is described below.

- 3) Break all expressions and assignments into a sequence of operations of the form:

```
Z := X op Y
```

For example:

```
VAR a, b, result:
SEQ
  a := 10
  b := 20
  result := (a + b) - 5
```

generates microcode field definitions to control the registers, 'constants box', and ALU, in addition to the 'next'

field. The following example is the definition of the register control field:

```
FIELD "Regfield" Microword[22, 23, 24, 25, 26, 27]
XbusFromR0 = #B100000
XbusFromR2 = #B010000
YbusFromR1 = #B001000
R0FromZbus = #B000100
R1FromZbus = #B000010
R2FromZbus = #B000001;
```

The register R0 is used for **a**, R1 for **b** and R2 for **result**. R0 and R2 can supply data to the Xbus, R1 to the Ybus. All three registers can be loaded from the Zbus.

A microinstruction is constructed by combining values from each of a number of fields; for example:

```
LAB1: XbusFromR0 YbusFromR1
      ZbusFromXbusPlusYbus R2FromZbus LAB2;
```

selects R0 and R1 as the sources for the Xbus and Ybus respectively, selects the ALU operation as Plus (ZbusFromXbusPlusYbus) and selects the R2 as the destination for the result. LAB2 indicates the next microinstruction to be executed.

The microcode for the above program is:

```
START: XbusFrom10 ZbusFromXbus
       R0FromZbus LAB0;
LAB0: XbusFrom20 ZbusFromXbus
       R1FromZbus LAB1;
LAB1: XbusFromR0 YbusFromR1
      ZbusFromXbusPlusYbus R2FromZbus LAB2;
LAB2: XbusFromR2 YbusFrom5
      ZbusFromXbusMinusYbus R2FromZbus END;
```

An example of the HDL generated is the registers a, b, result:

```
MODULE Registers (IN Clocks[4:1], ROMoutputs[27:22],
                  Zbus[31:0],
                  OUT Xbus[31:0], Ybus[31:0])
  Xreg32 R0 (IN Clocks[4:1], ROMoutputs[22],
              ROMoutputs[25], Zbus[31:0],
              OUT Xbus[31:0])
  Yreg32 R1 (IN Clocks[4:1], ROMoutputs[24],
              ROMoutputs[26], Zbus[31:0],
              OUT Ybus[31:0])
  Xreg32 R2 (IN Clocks[4:1], ROMoutputs[23],
              ROMoutputs[27], Zbus[31:0],
              OUT Xbus[31:0])
END REGISTERS
```

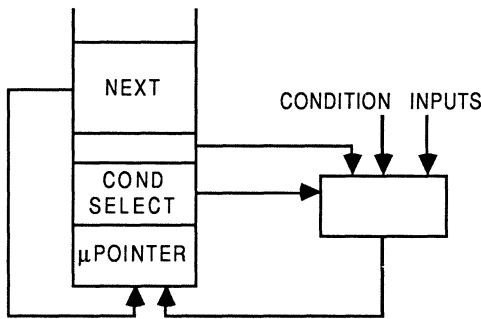
which defines the collection of the three registers and their control signals and bus connections. XReg32 is itself the name of a module which defines a 32-bit register with outputs to the Xbus; YReg32 similarly defines a register with outputs to the Ybus.

5.8 IF and WHILE

The occam IF and WHILE constructs can both be implemented by allowing the address of the next microinstruction to be determined by a selected condition.

Conditional behaviour is provided by arranging for the least significant bit of the microinstruction pointer to be loaded from a selected conditional input; the selection being made by a further microinstruction field connected to a multiplexor. To allow unconditional branching, one input from the multiplexor is derived from

the least significant bit in the 'next address' field.



An example is the following process which computes the greatest common divisor of two numbers:

```
VAR m, n, result:  
SEQ  
  m := 100  
  n := 35  
  WHILE (m <> n)  
    IF  
      (m > n)  
        m := m - n  
      (m < n)  
        n := n - m  
    TRUE  
    SKIP  
  result := m
```

which generates the following microcode, and requires three registers for **m**, **n** and **result**:

```
START: XbusFrom100 ZbusFromXbus  
       R0FromZbus LAB0;  
LAB0:  XbusFrom35 ZbusFromXbus  
       R1FromZbus LBL0;  
LBL0:  XbusFromR0 YbusFromR1 ZbusFromXbusMinusYbus  
       (CondFromNotZbusEq0 -> LAB1, LBL1);  
LAB1:  XbusFromR1 YbusFromR0 ZbusFromXbusMinusYbus  
       (CondFromZbusGr0 -> LBL2, LAB2);  
LAB2:  XbusFromR0 YbusFromR1 ZbusFromXbusMinusYbus  
       R0FromZbus LBL0;  
LBL2:  XbusFromR0 YbusFromR1 ZbusFromXbusMinusYbus  
       (CondFromZbusGr0 -> LBL0, LAB3);  
LAB3:  XbusFromR1 YbusFromR0 ZbusFromXbusMinusYbus  
       R1FromZbus LBL0;  
LBL1:  XbusFromR0 ZbusFromXbus  
       R2FromZbus END;
```

5.9 Arrays

Arrays are implemented by including a random access memory. Indexing operations are provided by constructing the bitwise OR of the base address and the subscript (the base being a literal and the subscript being held in a register), eliminating the need for address arithmetic and enabling a selected component of an array to be transferred to or from a register in a single cycle. The base address of each array in the process

is chosen to make this possible, and unused rows are omitted from the memory array.

5.10 Procedures

occam procedures can be implemented either by substitution of the procedure body prior to compilation or by a conventional closed procedure call.

As no recursion is permitted, the maximum depth of calling is known to the compiler, and it is possible to compile a stack of microinstruction pointer registers of the appropriate depth. Dedicated registers can be allocated for the variables in each procedure; temporaries can be shared by all procedures as occam does not contain functions.

5.11 PAR

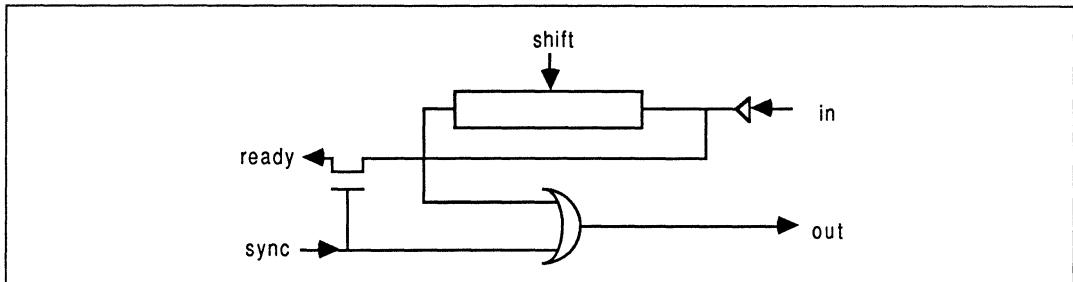
The easiest way to implement concurrent processes is to use one processing element for each process, and the present compiler does this.

5.12 Channels and communication

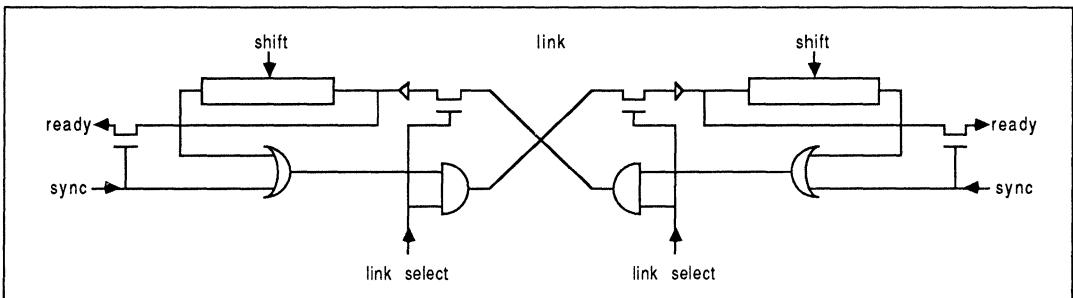
Synchronisation of input and output requires that the processor idles as the first process waits for the second. This is achieved by a microprogram polling loop.

It is clearly desirable to minimise the amount of hardware associated with each channel, and to minimise the number of connections needed to implement a channel.

For any process which includes channel communication, the compiler generates a shift register, two control signals, *sync* and *shift*, and an input to the condition multiplexor, *ready*.



For each pair of devices which communicate, two connections are used to form a link. Each link is connected to a device as shown; only one additional control signal is needed for each link on a device; this is used to select which link is in use.



An input or output is performed by asserting the *sync* signal together with the appropriate link select signal, and polling the *ready* signal. When a ready signal is detected, this indicates that both devices are ready to communicate. At this point the process at the other end of the link will also have detected a ready signal. Both devices now release their *sync* signals, and clock their shift registers using the *shift* signals. With only the link select signals asserted, the two shift registers at either end of the channel effectively form one long cyclic shift register, so the data in the two shift registers is exchanged. After the data has been exchanged, the link select signals are released.

Clearly, this operation is completely symmetrical. Each link between two devices can be used for both input and output; it is not necessary for these to be performed concurrently as each device implements only one process.

An example of a simple process which inputs a value, adds 1, and outputs the result is:

```
CHAN c, d:  
VAR x:  
SEQ  
  c ? x  
  d ! (x + 1)
```

The microcode is as follows:

```
SETUP0:    YbusFrom32 ZbusFromYbus T0FromZbus  
           SelectCh0 SYNC  
           (CondFromReady -> TRANSFER0, SETUP0) ;  
TRANSFER0: XbusFromT0 YbusFrom1 ZbusFromXbusMinusYbus  
           T0FromZbus  
           SelectCh0 ShiftChan  
           (CondFromNotZbusEq0 -> TRANSFER0, DONE0) ;  
DONE0:     XbusFromChan ZbusFromXbus  
           R0FromZbus LAB0 ;  
LAB0:      XbusFromR0 YbusFrom1 ZbusFromXbusPlusYbus  
           ChanFromZbus SETUP1 ;  
SETUP1:    YbusFrom32 ZbusFromYbus T0FromZbus  
           SelectCh1 SYNC  
           (CondFromReady -> TRANSFER1, SETUP1)  
TRANSFER1: XbusFromT0 YbusFrom1 ZbusFromXbusMinusYbus  
           T0FromZbus  
           SelectCh1 ShiftChan  
           (CondFromNotZbusEq0 -> TRANSFER1, END) ;
```

A temporary register (T0) is introduced to count the number of bits to be transferred to or from the channel register (CHAN). The value of *x* is held in a further register (R0). The ALU is used to decrement the count register and test for zero at the same time that each bit is shifted through the link.

5.13 ALT

Alternative input requires that the inputting processor can poll a number of channels in turn until one is found to be ready for input. The link implementation described above can be used for this purpose; an example is shown below:

```
ALT  
  in1 ? x  
  count := count + 1  
  in2 ? x  
  count := count - 1
```

The microcode for polling the channels is:

```

LBL1:      SelectCh0  SYNC
            (CondFromReady -> TRANSFER1, LBL2)
TRANSFER1:  ...
            ...

LBL2:      SelectCh0  SYNC
            (CondFromReady -> TRANSFER2, LBL1)
TRANSFER2:  ...
            ...

```

The microcode loop attempts to synchronise with each of the two links until it succeeds, in which case it continues with the input.

5.14 Example: the prime farm

Prime numbers can be generated concurrently using a 'processor farm'. A program is given in [6]. It uses a controller which farms out successive numbers to an arbitrary number of primality testers. Each tester stores all of the primes up to the square root of the number to be tested; it uses these to test whether or not the number is prime, and responds to the controller accordingly.

Here we use an even simpler program. Each tester divides its new number by all numbers up to the square root of the new number. This removes the need for an array to store the prime numbers up to the square root.

This is entirely justified because we are trying to optimise the use of silicon area; *the area taken for one tester with memory can be better used for many testers without*. This is certainly true for generating primes up to 2^{32} .

```

PROC primetest(CHAN from.controller, to.controller)
    DEF isprime = 0, notprime = 1:
    VAR maxtest, candidate, active:
    SEQ
        active := true
        WHILE TRUE
            SEQ
                from.controller ? maxtest; candidate
                IF
                    maxtest = 0
                    active := FALSE
                    maxtest <> 0
                    VAR nexttest:
                    SEQ
                        nexttest := 3
                        WHILE ((candidate REM nexttest) <> 0) AND
                            (nexttest < maxtest)
                            nexttest := nexttest + 2
                        IF
                            nexttest < maxtest
                                to.controller ! not.prime
                            TRUE
                                to.controller ! is.prime

```

The controller is about 1.5 mm \times 2.5 mm in area; each tester is about 1.2 mm \times 2.3 mm. The space occupied by a controller with sixteen testers is about 50 mm², and can easily be implemented on a single chip using a current manufacturing process. Such a chip would require very few external connections; a single link, clock and reset inputs, and power. There is a great deal of freedom in configuring the devices on the chip, as they communicate only by two-wire links. It seems likely that 'process farms' are an effective way of organising specialised VLSI systems.

5.15 Example: signal processing

The following example is a second-order filter which filters a stream of values. It would normally be used as a component in a pipeline in which each component filter has different parameters.

```

PROC Filter (Chan In, Out)
  VAR x, y, t1, t2, t3, z1, z2:
  SEQ
    z1 := 3
    z2 := 4
    WHILE TRUE
      SEQ
        in ? x
        t1 := x - (b2 * z2)
        t2 := a2 * z2
        z2 := t1 - (b1 * z1)
        t3 := t2 + (a1 * z1)
        out ! t3 + (a0 * z2)
        in ? x
        t1 := x - (b2 * z1)
        t2 := a2 * z1
        z2 := t1 - (b1 * z2)
        t3 := t2 + (a1 * z2)
        out ! t3 + (a0 * z1)

```

This requires 9 registers and 93 microinstructions; the relatively large number of microinstructions arises because each multiplication requires a short sequence of microinstructions including a loop. This could be improved by providing microcode subroutines (using an additional microinstruction pointer register). Multiplication speed could also be improved (at the expense of area) by use of a parallel multiplier.

The filter occupies 3 mm²; so a pipeline of 20 filters could be fitted on a single VLSI device.

5.16 Example: simple processor

Our final example is a simple programmable processor with a (very) reduced instruction set. Despite its tiny instruction set, it provides all of the functions needed to implement a sequential occam process; in fact it is very easy to compile an OCCAM process into the instruction set of this processor. The processor has four input and four output links, and 256 bytes of random-access memory.

```

PROC Processor(CHAN In0, In1, In2, In3,
               CHAN Out0, Out1, Out2, Out3)
  VAR Iptr, Wptr:
  VAR Areg, Breg:
  VAR Instruction, Function, Operand:
  VAR Memory[256]:
  SEQ
    Memory[0] := Boot
    Iptr := 0
    Operand := 0

    WHILE TRUE
      SEQ
        Instruction := Memory [ Iptr ]
        Iptr := Iptr + 1
        Function := Instruction /\ #F0
        Operand := (Instruction /\ #0F) \/ Operand

        IF
          Function=Prefix
            Operand := Operand << 4

```

```
TRUE
SEQ
IF
    Function=Loadavar
        Areg := Memory[Wptr+Operand]

    Function=Loadbvar
        Breg := Memory[Wptr+Operand]

    Function=Loadalit
        Areg := Operabd

    Function=Loadbbit
        Breg := Operand

    Function=Storeavar
        Memory[Wptr+Operand] := Areg

    Function=Loadaind
        Areg := Memory[Areg+Operand]

    Function=Storebind
        Memory[Areg+Operand] := Breg

    Function=Jump
        Iptr := Iptr + Operand

    Function=Jumpfalse
        IF
            Areg = 0
            Iptr := Iptr + Operand
        TRUE
        SKIP

    Function=Equalalit
        Areg := Areg = Operand

    Function=Addalit
        Areg := Areg + Operand

    Function=Adjust
        Wptr := Wptr + Operand

    Function=Call
        SEQ
        Areg := Iptr
        Iptr := Iptr + Operand

    Function=Operate
        IF
            Operand=input
            IF
                Areg=0
                    In0 ? Areg
                Areg=1
                    In1 ? Areg
                Areg=2
                    In2 ? Areg
                Areg=3
                    In3 ? Areg
```

```

Operand=output
IF
  Areg=1
    Out0 ! Breg
  Areg=2
    Out1 ! Breg
  Areg=4
    Out2 ! Breg
  Areg=8
    Out3 ! Breg

Operand=Alternative
ALT
  ((Areg /\ 1) <> 0) & In0 ? Areg
    Iptr := Iptr + 0
  ((Areg /\ 2) <> 0) & In1 ? Areg
    Iptr := Iptr + 1
  ((Areg /\ 4) <> 0) & In2 ? Areg
    Iptr := Iptr + 2
  ((Areg /\ 8) <> 0) & In3 ? Areg
    Iptr := Iptr + 3

Operand=Greater
  Areg := Areg > Breg

Operand=Shiftleft
  Areg := Areg << Breg

Operand=Shiftright
  Areg := Areg >> Breg

Operand=Xorbits
  Areg := Areg >< Breg

Operand=Andbits
  Areg := Areg /\ Breg

Operand=Add
  Areg := Areg + Breg

Operand=Subtract
  Areg := Areg - Breg

Operand= Boot
SEQ
  In0 ? Wptr
  Iptr := 0
  WHILE Iptr < Wptr
    SEQ
      In0 ? Memory [ Iptr ]
      Iptr := Iptr + 1
    Iptr := 0
  Operand := 0

```

On reset, the processor waits for a program to be supplied via link 0. It then loads a program, and executes it until a 'boot' instruction is executed.

There is obviously considerable scope for better optimisation in this case; in particular it would be desirable to implement the instruction decoding 'IF' construct with a mechanism which replaces the microinstruction pointer register with a value held in a register.

The processor requires 11 registers and has 140 microinstructions. The whole device including the memory occupies about 6.25 mm²; 10 such devices with their interconnections would take less area than a typical 32-bit microprocessor.

5.17 Conclusions

A communicating process language such as occam can be used to design VLSI devices, and can be compiled into silicon layout. Some parts of the design process are still performed by hand (such as the final placement of the functional blocks), but this cannot introduce errors. It is therefore possible to design concurrent VLSI systems using OCCAM, establish that the design behaves as intended using the formal semantics of occam (or in simple cases by experimental testing of the occam program), and finally compile the OCCAM source into correct silicon layout.

In order to simplify the construction of the compiler, many issues have been ignored. For example, the synchronous communication system is only appropriate for local communication between devices sharing a common clock. This problem can be overcome by using a different link implementation for 'long distance' communication (for example, the link used in the transputer itself).

Expressing an application in a form which efficiently exploits silicon area involves careful consideration of the relative costs of memory, processing and communication. Concurrent algorithms which perform 'redundant' calculations can be faster *and consume less area* than sequential algorithms which store values. An important use of a silicon compiler is to aid in the evaluation of 'silicon algorithms'.

5.18 References

- 1 *occam Programming Manual*, Prentice-Hall International 1984.
- 2 *The Laws of occam Programming*, A W Roscoe and C A R Hoare, Programming Research Group, Oxford University 1986.
- 3 *Formal Methods applied to a Floating Point Number System*, G Barrett, Programming Research Group, Oxford University 1986.
- 4 *The Transputer Implementation of occam*, INMOS Ltd, Technical Note 21.
- 5 *Compiling Communicating Processes into delay insensitive VLSI Circuits*, Alain J Martin, Journal of Distributed Computing 1986.
- 6 *Communicating Process Computers*, INMOS Ltd, Technical Note 22.



Practice

6 The development of occam 2

6.1 Introduction

The major reason for the design of the **occam 2** [1] programming language was a desire to incorporate floating-point arithmetic into **occam**. This had to be done without breaching the security of the language. As a result **occam 2** is well defined, many programming errors are detectable at compile time, and run-time errors are reliably and cheaply detectable. This paper describes various aspects of the language design which relate to security; some of these, such as 'channel protocols' overcome problems caused by the introduction of new data types to the language, others, such as alias checking, tackle security problems which are present in most other programming languages.

6.2 The data types of occam 2

The **occam 1** programming language provided concurrency, message passing and a limited set of data types; the word, the channel and vectors of words or of channels. Although this was sufficient for many purposes, there were instances where a language which had a richer set of data types would offer significant advantages. In particular:

- 1 We wanted to support numerical programming; for **occam** to become the FORTRAN of parallel processing we would have to support floating-point arithmetic and multi-dimensional arrays.
- 2 We wanted to be able to pass messages of length greater than a single word. This is because much of the cost of passing a message is due to process synchronisation rather than data transfer. A language which would permit several words to be communicated in a single transfer would be more efficient than one which could transfer only single words.
- 3 We wanted to program systems of processors which did not share a common wordlength. In such systems communication would have to be in terms of some unit other than the word.

As a result **occam 2** supports several primitive data types. There is a machine-dependent data type, INT, which loosely corresponds to the VAR of **occam 1**. INT is the type of signed integer values most efficiently provided by the machine. As this normally corresponds with the size of an address in the machine, values of type INT are used for such purposes as replicator indices and array subscripts. Unlike **occam 1**, there is a separate type BOOL, which represents boolean values, and a BYTE type, which represents unsigned integers in the range 0 to 255. (Note that although **occam 1** could pack and unpack values into bytes, the byte was not a proper data type; all arithmetic and message passing was done in terms of words.)

There are occasions where the use of a machine-dependent type such as INT is not satisfactory; for example, where a message is to be passed between two machines of differing wordlength, or where a calculation has to be performed to a particular precision, regardless of the machine on which it is to be performed. To cope with these situations, **occam 2** has a further three integer types, INT16, INT32 and INT64, which represent signed integers with a length of 16, 32 and 64 bits respectively.

There are two floating-point types called REAL32 and REAL64. These correspond to the single- and double-length floating-point numbers of the IEEE Standard for Binary Floating Point. In fact **occam** does not support the multiple error symbols of the standard as this would undermine the substitution semantics of the language; for example, in full IEEE arithmetic, $x = y$ does not imply that $x \text{ op } z = y \text{ op } z$.

In addition to the primitive types, **occam** has array types. The components of an array may be of any single type. As an array may have components of an array type, **occam 2** does provide multi-dimensional arrays. An array may be subscripted (giving a component of the array), assigned, passed as a message and used as a parameter to a procedure or a function.

6.3 Channel protocols

The presence of several different data types in occam 2 introduces the problem of how to extend the communication model to handle them. The problem arises from the need to ensure that when a message is passed, the type of data sent by the transmitter matches the type of data expected by the receiver. It is desirable to provide some sort of checking of channel communication for two reasons. Firstly, it is very easy to make mistakes in communication and *anything* which enables these mistakes to be detected at compile time is helpful. Secondly, the effect of run-time errors in communication can be at least as devastating as subscript errors; it can cause store to be overwritten arbitrarily, or can cause the breakdown of process synchronisation.

A number of different proposals were considered during the design of occam 2. Some, such as restricting a channel to communicating a single type, were rejected due to lack of flexibility. Others, such as type-checking all communication at run-time, were rejected as they carried too much run-time overhead.

The solution adopted in occam 2, the *channel protocol*, allows great freedom over what is communicated on a channel, but ensures security. Whenever a channel is declared the structure of all communication occurring on that channel must also be declared as a channel protocol. This enables most communication to be checked at compile-time, and simplifies any remaining run-time checks.

The simplest protocol permits communication of a single type which may, of course, be an array type. For example, if the channel **greeting** is being used to communicate strings which are to be displayed on a 12-character LCD display, it might be declared:

```
CHAN OF [12]BYTE greeting :
```

The compiler can subsequently check that all inputs and outputs correspond to this protocol. Thus the compiler would accept:

```
greeting ! "Hello world!"
```

but would reject:

```
greeting ! "Goodbye world!"
```

as the string has too many characters.

Whilst this example is perfect for the application described it does raise the question of how to deal with arrays whose size is determined at run-time. As this is a fairly general requirement occam has a protocol which corresponds to a counted array. When a message is passed on a channel with a counted array protocol, the length of the array is communicated and then the array is communicated. Suppose in the previous example, the string was to be displayed, not on a 12-character LCD display, but as a line on a terminal. We might then declare a channel **terminal** as:

```
CHAN OF INT::[]BYTE terminal :
```

indicating that inputs would be of the form:

```
terminal ? count::array
```

which first receives the number of elements to be input into the variable **count** and then inputs into the first **count** elements of **array**. Similarly, outputs would be of the form:

```
terminal ! count::array
```

which first outputs the value of **count** and then outputs the first **count** elements of the array **array**.

Where a channel has been declared with a counted array as its protocol, some checks can be made at compile-time but others must wait until run-time. For example, with the channel **terminal** declared above, the compiler would reject the output:

```
terminal ! "Hello world!"
```

as it is not of the correct form. However, it would accept the following:

```
[4000]BYTE lengthy :
SEQ
...
terminal ! SIZE lengthy::lengthy
```

which would cause a run-time error when input by the following fragment of program:

```
[80]BYTE line.buffer :
SEQ
...
terminal ? count::line.buffer
```

as the compiler inserts code to check the value of **count** against the length of the array **line.buffer**.

In addition to the protocols described above, called *simple protocols*, which permit a single item to be communicated, occam 2 has *sequential protocols* which permit a specified number of items to be transmitted by a single input or output. Suppose that we wanted to extend the previous example so that the message passed along the channel **terminal** specified the line on which the string was to be displayed. We want to send messages which first send the line number and then the text to be displayed. We can name a suitable protocol and then use it to declare the channel. (The previous examples have used simple protocols which do not require naming.)

```
PROTOCOL line IS INT; INT::[]BYTE :
CHAN OF line terminal :
```

As a result of this declaration the compiler is able to check that all communications are of the correct form. For example, the following would be detected as an error:

```
terminal ! 12::"Hello world!"; line.number
```

since the order of the line number and the line has been swapped.

Often a single channel is used to pass messages with different structures. For example, suppose we are writing a program to control a pen plotter which has a number of simple operations of the form 'pen up' or 'pen down', and a single draw operation 'move pen' which requires a pair of coordinates. We can indicate that we wish to send messages of two different structures by using a *variant protocol*. In this case we would declare a protocol **plotter.control** consisting of two *tagged protocols*:

```
PROTOCOL plotter.control
CASE
  simple.command; INT
  move.command; REAL32; REAL32
```

The compiler only permits outputs which first output one of the tags, followed by an output matching the remainder of the tagged protocol. For example, the following output will cause the plotter to move to the origin:

```
plotter ! move.command; 0.0 (REAL32); 0.0 (REAL32)
```

If we had made a mistake here such as omitting the **move.command** tag or sending the coordinates as integers, the compiler would detect the error.

An input on a channel which has a variant protocol is necessarily more complex than an output. The actual form the input will take depends on the tag received. To cope with this OCCAM 2 has a 'case input' which first inputs a tag and then selects a matching input and then executes an associated process. For example, the program which actually drives the plotter would have an input such as:

```
plotter ? CASE
  simple.command; command
    execute.command(command)
  move.command; x; y
    move.pen.to(x, y)
```

When this is executed a tag is input from the channel **plotter** and used to select the matching input. For example, if the tag input is **move.command** then an input to **x** and **y** will occur, followed by the execution of the procedure **move.pen.to**.

It is not necessary to list all possible tags in a case input. When a case input receives a tag which does not match any of the tagged inputs this is treated as an error. There are occasions where a program is expecting a specific tag to be received; in these cases a special form of input can be used. For example, if the pen plotter driver is expecting a **simple.command** then the program would look like:

```
SEQ
  ...
  plotter ? CASE simple.command; command
  ...
```

This special case input is equivalent to:

```
plotter ? CASE
  simple.command; command
  SKIP
```

6.4 Numerical behaviour

The numerical behaviour of operations in OCCAM 2 is well defined. Usually overflow, division by zero, et cetera are treated as errors. However, it is recognised that sometimes it is necessary to perform calculations where these events are not considered to be errors. To this end OCCAM 2 provides the PLUS, MINUS and TIMES operators which are unchecked.

The presence of the large number of concrete data types in the language raises the question of how constant values should be represented. In OCCAM 2, only INT literals expressed as undecorated decimal strings (e.g. **123**) or as hexadecimal strings (e.g. **#FA77FE16**), BYTE literals expressed as a quoted character (e.g. **'Z'**) and the BOOL constants (**TRUE** and **FALSE**) take their types implicitly; all other constants are explicit about their type. Whilst the requirement for explicit typing may seem unnecessary it does ensure that arithmetic on constants is performed correctly. For example, the result of the calculation **16777216.0 + 1.0** depends on whether the numbers are interpreted as REAL32s (in which case the result is **16777216.0**) or as REAL64s (in which case the result is **16777217.0**). It is for similar reasons that OCCAM requires that all conversions between types are stated explicitly.

A similar decision has been taken in deciding to perform arithmetic according to the type of data on which it is being performed. All intermediate results are calculated as if they were of the same type as the result. This is unusual; it is quite common for the intermediate results of floating-point calculations to be held in an extended format. Whilst this may seem advantageous, it actually has two important drawbacks. The first is that it can lead to 'double rounding' and thus a *less* accurate result than if the arithmetic were performed to the correct precision. For example, in the program:

```
VAL REAL32 a IS 2-100(1 + 2-23) :
VAL REAL32 b IS 2-27(1 - 3 × 2-23) :
x := a * b
```

a different result from that obtained by rounding straight into single-length format is obtained if the calculation is first rounded to an extended precision and then into the correct precision. The second drawback is that storing a result becomes an *arithmetic* operation which undermines the substitution semantics of the language. This has important consequences as various common optimisations, such as common subexpression elimination, would no longer be valid. For example, the two programs below would not be equivalent:

```
REAL32 dummy :
SEQ
    dummy := x * y           SEQ
    a := dummy + a           a := (x * y) + a
    b := dummy + b           b := (x * y) + b
```

6.5 Abbreviations

The occam 2 language defines procedure calling in terms of 'abbreviation' and textual substitution. An abbreviation enables a name to be given to a variable or array element or to an expression. For example:

```
INT element IS array[subscript] :
```

introduces the name **element** to identify the array component **array [subscript]** and:

```
VAL INT twice.x IS 2 * x :
```

introduces **twice.x** as a name for the expression **2 * x**.

In order to keep the semantics of abbreviation simple, and the implementation of abbreviation and parameter passing efficient, various rules concerning abbreviations are enforced. One such rule is that the abbreviation of an expression is only valid if its scope contains no assignment to a variable in the expression. For example, consider the following program:

```
VAL x IS y[i][j] :
SEQ
  ...
  z := x
  ...
```

The rule mentioned above ensures that there are (at least) three possible implementations of the abbreviation. The first simply replaces the occurrence of **x** in **z := x** with **y[i][j]**. The second assigns the value of **y[i][j]** to a new 'variable' **x** when the abbreviation is executed and uses that variable in the assignment. Finally, the third sets up a pointer to **y[i][j]** when the abbreviation occurs and de-references that pointer when the assignments occurs.

One important consequence of defining parameter passing in terms of abbreviation is that VAL parameters can be passed either by copying the value (suitable for single-word values), or by passing a pointer to the value (suitable for arrays).

6.6 Alias checking

Aliasing occurs when, within a scope, there are two or more names which identify the same object. When aliasing is present, the meaning of programs becomes obscure, because assignment to one name can affect the value of another name.

For example, the following procedure *clearly* leaves the value of its parameter **x** unchanged (note that the use of the **PLUS** and **MINUS** operators ensures that arithmetic overflow is not a problem):

```
PROC nonsense(INT x, VAL INT y)
SEQ
  x := x PLUS y
  x := x MINUS y
:
```

as is demonstrated by the following expansion of **nonsense(n, 3)**:

```
INT x IS n :
VAL INT y IS 3 :
SEQ
  x := x PLUS y
  x := x MINUS y
```

which is equivalent to:

```
SEQ
  n := n PLUS 3
  n := n MINUS 3
```

which can be shown to be equivalent to **n := n** which is, in turn, equivalent to **SKIP**.

However, consider what would be the expansion of **nonsense(n, n)**:

```
INT x IS n :
VAL INT y IS n : -- invalid abbreviation
SEQ
  x := x PLUS y
  x := x MINUS y
```

which would be equivalent to:

```
SEQ
  n := n PLUS n
  n := n MINUS n
```

The value of **n** after this instance of **nonsense**, were it valid, would be 0.

Similarly, the instance **nonsense(i, v[i])**, were it valid would be equivalent to:

```
SEQ
  i := i PLUS v[i]
  i := i MINUS v[i]
```

the effect of which is very difficult to predict as in each of the assignments **v[i]** would probably reference a different component of **v**.

It is now recognised that aliasing can be the source of particularly insidious program bugs and, to counter this, aliasing is forbidden in some modern languages, for example, Euclid [2]. In OCCAM 2, aliasing is restricted, not only for the reason outlined above but also to simplify checking the validity of parallel constructs. The rules imposed in OCCAM 2 forbid the use of an element which has been abbreviated within the scope of that abbreviation. In the expansion of the instance of **nonsense(n, n)** given above, the abbreviation:

```
VAL INT y IS n :
```

is invalid because the name **n** has occurred on the right-hand side of an abbreviation which is currently in scope.

The majority of the anti-aliasing rules of OCCAM 2 can be checked at compile-time, however, those which permit an array to be used in a second abbreviation *provided that the same element of the array is not abbreviated* can require run-time checking. For example, consider:

```
first IS order[1] :
second IS order[2] :
```

which can be checked at compile-time. However, the abbreviations:

```
first IS order[1] :
n.th IS order[n] :
```

cannot as the second abbreviation is only valid if **n** is not equal to 1. The compiler will insert code to check this at run-time. (Although it may seem strange to perform this sort of check at run-time, rather than compile-time, it is really no different from range checking subscripts at run-time!)

The imposition of rules forbidding aliasing does have a perhaps unexpected impact in the use of procedures. The anti-aliasing rules require that when calling a procedure all non-VAL parameters are distinct, and are distinct from any VAL-parameters. These rules can lead to some procedure instances being unexpectedly rejected. For example, the procedure:

```
PROC factorial(INT result, VAL INT argument)
SEQ
    result := 1
    SEQ i = 1 FOR argument
        result := result * i
:
```

returns as its result the factorial of its argument. (Note that a negative argument will cause the replicated sequence to behave like STOP.) The instance **factorial(result, 3)** will set **result** to 6. However, consider **factorial(n, n)** which is supposed to set **n** to **n** factorial. This instance is, in fact, illegal according to the anti-aliasing rules. The reason for this can be seen if the instance is expanded:

```
INT result IS n :
VAL INT argument IS n : -- invalid abbreviation
SEQ
    result := 1
    SEQ i = 1 FOR argument
        result := result * i
```

which, if it were legal, would be equivalent to:

```
SEQ
    n := 1
    SEQ i = 1 FOR n
        n := n * i
```

which, in turn, would be equivalent to **n := 1**, not **n** factorial!

To get the effect originally desired we have to write:

```
INT temp :
SEQ
  factorial(temp, n)
  n := temp
```

The explicit introduction of temporary variables is undesirable and can be avoided in OCCAM 2 because of the presence of functions. These permit us to define:

```
INT FUNCTION factorial(VAL INT argument)
  INT product :
  VALOF
    product := 1
    SEQ i = 1 FOR argument
      product := product * i
  RESULT product
:
```

and to write `n := factorial(n)`.

In OCCAM 2 the functions are proper functions; they are side-effect free and deterministic. This is of great practical importance as it means that the compiler can compile replicated alternatives. Consider, for example the following alternative:

```
ALT i = 0 FOR n
  f(i) & c ? a
  P(i)
```

where `n` is a variable. The OCCAM compiler will generate code which evaluates each function instance, `f(i)`, twice; once when enabling the guards, once when disabling. Similarly, a compiler which used a polling implementation of alternative would also be correct.

6.7 Checking the validity of parallel constructs

The OCCAM 2 language specifies that if a variable is assigned to or is used in an input then that variable may not be used in any parallel process. Thus the compiler will reject a program such as:

```
PAR
  x := 42
  x := 69
```

However, consider the procedure `parallel.assignment`:

```
PROC parallel.assignment(INT x, y)
  PAR
    x := 42
    y := 69
:
```

The validity of any instance of this procedure will depend on the parameters used in that instance. This suggests that the compiler must check the validity of each procedure instance by substituting the parameters into the body of the procedure. However, the fact that alias checking is performed means that the compiler can check the validity in two stages.

During the first stage of checking each procedure is checked on the assumption that all parameters and free variables are distinct. This will accept the procedure **parallel.assignment** but, for example, would reject:

```
PROC invalid.parallel.assignment(INT x, y)
PAR
  x := 42
  x := 69
:
```

The second stage of checking is performed by the alias check which occurs for each procedure instance. For example:

```
parallel.assignment(x, y)
```

would be accepted, but:

```
parallel.assignment(x, x)
```

would be rejected.

It is important to notice that little more information is required about a procedure in order to perform parallel disjointness checking than is required for simple type-checking of its parameters. This opens the possibility of constructing a completely secure system for the dynamic loading and execution of procedures.

6.8 Run-time error handling in occam 2

When a language such as occam 2 is used for the programming of secure or reliable systems, the behaviour of that system when an error occurs is of great concern. There seems to be no single method of dealing with errors which is universally applicable to all systems. For this reason, occam 2 specifies that run-time errors are to be handled in one of three ways, each of which is suitable for use at different times.

The first mode is to ignore all run-time errors. This is potentially very dangerous and it is to be hoped that this will, one day, be made illegal except for systems which have been proved to be correct. This mode will most probably be used for benchmarking.

The other two modes detect run-time errors and prevent them from corrupting non-errant parts of the system. The first of these respectable modes causes all run-time errors to be signalled and to bring the whole system to a halt. This is known as 'halt' mode. In this mode the primitive process STOP is treated as if it caused an error. This mode is extremely useful for program debugging and is suitable for any system where an error is to be handled externally. For example, in at least one existing automobile engine management system, if the processor signals an error then the system reverts to its default settings by external *analogue* circuitry.

The second of the respectable modes, 'stop' mode, allows more control and containment of errors than does 'halt' mode. In stop mode all errant processes are mapped on to the process STOP. This will have the effect of gradually propagating the STOP process throughout the system. Although, at first sight, this does not seem very useful, it is possible for other parts of system to detect that one part has gone wrong, for example, by use of 'watchdog' timers. This allows multiply redundant systems, or gracefully degrading systems to be constructed.

6.9 Conclusions

The design of the OCCAM 2 programming language has been influenced by the need to ensure that programming errors are as difficult to make as possible and that when they are made they should be detectable. The properties of the data types in the language have been carefully specified to ensure that they are consistent with the semantics of OCCAM. The use of channel protocols makes possible the detection of many programming errors at compile-time and ensures that total security can be attained at run-time with little cost. The insistence that names are not aliased detects some particularly obscure programming errors and greatly simplifies checking the validity of parallel constructs.

6.10 References

- 1 *occam 2 Language Definition*, David May, INMOS Ltd 1987.
- 2 *occam 2 Reference Manual*, INMOS Ltd 1988.
- 3 *Compile-Time Detection of Aliasing in Euclid Programs*, James R Cordy, Software – Practice and Experience, Vol. 14(8) pp. 755-768.

7 IMS T800 architecture

7.1 Introduction

The INMOS transputer family is a range of system components each of which combines processing, memory and interconnect in a single VLSI chip. The first member of the family, the IMS T414 32-bit transputer [1], was introduced in September 1985, and has enabled concurrency to be applied in a wide variety of applications such as simulation, robot control, image synthesis, and digital signal processing. These numerically intensive applications can exploit large arrays of transputers; the system performance depending on the number of transputers, the speed of inter-transputer communication and the floating-point performance of each transputer.

The latest addition to the INMOS transputer family, the IMS T800, can increase the performance of such systems by offering greatly improved floating-point and communication performance. The IMS T800-20, available in the second half of 1987, is capable of sustaining over one and a half million floating-point operations per second; the IMS T800-30, available in the second half of 1988, is capable of sustaining over two and a quarter million floating-point operations per second. The comparative figure for the IMS T414 transputer is somewhat less than one hundred thousand floating-point operations per second.

The IMS T800 is pin-compatible with, and retains all the capabilities of, the established IMS T414 transputer. In addition, the IMS T800 incorporates an on-chip floating-point unit, novel instructions to support graphics, and twice the on-chip RAM of the IMS T414.

To minimise development time and risk, the design of the IMS T800 employs many of the component modules used in the IMS T414. The design of the floating-point unit makes extensive use of formal techniques, to ensure that each floating-point operation produces the correct result as specified by the IEEE 754 floating-point standard [2].

The design of the IMS T800 forms part of the P1085 European ESPRIT parallel computer architecture project [3]. The goal of this project is to develop a low-cost, high-performance super-computer, based on reconfigurable nodes of transputers. The intention is that single nodes (typically of 20 or so transputers) would be used as powerful workstations, and that up to 64 nodes could be connected together, offering a machine with a performance greatly in excess of one giga-flop. Within the project, software is being developed for applications in physics, engineering, CAD, CAM, and image processing.

7.2 The transputer: basic architecture and concepts

7.2.1 A programmable device

The transputer is a component designed to exploit the potential of VLSI. This technology allows large numbers of *identical* devices to be manufactured cheaply. For this reason, it is attractive to implement a concurrent system using a number of identical components, each of which is customised by an appropriate program. The transputer is, therefore, a VLSI device with a processor, memory to store the program executed by the processor, and communication links for direct connection to other transputers. Transputer systems can be designed and programmed using Occam (see section 7.11) which allows an application to be described as a collection of processes which operate concurrently and communicate through channels. The transputer can therefore be used as a building block for concurrent processing systems, with occam as the associated design formalism.

7.2.2 Processor and memory on a single chip

One important property of VLSI technology is that communication between devices is very much slower than communication within a device. In a computer, almost every operation that the processor performs involves the use of memory. For this reason a transputer includes both processor and memory in the same integrated circuit device.

7.2.3 Serial communication between transputers

In any system constructed from integrated circuit devices, much of the physical bulk arises from connections between devices. The size of the package for an integrated circuit is determined more by the number of connection pins than by the size of the device itself. In addition, connections between devices provided by paths on a circuit board consume a considerable amount of space.

The speed of communication between electronic devices is optimised by the use of one-directional signal wires, each connecting two devices. If many devices are connected by a shared bus, electrical problems of driving the bus require that the speed is reduced. Also, additional control logic and wiring are required to control sharing of the bus.

To provide maximum speed with minimal wiring, the transputer uses point-to-point serial communication links for direct connection to other transputers. The protocols used on the transputer links are discussed later.

7.2.4 Simplified processor with microcoded scheduler

The most effective implementation of simple programs by a programmable computer is provided by a sequential processor. Consequently, the transputer has a fairly conventional microcoded processor. There is a small core of about thirty-two instructions which are used to implement simple sequential programs. In addition there are other, more specialised groups of instructions which provide facilities such as long arithmetic and process scheduling.

As a process executed by a transputer may itself consist of a number of concurrent processes the transputer has to support the OCCAM programming model internally. The transputer, therefore, has a microcoded scheduler which shares the processor time between the concurrent processes. The scheduler provides two priority levels; any high-priority process which can run will do so in preference to any low-priority process.

7.2.5 Transputer products

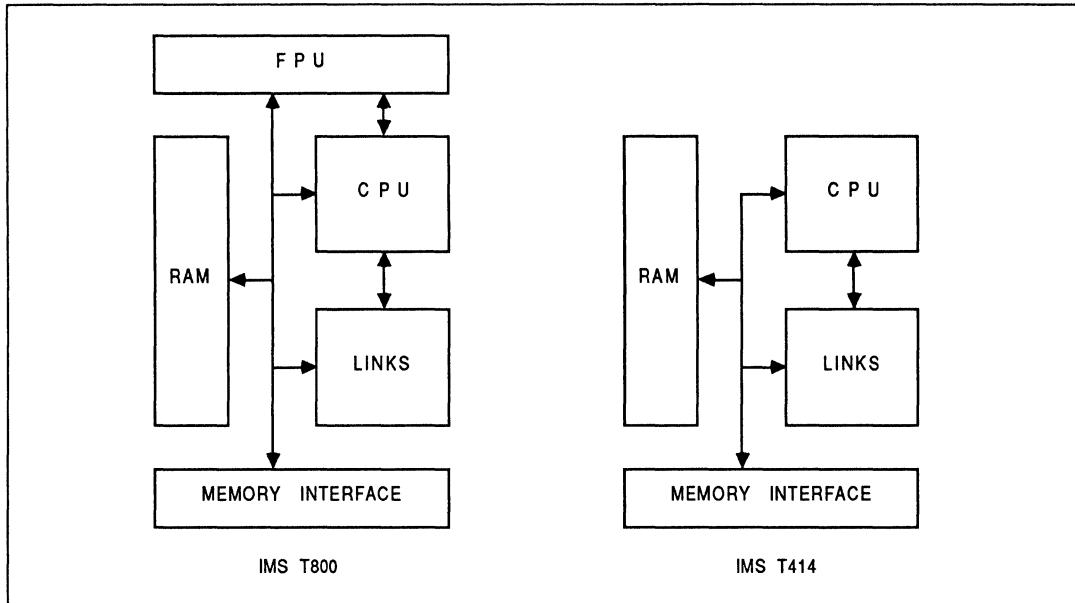
The first transputer to become available was the INMOS IMS T414. This has a 32-bit processor, 2 Kbytes of fast on-chip memory, a 32-bit external memory interface and four links for connection to other transputers. The current fastest available version of this product, the IMS T414-20, has a 50 nS internal cycle time, and achieves about 10 MIPS on sequential programs. The second transputer to become available was the IMS T212; this is very similar to the IMS T414 but has a 16-bit processor and 16-bit external memory interface. The remaining transputer in the family is the IMS M212 disk processor. This contains a 16-bit processor, RAM, ROM, two inter-transputer links and special hardware to control both winchester and floppy disks.

In addition the transputer family includes a number of transputer link related products. There are the 'link adaptors' which convert between handshaken 8-bit parallel data and INMOS link bit-serial data. These allow transputers to be connected to conventional, bus-based systems, and also allow conventional microprocessors to use transputer links as a system interconnect. In addition there is the IMS C004, which is a link exchange.

7.3 IMS T800 architecture

The IMS T800, with its on-chip floating-point unit, is only 20% larger in area than the IMS T414. The small size and high performance come from a design which takes careful note of silicon economics. This contrasts starkly with conventional co-processors, where the floating-point unit typically occupies more area than a complete micro-processor, and requires a second chip (or in the case of the Weitek 1167 floating-point processor for the Intel 80386, second, third and fourth chips).

The architecture of the IMS T800 is similar to that of the IMS T414. However, in addition to the memory, links, central processing unit (CPU), and external memory interface, there is a microcoded floating-point unit (FPU) which operates concurrently with and under the control of the CPU. The block diagrams opposite indicate the way in which the major blocks of the IMS T800 and IMS T414 are interconnected.



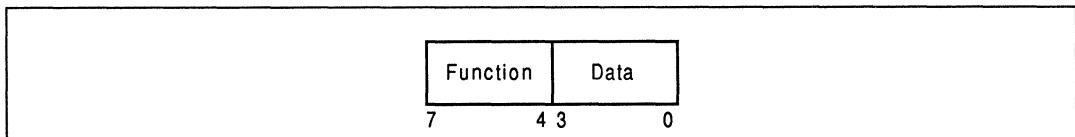
The CPU of the IMS T800, just like that of the IMS T414, contains three registers (A, B and C) used for integer and address arithmetic, which form a hardware stack. Loading a value into the stack pushes B into C, and A into B, before loading A. Storing a value from A pops B into A and C into B. In addition there is an O register which is used in the formation of instruction operands. Similarly, the FPU includes a three-register floating-point evaluation stack, containing the AF, BF, and CF registers. When values are loaded on to, or stored from the stack the AF, BF and CF registers push and pop in the same way as the A, B and C registers.

The addresses of floating-point values are formed on the CPU stack, and values are transferred between the addressed memory locations and the FPU stack under the control of the CPU. As the CPU stack is used only to hold the addresses of floating-point values, the wordlength of the CPU is independent of that of the FPU. Consequently, it would be possible to use the same FPU together with, for example, a 16-bit CPU such as that used on the IMS T212 transputer.

The IMS T800, like the IMS T414, operates at two priority levels. The FPU register stack is duplicated so that when the IMS T800 switches from low to high priority none of the state in the floating-point unit is written to memory. This results in a worst-case interrupt response of only 2.5 μ s (-30), or 3.7 μ s (-20). Furthermore, the duplication of the register stack enables floating-point arithmetic to be used in an interrupt routine without any performance penalty.

7.3.1 Instruction encoding

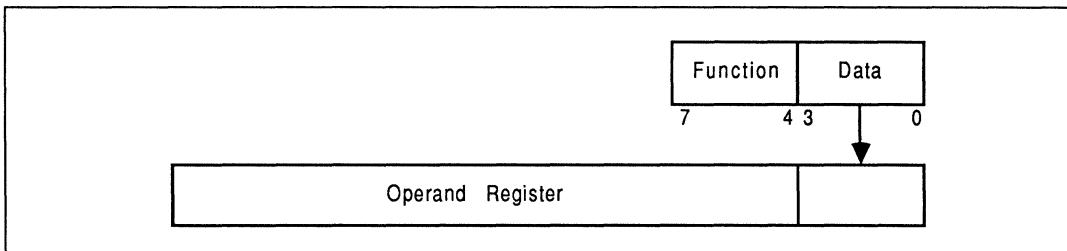
All transputers share the same basic instruction set. It contains a small number of instructions, all with the same format, chosen to give a compact representation of the operations most frequently occurring in programs. Each instruction consists of a single byte divided into two 4-bit parts.



The four most significant bits are a function code, and the four least significant bits are a data value. The sixteen functions include loads, stores, jumps and calls and enable the most common instructions to be

represented in a single byte. As this encoding permits only 4 bits of operand per instruction two of the function codes (*prefix* and *negative prefix*) are used to allow the data part of any instruction to be extended in length. Another of the sixteen functions (*operate*) treats its data portion as an operation on values held in the processor registers. This allows up to sixteen such operations to be encoded in a single byte instruction.

All instructions are executed by loading the four data bits into the least significant four bits of the O register, which is then used as the the instruction's operand. All instructions except the prefix instructions end by clearing the O register, ready for the next instruction.



The *prefix* instruction loads its four data bits into the O register, and then shifts the O register up four places. The *negative prefix* instruction is similar, except that it complements the operand register before shifting it up. Consequently operands can be extended to any length up to the length of the operand register by a sequence of prefix instructions.

The prefix functions can be used to extend the operand of an *operate* instruction just like any other. The instruction representation therefore provides for an indefinite number of operations. The encoding of operations is chosen so that the most common operations, such as *add* and *greater than*, are represented without a *prefix* instruction.

The IMS T800 has additional instructions which load into, operate on, and store from, the floating-point register stack. It also contains new instructions which support colour graphics, pattern recognition and the implementation of error-correcting codes. These instructions have been added whilst retaining the existing IMS T414 instruction set. This has been possible because of the extensible instruction encoding used in transputers.

7.3.2 Floating-point instructions

The core of the floating-point instruction set was established fairly early in the design of the IMS T800. This core includes simple load, store and arithmetic instructions. Examination of statistics derived from FORTRAN programs suggested that the addition of some more complex instructions would improve performance and code density. Proposed changes to the instruction set were assesed by examining their effect on a number of numerical programs. For each proposed instruction set, a compiler was constructed, the programs compiled with it, and the resulting code then run on a simulator. The resulting instruction set is now described.

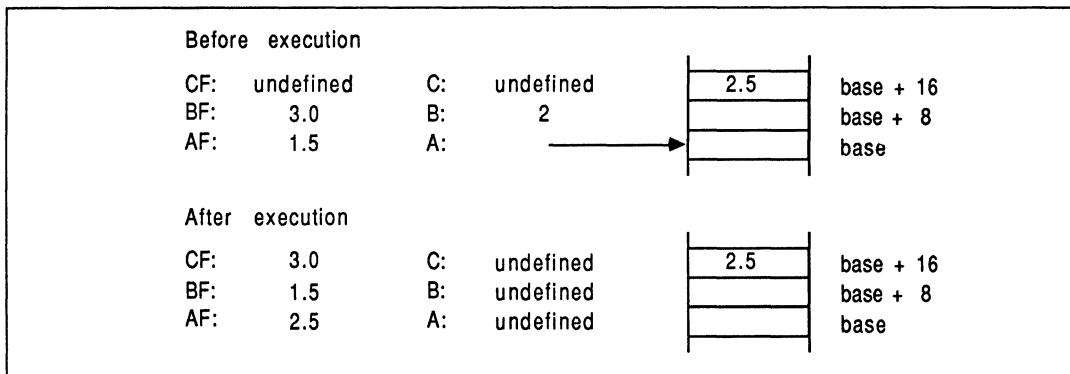
In the IMS T800, operands are transferred between the transputer's memory and the floating-point evaluation stack by means of floating-point load and store instructions. There are two groups of such instructions, one for single-length numbers, one for double-length. In the description of the load and store instructions, which follow, only the double-length instructions are described. However, there are single-length instructions which correspond with each of the double-length instructions.

The address of a floating-point operand is computed on the CPU's stack and the operand is then loaded, from the addressed memory location, on to the FPU's stack. Two new addressing operations have been added to the CPU to improve access to double-word (64-bit real and integer) values. The first of these, *word subscript double*, is used to index double-word values. The second of these, *duplicate*, is used when the CPU has to manipulate the addresses of both the more significant and less significant words of a double-word object.

Operands in the floating-point stack are tagged with their length. The operand's tag will be set when the operand is loaded or is computed. The tags allow the number of instructions needed for floating-point

operations to be reduced; there is no need, for example, to have both *floating add single* and *floating add double* instructions; a single *floating add* will suffice.

There are two instructions to load double-length floating-point numbers into the floating-point evaluation stack from the transputer's memory. These are *floating load non-local double* and *floating load indexed double*. The *floating load non-local double* instruction loads the value pointed to by the A register of the CPU's stack. The *floating load indexed double* instruction has the same effect as the instruction *word subscript double* followed by *floating load non-local double*. The value in the B register is used as a double-word offset from the base pointer in the A register and the selected double-length value is loaded into the AF register. The diagram below shows the effect of executing a *floating load indexed double* instruction.



The effect of the *floating load indexed* instructions can be achieved by a sequence of just two instructions. However, their presence does decrease code size; the *floating load indexed double* instruction is encoded in only two bytes, whereas the equivalent instruction sequence would require four bytes. This appears to be a worthwhile optimisation as this instruction sequence would be compiled for every array access.

However, there are just two floating store instructions, *floating store non-local single* and *floating store non-local double*. These both store the value in the AF register into the location pointed to by the A register. There are no *floating store indexed* instructions. This may be surprising given that the *floating load indexed* instructions exist; however, in any program there are less store operations than load operations and, therefore, there is less to be gained by optimising store (write to memory) operations than optimising load (read from memory) operations.

The common floating-point operations of addition, subtraction, multiplication and division are provided by single instructions. These instructions operate on values in the AF and BF registers, storing the result of the operation into the AF register and popping the CF register into the BF register. Similarly, the floating-point comparison operations, *floating-point greater than* and *floating-point equality*, compare values stored in the AF and BF registers; however, they load the result of the comparison into the A register of the CPU.

As an example, consider the following fragment of occam which sets a boolean variable **converged** to indicate whether the value of the 32-bit floating-point variable **absolute.error** is less than the value of the variable **epsilon**.

```

BOOL converged :
REAL32 absolute.error, epsilon :
SEQ
...
converged := absolute.error < epsilon
...

```

The compiled code for this fragment would be:

load local pointer	epsilon	<i>address of epsilon</i>
floating load non-local single		<i>load value into FPU</i>
load local pointer	absolute.error	<i>address of absolute.error</i>
floating load non-local single		<i>load value into FPU</i>
floating greater than		<i>result pushed on to CPU stack</i>
store local	converged	<i>store in converged</i>

There are four instructions which combine loading and operating. These exist, as do the *load indexed* instructions, to improve code compactness. The effect of the *floating load and add single* instruction is just the same as the sequence *floating load non-local single* followed by *floating add*. The remaining instructions complete the set needed to load and add or multiply single- and double-length values. The choice of optimising only addition and multiplication in this way reflects the high usage of these operators in programs.

7.3.3 Optimising use of the stack

The depth of the register stacks in the CPU and FPU is carefully chosen. Floating-point expressions commonly have embedded address calculations, as the operands of floating-point operators are often elements of one dimensional or two-dimensional arrays. The CPU stack is deep enough to allow most integer calculations and address calculations to be performed within it. Similarly, the depth of the FPU stack allows most floating-point expressions to be evaluated within it, employing the CPU stack to form addresses for the operands.

No hardware is used to deal with stack overflow. A compiler can easily examine expressions and introduce temporary variables in memory to avoid stack overflow. The number of such temporary variables can be minimised by careful choice of the evaluation order; an algorithm to perform this optimisation is given in [4]. The algorithm, already used to optimise the use of the integer stack of the IMS T414, is also used for the main CPU of the IMS T800.

7.3.4 Concurrent operation of FPU and CPU

In the IMS T800 the FPU operates concurrently with the CPU. This means that it is possible to perform an address calculation in the CPU whilst the FPU performs a floating-point calculation. This can lead to significant performance improvements in real applications which access arrays heavily. This aspect of the IMS T800's performance was carefully assessed, partly through examination of the 'Livermore Loops' (see section 7.12 and [5]). These are a collection of small kernels designed to represent the types of calculation performed on super-computers. They are of interest because they contain constructs which occur in real programs which are not represented in such programs as the Whetstone benchmark (see below). In particular, they contain accesses to two- and three-dimensional arrays, operations where the concurrency within the IMS T800 is used to good effect. In some cases the compiler is able to choose the order of performing address calculations so as to maximise overlapping; this involves a modification of the algorithm mentioned earlier.

As a simple example of overlapping consider the implementation of Livermore Loop 7 (see section 7.12). The IMS T800-30 achieves a speed of 2.25 Mflops on this benchmark; for comparison the IMS T800-20 achieves 1.5 Mflops, the T414-20 achieves 0.09 Mflops and a VAX 11/780 (with fpa) achieves 0.54 Mflops. The OCCAM program for loop 7 is as follows:

```
-- LIVERMORE LOOP 7
SEQ k = 0 FOR n
  x[k] :=  u[k] + ((( r*(z[k] + (r*y[k])) ) +
    (t*((u[k+3] + (r*(u[k+2] + (r*u[k+1])))))) ) +
    (t*((u[k+6] + (r*(u[k+5] + (r*u[k+4])))))) )
```

The following explains how this program fragment is executed on the IMS T800. The explanation assumes that the floating-point variable *r* and the floating-point arrays (*x*, *y*, *z*, and *u*) are located in a global data area and must be accessed via a static link, but that the loop count *k*, is in the process workspace. A compiler will generate code which first evaluates the subexpression *z[k] + (r*y[k])*.

The first stage in the computation of this is to load the value $y[k]$. The code to do this is:

```
load local           k
load local           static.link
load non-local pointer      y
floating load indexed single
```

The first load pushes the subscript k on to the CPU stack. The next load pushes the static link on to the CPU stack; the static link will contain a pointer to the base of the area of memory which contains the floating-point variables and arrays. The *load non-local pointer* instruction generates a pointer to the y th element of that area of memory; this will be the base of the array y . The CPU stack now has its A register containing a pointer to the base of the array y , and its B register containing the subscript, k . The *floating load indexed single* instruction pushes the single-length floating-point number stored in $y[k]$ on to the FPU stack.

The next segment of code pushes the value r on to the floating-point stack and multiplies the number r in AF by $y[k]$ in BF.

```
load local           static.link
load non-local pointer      r
floating load and multiply single
```

Although the floating-point multiplication takes several cycles to complete, the CPU is able to continue executing instructions whilst the FPU performs the multiplication. Thus the whole of the next segment of code can be executed whilst the multiplication is being performed.

```
load local           k
load local           static.link
load non-local pointer      z
word subscript
```

This code is similar to the first section of code illustrated above. However, it explicitly executes a *word subscript* to compute the address of $z[k]$; this allows the code following to use a *floating load and add single* instruction which saves 2 bytes of code.

Finally, the value $z[k]$ is pushed on to the floating-point stack and added to the previously computed subexpression $r*y[k]$. It is not until the value $z[k]$ is loaded that the CPU needs to synchronise with the FPU.

The computation of the remainder of the expression proceeds in the same way, and the FPU never has to wait for the CPU to perform an address calculation.

The overlapping of address calculation with floating-point computation is effective even when access is being made to multi-dimensional arrays. The IMS T800 retains the fast multiplication instruction (*product*) of the IMS T414 which is used for the multiplication implicit in multi-dimensional array access. This instruction executes in a time dependent on the highest bit set in its second operand.

For example, in the execution of the following fragment of occam:

```
[20] [20]REAL32 A :
SEQ
...
B := A[I, J] + (C / D)
...
```

loading the element **A[I, J]** involves computing the offset of the element from the base of the array **A**. The transputer compiler would generate the following code for this computation:

load local	I	<i>load I onto CPU stack</i>
load constant	20	<i>load 20 onto CPU stack</i>
product		<i>multiply I by 20</i>
load local	J	<i>load J onto CPU</i>
add		<i>add J to I * 20</i>

In this case the *product* instruction will execute in only 8 cycles (267 nS (-30), or 400 nS (-20)) and the whole address calculation will take 19 cycles which would be overlapped with the execution of the division **C / D**. Effectively, the overlapping allows the array accessing to be performed in only one cycle.

7.4 Floating-point unit design

In designing a concurrent systems component such as the IMS T800, it is important to maximise the performance obtained from a given area of silicon; many components can be used together to deliver more performance. This contrasts with the design of a conventional coprocessor where the aim is to maximise the performance of a single processor by the use of a large area of silicon. (Interestingly, however, the IMS T800-20 achieves similar performance to the 80386 with its Weitek 1167 coprocessor chip set.) As a result, in designing the IMS T800, the performance benefits of silicon-hungry devices such as barrel shifters and flash multipliers were carefully examined.

A flash multiplier is too large to fit on chip together with the processor, and would therefore necessitate the use of a separate coprocessor chip. The introduction of a coprocessor interface to a separate chip slows down the rate at which operands can be transferred to and from the floating-point unit. Higher performance can, therefore, be obtained from a slow multiplier on the same chip as the processor than from a fast one on a separate chip. This leads to an important conclusion: *a separate coprocessor chip is not appropriate for scalar floating-point arithmetic*. A separate coprocessor would be effective where a large amount of work can be handed to the coprocessor by transferring a small amount of information; for example a vector coprocessor would require only the addresses of its vector operands to be transferred via the coprocessor interface.

It turns out that a flash multiplier also operates much more quickly than is necessary. Only a pipelined vector processor can deliver operands at a rate consistent with the use of such devices. In fact, any useful floating-point calculation involves more operand accesses than operations. As an example consider the assignment **y[i] := y[i] + (t * x[i])** which constitutes the core of the LINPACK floating-point benchmark. To perform this it is necessary to load three operands, perform two operations and to store a result. If we assume that it takes twice as long to perform a floating-point operation as to load or store a floating-point number then the execution time of this example would be evenly split between operand access time and operation time. This means that there would be at most a factor of two available in performance improvement from the use of an infinitely fast floating-point unit!

Unlike a flash multiplier, a fast normalising shifter is important for fast floating-point operation. When implementing IEEE arithmetic it may be necessary to perform a long shift on every floating-point operation and unless a fast shifter is incorporated into the floating-point unit the maximum operation time can become very long. Fortunately, unlike a flash multiplier, it is possible to design a fast shifter in a reasonable area of silicon. The shifter used in the IMS T800 is designed to perform a shift in a single cycle and to normalise in two cycles.

Consequently, the floating-point unit of the IMS T800 contains a fast normalising shifter but not a flash multiplier. However, there is a certain amount of logic devoted to multiplication and division. Multiplication

is performed 3-bits per cycle, and division is performed 2-bits per cycle. This gives rise to a single-length multiplication time of 13 cycles (367 nS (-30), or 550 nS (-20)) and a double-length divide time of 34 cycles (1.07 μ S (-30), or 1.6 μ S (-20)).

One other aspect of floating-point arithmetic which was carefully examined was the implementation of standard scientific functions (*sqrt*, *sin*, etc). Trigonometric functions are generally implemented by algorithms which make use of an approximation which is only accurate over a small part of the function's domain. This is possible because mathematical identities enable the full function to be computed from the partial approximation. Algorithms differ in the way in which they compute the approximation; two methods of computing the approximation were considered during the design of the IMS T800.

The first method is called CORDIC which was developed for hardware implementation and is used in some floating-point coprocessor chips such as the i8087. This requires the addition of significant quantities of hardware into the datapath and the storage of large look-up tables. Even with this hardware the best performance which could be achieved would be to generate one bit of result every four cycles, resulting in a minimum evaluation time for the reduced function of about 230 cycles (double length).

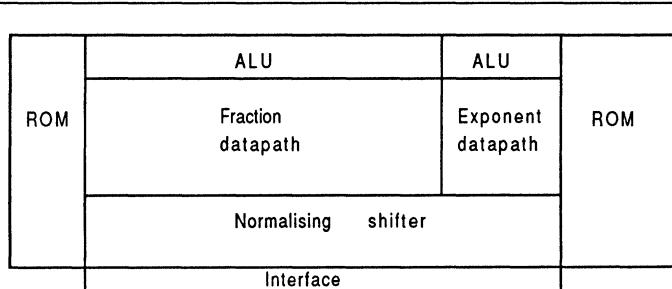
The second method is polynomial approximation. This requires no additional hardware in the FPU. The evaluation time will vary from function to function, but for the *sin* function only about twice the number of cycles required by CORDIC would be used since the implementation of multiplication generates three bits of product every one cycle.

In practice we are interested in the evaluation time for the function proper, not just the reduced function. Once the time for argument reduction and function generation have been added to the time for the evaluation of the partial function it is clear that there is no possibility of a CORDIC based implementation being even twice as fast as polynomial-based implementation. For this reason the IMS T800 contains no special support for trigonometric function evaluation.

This comparison can also be extended to show that if a processor with even faster trigonometric function evaluation were required it should be achieved by increasing the speed of the processor's multiplication. This would have the additional benefit that it would increase performance on virtually all applications, not just those which make heavy use of trigonometric functions.

The situation is rather different with regard to the square-root function. Here the IEEE standard requires that the result is produced correct to the last bit and this is not easy to achieve by simple polynomial evaluation. Furthermore it only requires a small amount of additional hardware to perform square root in hardware and this has been done in the IMS T800.

The block diagram below illustrates the physical layout of the floating-point unit.



Block diagram of floating-point unit

The datapaths contain registers and shift paths. The fraction datapath is 59 bits wide, and the exponent datapath is 13 bits wide. The normalising shifter interfaces to both the fraction datapath and the exponent datapath. This is because the data to be shifted will come from the fraction datapath whilst the magnitude of the shift is associated with the exponent datapath. One further interesting aspect of the design is the

microcode ROM. Although the diagram shows two ROMs, they are both part of the same logical ROM. This has been split in two so that control signals do not need to be bussed through the datapaths.

7.5 Floating-point performance

The IMS T414 has microcode support for 32-bit floating-point arithmetic which gives it performance comparable with the current generation of floating-point coprocessors. It achieves an operation time of about 10 microseconds on single-length IEEE 754 floating-point numbers. The IMS T800-20 betters the floating-point operation speed of the IMS T414 by more than an order of magnitude; its operation times are shown below

operation	IMS T800-30		IMS T800-20		IMS T414-20	
	single	double	single	double	single	double
add	233 nS	233 nS	350 nS	350 nS	11.5 μ S	28.3 μ S
subtract	233 nS	233 nS	350 nS	350 nS	11.5 μ S	28.3 μ S
multiply	367 nS	667 nS	550 nS	1000 nS	10.0 μ S	38.0 μ S
divide	567 nS	1067 nS	850 nS	1600 nS	12.3 μ S	55.75 μ S

The operation time is not a reliable measure of performance on real numerical programs. For this reason, floating-point performance is often measured by the Whetstone benchmark. The Whetstone benchmark provides a good mix of floating-point operations, and also includes procedure calls, array indexing and transcendental functions. It is, in some senses, a 'typical' scientific program.

The performance of the IMS T414 and IMS T800 compared with other processors as measured by the Whetstone benchmark is shown below:

Processor	Whetstones/second single length
Intel 80286/80287	8 MHz
IMS T414-20	20 MHz
NS 32332-32081	15 MHz
MC 68020/68881	16/12 MHz SUN 3
VAX 11/780 FPA	UNIX 4.3 BSD
IMS T800-20	20 MHz
IMS T800-30	30 MHz
	300K
	663K
	728K
	755K
	1083K
	4000K
	6000K

This table shows that although the IMS T414 has an operation time three times slower than the MC68881 coprocessor it performs only 25% worse than the MC68020 + MC68881 coprocessor (as measured by the Whetstone benchmark). This is because the speed of evaluating a floating-point expression depends on two factors; the speed at which operands are transferred to and from the floating-point unit and the speed of the unit itself. By careful balancing of these the single chip IMS T800-20 achieves more than five times the Whetstone performance of the MC68020/MC68881 combination.

Another important measure is the performance obtained from a given area of silicon. For example, four IMS T800-30 chips occupy an area similar to the i80386 together with the Weitek 1167 chip set, and on single-length floating-point will deliver six times the performance in any concurrent application. In terms of circuit board area, the effect is even more dramatic; the IMS T800 requires negligible support circuitry and can even be used without external memory.

7.6 Formal methods ensure correctness and quick design

One of the concerns of engineers designing microprocessors into life-critical systems is the correctness of the implementation of those microprocessors. The complexity of a floating-point unit is such that it is impossible to validate by exhaustive testing. The approach which INMOS has taken is to make use of the most advanced formal methods to ensure the correct implementation of the IEEE 754 standard for floating-point arithmetic. This work has been undertaken in cooperation with the Programming Research Group at Oxford University

and has made use of the formal semantics of the occam programming language. INMOS has found that the use of formal methods in complex designs greatly decreases design time as well as ensuring correctness.

The specification language Z (see section 7.13) was used extensively during the design of the IMS T800, both to express the IEEE 754 standard mathematically and to specify instructions precisely. The first stage of implementation was to write a software package in the occam language and to prove that it met the specification. (This package is used to provide floating-point arithmetic for various occam implementations, including that for the IMS T414.) Using an interactive program transformation system, the occam package was then transformed into the microcode for the IMS T800.

This design process is illustrated using one instruction from the sequence of instructions executed by the IMS T800 to perform floating-point to integer conversions. This instruction occurs in the middle of the sequence, after the floating-point number has been rounded into an integer in floating-point format. The instruction checks that the rounded value lies within the range of numbers representable as an integer and, if not, sets the error flag.

7.6.1 Z specification

The precise specification of range checking is expressed in Z as:

$\text{Floating.Check_Integer_Range}$
$\text{Areg, Areg' : Floating_Point_Register}$ $\text{Error_Flag, Error_Flag' : bool}$
$\text{fv} \in \mathbb{Z}$ $\text{Areg}' = \text{Areg}$ $\text{fv Areg} \in [\text{MinInt}, \text{MaxInt}] \Rightarrow \text{Error_Flag}' = \text{Error_Flag}$ $\text{fv Areg} \notin [\text{MinInt}, \text{MaxInt}] \Rightarrow \text{Error_Flag}' = \text{true}$

In this specification the primed variables Areg' and Error_Flag' denote the values of registers after the operation, and the unprimed variables denote the values before. Maxint and Minint are constants defined by the integer format and fv is a function that returns the value of a floating-point register. The predicates state that the operation is only defined when Areg contains an 'integer' value, that Areg is unchanged by the operation and that Error_Flag is set if Areg lies outside the storable integer range and is unchanged otherwise.

7.6.2 High-level occam implementation

The high-level occam implementation is as shown below. Its correctness depends on proving two assertions. Firstly, that there is an exponent, LargestINTExp, such that every floating-point register with a smaller exponent lies in [MinInt,MaxInt], and secondly, that a register with a negative sign bit, an exponent equal to LargestINTExp and a fraction with only the implied msb set, has an fv of MinInt.

```

IF
  Areg.Exp < LargestINTExp
    SKIP
  (Areg.Sign = NEGATIVE) AND (Areg.Exp = LargestINTExp) AND
    (Areg.Frac = MSBit)
    SKIP
  TRUE
    SetError (ErrorFlag)

```

In the above code, the occam variable **Areg.Exp** is used to represent the contents of the exponential part of the FA register of the FPU. Similarly, **Areg.Sign** and **Areg.Frac** represent the sign bit and fraction part. This code first checks to see if the exponent is smaller than **LargestINTExp**; if it is then the value in the FA register is in range and no further action is to be performed. Otherwise, the code checks if the value in the FA register has a negative sign bit, an exponent equal to **LargestINTExp** and a fraction with only the most significant bit set; if it does then, again, no further action is performed. Otherwise, the value in the FA register is out of range and the error flag is set.

7.6.3 Low-level OCCAM implementation

The program above can be transformed using the laws of OCCAM. First the condition on **Areg.Sign** is pulled to the outside. Then the program is transformed into processes and variables defined in terms of operations found in the floating-point microcode. This involves the use of register and bus operations to perform the comparisons together with explicit tests of the resulting flags. At this stage the processes are also grouped into the sequences of operations that form each microinstruction. For brevity the negative case is omitted in this illustration:

```

SEQ
  AregSignNEGATIVE := (Areg.Sign = NEGATIVE)
  ExpZbus := (Areg.Exp - LargestINTExp)
  ExpZbusNeg := ExpZbus < 0
  IF
    AregSignNEGATIVE
      ... negative case
    NOT AregSignNEGATIVE
    IF
      ExpZbusNeg
        SKIP
      NOT ExpZbusNeg
        SetError(ErrorFlag)
  
```

7.6.4 Flattened low-level implementation

The low-level OCCAM implementation is then transformed into a ‘flattened’ form that makes explicit use of a microinstruction pointer. This form uses a **WHILE** loop and explicit testing of the next instruction register (**NextInst**) to simulate the sequencing of the microcode. If the resulting microcode involves no loops it is possible to transform it back into the original form mechanically. In the program below the **SetError(ErrorFlag)** process has been moved into a separate microinstruction, **OutOfRange**.

```

INT NextInst :
SEQ
  NextInst := FloatingPointCheckIntegerRange
  WHILE NextInst <> NOWHERE
    IF
      NextInst = FloatingPointCheckIntegerRange
      SEQ
        AregSignNEGATIVE := (Areg.Sign = NEGATIVE)
        ExpZbus := (Areg.Exp - LargestINTExp)
        ExpZbusNeg := ExpZbus < 0
        IF
          AregSignNEGATIVE
            ... negative case
          NOT AregSignNEGATIVE
          IF
            ExpZbusNeg
              NextInst := NOWHERE
            NOT ExpZbusNeg
              NextInst := OutofRange
        NextInst = OutofRange
        SEQ
          SetError(ErrorFlag)
          NextInst := NOWHERE
        ... negative case micro instructions
  
```

7.6.5 Microcode

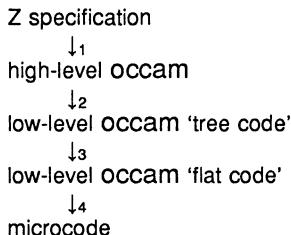
The flattened occam code is then transformed into microcode assembler. This is done by a pattern-matching and textual substitution program. Without the use of mechanical assistance this is a very laborious and error-prone task. The program below shows the microcode which results from translating the **FloatingPointCheckIntegerRange** microinstruction of the previous example.

```
FloatingPointCheckIntegerRange:
ConstantLargestINTExp
ExpXbusFromAreg           ExpYbusFromConstant
ExpZbusFromXbusMinusYbus

GOTO Cond1FromAregSign ->
    (Cond0FromExpZbusNeg -> (... , ...),
     Cond0FromExpZbusNeg -> (NOWHERE, OutofRange))
```

7.6.6 Summary

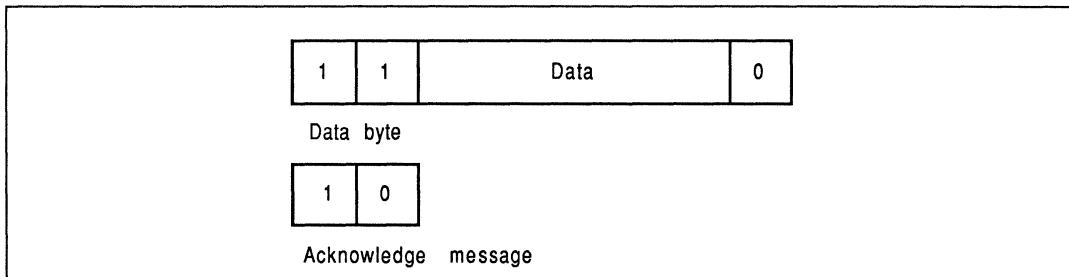
The use of the high-level specification language Z provides short and precise specifications of instructions, and, being mathematically based, avoids the problems of interpreting natural language specifications. This specification can be implemented fairly naturally, at a high level, in occam. This implementation can be proved correct, using occam's denotational semantics. The algebraic semantics of OCCAM then allow the occam to be transformed into a form that corresponds to the microcode.



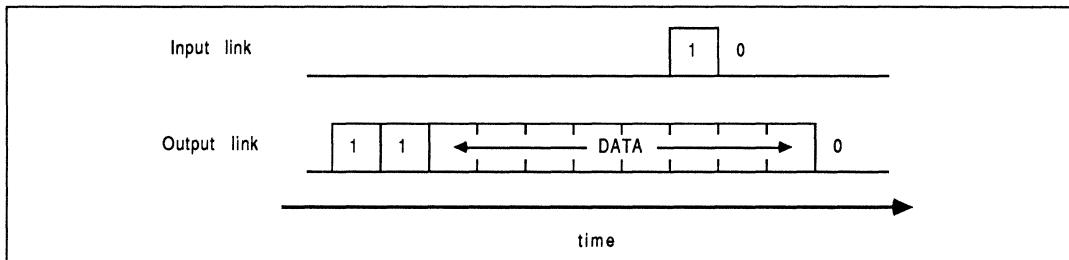
Each of steps 1 to 3 can be proven correct using the formal semantics of occam. The translation and compilation of step 4 could also be proved correct. In practice, both steps 1 and 2 were performed backwards; that is, an implementation was written and then transformed back into the previous specification. This process made use of an occam source transformation system, written in ML and implemented by the Programming Research Group at Oxford University. Steps 3 and 4 are performed semimechanically by programs. Although these have not been formally proved, their use is more reliable than doing the same work by hand; computers do not mistakenly miss out lines of microcode due to boredom!

7.7 Communication links

A link between two transputers is implemented by connecting a link interface on one transputer to a link interface on the other transputer by two one-directional signal wires, along which data is transmitted serially. The two wires provide two occam channels, one in each direction. This requires a simple protocol to multiplex data and control information. Messages are transmitted as a sequence of bytes, each of which must be acknowledged before the next is transmitted. A byte of data is transmitted as a start bit followed by a one bit followed by eight bits of data followed by a stop bit. An acknowledgement is transmitted as a start bit followed by a stop bit. An acknowledgement indicates both that a process was able to receive the data byte and that it is able to buffer another byte.



The protocol permits an acknowledgement to be generated as soon as the receiver has identified a data packet. In this way the acknowledgement can be received by the transmitter before all of the data packet has been transmitted and the transmitter can transmit the next data packet immediately. The IMS T414 transputer does not implement this overlapping and achieves a data rate of 0.8 Mbytes per second using a link to transfer data in one direction. However, by implementing the overlapping and including sufficient buffering in the link hardware, the IMS T800 more than doubles this data rate to 1.8 Mbytes per second in one direction, and achieves 2.4 Mbytes per second when the link carries data in both directions. The diagram below shows the signals that would be observed on the two link wires when a data packet is overlapped with an acknowledgement.



7.8 Graphics capability

The 'bit-blit' operations of a conventional graphics processor no longer seem appropriate in these days of byte (or greater) per pixel colour displays. The fast block move of the IMS T414 make it suitable for use in graphics applications using byte-per-pixel colour displays. Indeed, the IMS B007 colour graphics evaluation board uses it in such a manner.

The block move on the IMS T414 is designed to saturate the memory bandwidth, moving any number of bytes from any byte boundary in memory to any other byte boundary using the smallest possible number of word read and write operations. Using the transputer's internal memory the block move sustains a transfer rate of 60 Mbytes per second (-30), or 40 Mbytes per second (-20); using the fastest possible external memory the block move sustains 20 Mbytes per second (-30) or 13.3 Mbytes per second (-20).

The IMS T800 extends this capability by incorporation of a two-dimensional version of the block move which can move windows around a screen at full memory bandwidth, and conditional versions of the same block move which can be used to place templates and text into windows. One of these operations copies bytes from source to destination, writing only non-zero bytes to the destination. A new object of any shape can therefore be drawn on top of the current image. All of these instructions achieve the speed of the simple IMS T414 move instruction, enabling a 1 million pixel screen to be drawn thirteen times per second.

7.8.1 Instruction description

The three new instructions are concerned with moving a two-dimensional block of data from source to destination. The instructions differ in how the source is used to modify the destination. Unlike the conventional 'bit-blit' instruction, it is never necessary to read the destination data.

The instructions are described in occam:

```
PROC Move2d ([][]BYTE Source, sx, sy,
              [][]BYTE Dest, dx, dy,
              width, length)
  SEQ y = 0 FOR length
    [Dest[y+dy] FROM dx FOR width] :=
      [Source[y+sy] FROM sx FOR width]
```

This moves a block of size **width** × **length** which starts at byte **Source** [**sy**] [**sx**] to the block starting at byte **Dest** [**dy**] [**dx**].

```
PROC Draw2d ([][]BYTE Source, sx, sy,
              [][]BYTE Dest, dx, dy,
              width, length)
  BYTE temp:
  SEQ line = 0 FOR length
    SEQ point = 0 FOR width
      SEQ
        temp := Source[line+sy] [point+sx]
        IF
          temp = 0(BYTE)
            SKIP
          TRUE
            Dest[line+dy] [point+dx] := temp
```

This moves a block of size **width** × **length** which starts at byte **Source** [**sy**] [**sx**] to the block starting at byte **Dest** [**dy**] [**dx**]. However for every byte transferred a check is made to see if it is zero. If this is the case then the byte is not copied, and the destination remains unaltered.

```
PROC Clip2d ([][]BYTE Source, sx, sy,
              [][]BYTE Dest, dx, dy,
              width, length)
  BYTE temp:
  SEQ line = 0 FOR length
    SEQ point = 0 FOR width
      SEQ
        temp := Source[line+sy] [point+sx]
        IF
          temp = 0(BYTE)
            Dest[line+dy] [point+dx] := temp
          TRUE
            SKIP
```

This moves a block of size **width** × **length** which starts at byte **Source** [**sy**] [**sx**] to the block starting at byte **Dest** [**dy**] [**dx**]. However, for every byte transferred a check is made to see if it is zero. If this is the case then that byte is copied.

Draw2d and **Clip2d** are complementary and are used for the copying of irregular shapes on to the screen and the creation of templates.

Like the transputer's one-dimensional block move, the **Move2d**, **Draw2d** and **Clip2d** instructions move data from any byte address in memory to any byte address using the smallest possible number of single-word transfers. When executing a **Draw2d** operation, data is written in whole words, and hardware is used to suppress the generation of individual byte write signals corresponding to zero bytes in the source. Further, the write cycle is omitted completely if all bytes in the source word are found to be zero. **Clip2d** is similarly implemented using the smallest number of word read and write operations. Consequently, **Draw2d** and **Clip2d** normally operate faster than simple moves.

Move2d, **Draw2d** and **Clip2d** are not restricted to operations on single byte pixels. For example, 3 byte (24 bit) pixels can be treated in exactly the same way as single byte pixels, with a zero pixel being represented

by three zero bytes, and non-zero pixels being represented by three non-zero pixels. Pixels less than a byte can be implemented by omitting unnecessary bit planes from the video memory. By regarding an image as a two-dimensional array of pixels, each of which is itself an array of bytes, it is possible to use the same graphics software on systems with differing pixel sizes.

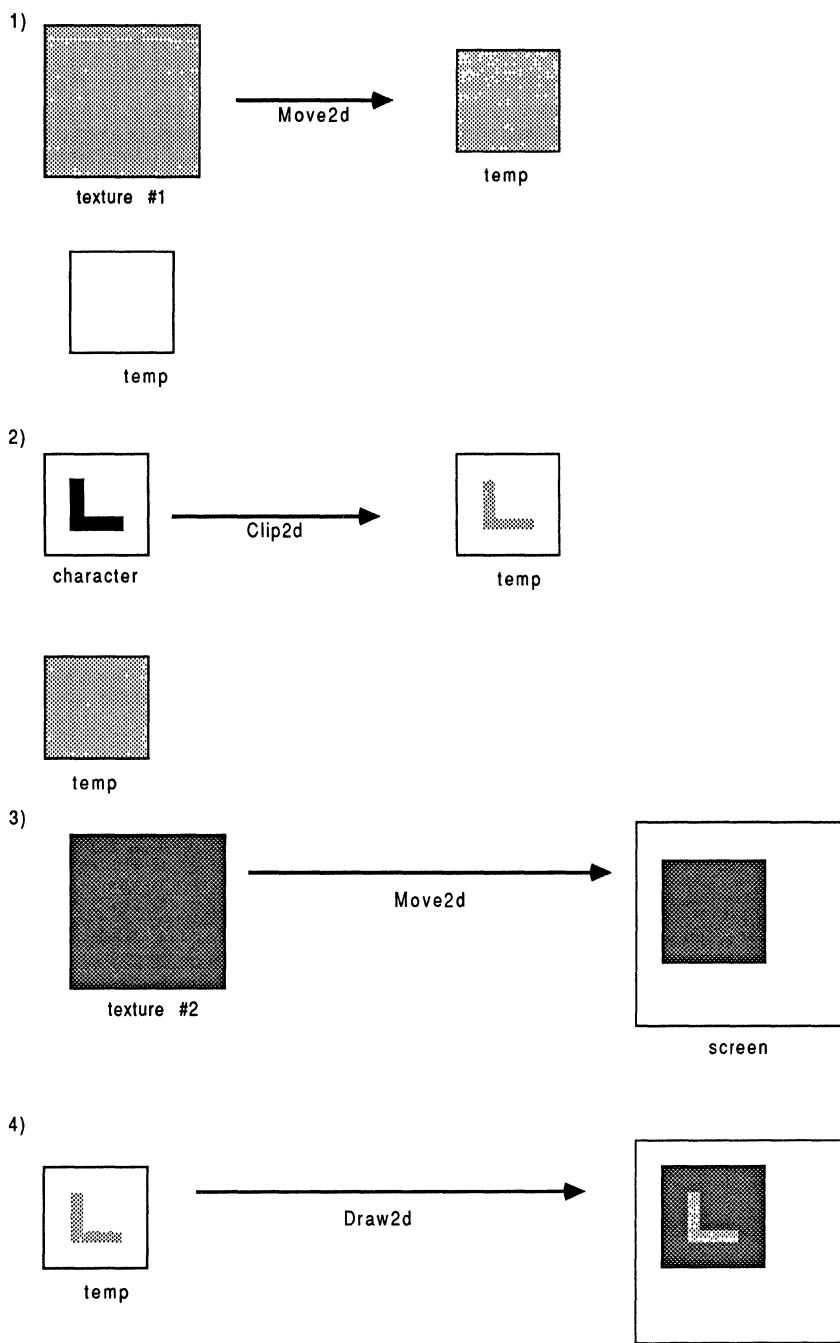
7.8.2 Drawing coloured text

Drawing proportional spaced text provides a simple example of the use of the IMS T800 instructions. The font is stored in a two-dimensional array **Font**; the height of **Font** is the fixed character height, and the start of each character is defined by an array **start**. The textures of the character and its background are selected from an array of textures; the textures providing a range of colours or even stripes and tartans!

An occam procedure to perform such drawing is given below and the effect of each stage in the drawing process is illustrated by the diagrams opposite. First, (1) the texture for the character is selected and copied to a temporary area and (2) the character in the font is used to clip this texture to the appropriate shape. Then (3) the background texture is selected and copied to the screen, and (4) the new character is drawn on top of it.

```
-- Draw character ch in texture F on background texture B
PROC DrawChar(VAL INT Ch, F, B)
    SEQ
        IF
            (x + width[ch]) > screenwidth
                SEQ
                    x := 0
                    y := y + height
            (x + width[ch]) <= screenwidth
                SKIP
        [height] [maxwidth] BYTE Temp :
    SEQ
        Move2d(Texture[F], 0, 0, Temp, 0, 0, width[ch], height)
        Clip2d(Font[ch], start[ch], 0, Temp, 0, 0, width[ch], height)
        Move2d(Texture[B], 0, 0, Screen, x, y, width[ch], height))
        Draw2d(Temp, 0, 0, Screen, x, y, width[ch], height)
        x := x + width[ch]
```

This procedure will fill a 1 million pixel screen with proportionally spaced characters in about 1/6 second. Obviously, a simpler and faster version could be used if the character colour or background colour was restricted. The operation of this procedure is illustrated on the next page.



7.9 Conclusions

The IMS T800 floating-point transputer provides a very high-performance building block for concurrent systems. The design of the IMS T800 demonstrates that it is not desirable to use coprocessors to achieve high-performance floating-point capability. The careful consideration of silicon economics has enabled the IMS T800 to incorporate a floating-point unit, a central processing unit, memory and a communication system in a single device; it is a complete scientific computer on a single chip. For example, the 4 Kbytes of on-chip memory allows the IMS T800 to be used, without external memory, in a number of signal processing applications. The fact that the floating-point performance of the IMS T800 exceeds its fixed-point performance on multiply-accumulates removes the need to design algorithms which use fixed-point arithmetic.

The IMS T800 forms the basis of the most powerful super-computer in Europe, currently under construction at Edinburgh University. This will contain 1000 transputers operating on one giga-byte of main store and should be operational by April 1988. Whilst this may seem to be a very large machine, the continuing improvement in VLSI technology means that such a machine will occupy only a few cubic feet in the early 1990s. Even today, using conventional packaging and printed circuit board technology, machines built from the IMS T800-20 can achieve a 'performance density' of 1.5 Gflop per cubic foot.

7.10 References

- 1 *The Transputer Databook*, INMOS Ltd 1989.
- 2 *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI IEEE Std 754-1985.
- 3 *Phase 1 of the development and application of a low cost, high performance multiprocessor machine*, J G Harp et al, ESPRIT 1986: Results and Achievements, pp. 551-562, Elsevier Science Publishers B V.
- 4 *The Transputer Instruction Set - A Compiler Writers' Guide*, INMOS Ltd, Prentice Hall 1988.
- 5 *The Livermore Fortran Kernels: A Computer Test of the Numerical Performance Range*, Frank H McMahon, Lawrence Livermore National Laboratory, UCRL-53745.

7.11 Note on OCCAM

It is not possible to give a comprehensive description of OCCAM in the space available. However, the following overview explains the basic concepts of the language and explains those details which are required in order to understand the examples in the paper.

The OCCAM programming language was developed to allow concurrent, *distributed*, systems to be programmed. The emphasis is placed on distributed because it was for this area that previous languages are unsuited. The OCCAM language enables a system to be described as a collection of concurrent processes, which communicate with each other and with peripheral devices through channels. The concurrent processes do not communicate via shared variables, and thus OCCAM is a suitable language for programming systems where there is no store which is shared between processors in the system.

OCCAM programs are built from three primitive processes:

```
v := e    assign expression e to variable v
c ! e    output expression e to channel c
c ? v    input from channel c to variable v
```

The primitive processes are combined to form constructs:

SE quential	components executed one after another
P ARallel	components executed together
A Lternative	component first ready is executed

A construct is itself a process, and may be used as a component of another construct. The syntax of

OCCAM uses indentation to indicate program structure, thus the OCCAM program below consists of two parallel processes. The first process inputs from the channel **source** into **next.problem**. The second process itself consists of two processes to be executed sequentially. The first is an instance of the procedure **compute.next.solution**, and the second, which is executed after the first has terminated, outputs **solution** on to channel **result**.

```
PAR
  source ? next.problem
  SEQ
    compute.next.solution(this.problem, solution)
    result ! solution
```

Conventional sequential programs can be expressed in OCCAM with variables and assignments, combined in sequential constructs. **IF** and **WHILE** constructs are also provided. The **IF** construct test a number of conditions in sequence; when one is found to be true, the associated process is executed. The example below shows how this might be used to compare two numbers, **a** and **b**, and to record their order.

```
IF
  a > b
    order := gt
  a < b
    order := lt
  TRUE
    order := eq
```

Concurrent programs can be expressed with channels, inputs and outputs, which are combined in parallel and alternative constructs.

Each OCCAM channel provides a communication path between two concurrent processes. Communication is synchronised and takes place when both the inputting process and the outputting process are ready. The data to be output is then copied from the outputting process to the inputting process, and both processes continue.

An alternative process may be ready for input from any one of a number of channels. In this case, the input is taken from the channel which is first used for output by another process.

Although the first version of OCCAM (as described in the occam Programming Language) had only a single data type and only one-dimensional arrays, the version of the language used in this paper, OCCAM 2, supports several data types and multi-dimensional arrays. Arrays may be assigned, communicated between processes and passed as parameters to procedures. OCCAM permits a subarray of an array to be used as an array. For example, the following program declares a 10 element array of integers, **a**, and then, in parallel inputs to the first 5 elements of **a** from the channel **c**, and to the second 5 elements from the channel **d**. (Note that in OCCAM the first element of an array is element 0.)

```
[10] INT a :
PAR
  c ? [a FROM 0 FOR 5]
  d ? [a FROM 5 FOR 5]
```

One further feature of OCCAM which requires explanation is the replicated constructor. The examples in the paper are all of replicated SEQs which have a similar effect to a FOR loop in a conventional language. The replicated SEQ:

```
SEQ i = base FOR count
  a[i] := i
```

is implemented as a loop and is equivalent to the following:

```
SEQ
  a[base]           := base
  a[base + 1]       := base + 1
  .....
  a[base + count - 1] := base + count - 1
```

7.12 Note on the 'Livermore Loops'

The Livermore Fortran Kernels [5] (commonly known as the Livermore Loops) are a set of 24 computation kernels designed to measure realistic floating-point performance on FORTRAN applications. They differ from a number of other standard benchmark programs in that they do not produce a single figure of merit, but a set of figures, one for each kernel. They represent a useful source of information about the structure of scientific programs, and as such, were studied during the design of the IMS T800.

Livermore Loop 7, mentioned in this paper, is an 'equation of state' fragment. The FORTRAN code for this loop is:

```
DO 7 k= 1,n
  X(k)=      U(k    ) + R*( Z(k    ) + R*Y(k    )) +
  .          T*( U(k+3) + R*( U(k+2) + R*U(k+1)) +
  .          T*( U(k+6) + R*( U(k+5) + R*U(k+4))))+
7 CONTINUE
```

The program in the paper is written in occam 2 and it is for this version that the code and performance figure is given. The implementation of OCCAM and FORTRAN will differ slightly as the two languages allocate store differently.

7.13 Note on the formal specification language Z

The specification notation Z has been developed to tackle the problems of specifying actual systems. Z originated with Jean-Raymond Abrial and has been developed and used extensively by members of the Programming Research Group, Oxford University.

A Z specification consists of a combination of a formal text and a natural language description. This formal text provides the precise specification while the natural language text introduces and explains the formal parts. The formal text has two parts: the schema language, which provides a means of structuring the specification, and the mathematical language, which allows for the preciseness of the specification. The mathematical language is based largely on set theory and enables an abstract mathematical view of the objects being specified to be taken. The schema language enables specifications of large systems to be broken into more manageable sections.

The combination of natural language for explanation, and the schema language produces specifications that are more readable than pure mathematics. In addition, the mathematical nature of the specifications enables implementors to use mathematical proofs to ensure the correspondence of their implementations with the specification.

The formal part of a Z specification makes use of 'schemas'. The schema consists of two 'boxes'. The top box contains the *signature* which introduces the variables of the specification. The lower box contains a list of predicates which constrain the values that the variables may take.

The following is an example of a Z specification:

If the reset signal is set then the **Count** operation sets the register to 0, otherwise it increments the register.

Count	
value, value'	: N
reset?	: bool
reset? \implies	value' = 0
\neg reset? \implies	value' = value + 1

This schema, named **Count**, introduces three variables; value, value' and reset. Conventionally, primed variables such as value' represent values of state variables after an operation, while their unprimed counterparts represent the values before the operation. Variables with names ending in '?', such as reset, are conventionally inputs to an operation. The values are constrained by two predicates (conventionally these are and-ed together, unless explicitly written otherwise). These predicates formally specify the behaviour of the operation described in the informal text that precedes the schema. This schema gives a precise specification of the operation; what it has not done is to dictate how the counter is implemented (number of bits etc.) as these are implementation details.

8 The role of occam in the design of the IMS T800

8.1 Introduction

Recent research has demonstrated the possibilities of producing hardware designs that have been *verified* as opposed to tested. Examples of this approach include the proof of correctness of a simple microcoded processor [1] and the verification of the design of various low-level hardware modules [2]. The tools that have been used in this work are LCF.LSM [3], VERITAS [4] and HOL [5].

Most people would agree that it is desirable for a manufacturer's products to meet some form of specification. This requirement becomes vital when the product is used in a life-critical situation — users must know what the behaviour of the product will be. This has resulted in the emergence of a disciplined approach to design in many engineering professions. An architect checks that a new building will not fall down, an aircraft designer does detailed calculations to ensure that the wings produce enough lift. At each step of the construction process checks are made to ensure that the components used meet their specifications in the design.

Now that computers are being used in life-critical applications, such as fly-by-wire aircraft or complex life support systems, it is vital for the underlying hardware to be correct. It is impossible to exhaustively test components as simple as a 32-bit multiplier — never mind an entire processor — so different techniques must be used to verify designs. As E.W.Dijkstra has remarked [6]

(non-exhaustive) *testing can be used to show the presence of bugs
but never to show their absence.*

Starting from an agreed formal specification a correct design can be produced if the implementation is produced by a sequence of provably correct steps. This will bring the standard of computer design to the levels expected in other branches of engineering [7]. Use of verified design methods can produce savings in time and expenditure. The need to redesign part of a VLSI device may cause a 2 or 3 month delay in its launch and several such iterations can make a device obsolete before it comes to market.

This chapter details how a verified design approach was used on sections of the IMS T800 floating-point unit microcode. The formal semantics of the occam language [8][9] and the use of program transformations are described. Then a simple example is used to show how a high-level specification can be developed into microcode using formal design methods that guarantee the correctness of the final design.

8.2 occam

The occam language [8] allows a system to be hierarchically decomposed into a collection of concurrent processes communicating via channels. This allows it to be used to represent the behaviour of a VLSI device in a very natural way — the various top-level modules can be mapped on to individual processes with their interfacing handled by channel communication. In more traditional languages the inherent parallelism of a VLSI device has to be handled by explicit programming. occam has a very efficient implementation permitting fast execution of such a behavioural description to allow for simulation. Most importantly, for the purposes of this paper, occam has rich formal semantics [9] which facilitate program transformation and proof.

8.2.1 OCCAM transformations

The algebraic semantics of occam given in [9] consists of a set of laws which define the language constructs. The algebraic semantics have been shown to be consistent with the denotational semantics establishing the validity of these laws. These transformation laws enable a normal form for finite occam programs to be defined.

A transformation law can be used to transform one program into another whose observable behaviour is equivalent. Many transformation laws are 'obviously true' and are regularly used by programmers — for example sequential composition of processes is associative:

$$\begin{array}{ccc} \text{SEQ} & & \text{SEQ} \\ \text{P} & & \text{SEQ} \\ \text{SEQ} & \equiv & \text{P} \\ \text{Q} & & \text{Q} \\ \text{R} & & \text{R} \end{array}$$

This is the law *SEQ binassoc*. Others are more complex and include preconditions for validity but, with a bit of effort, can be seen to be true.

If a sequence of transformations can be found to transform one program into another then the two programs are known to be equivalent. If, in addition, one of these programs is known to be a correct implementation of a specification then the correctness of the other can be inferred.

Using these techniques it is possible to demonstrate the correctness of implementations by transformation — doing this by experimental testing takes far too long for problems like floating-point arithmetic.

An example transformation

As an example consider the following program fragment:

```
SEQ
  X := A
  Y := Y + X
```

These two assignment statements can be merged into one multiple assignment statement. First the law *AS id* is used to add an identity assignment to each statement:

```
AS id  X, Y := e, Y ≡ X := e
```

giving the program:

```
SEQ
  X, Y := A, Y
  Y, X := Y + X, X
```

Next the law *AS perm* is applied to the second statement:

```
AS perm    < x_i | i = 1..n > := < e_i | i = 1..n >
            =
< x_{\pi_i} | i = 1..n > := < e_{\pi_i} | i = 1..n >
  for any permutation \pi of {1..n}
```

giving:

```
SEQ
  X, Y := A, Y
  X, Y := X, Y + X
```

Finally these two statements are merged by the law *SEQ comb*:

```
SEQ comb  SEQ(x := e, x := f) ≡ x := f[e/x]
```

giving:

```
X, Y := A, Y + A
```

8.2.2 The occam transformation system

To aid the process of transforming programs a simple interactive transformation system has been implemented in the language ML[10]. A program can be parsed into this system and then manipulated by the user. All the basic laws in [9] are implemented inside the system along with some extra ones — the system is extensible and new laws (that have been proven correct) can be coded and added if required. Regularly executed sequences of transformations can be coded as ML functions giving higher-level transformations. The example transformation shown above has been coded up as the transformation law *combas* which itself is used in more powerful transformations. The basic transformations often have only a small localised effect but when suitably combined they can perform significant transformations which being constructed from correct component transformations are known to be correct.

The transformation system user can select which transformation laws to apply and examine the effects of these transformations. The fact that the transformation system is being used provides the verification of the equivalence between the initial program and the transformed end result — but if necessary it would be feasible to produce the list of transformations which constitute the proof.

8.3 Instruction development

The instruction development process consists of specifying the operation of the instruction in the Z specification language [11]. Since Z is a mathematically based language it allows precise unambiguous statements about operations to be made concisely and — if used in a sympathetic manner — clearly.

Along with the specifications of the instructions there will be a set of specifications of system constants, system state and other global features of the design. In the case of the IMS T800 floating-point unit this consists of a formal specification of the IEEE floating-point standard — such as in [12], a specification of the internal representation of floating-point numbers in registers, a specification of the floating-point unit state — i.e. the registers and flags, and definitions of various constants that are used. This corresponds to formally describing the overall architecture.

Each instruction specification is refined into a high-level occam implementation. This can involve going via a guarded command language using pre- and post-conditions as in [13]. This high-level implementation is often the sort of implementation that a competent programmer would produce from the specification but the formal derivation ensures that no mistakes are made.

The occam program is then transformed inside the transformation system into a form equivalent to the microcode assembler source. The steps in this process are motivated by the functions available in the microcode machine. This involves:

- 1 refining *IF* conditions into the conditions available on the microcode machine
- 2 refining the expressions so that they use the alu and bus operations available on the microcode machine
- 3 refining the sequential control of the program into a form that simulates the microinstruction control in the microcode machine

The various stages of simple development used as an example are shown in the next section.

8.4 An example instruction development

The following example demonstrates the methods that have been found to be useful in the IMS T800 design. This example takes a high-level specification in the Z specification language [11] and refines it in a sequence of steps into a microcoded implementation that will run on a microcode machine similar to the IMS T800 floating-point unit. For brevity certain simplifications have been made — notably that infinities, Not-a-Numbers and denormalised numbers are ignored.

8.4.1 Preliminary definitions

Before any instructions are specified and implemented it is necessary to make a few preliminary definitions. There is a need to specify the format of registers, various constants and methods for interpreting data. This is a formalisation of the top level of architectural description of the device. Only the subset of definitions relevant to this example will be given.

The definition of the real format will contain the specification of the number of bits in the fractional part of a floating-point number and the exponent bias:

bitsinfrac, bias : N

Now the floating-point register format can be specified:

Floating_Point_Register

frac, exp : N

sign : { -1, +1}

(exp = 0 \wedge frac = 0)

V

($2^{\text{bitsinfrac}-1} \leq \text{frac} < 2^{\text{bitsinfrac}}$)
--

This states that a *Floating_Point_Registers* has three fields. Two of which, *frac* and *exp*, are positive integers and the third, *sign*, is either -1 or $+1$. The predicate states that both the exponent and fraction are 0 or that *frac* is between $2^{\text{bitsinfrac}-1}$ and $2^{\text{bitsinfrac}}$ — this ensures that the fraction is normalised.

The valuation function on a floating-point register *fv* establishes the link between a *Floating_Point_Register* and the value it ‘holds’:

<i>fv</i> : Floating_Point_Register \rightarrow R

$\forall x : \text{Floating_Point_Register}.$

$f(x) = x.\text{sign} \times$

$(x.\text{frac} \times 2^{1-\text{bitsinfrac}}) \times 2^{\text{exp}-\text{bias}}$
--

Two constants are used to represent the largest and smallest integers in the integer format. As the IMS T800 uses 32-bit 2s complement integers these are specified by:

MinInt, MaxInt : Z

MinInt = -2^{31}

MaxInt = $2^{31} - 1$

8.4.2 The instruction specification

The instruction under consideration here is a component of the real to integer conversion instruction sequence. It checks that the value of Areg lies within integer range — if it doesn't then the error flag must be set to indicate a conversion error.

The Z specification of this instruction is very simple:

Floating_Check_Integer_Range	
Areg, Areg'	: Floating_Point_Register
Error_Flag, Error_Flag'	: bool
$\text{fv(Areg)} \in \mathbb{Z}$	
Areg' = Areg	
$\text{fv(Areg)} \in [\text{MinInt}, \text{MaxInt}] \Rightarrow$	
Error_Flag' = Error_Flag	
$\text{fv(Areg)} \notin [\text{MinInt}, \text{MaxInt}] \Rightarrow \text{Error_Flag}' = \text{true}$	

The first predicate is a precondition to this operation. If fv(Areg) is not an integer then the effect of this operation will be undefined. In this way the precise conditions for the correct execution of an operation are stated. This instruction is intended for use in a particular sequence of instructions and the previous instruction will have established this precondition.

It is easy to see that this specification satisfies the requirements for the instruction. Once this has been agreed to be 'correct' the development process will ensure that the final implementation will also satisfy the requirements.

8.4.3 Refining to procedural form

A refinement of a specification can consist of either refining a data type or decomposing the procedural form. As the major data type — reals — has already been refined into its machine representation, by using *Floating_Point_Register* and the abstraction function *fv*, the specification can be decomposed into procedural form. The specification can be easily implemented by:

```

if
  fv(Areg) ∈ [MinInt, MaxInt] → skip
  || fv(Areg) ∉ [MinInt, MaxInt] →
    Error_Flag := true
fi

```

Using the pre/post-condition laws in [13] this can be shown to implement the Z specification.

8.4.4 Refining to occam

This has produced a procedural implementation but the conditionals used in the *if .. fi* construct are not available in occam so they need to be refined into equivalent occam expressions.

To do this the lemmas about integer range shown below will be useful.

- lemma 1 $\vdash \forall x, y : \text{Floating_Point_Register}.$
 $(x.\text{exp} < y.\text{exp} \vee (x.\text{frac} < y.\text{frac} \wedge x.\text{exp} = y.\text{exp})) \Leftrightarrow |\text{fv}(x)| < |\text{fv}(y)|$
- lemma 2 $\vdash \forall x : \text{Floating_Point_Register}.$
 $\text{fv}(x) = \text{MinInt} \Leftrightarrow (x.\text{sign} = -1 \wedge x.\text{frac} = \text{MSBit} \wedge x.\text{exp} = \text{LargestINTExp})$
- lemma 3 $\vdash \text{MaxInt} = -(\text{MinInt} + 1)$
where MSBit = $2^{\text{bitsInfrac}-1}$
 $\text{LargestINTExp} = 32 + \text{bias}$

From lemmas 1 and 2 obtain:

$$\vdash \forall x : \text{Floating_Point_Register} . \\ x.\text{exp} < \text{LargestINTExp} \Leftrightarrow |fv(x)| < |\text{MinInt}|$$

The fact that $\text{MSBit} \leq x.\text{frac}$ is part of the invariant of `Floating_Point_Register` is used to eliminate the disjunct where $x.\text{exp} = \text{LargestINTExp}$.

Now using lemma 3 and adding an extra condition obtain:

$$\vdash \forall x : \text{Floating_Point_Register} . \\ fv(x) \in \mathbb{Z} \Rightarrow x.\text{exp} < \text{LargestINTExp} \\ \Leftrightarrow |fv(x)| \leq \text{MaxInt}$$

From these obtain:

$$\vdash \forall x : \text{Floating_Point_Register} . \\ fv(x) \in \mathbb{Z} \Rightarrow fv(x) \in [\text{MinInt}, \text{MaxInt}] \\ \Leftrightarrow (x.\text{exp} < \text{LargestINTExp}) \\ \vee fv(x) = \text{MinInt}$$

8.4.5 High-level occam implementation

The previous section allows the high-level occam implementation below to be derived.

```
IF
  (Areg.Exp < LargestINTExp) OR
    ((Areg.Sign = 1) AND
      (Areg.Exp = LargestINTExp) AND (Areg.Frac = MSBit))
  SKIP
  NOT ((Areg.Exp < LargestINTExp) OR
        ((Areg.Sign = 1) AND
          (Areg.Exp = LargestINTExp) AND (Areg.Frac = MSBit)))
ErrorFlag := TRUE
```

Using two laws *IF pri* and *IF or-dist*:

$$\begin{aligned} \text{IF pri} \quad & IF(b_1 P_1, \dots, b_n P_n) \\ & \equiv IF(b_1^* P_1, \dots, b_n^* P_n) \\ & \text{where } b_i^* = \neg b_1 \wedge \dots \wedge \neg b_{i-1} \wedge b_i \end{aligned}$$

$$\begin{aligned} \text{IF or-dist} \quad & IF(b_1 P, b_2 P, C) \\ & \equiv IF(b_1 \vee b_2 P, C) \end{aligned}$$

this can be simplified to the program:

```
IF
  (Areg.Exp < LargestINTExp)
  SKIP
  (Areg.Sign = 1) AND
    (Areg.Exp = LargestINTExp) AND (Areg.Frac = MSBit)
  SKIP
TRUE
ErrorFlag := TRUE
```

which is probably the implementation of the specification that a competent programmer would produce — but the ‘special’ case of *MinInt* is frequently omitted.

8.4.6 Transformations towards microcode

The previous sections have developed an occam program that correctly implements the specification. This can now be transformed into an equivalent form that corresponds to microcode assembler source. Full details of this process will not be given here.

Each step consists of transforming one aspect of the program towards the form used in the microcode machine. Ideally this occam program would be transformed into the final program. As the transformation system is still under development most of the laws that it contains are those that are 'general' — i.e. are correct in all environments. This does not allow the required transformation to be performed in a forwards manner. Instead at each step a proposed implementation was constructed and this was then verified by transforming it back into the current 'correct' implementation.

Refining the conditionals

The occam program given contains a three-way *IF* statement with the conditionals:

```

1 (Areg.Exp < LargestINTExp)
2 (Areg.Sign = 1) AND
   (Areg.Exp = LargestINTExp)
   AND (Areg.Frac = MSBit)
3 TRUE

```

The structure of the program must be transformed to take account of the conditional signals available on the microcode machine — i.e. that conditionals are available to signal that the result of an ALU operation is less than 0 or that the result of an ALU subtraction is 0 etc.

This program for implementation with refined conditionals is shown below. The various laws for *IF* constructs in [9] enable this to be verified:

```

IF
  (Areg.Sign = 1)
    IF
      ((Areg.Exp - LargestINTExp) < 0)
        SKIP
      NOT ((Areg.Exp - LargestINTExp) < 0)
        IF
          ((Areg.Exp - LargestINTExp) = 0)
            IF
              ((MSBit - Areg.Frac) = 0)
                SKIP
              NOT ((MSBit - Areg.Frac) = 0)
                ErrorFlag := TRUE
              NOT ((Areg.Exp - LargestINTExp) = 0)
                ErrorFlag := TRUE
            NOT (Areg.Sign = 1)
              IF
                ((Areg.Exp - LargestINTExp) < 0)
                  SKIP
                NOT ((Areg.Exp - LargestINTExp) < 0)
                  ErrorFlag := TRUE

```

Refining the expressions

The previous section has produced conditionals that are available in the microcode machine. The next step is to take account of how the expressions producing these conditionals are evaluated. This stage involves introducing variables to represent the various buses and conditional flags. The conditional flags appear as the *IF* conditionals and are evaluated in terms of the results of the ALU operations before the *IF* statement.

This program for implementation with refined expressions is shown below: The laws for *SEQ*, *VAR* and assignment in [9] verify this step:

```

VAR AregNegative, ExpZbus, ExpZbusNeg, ExpZbusEqZ, FracZbusEqZ :
VAR FracZbus :
SEQ
  AregNegative := (Areg.Sign = 1)
  ExpZbus := (Areg.Exp - LargestINTExp)
  ExpZbusNeg := ExpZbus < 0
  IF
    AregNegative
    IF
      ExpZbusNeg
      SKIP
      NOT ExpZbusNeg
      SEQ
        ExpZbus := (Areg.Exp - LargestINTExp)
        FracZbus := (MSBit - Areg.Frac)
        ExpZbusEqZ := ExpZbus = 0
        IF
          ExpZbusEqZ
          SEQ
            FracZbusEqZ := FracZbus = 0
            IF
              FracZbusEqZ
              SKIP
              NOT FracZbusEqZ
              ErrorFlag := TRUE
            NOT ExpZbusEqZ
            ErrorFlag := TRUE
        NOT AregNegative
        IF
          ExpZbusNeg
          SKIP
          NOT ExpZbusNeg
          ErrorFlag := TRUE

```

Introducing sequencing

The program now contains expressions and conditionals that can be formed in the microcode machine. However, the program does not define microwords. The final step is to mimic the microsequencing in the microcode machine by use of a variable as a microprogram counter and a *WHILE* loop containing an *IF* microinstruction selector. Each branch of the *IF* statement contains the 'code' for one microinstruction — i.e. it can have one fractional ALU operation, one exponential ALU operation and defines the next microinstruction to execute — possibly with one or two conditionals.

The laws for *WHILE* and *IF* allow this program to be 'unwound' back into its previous form.

8.4.7 Translation to microcode

The final program for low level OCCAM implemenation from the previous transformations is:

```

VAR NextInst :
VAR AregNegative, ExpZbusNeg, ExpZbusEqZ, FracZbusEqZ :
VAR FracZbus, ExpZbus :
SEQ
NextInst := FloatingPointCheckIntegerRange
WHILE NextInst <> NOWHERE
IF
    NextInst = FloatingPointCheckIntegerRange
    SEQ
        AregNegative := (Areg.Sign = 1)
        ExpZbus := (Areg.Exp - LargestINTExp)
        ExpZbusNeg := ExpZbus < 0
        IF
            AregNegative
            IF
                ExpZbusNeg
                NextInst := NOWHERE
                NOT ExpZbusNeg
                NextInst := CheckMinInt
            NOT AregNegative
            IF
                ExpZbusNeg
                NextInst := NOWHERE
                NOT ExpZbusNeg
                NextInst := OutofRange
            NextInst = OutofRange
            SEQ
                ErrorFlag := TRUE
                NextInst := NOWHERE
            ...
            negative case micro instructions
        .
    
```

This corresponds in an almost one-to-one manner with the source format for the microcode assembler. A pattern-matching program is used to translate the stylised OCCAM of the above program into the source for the microcode assembler. The microcode assembler then produces the definition of the microcode ROM from this source.

8.4.8 Microcode assembler source

Finally the microcode can be derived:

```

FloatingPointCheckIntegerRange:
    ExpConstantFromLargestINTExp
    ExpXbusFromAreg
    ExpZbusFromXbusMinusYbus
    GOTO Cond1FromAregSign -> ExpYbusFromConstant
                                            (Cond0FromExpZbusNeg -> (NOWHERE, CheckMinInt),
                                             Cond0FromExpZbusNeg -> (NOWHERE, OutofRange))

CheckMinInt:
    ExpConstantFromLargestINTExp
    ExpXbusFromAreg
    ExpZbusFromXbusMinusYbus
    FracXbusFromMSBit
    FracZbusFromXbusMinusYbus
    GOTO Cond1FromExpZbusEqZ -> ExpYbusFromConstant
                                            (CheckMinInt2, OutofRange)

FracYbusFromAreg
                                            (CheckMinInt2, OutofRange)

CheckMinInt2:
    GOTO Cond1FromFracZbusEqZ -> (NOWHERE, OutofRange)

OutofRange:
    SetErrorFlag
    GOTO NOWHERE
  
```

This process has ensured that the 'program' in the microcode ROM correctly implements the initial specification. It might seem possible to do this informally in this simple case which only produces four microwords. Other instructions contain up to ninety microwords where informal development can easily introduce subtle bugs. The ability to verify an implementation using program transformations has proved invaluable.

8.5 Current and future work

Work on the IMS T800 has shown how correct microcode can be derived from a high-level specification. However, this has assumed that the hardware implementing the microcode machine is correct. To produce a verified processor design it will be necessary to apply the same degree of rigour to the design of the microcode machine. This necessitates refining the specifications of microfunctions into hardware description language (HDL) implementations. The INMOS CAD system already ensures that silicon layout is equivalent to its HDL specification.

This correctness of design can be achieved by defining axioms for the behaviour of low-level modules in the HDL module library if necessary down to transistor level. Larger modules and circuits can then be specified in terms of compositions of these 'axiomatic' modules. Then a logic tool, such as HOL [5], can be used to derive the behaviour of the design. Checking this against an original specification enables the correctness — or otherwise — of the design to be established.

8.6 Conclusions

Work at INMOS using the transformation system and a formal design strategy has been seen to be of benefit. The correctness of the microcode for the IMS T800 floating-point unit was established in far less time than would be needed by an 'adequate' amount of testing. In addition, any amount of non-exhaustive testing leaves the possibility that certain erroneous operations have not been exercised. This has enabled INMOS to produce the IMS T800 well ahead of schedule with a high degree of confidence in the correctness of the microcode — this would not have been possible by other design methods.

Work is now in progress to incorporate this formal design strategy into the other levels of the design process to maintain the correctness of a complete design. It seems clear that the CAD system will need to incorporate a theorem prover and work is progressing at INMOS to ensure that this is the case.

8.7 References

- 1 *Proving a computer correct*, M Gordon, University of Cambridge Computer Laboratory Technical Report 42, 1983.
- 2 *Specification and Verification using Higher-Order Logic*, F K Hanna, N Daeche, Proceedings of the 7th International Conference on Computer Hardware Design Languages, Tokyo, 1985.
- 3 *LCF_LSM* M Gordon, University of Cambridge Computer Laboratory, Technical Report 41, 1983.
- 4 *The VERITAS theorem Prover*, F K Hanna, N Daeche, Electronics Laboratory, University of Kent at Canterbury, 1984 onwards.
- 5 *HOL: A machine orientated formulation of Higher-Order Logic*, M Gordon, University of Cambridge Computer Laboratory, Technical Report 68, 1985.
- 6 Dijkstra, E W, quotation taken from 7
- 7 *Programming is an engineering profession*, C A R Hoare, Oxford University Computing PRG, Technical Monograph PRG-27, 1982.
- 8 *The occam Programming Manual*, INMOS Ltd, Prentice Hall, 1984.
- 9 *The laws of Occam programming*, A W Roscoe, C A R Hoare, Oxford University Computing PRG, Technical Monograph PRG-53, 1986.
- 10 *Edinburgh LCF* – chapter 2, M Gordon, R Milner, C Wadsworth, LCNS 78, Springer Verlag, 1979.
- 11 *The Z Handbook* B A Sutrin (editor), Oxford University Computing PRG, 1986.
- 12 *Formal methods applied to a floating point number system*, G Barrett, Oxford University Computing PRG, Technical Monograph, 1987.
- 13 *The science of programming*, D Dries, Springer-Verlag, 1981

9 Simpler real-time programming with the transputer

9.1 Introduction

INMOS manufactures a range of high performance microprocessors, called transputers, which combine all the essential elements of a computer (processor, memory and i/o) in a single component. Transputers provide support, in hardware and microcode, for concurrency and communication. This support includes communication links for connecting transputers together and two hardware timers which can be used for interval measurement or for real-time scheduling.

The OCCAM language was designed for programming systems composed of concurrently executing, communicating processes and, as such, is especially suitable for transputer-based systems. An important application of modern microprocessor systems is real-time control and occam provides many features for this purpose. One of these is the *timer*, a means of measuring time periods and generating time delays.

This technical note describes some aspects of timers on the transputer, using occam. It introduces the basics of the occam language and then goes on to show some simple ways in which timers can be used in programs. The next section describes how the transputer implements timers. Finally there are some examples taken from occam programs which illustrate various aspects of the use of timers.

9.2 The occam programming language

The occam language enables a system to be described as a collection of concurrent processes which communicate with one another, and with the outside world, via communication channels.

9.2.1 occam programs

This section is a brief introduction to occam and, as such, can be overlooked by those familiar with the language. OCCAM programs are built from three primitive processes:

variable := expression	assign value of expression to variable
channel ? variable	input a value from channel to variable
channel ! expression	output the value of expression to channel

Each occam channel provides a one way communication path between two concurrent processes. Communication is synchronised and unbuffered. The primitive processes can be combined to form constructs which are themselves processes and can be used as components of other constructs.

Conventional sequential programs can be expressed by combining processes with the sequential-constructs **SEQ**, **IF**, **CASE** and **WHILE**. Concurrent programs are expressed using the parallel-construct **PAR**, the alternative-construct **ALT** and channel communication. **PAR** is used to run any number of processes in parallel and these can communicate with one another via communication channels. The alternative-construct allows a process to wait for input from any number of input channels. Input is taken from the first of these channels to become ready and the associated process is executed.

Sequence

A sequential-construct is represented by:

```
SEQ
  P1
  P2
  P3
  ...

```

The component processes **P1**, **P2**, **P3** ... are executed one after another. Each component process starts after the previous one terminates and the construct terminates after the last component process terminates.

For example:

```
SEQ
  c1 ? x
  x := x + 1
  c2 ! x
```

inputs a value, adds one to it, and then outputs the result.

Sequential-constructs in occam are similar to programs written in conventional programming languages.

Parallel

A parallel-construct is represented by:

```
PAR
  P1
  P2
  P3
  ...
```

The component processes **P1**, **P2**, **P3** ... are executed together, and are called concurrent processes. The construct terminates after all of the component processes have terminated, for example:

```
PAR
  c1 ? x
  c2 ! y
```

allows the communications on channels **c1** and **c2** to take place together.

The parallel-construct is unique to occam. It provides a straightforward way of writing programs which directly reflects the concurrency inherent in real systems. Concurrent processes communicate only by using channels, and communication is synchronized. If a channel is used for input in one process, and output in another, communication takes place when both the inputting and the outputting processes are ready. The value to be output is copied from the outputting process to the inputting process, and the processes then proceed.

Conditional

A conditional-construct

```
IF
  condition1
    P1
  condition2
    P2
  ...
```

means that **P1** is executed if **condition1** is true, otherwise **P2** is executed if **condition2** is true, and so on. Only one of the processes is executed, and then the construct terminates, for example:

```
IF
  x = 0
    y := y + 1
  x <> 0
    SKIP
```

increases **y** only if the value of **x** is 0.

Alternation

An alternative construct

```
ALT
  input1
    P1
  input2
    P2
  input3
    P3
  ...
  ...
```

waits until one of **input1**, **input2**, **input3** ... is ready. If **input1** first becomes ready, **input1** is performed, and then process **P1** is executed. Similarly, if **input2** first becomes ready, **input2** is performed, and then process **P2** is executed. Only one of the inputs is performed, then its corresponding process is executed and then the construct terminates, for example:

```
ALT
  count ? signal
    counter := counter + 1
  total ? signal
    SEQ
      out ! counter
      counter := 0
```

either inputs a signal from the channel **count**, and increases the variable **counter** by 1, or alternatively inputs from the channel **total**, outputs the current value of the counter, then resets it to zero. The **ALT** construct provides a formal language method of handling external and internal events that must be handled by assembly level interrupt programming in conventional languages.

Loop

```
WHILE condition
  P
```

repeatedly executes the process **P** until the value of the condition is false, for example:

```
WHILE (x - 5) > 0
  x := x - 5
```

leaves **x** holding the value of (**x** remainder 5) if **x** were positive.

Selection

A selection construct

```
CASE s
  n
    P1
  m, q
    P2
  ...
  ...
```

means that **P1** is executed if **s** has the same value as **n**, otherwise **P2** is executed if **s** has the same value as **m** or **q**, and so on.

For example:

```
CASE direction
  up
    x := x + 1
  down
    x := x - 1
```

increases the value of **x** if **direction** is equal to **up**, otherwise if **direction** is equal to **down** the value of **x** is decreased.

Replication

A replicator is used with a **SEQ**, **PAR**, **IF** or **ALT** construction to replicate the component process a number of times. For example, a replicator can be used with **SEQ** to provide a conventional loop:

```
SEQ i = 0 FOR n
  P
```

causes the process **P** to be executed **n** times.

A replicator may be used with **PAR** to construct an array of concurrent processes:

```
PAR i = 0 FOR n
  Pi
```

constructs an array of **n** similar processes **P0, P1, ..., Pn-1**. The index **i** takes the values 0, 1, ..., **n-1**, in **P0, P1, ..., Pn-1** respectively.

This note contains some short program examples written in occam. These should be readily understandable but, if necessary, a full definition of the occam language can be found in the occam reference manual [1].

9.2.2 Timers in occam

This section gives more detail of the **TIMER** in occam.

An occam timer provides a clock which can be read to provide a value representing the time. The timer is read by an input statement similar to that used for receiving data from a channel. Unlike a communication channel, a single timer can be shared by any number of concurrent processes. Timers are declared in an occam program to be of type **TIMER** in the same way as channels and variables are declared. An example of the use of timers is shown below:

```
TIMER clock :
INT t :
SEQ
  ...
  clock ? t -- read value of timer 'clock' into 't'
  ...
```

9.2.3 Timer values

The value input from a timer is of type **INT**. The value is derived from a clock which increments by a fixed amount at regular intervals. The value of the clock is cyclic, that is when the time reaches the most positive integer value then the next increment results in the most negative value. An analogy can be drawn here with a real clock. We normally understand whether a particular time is before or after another from the context. For example 11 o'clock would normally be considered to be before 12 o'clock, and 12 o'clock to be before 1 o'clock. This comparison only works for limited ranges of times. For example we may consider 6 pm to be after 12 noon, but 7 am to be *before* noon (i.e. 7 am is before 6 pm even though 6 is less than 7).

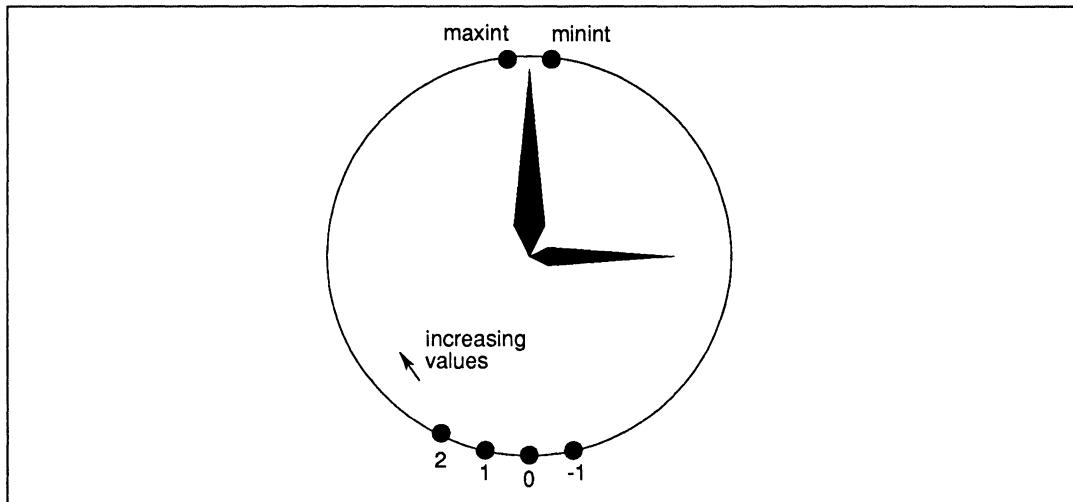


Figure 9.1 Cyclic timer values

9.2.4 Modulo operators

A special operator, **AFTER**, can be used to compare times in **occam**. **AFTER** is one of a set of *modulo* operators, these perform arithmetic with no overflow-checking and thus produce cyclic results. Two other modulo operators useful with timer values are **PLUS** and **MINUS** which perform addition and subtraction respectively. For example, if **maxint** is the largest value of type **INT** that can be represented, then **maxint PLUS 1** wraps around and becomes the most-negative representable integer (**minint**), this is illustrated in Figure 9.1. **a AFTER b** is defined to be equivalent to **(b MINUS a) > 0**. The value **t2 AFTER t1** is true if the value of **t2** represents a later time than the value of **t1**. This comparison is only valid for times within half a timer cycle of one another because **(b MINUS a)** must be positive.

The **AFTER** operator can also be used in a timer input to create a *delayed input*. This specifies a time after which the input terminates. For example:

```
TIMER clock :  
SEQ  
...  
clock ? AFTER t  
...
```

This example will wait until the value of the timer **clock** is later than the value of **t**.

9.3 Using timers

This section outlines the basic applications of timers in occam programs.

9.3.1 Measuring time intervals

Perhaps the most obvious use of a timer is for measuring time intervals. Different timers are not guaranteed to have the same value so time intervals must be measured using a single timer.

For example, when benchmarking programs written in occam, the timer can be read before and after executing the main body of the code:

```
TIMER clock :
INT t1, t2, time :
SEQ
    clock ? t1          -- read start time into t1
    ... run benchmark
    clock ? t2          -- read end time into t2
    time := t2 MINUS t1 -- calculate elapsed time
    ... print time taken
```

There are a few important points to note about this example.

- The use of the modulo operator, **MINUS**, to calculate the time taken. If, at the start of the program, the timer has a very large positive value then it may have 'wrapped-round' to a negative value the second time it is read. Using a normal subtraction on these values would cause an arithmetic-overflow error. The modulo operator gives the correct elapsed time.
- As explained in Section 9.2.4 the time interval measured in this way must be less than half the cycle time of the timer.
- The time measured in this way is elapsed time, not processor time used by this process. This may cause 'incorrect' results if there are other processes running in parallel.

9.3.2 Generating a known delay

The next application of timers is to use the delayed input to generate a known time delay. This is very simple as shown below:

```
TIMER clock :
INT now :
VAL delay IS 1000 : -- delay time in clock 'ticks'
SEQ
    clock ? now
    clock ? AFTER now PLUS delay
```

This example reads the current value of the timer, then the delayed input waits until the value of the timer is later than the value of **now PLUS delay**. The process is descheduled while waiting so other processes can be executed. An important practical point here is that there may be a delay before the process is rescheduled. This *latency* may be due to a number of factors, e.g. the number of other processes executing at the time, and may be variable. The transputer implements process scheduling in hardware and so the latency can be very small (see Section 9.4.1).

Again, note the use of the modulo operator **PLUS** to calculate the time to wait until and the fact that the greatest delay is half the timer's cycle time. A technique for generating delays of arbitrary length is given in Section 9.5.5.

9.3.3 Generating events at regular intervals

A program which must perform a task at regular intervals cannot do so simply by means of a fixed delay between processing, as in the previous example. If a simple delay were used then the time at which the task happens will slip gradually because the delay does not account for the time taken by the task itself (which may vary) and this error accumulates. This is illustrated in Figures 9.2 & 9.3.

To make this more explicit, assume the task must be scheduled every millisecond and will execute for 10 μ s. The task executes and is then descheduled for 1ms (plus the time required to reschedule the process). The interval between tasks is therefore at least 1.01ms and this error will accumulate so, after 1 second the task will have been executed only 990 times instead of 1000 times. It would be possible to adjust the delay to take the processing time of the task into account, but this implies that the processing time is both known and fixed. This is unlikely to be the case in a real system. Consider the following example:

```

TIMER clock :
INT time :
SEQ
  WHILE active
    SEQ
      ... perform process P at intervals
      -- wait for 'delay' clock ticks
      clock ? time
      clock ? AFTER time PLUS delay
  
```

The time taken to execute the loop is the delay time plus the execution time of process P. Any variation in the processing required in P will vary the frequency at which it is executed.

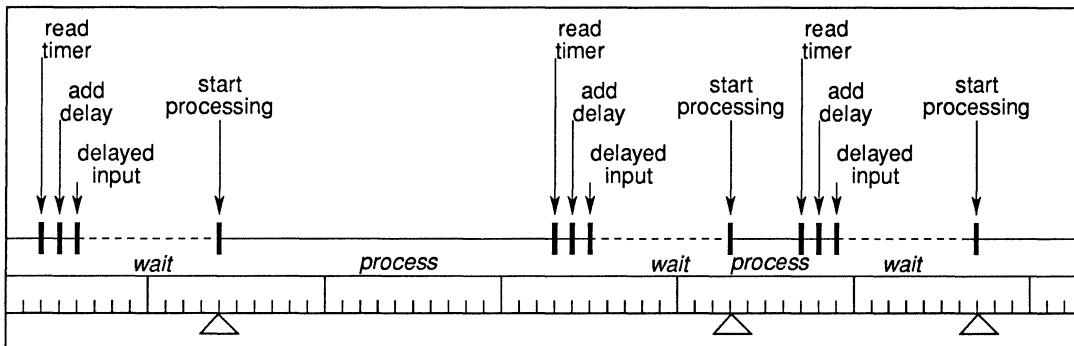


Figure 9.2 Using timer to generate delays between processing

A far more accurate way to achieve the desired effect is shown below:

```

TIMER clock :
INT time :
SEQ
  clock ? time
  WHILE active
    SEQ
      ... perform process P at regular intervals
      -- add interval to the time the process started
      time := time PLUS interval
      -- and wait until it is time to execute the process again
      clock ? AFTER time
  
```

The important point to note here is that the value of the timer is only read once, before the loop is entered. After that the time is updated by adding a constant increment to the current value. This ensures that the delayed input always waits *until the desired starting time*, rather than for a fixed delay. This prevents any drift in the timing of the processing.

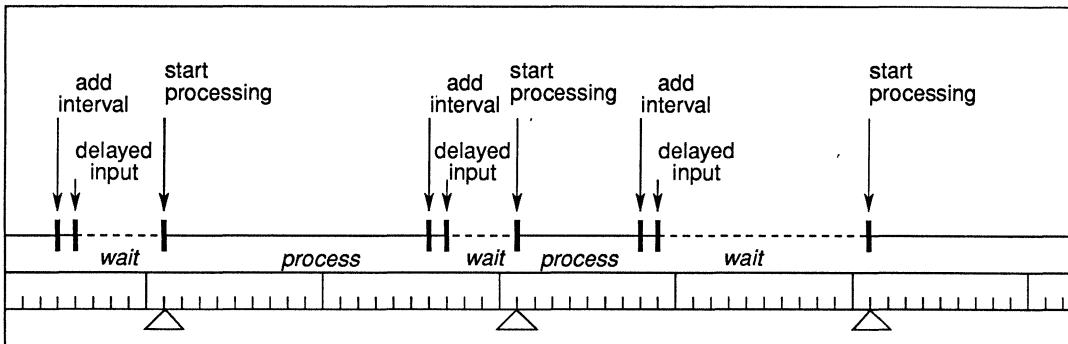


Figure 9.3 Using timer to perform processing at fixed intervals

To take the previous example of a task being scheduled every millisecond, it can be seen that the task is initiated at (or shortly after, because of scheduling latency) the time specified by the value of `time`. When the task has completed a constant amount is added to the value of `time` to calculate the time the task should next be scheduled. This time is independent of the time taken by the task. The possible variation in the time taken to schedule a process may introduce some jitter into the timing of the task, but will not cause it to slip.

9.3.4 Use in ALTs

Delayed timer inputs are often used in alternative constructs.

Interleaving-processing

An alternative may be used to interleave-processing at fixed times with processing performed when data is received. As an example, a data logging process may need to record data received from a channel and, at suitable intervals, insert a time stamp in the recorded data. This could be written with an **ALT** very simply:

```
TIMER clock :
INT time, data :
SEQ
  clock ? time
  WHILE active
    SEQ
      time := time PLUS one.second
      PRI ALT
        clock ? AFTER time
        ...   insert time stamp in file
        in ? data
        ...   store data in file
```

Note that the delayed input is prioritised with respect to the channel input; this ensures that, even if the channel `in` is always ready, the time stamping process will be selected when it becomes ready.

Timeouts on channels

Another use of delayed inputs in alternatives is to provide some sort of timeout on channel communication. This may be to execute a process if no user command is received, or to detect an error condition. For example, a disk controller may wish to 'park' the heads (i.e. move them to a safe position on the disk) if no

commands are received within a time limit:

```
WHILE active
SEQ
  clock ? time
  ALT
    (headsNotParked) & clock ? AFTER time PLUS timeout
      ... move heads to shipping track
    in ? command
      ... execute command from file system
```

Multiple delayed inputs

An alternative may contain several delayed inputs with different delays. This may be useful if it is necessary to handle a number of devices at different, fixed intervals. For example, if the processor needs to be scheduled to service two peripherals at different periods then an **ALT** can be used to correctly interleave the handling of these devices:

```
TIMER clock :
INT timeA, timeB :
VAL intervalA IS 96 :
VAL intervalB IS 42 :
SEQ
  clock ? timeA
  clock ? timeB
  WHILE active
    ALT
      clock ? AFTER timeA
      SEQ
        timeA := timeA PLUS intervalA
        ... handle device A at fixed intervals
      clock ? AFTER timeB
      SEQ
        timeB := timeB PLUS intervalB
        ... handle device B at fixed intervals
```

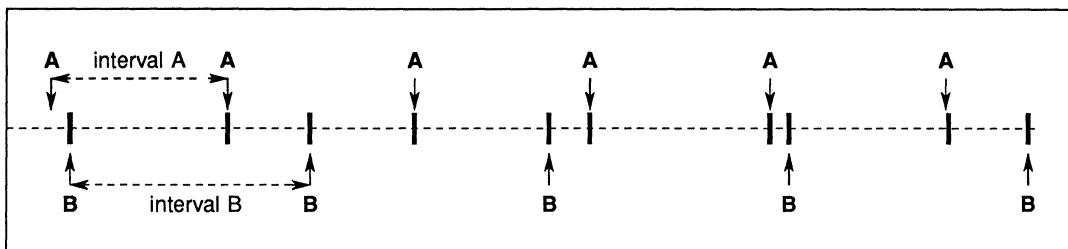


Figure 9.4 Scheduling two processes, **A** and **B**, at different intervals

Only times that are within half a timer-cycle can be compared by **AFTER** so, if several times are being compared, they must all be within half a cycle of one another. If an **ALT** contains more than one delayed input then *all* of the times involved (including the present timer value) must be within half a cycle of one another. A simpler, but sometimes more restrictive, rule is to ensure that all times in the delayed inputs are within a quarter of a cycle of the current timer value.

9.4 Transputer implementation of timers

The transputer [2] has hardware and microcode support for occam timers. This allows timer instructions to be fast and, more importantly, delayed inputs to be *non-busy* (i.e. to consume no processor time whilst waiting). There are two timer clocks, with the same wordlength as the particular device, which tick periodically. One timer is accessible only to high-priority processes and is incremented every microsecond. The other can only be accessed by low-priority processes and ticks every $64\mu s$, giving exactly 15,625 ticks per second. The cycle time of these timers depends on the wordlength of the device. The approximate cycle times, for the current range of 16 and 32 bit transputers, are shown in the table below.

Transputer type	Priority	
	High	Low
IMS T800 & IMS T414	1.2 hours	76 hours
IMS T212 & IMS M212	65.5 ms	4.2 s

It is important to have a resolution of $1\mu s$ for precise timing. However, on a 16 bit processor, this means a cycle time of only 65ms — too short for many applications. To provide both high resolution and a long cycle time, two timer rates were introduced. The same method was used on the 32 bit processors, so the timers behave similarly on all transputer types.

Timers are local to each processor, so the absolute time values read by processes on different transputers in a network will be different. However, the rates of the timers on each transputer will be the same, independent of processor speed etc.

Although timers can be shared between parallel processes, this can appear rather odd if a timer is shared between processes at different priorities. This would have the effect of a single timer producing different values in each process. To make it clear which timer is being used within a process it is good practice to declare timers local to each priority, for example:

```
PRI PAR
  TIMER hiClock :
    SEQ
      ... high-priority process

  TIMER loClock :
    SEQ
      ... low-priority process
```

9.4.1 Scheduling latency

The transputer has a microcoded scheduler which enables any number of concurrent processes to be executed together, sharing processor time. Processes which are descheduled, waiting for a communication or delayed input, do not consume any processor time. The scheduler supports two levels of priority.

The latency between the time a process becomes ready to execute and the time it begins processing depends on the priority at which it is executing. Low priority processes are executed whenever there are no high-priority processes which are ready to execute. A high-priority process runs until it has to wait for a communication or timer input, or until it has completed processing.

Low-priority processes

Low-priority tasks are periodically timesliced to provide an even distribution of processor time between computationally intensive processes. If there are n low-priority processes then the maximum latency is $2n - 2$ timeslice periods. The latency will generally be much less than this as processes are usually descheduled for communication or by a delayed input before the end of their timeslice (see, for example, Section 9.5.2 on polling). The timeslice period is approximately 1ms.

High-priority processes

High-priority processes run whenever they are able to, interrupting any currently executing low-priority process if necessary. If a high-priority process is waiting on a timer input, and no other high-priority processes are running, then the interrupt latency is typically 19 processor cycles ($0.95\mu s$ with a 20MHz processor clock). The maximum latency depends on the processor type as shown in the following table.

Transputer type	Maximum interrupt latency	
	processor cycles	microseconds (at 20MHz)
IMS M212, IMS T212	53	2.65
IMS T414	58	2.9
IMS T800 (FPU in use)	78	3.9
IMS T800 (FPU not in use)	58	2.9

These times indicate that a transputer can handle many tens of thousands of interrupts per second, even while engaged in computationally-intensive tasks involving floating-point calculations.

9.4.2 Timer instructions

The user programming in occam (or other high-level language) does not need to know how the timers are implemented. However, the following description of their implementation in terms of the transputer instruction set may be of interest. Further details of the implementation of occam for the transputer can be found in [3] and a complete description of the transputer instruction set in [4].

The timers are initialised using the **store timer** instruction. This sets the timer to a known value and starts it 'ticking'. This is normally done by the bootstrap or loader-code rather than by a user-program. The value of a timer can be read at any time with the **load timer** instruction.

Delayed inputs

Delayed inputs are supported directly by the **timer input** instruction. The transputer maintains a linked list of processes waiting on each timer, in order of increasing time. The process at the front of each queue is pointed to by a register in the CPU. Another register holds the time that this process is waiting for. A comparator continuously performs the **AFTER** test between this 'alarm' time and the value of the clock, causing the process to be rescheduled when the time is reached.

The **timer input** instruction requires a time to be specified. If this time is in the 'past' then the instruction does nothing, otherwise it deschedules the process and adds it to the list of processes waiting on the timer. The instruction searches down the list of processes and inserts the current process and time value in the appropriate place. If this time is earlier than the current value in the 'alarm' register then the new value will be put in the register.

An important feature of the **timer input** instruction is that it is *interruptable*. Because there can be any number of processes in a timer queue, it is important that searching the queue does not affect the interrupt latency of the system. For this reason, unbounded instructions like this and the 2D block-moves of the IMS T800 can be interrupted by a higher-priority process becoming ready.

9.5 Some application examples

This section is intended to show how some real problems can be solved efficiently. The traditional approaches to handling these problems would either be through polling or interrupts. The disadvantages of these approaches are described as follows, together with the ways in which occam can provide simple solutions.

9.5.1 Interrupts

Interrupts are the usual way of handling devices that require infrequent but fast servicing. Interrupt handlers are notoriously difficult to write and debug, they are usually only supported by programming in assembler and this is often very difficult to integrate with other code written in a high-level language. occam and

the transputer support both internal and external interrupts in a very simple and efficient way. An example of an internal interrupt is a communication or delayed input; external interrupts can be generated from the transputer's links or the *event* input. A transition on the **EventReq** pin behaves just like a channel communication and can be used to synchronise with an OCCAM process. It is, therefore, very easy to write an OCCAM process which handles events — it simply has to perform an input from the channel mapped on to **EventReq** and, when both the event channel and the process are ready, the process is scheduled. The following example shows how a UART¹, which has its *data received* interrupt connected to the transputer's event input, would be handled in OCCAM.

```
{{{ event handler
CHAN OF BYTE error :
PLACE event AT 8 : -- event channel control word

BYTE sync :
WHILE active
SEQ
  event ? sync      -- wait for input from EventReq
  read.data (char)  -- read data from UART
  to.buffer ! char  -- output to waiting process
}}}
```

If this process is run at high-priority then it can interrupt a low-priority process:

```
PRI PAR
... event handler
PAR
...
... low-priority (background) processes
...
```

The performance of transputer interrupts was detailed in Section 9.4.1.

Interrupts can have various disadvantages. With multiple sources of interrupts there is inevitably a cost in determining which device generated the interrupt. This may be extra hardware to encode and prioritise the interrupts, or software to poll the devices on receipt of an interrupt to see which are ready.

9.5.2 Polling

The main disadvantage of polling is that it is *busy*, i.e. it consumes processor time. In the transputer this can have a wide impact on performance because it will affect the scheduling of processes. Low-priority processes are timesliced to ensure that all processes get a fair share of processor time. However, in most real OCCAM programs, processes are frequently descheduled before the end of the timeslice period because they perform some communication. A process which is continuously polling a memory-mapped device, for example, can get a disproportionate amount of the processing resource simply because other processes are descheduled more frequently for communication purposes. If a process in parallel with the polling process is transmitting individual bytes down a link, then each communication may appear to take several milliseconds. This is because the polling process will be scheduled between each byte-transfer and not be descheduled for one or two timeslice periods.

If a peripheral device must be polled then it is much more efficient to use a delayed input to control exactly when, and how often, polling takes place. In most cases this can be done with no degradation in the performance of the device, as the maximum rate at which data can arrive is known. There is no point polling the device more frequently than this as the data will not be there.

An example of this is polling a UART. The maximum rate at which characters arrive is $\frac{\text{baudrate}}{10}$ characters per second (assuming 8 data bits, 1 start bit and 1 stop bit). In the following example the value **interval**

¹A peripheral device which controls a serial communications port, such as an RS232 interface.

is set to be slightly less than the shortest possible time between received characters (i.e. $\frac{10}{\text{baudrate}} - \Delta$):

```

SEQ
clock ? time
WHILE active
SEQ
-- wait until a character might be ready
time := time PLUS interval
clock ? AFTER time
{{ poll and read data from UART
data.ready (ready) -- check UART status register
IF
  ready
  SEQ
    read.data (char)
    to.buffer ! char
  TRUE
  SKIP
}}}

```

This loop only consumes processor time whilst it is actually reading the UART registers. After a character has been received and passed on, it is descheduled until just before the next character is ready, freeing the processor for other work.

This example can be readily extended to allow mixing of data from the serial port and from an occam channel:

```

SEQ
clock ? time
WHILE active
SEQ
time := time PLUS interval
PRI ALT
  clock ? AFTER time
  ... poll and read data from UART
  source ? char
  -- insert character from channel into buffer
  to.buffer ! char

```

Another simple example is a program communicating with a transputer system, emulating a terminal, and simultaneously checking the error flag of the system. The system-error flag only needs to be checked occasionally, say 10 times a second, to give the impression of instant response to an error. The following code shows how the two data sources and the error flag are all handled in a single loop:

```

SEQ
clock ? time
WHILE active
SEQ
ALT
  clock ? AFTER time
  SEQ
    ... check error pin
    time := time PLUS interval
  keyboard ? char
    ... send character to system
  link ? char
    ... display character on screen

```

This process is only scheduled when data arrives (from the keyboard or the transputer system) or it is time to check the error-flag.

It is worth noting here why this code is structured as a single **WHILE** loop rather than three parallel processes:

```
PAR
... check error-flag
... copy data from keyboard to system
{{ copy data from system to screen
WHILE active
SEQ
link ? char
... display character on screen
}}
```

Although this approach appears simpler, it introduces the problem of causing three concurrently-executing loops to terminate correctly. The solution that would usually be adopted is for each process to have an extra input channel and to terminate when a message arrives on that channel. This then means that each loop requires an **ALT** and the initial simplicity of this approach disappears.

9.5.3 A real-time clock/calendar

This example is taken from a simple disk filing system for transputers. It is a process which uses the Occam timer to maintain the date and time. The program is organised as a number of communicating processes, so the real-time clock can be interrogated by any of a number of processes which wish to know the current time or date:

```
INT hours, minutes, seconds, date :

PROC update.time (INT now)
INT new.now, delta :
SEQ
timer ? new.now
delta := new.now MINUS now
now := new.now
... use 'delta' to update hours, minutes, seconds, and date
:

VAL one.hour IS ticks.per.second * 3600 :
INT now :
SEQ
... initialise
WHILE running
ALT
-- wait for a timeout
timer ? AFTER now PLUS one.hour
SEQ
update.time (now)

-- or commands from users
ALT i = 0 FOR users
request[i] ? command
SEQ
update.time (now)
CASE command
read.time
... output time to user i
... handle other requests
```

As the OCCAM timer can only be used to measure relative times, the process keeps track of the current time and date. Whenever a user requests the time or date the timer is read. This value is subtracted from the previous timer value and this difference used to update the stored time and date values before the reply is returned to the requestor.

The OCCAM timer will eventually wrap-round, so it is important that the stored time and date values are updated periodically. To ensure that this happens, even if no requests are received from the users, there is a delayed input in the **ALT** which times out after one hour. When this happens the stored values are updated and the **ALT** reentered to wait for another request or timeout.

9.5.4 A task scheduler

The use of multiple delayed-inputs can even be extended to use a replicated **ALT** where all the times and intervals are stored in arrays. This could form the basis of a scheduler for handling a large number of peripheral devices. For example:

```
WHILE active
  ALT
    control ? CASE
      ... change time interval for a device
      ... modify enable mask for a device
      ... other commands
    ALT i = 0 FOR N
      enabled[i] & clock ? AFTER time[i]
      SEQ
        ... handle device i
        time[i] := time[i] PLUS interval[i]
```

This loop schedules tasks to handle various peripheral devices at intervals. Each peripheral has associated with it: a next time value; a boolean flag which enables its task; and a frequency at which it needs attention. These are stored in the arrays **time**, **enabled** and **interval**. There is also a channel, **control**, for modifying these parameters of the tasks associated with each device.

9.5.5 Very long delays

The example below is a procedure that can be used to generate arbitrarily long delays. As noted earlier, the greatest delay that can be generated directly by a delayed input is half the timer-cycle time. This procedure generates the desired delay as a number of shorter (in this case, one second) delays. This prevents the duration of any one delayed input being a problem and, on the transputer, is still very efficient. This process will be scheduled once a second during the delay period to perform another delayed input — this will amount to only about 2.5 μ s of processor time per second:

```
PROC delay (VAL INT seconds)
  TIMER clock :
  INT time :
  SEQ
    clock ? time
    SEQ i = 0 FOR seconds
    SEQ
      time := time PLUS ticks.per.second
      clock ? AFTER time
    :
```

9.6 Conclusions

An important application of microprocessors is in real-time control. The OCCAM language provides support for programming real-time systems. An important aspect of this is the timer. This allows measurement of

time intervals, creation of delays and scheduling of processes for given times. The timer operations are fully integrated with the control structures of the language, providing many powerful facilities especially when used with an alternative.

The transputer provides hardware and instruction level support for the timer operations. This allows them to be fast (sub-microsecond process scheduling) and efficient (processes use no processor time whilst waiting for a timer). Because the transputer has microcode and hardware support for OCCAM timers, any language executing on a transputer can be provided with the same facilities.

9.7 References

- 1 *occam 2 reference manual*, INMOS Limited, Prentice Hall 1988.
- 2 *The Transputer Databook*, INMOS Limited, 1989.
- 3 *The transputer implementation of occam*, Technical Note 21, INMOS Limited.
- 4 *The transputer instruction set: a compiler writers guide*, INMOS Limited, Prentice Hall 1988

10 Long arithmetic on the transputer

10.1 Introduction

This note describes how to use the facilities provided in the transputer and occam to implement long arithmetic, i.e. arithmetic on arbitrarily large integers.

The transputer family naturally handles integers of the wordlength of the machine (16 bit on T2xx family, 32 bit on T4xx and T8xx families). There is also particular support from a communications point of view for bytes and messages of bytes, into which all other types can be mapped. The T4xx family has special instructions to accelerate software implementation of floating point numbers, and the T8xx family has hardware floating point facilities.

occam supports bytes, 16-bit integers, 32-bit integers, 64-bit integers, 32-bit floating point and 64-bit floating point on all transputer types.

Floating point representation allows the expression and manipulation of very large values, but in so doing trades precision for range. Thus for absolute integer precision with very large number range, floating point is not appropriate.

Certain applications use long integers for other reasons, such as generating Cyclic Redundancy Checks, cryptography, spread-spectrum radio etc.

(Note that later members of the T4xx family, and all the T8xx family, include dedicated instructions for cyclic redundancy checking.)

The INMOS occam compilers give direct access to the transputer instructions through predefined procedures that compile into inline code, rather than a procedure call. This note demonstrates that the efficiency of these is such that the performance cannot be significantly improved by using assembly language.

10.2 Requirements

The need is to be able to perform arithmetic on integers of any length, not limited by the wordsize of the cpu. For simplicity, however, it is usual to implement an integer length that is a multiple of the cpu wordlength.

Clearly, arithmetic on such integers is going to be slower than on the machine's natural length, but it is a requirement that the overhead in such operations is not excessive (exact figures would be application dependent).

Certain applications have indicated needs for up to 3200 bit integers for cryptography, and 5115 bits for spread spectrum communications, so clearly the 64 bit facilities provided by OCCAM need extension.

10.3 Facilities available on the transputer

The transputer instruction set has support for long arithmetic. These include instructions which perform addition and subtraction, with carries and borrows to allow extension to arbitrary length operations. These instructions are directly available in occam as predefined procedures, for which the compiler generates in-line code, with no procedure call overhead.

Both signed and unsigned versions are available when appropriate, and the procedures are listed below.

occam predefines used for long arithmetic

LONGADD	signed add with carry
LONGSUM	unsigned add with carry
LONGSUB	signed subtract with borrow
LONGDIFF	unsigned subtract with borrow
LONGPROD	unsigned multiply with carry-in
LONGDIV	unsigned divide
SHIFTRIGHT	double word shift right
SHIFTLEFT	double word shift left
NORMALISE	double word normalise
ASHIFTRIGHT	single word arithmetic shiftright -instruction sequence
ASHIFTLEFT	single word arithmetic shiftleft -instruction sequence

The add, sub and ashift predefines would be used for the most significant word of a long integer for signed work, using unsigned for the body of the integer.

For unsigned operands, sum,diff and shifts would be used throughout.

The next section gives the occam interface for these routines. A detailed definition of their operation, in occam, is given in section 10.10, taken from reference (1). Note that this is a definition; in general the compiler inserts only parameter loads and the instruction itself.

10.4 Interface description for the Occam Predefines

10.4.1 The integer arithmetic functions

LONGADD performs the addition of signed quantities with a carry in. The function is invalid if arithmetic overflow occurs.

```
INT FUNCTION LONGADD (VAL INT left, right, carry.in)
-- Adds (signed) left word to right word with least significant bit of carry.in.
```

LONGSUM performs the addition of unsigned quantities with a carry in and a carry out. No overflow can occur.

```
INT, INT FUNCTION LONGSUM (VAL INT left, right, carry.in)
```

```
-- Adds (unsigned) left word to right word with the least significant bit of carry.in.
-- Returns two results, the first value is one if a carry occurs, zero otherwise,
-- the second result is the sum.
```

LONGSUB performs the subtraction of signed quantities with a borrow in. The function is invalid if arithmetic overflow occurs.

```
INT FUNCTION LONGSUB (VAL INT left, right, borrow.in)
```

```
-- Subtracts (signed) right word and borrow.in from left word.
```

LONGDIFF performs the subtraction of unsigned quantities with borrow in and borrow out. No overflow can occur.

```
INT, INT FUNCTION LONGDIFF (VAL INT left, right, borrow.in)
```

- Subtracts (unsigned) **right** word and **borrow.in** from **left** word.
- Returns two results, the first is one if a borrow occurs, zero otherwise,
- the second result is the difference.

LONGPROD performs the multiplication of two unsigned quantities, adding in an unsigned carry word. Produces a double length unsigned result. No overflow can occur.

```
INT, INT FUNCTION LONGPROD (VAL INT left, right, carry.in)
```

- Multiplies (unsigned) **left** word by **right** word and adds **carry.in**.
- Returns the result as two integers most significant word first.

LONGDIV divides an unsigned double length number by an unsigned single length number. The function produces an unsigned single length quotient and an unsigned single length remainder. An overflow will occur if the quotient is not representable as an unsigned single length number. The function becomes invalid if the divisor is equal to zero.

```
INT, INT FUNCTION LONGDIV (VAL INT dividend.hi, dividend.lo, divisor)
```

- Divides (unsigned) **dividend.hi** and **dividend.lo** by **divisor**.
- Returns two results the first is the quotient and the second is the remainder.

SHIFTRIGHT performs a right shift on a double length quantity. The function must be called with the number of places in range, otherwise the implementation can produce unexpected effects.

i.e. **0 <= places <= 2*bitsperword**

```
INT, INT FUNCTION SHIFTRIGHT (VAL INT hi.in, lo.in, places)
```

- Shifts the value in **hi.in** and **lo.in** right by the given number of **places**.
- Bits shifted in are set to zero.
- Returns the result as two integers most significant word first.

SHIFTLEFT performs a left shift on a double length quantity. The function must be called with the number of places in range, otherwise the implementation can produce unexpected effects.

i.e. **0 <= places <= 2*bitsperword**

```
INT, INT FUNCTION SHIFTLEFT (VAL INT hi.in, lo.in, places)
```

- Shifts the value in **hi.in** and **lo.in** left by the given number of **places**.
- Bits shifted in are set to zero.
- Returns the result as two integers most significant word first.

NORMALISE normalises a double length quantity. No overflow can occur.

INT, INT, INT FUNCTION NORMALISE (VAL INT hi.in, lo.in)

- Shifts the value in **hi.in** and **lo.in** left until the highest bit is set.
- The function returns three integer results
- The first returns the number of places shifted.
- The second and third return the result as two integers with the most significant word first;
- If the input value was zero, the first result is **2*bitsperword**.

10.4.2 Arithmetic shifts

ASHIFTRIGHT performs an arithmetic right shift, shifting in and maintaining the sign bit. The function must be called with the number of places in range, otherwise the implementation can produce unexpected effects.

i.e. **0 <= places <= bitsperword**

No overflow can occur.

N.B the result of this function is NOT the same as division by a power of two.

- Shifts the value in **operand** right by the given number of **places**.
- The status of the high bit is maintained

INT FUNCTION ASHIFTRIGHT (VAL INT operand, places) IS

ASHIFTLEFT performs an arithmetic left shift, shifting out the most significant bits, and filling the least significant bits with zeroes. The function is invalid if significant bits are shifted out, i.e. if the most significant bit changes value at any point in the shift operation, signalling an overflow. The function must be called with the number of places in range, otherwise the implementation can produce unexpected effects.

i.e. **0 <= places <= bitsperword**

N.B the result of this function is the same as multiplication by a power of two.

INT FUNCTION ASHIFTLEFT (VAL INT argument, places)

- Shifts the value in **argument** left by the given number of **places**.
- Bits shifted in are set to zero.

10.5 Methodology

This section will show the code required to provide general purpose long arithmetic. At this point the code will be written for algorithmic efficiency but Occam clarity. For ultimate performance see the performance section for optimisations.

For each operation, a procedure will be demonstrated that takes as input arrays of integers being the operands, and returns an array of integers that is the result. All operands are arbitrarily sized integer arrays, so that if fed arrays of 100 words, that is the size for which the operation will be performed. To be safe, such routines should test for compatibility of the sizes of the three arrays passed, which should be identical for add and subtract, n,n and 2n for multiply and divide. The code for this is omitted from the examples below for clarity.

The first two, add and subtract, are trivial, and as the result has the same length as the operands, are shown for the operation **a := a op b**. Multiply is simple to program, but there are subtleties of the powerful transputer operations to be noted to achieve algorithmic efficiency. Divide accentuates this even further, and is more dependent on the algorithm, for which one is referred to ref(2). As the result has a different length

from the operands, multiply and divide are illustrated for the operation $a := b \text{ op } c$.

All arrays are assumed to be stored little-endian, i.e. the element with the lowest index is the least significant. The code could equally support big-endian arrays if required. Within a word, the bits MUST be stored little-endian to match the transputer hardware.

10.5.1 Addition

As an example, the general addition code will be derived from first double-length, then triple-length.

The following example adds two double length unsigned quantities explicitly. occam actually supports this implicitly as type **INT64**.

```

PROC add.double.unsigned ([2]INT result,VAL [2]INT righthop)
  INT carry:
  SEQ
    carry , result[0] := LONGSUM (result[0],righthop[0],0)
    carry , result[1] := LONGSUM (result[1],righthop[1],carry)
  .

```

To make this a signed operation, the final operation, i.e. that on the most significant word, is performed using the **LONGADD** predefine. This produces exactly the same result, but will raise the error flag if overflow should occur.

```

PROC add.double.signed ([2]INT result,VAL [2]INT rightop)
  INT carry:
  SEQ
    carry , result[0] := LONGSUM (result[0],rightop[0],0)
    result[1] := LONGADD (result[1],rightop[1],carry)
  :

```

To extend beyond double length simply involves using the same operation again.

```

PROC add.triple.signed ([3]INT result,VAL [3]INT righthop)
  INT carry:
  SEQ
    carry , result[0] := LONGSUM (result[0],righthop[0],0)
    carry , result[1] := LONGSUM (result[1],righthop[1],carry)
    result[2] := LONGADD (result[2],righthop[2],carry)

```

To create the general purpose case, one uses a loop rather than in-line code, and controls the loop with the length of the array passed in, which is accessible at runtime in occam. The following code performs a long-add on compatible arrays of integers. This version operates on signed integers, clearly one could make it slightly smaller for unsigned integers by looping one more time and omitting the last statement.

10.5.2 Subtraction

Similarly for subtraction, using the appropriate pair of predefines in exactly the same harness, with the same comments applying.

```
PROC subtract.long.signed.int ([]INT result,VAL []INT rightop)

INT borrow:
VAL last.index IS (SIZE rightop) -1:

SEQ
borrow := 0

SEQ i = 0 FOR last.index
borrow,result[i] := LONGDIFF (result[i],rightop[i],borrow)

result[last.index] := LONGSUB(result[last.index],
rightop[last.index],borrow)
:
```

10.5.3 Multiplication

Multiplication is more complex. The algorithm is best developed by mimicking a child doing long multiplication. Imagine multiplying 12 by 34. First, one removes the sign from both operands, generating the result sign as s1 XOR s2.

Then one takes the least significant digit of the multiplier(4) and multiplies all the digits of the multiplicand by it, writing the answer down with no shift to the left. One then takes the next digit to the left(3) and multiplies all digits of the multiplicand by it, writing the result down with a one digit shift to the left. This continues until all digits have been multiplied, then all the partial results are added for a final result.

$$\begin{array}{r}
12 & 12 & 12 & 12 \\
\times 34 & 34 & 34 & 34 \\
\hline
& 48 & 48 & 48 \\
& 36 & 36 & 36 + \\
\hline
& & & 408 \\
\hline
\end{array}$$

To analyse this operation, consider the position we write each result. Units times units we write in the units column, tens times tens in the hundreds column, so clearly the destination column is 10^{m+n} , where m and n are the powers of ten of the corresponding operand columns.

To implement this directly on a computer would require a large amount of memory. A 100 word integer would require 100 rows of intermediate results, and each row would be around 100 words... i.e. ten thousand words of memory, or 40K bytes on a 32 bit machine. Also, the control and implementation of the final add would use excessive cpu time.

The solution on the transputer instruction set, mapped into OCCAM via the predefined procedures, is to incorporate the add operation into each multiply, so that only one intermediate result is maintained, and that can clearly occupy the same space as the final result will occupy eventually. In order to achieve this, the result array must be cleared before use. Remember this code is designed for clarity... efficiency comes later. Clearly $(i+j)$ could be evaluated once in the inner loop, and clearly `leftop[i]` need only be accessed once per outer loop.

```

PROC multiply.long.unsigned.int ([]INT result,VAL []INT leftop,rightop)
SEQ
  SEQ i = 0 FOR SIZE result
    result[i] := 0
  SEQ i = 0 FOR SIZE leftop
    INT carry:
    SEQ
      carry := 0
    SEQ j = 0 FOR SIZE rightop
      INT temp:
      SEQ
        temp, result[ i+j ] :=
          LONGPROD(leftop[i], rightop[j], result[i+j])
        carry, result[ i+j+1 ] := LONGSUM(result[i+j+1],temp,carry)
      :

```

The reason this comes out so simply comes from the design of the instructions. Note that the multiply operation performs an accumulation for us, as its carry input takes a full word width operand. We need a double width accumulate however, so a long-sum is used to complete the operation for the upper word of the result. Note that the carry from the long-sum is required TWO words further up the result. This is achieved by holding it until the next iteration, when the index will be one higher, and storing it one higher than the index as usual.

Note that this algorithm will actually multiply arrays of differing length, providing the length of the result array is appropriate. Note also that as the loop counts start at zero, the highest loopcontrol value being $n-1$, the final access of the final carry operation is to `imax+jmax+1`. The result array is of size $2n$, but the last access is to index $(n-1) + (n-1) + 1$, i.e. $2n-1$, the correct last address.

There is a more efficient way of dealing with signed multiply than mimicking the human. Deleting a minus sign is easy, but negating a very long twos- complement integer may not be, as it could affect the bit pattern of every word.

The solution is to perform a multiply assuming the operands are both positive, and then to correct the result by subtraction if either of them was negative, as shown below :

```
PROC multiply.long.signed.int ([]INT result, VAL []INT leftop,rightop)
  SEQ
    multiply.long.unsigned.int (result,leftop,rightop)

    result.top.half IS
      [result FROM SIZE leftop FOR SIZE rightop]:
    VAL leftop.top.word IS leftop[(SIZE leftop) -1]:
    IF
      leftop.top.word < 0
        subtract.long.unsigned.int ( result.top.half,rightop)
    TRUE
    SKIP

    result.top.half IS
      [result FROM SIZE rightop FOR SIZE leftop]:
    VAL rightop.top.word IS rightop[(SIZE rightop) -1]:
    IF
      rightop.top.word < 0
        subtract.long.unsigned.int ( result.top.half,leftop)
    TRUE
    SKIP
  :
```

Note that the above code handles differing length arrays, but assumes that the long subtract procedure can also, which the example given thereof cannot.

10.5.4 Division

Division is another order of magnitude more complex than multiplication. As with multiplication, one mimics the human, extracting the sign operation first. To perform the division, a human makes a guess at a partial result, multiplies back and subtracts to achieve a remainder. If the remainder is negative, the guess was too large, so is repeated. If the remainder is greater than the divisor, the guess was too small.

The computer does exactly the same, but it can be 'helped' in making its initial guess appropriately.(Ref 2, Knuth)

```

PROC divide.long.int ([]INT result, VAL []INT lefttop,righttop)
  ... declarations
  SEQ
    ... extract signs from operands -> s.r, left.u,right.u
    ... normalise divisor (right.u)
    higher.word.left := 0
    SEQ i = 0 FOR SIZE lefttop
      VAL i.l.rev IS (SIZE lefttop)-1) - i:
      VAL i.r.rev IS (SIZE righttop)-1) - i:
    SEQ
      IF
        higher.word.left = right.u[i.r.rev]
        temp.result := MAX INT
      TRUE
        temp.result,remainder := LONGDIV ( higher.word.left,
                                             left.u[i.r.rev], right.u[i.l.rev])

      IF
        temp.result<>0
        SEQ
          multiply.long.unsigned.int( temp.vec,
                                       [temp.result],right.u)
          subtract.long.unsigned.int(left.u,temp.vec)

        WHILE (left.u[(SIZE left.u)-1] /\ signbit) <> 0
          SEQ
            temp.result := temp.result - 1
            add.long.unsigned.int( left.u, right.u)

        TRUE
        SKIP

      result[i.l.rev] := temp.result
      higher.word.left := left.u[i.l.rev]

    ... unnormalise wrt previous normalisation
    ... replace sign of result.
  :

```

The trial result **temp.result** can be either one or two units too large. To cover this possibility, the **WHILE** loop tests if the current remainder is negative, and adds back the divisor as many times as necessary to remedy the situation, adjusting the estimated quotient each time.

There is also a marginally faster version in Knuth that adds an extra test to improve the guess, and thus guarantees to be accurate or one high, never two high.

10.6 Shift Operations

The shift operations are simple in that there is only a single primitive to be used per loop, but note should be taken that the same performance enhancements as will be demonstrated on multiplication in section 10.7.1, particularly opening out the loops and using abbreviations on blocks of sixteen words, should be used here, as the actual shift operation is dwarfed in CPU time by the loop control around it otherwise.

Note that the examples given here assume a shift of less than the natural wordlength of the machine. In fact for any shift of 16 bits or greater, it is more efficient to do a block move assignment to handle the byte offset, and only tidy up the remaining bit-shift using this code.

It should also be noted that the shift left and rotate left operations should be performed from the 'top' of the vector, which means reversing the loop index, whilst the shift/rotate right operations are done from the lowest index.

```
PROC rotate.long.int.left ([]INT buffer, VAL INT n)
  INT highest,dump:
  VAL last IS (SIZE buffer) -1:
  SEQ
    highest := buffer[last]
    SEQ ii = 0 FOR last
      VAL i IS last - ii:
      buffer[i],dump := SHIFTLEFT(buffer[i],buffer[i-1],n)
      buffer[0],dump := SHIFTLEFT(buffer[0],highest,n) :
```

To make it an unsigned shift left, one simply omits the wrap around. Note that as a result, the last operation can be done with a conventional single length shift.

```
PROC shift.long.int.left ([]INT buffer, VAL INT n)
  INT dump:
  VAL last IS (SIZE buffer) -1:
  SEQ
    SEQ ii = 0 FOR last
      VAL i IS last - ii:
      buffer[i],dump := SHIFTLEFT(buffer[i],buffer[i-1],n)
      buffer[0] := buffer[0] << n :
```

For arithmetic shift, one simply handles the first (most significant) word separately. Because the arithmetic shift on the transputer is single length, it is appropriate to use this for overflow checking only, and repeat the operation with a logical shift. Thus one simply inserts the following line before the loop. (This does not apply to right shift, see later)

```
dump := ASHIFTLEFT(buffer[last],n)
```

Similarly, for right shifts, noting that we progress the other direction along the vector:

```
PROC rotate.long.int.right ([]INT buffer, n)
  INT first,dump:
  VAL last IS (SIZE buffer) -1:
  SEQ
    first := buffer[0]
    SEQ i = 0 FOR last
      dump,buffer[i] := SHIFTRIGHT(buffer[i+1],buffer[i],n)
      dump,buffer[last] := SHIFTRIGHT(first,buffer[last],n) :
```

Again, shift and arithmetic shift involve removing the wraparound, and using **ASHIFTRIGHT** for the last line, respectively. Note that the arithmetic shift is done last, not first, with the right shift, as we progress up the vector from least significant to most significant.

10.6.1 Normalisation

Normalisation is similar to shifting, but is conditional on the data, so is split into five sections. Firstly, from the most significant end, the first non-zero word is found. That word is then normalised, using a single application of the normalise predefined, which also pulls in any bits needed from the next word down. The third operation is to shift all the remaining words left by the number of places returned by the normalise routine, and then the fourth is to move the active words, now correctly word aligned, to the top of the array. The final operation is to clear the vacated words at the bottom of the array.

```

PROC normalise.long.integer ([]INT buffer)
  INT pointer, trash :
  INT places :
  VAL len IS SIZE buffer :
  SEQ
    --find first non-zero word
    IF
      IF i = 1 FOR len
        buffer[len MINUS i] <> 0
        pointer := len MINUS i
      TRUE
      pointer := 0

    --normalise that word, pulling in bits as
    --needed from next word down
    places,buffer[len],trash :=
      NORMALISE(buffer[pointer],buffer[pointer-1])

  VAL diff IS (len) MINUS (pointer) :
  SEQ
    --shift the rest of the buffer left by the
    --same number of bits
    shift.left ([buffer FROM 0 FOR pointer],places)

    --block move up to the top of the buffer
    SEQ i = 0 FOR (pointer PLUS 1)
      VAL ii IS (pointer) MINUS (i) :
      buffer[ii PLUS diff] := buffer[ii]

    --fill the vacated words with zeros
    SEQ ii = 0 FOR (diff MINUS 1 )
      buffer [ii] := 0
:
:
```

10.7 Performance

10.7.1 Optimisation, using multiplication as an example

Addition and subtraction are sufficiently simple, and sufficiently directly built on the transputer instructions that little algorithmic optimisation can be done. However, as the arithmetic operations are so fast, they suffer greatly from the loop control overhead, so benefit greatly from opening out the loops.

However multiplication can be considerably optimised, in three steps. The first is to take invariant expressions outside loops, saving both indexing and arithmetic. The second is to set up abbreviations (or pointers) to frequently accessed arrays.

The third, and most beneficial, is to open out the inner loop by some factor, ideally sixteen. This both saves loop control, but combined with the abbreviations means that many array accesses are reduced to constant indices, which are very fast on the transputer. This does have the restriction that the arrays must be a multiple of the opening factor, rather than truly variable size.

The second method is a stepping stone to the third, and the performance benefit of it is not seen until the loop is opened.

The following sections show the inner two loops suitably modified:

Simple Code

```

SEQ i = 0 FOR SIZE leftop

SEQ
  carry := 0
  VAL leftop.i IS leftop[i]: 

SEQ j = 0 FOR SIZE rightop    --usually the same size

  VAL ij  IS i + j :
  VAL ij1 IS ij + 1 :
  SEQ
    temp, result[ij] :=
      LONGPROD(leftop.i, rightop[j], result[ij])

    carry, result[ij1] := LONGSUM(result[ij1],temp,carry)

```

Using Array Abbreviations and opened loops

```

SEQ i = 0 FOR SIZE leftop

SEQ
  carry := 0
  VAL leftop.i IS leftop[i]: 

SEQ j = 0 FOR (SIZE rightop)/opening.factor

  VAL ij  IS i + j :
  VAL ij1 IS ij + 1 :
  VAL rightop.j IS [rightop FROM (J TIMES opening.factor )
                    FOR opening.factor]:
  result.ij IS [result FROM i+(j TIMES opening.factor)
                FOR opening.factor + 1]:
  SEQ

    VAL k IS 0:
    SEQ
      temp, result.ij[k] :=
        LONGPROD(leftop.i, rightop.j[k],
                  result.ij[k])

      carry, result.ij[k+1] :=
        LONGSUM(result.ij[k+1],temp,carry)

    VAL k IS 1:
    SEQ
      .
      .
      .
    VAL k IS opening.factor-1:
    SEQ
      .
      .

```

Points to note here are that now all access to the arrays are of the form k or k + constant. As k itself is a constant, this will be folded at compile time, so all array accesses in the inner loop have constant indices and are thus very fast, using instructions designed for that operation.

10.7.2 Performance Figures

The following table gives the performance of each of the algorithms mentioned above, and for the optimised versions with open loops, and finally for the same operation in assembly language. These numbers are all for a 20MHz 32 bit transputer.

Performance in microseconds for 3200 bit integer

OPERATION	SIMPLE	OPTIMISED OCCAM	ASSEMBLER
ADD/SUBTRACT	305	164	164
MULTIPLY	57700	43500	43485
DIVIDE	36-72ms	—	—
SHIFT/ROTATE 1 bit	300	167	153
SHIFT/ROTATE 8 bit	336	202	187
SHIFT/ROTATE 15 bit	372	236	211
SHIFT/ROTATE 8*N (Block Move)	41	41	41
NORMALISE	384	—	—

Note that assembler coding gains very little, and also the spectacular performance of the shift 8*N version, due to the block move hardware in the transputer.

Divide spends most of its time multiplying out the estimated results, so has little scope for optimising other than in the multiply. Divide and Normalise have data-dependent execution times. The figures given are mid-range, i.e. (max.time+min.time)/2.

10.8 Conclusions

The long arithmetic facilities in occam allow very efficient implementation of arbitrary length integers, meaning that there is no benefit in using assembler.

The underlying instructions of the transputer are directly accessible as in-line procedures in occam, and are themselves sophisticated primitives allowing maximum performance in such computationally intense applications.

The assistance of Roger Shepherd (INMOS, Architecture Group) is gratefully acknowledged, particularly for help with the divide algorithms, and of Andy Hamilton and John Carey, INMOS Central Applications, for benchmarking and verification work.

10.9 References

- 1 occam 2 Reference Manual, INMOS Limited, Prentice Hall 1988, ISBN 0-13-629312-3
(Particularly Appendix L,pp105-113.)
- 2 Seminumerical Algorithms, The Art of Computer Programming Vol 2, Donald Knuth,
Addison-Wesley 1969,1981, ISBN 0-201-03822-6(v.2) (Particularly Section 4.3)

10.10 The Occam Predefined Procedures

10.10.1 Definition of terms

For the purpose of explanation imagine a new type *INTEGER*, and the associated conversion. This imaginary type is capable of representing the complete set of integers and is presumed to be represented as an infinite bit two's complement number. With this one exception the following are OCCAM descriptions of the various arithmetic functions.

```
-- constants used in the following description
VAL bitsperword IS machine.wordsizes(INTEGER) :
VAL range IS storeable.values(INTEGER) :
-- range = 2bitsperword
VAL maxint IS INTEGER (MOSTPOS INT) :
-- maxint = (range/2 - 1)
VAL minint IS INTEGER (MOSTNEG INT) :
-- minint = -(range/2)

-- INTEGER literals
VAL one IS 1(INTEGER) :
VAL two IS 2(INTEGER) :
-- mask
VAL wordmask IS range - one :
```

In occam, values are considered to be signed. However, in these functions the concern is with other interpretations. In the construction of multiple length arithmetic the need is to interpret words as containing both signed and unsigned integers. In the following the new *INTEGER* type is used to manipulate these values, and other values which may require more than a single word to store.

The sign conversion of a value is defined in the functions *unsign* and *sign*. These are used in the description following but they are NOT functions themselves.

10.10.2 The integer arithmetic functions

LONGADD performs the addition of signed quantities with a carry in. The function is invalid if arithmetic overflow occurs.

The action of the function is defined as follows:

```
INT FUNCTION LONGADD (VAL INT left, right, carry.in)
-- Adds (signed) left word to right word with least significant bit of carry.in.
```

```
INTEGER sum.i, carry.i, left.i, right.i :
VALOF
SEQ
  carry.i := INTEGER (carry.in /\ 1)
  left.i := INTEGER left
  right.i := INTEGER right
  sum.i := (left.i + right.i) + carry.i
-- overflow may occur in the following conversion
-- resulting in an invalid process
RESULT INT sum.i
:
```

LONGSUM performs the addition of unsigned quantities with a carry in and a carry out. No overflow can occur.

The action of the function is defined as follows:

```
INT, INT FUNCTION LONGSUM (VAL INT left, right, carry.in)

-- Adds (unsigned) left word to right word with the least significant bit of carry.in.
-- Returns two results, the first value is one if a carry occurs, zero otherwise,
-- the second result is the sum.

INT carry.out :
INTEGER sum.i, left.i, right.i :
VALOF
SEQ
  left.i := unsign (left)
  right.i := unsign (right)
  sum.i := (left.i + right.i) + INTEGER (carry.in /\ 1)
  IF                      -- assign carry
    sum.i >= range
    SEQ
      sum.i := sum.i - range
      carry.out := 1
    TRUE
    carry.out := 0
  RESULT carry.out, sign (sum.i)
:
```

LONGSUB performs the subtraction of signed quantities with a borrow in. The function is invalid if arithmetic overflow occurs.

The action of the function is defined as follows:

```
INT FUNCTION LONGSUB (VAL INT left, right, borrow.in)

-- Subtracts (signed) right word from left word and subtracts borrow.in from the result.

INTEGER diff.i, borrow.i, left.i, right.i :
VALOF
SEQ
  borrow.i := INTEGER (borrow.in /\ 1)
  left.i   := INTEGER left
  right.i  := INTEGER right
  diff.i   := (left.i - right.i) - borrow.i
  -- overflow may occur in the following conversion
  -- resulting in an invalid process
  RESULT INT diff.i
:
```

LONGDIFF performs the subtraction of unsigned quantities with borrow in and borrow out. No overflow can occur.

The action of the function is defined as follows:

```
INT, INT FUNCTION LONGDIFF (VAL INT left, right, borrow.in)

-- Subtracts (unsigned) right word from left word and subtracts borrow.in from the result.
-- Returns two results, the first is one if a borrow occurs, zero otherwise,
-- the second result is the difference.

INTEGER diff.i, left.i, right.i :
VALOF
SEQ
left.i := unsign (left)
right.i := unsign (right)
diff.i := (left.i - right.i) - INTEGER (borrow.in /\ 1)
IF      -- assign borrow
diff.i < 0
SEQ
diff.i := diff.i + range
borrow.out := 1
TRUE
borrow.out := 0
RESULT borrow.out, sign (diff.i)
:
```

LONGPROD performs the multiplication of two unsigned quantities, adding in an unsigned carry word. Produces a double length unsigned result. No overflow can occur.

The action of the function is defined as follows:

```
INT, INT FUNCTION LONGPROD (VAL INT left, right, carry.in)

-- Multiplies (unsigned) left word by right word and adds carry.in.
-- Returns the result as two integers most significant word first.

INTEGER prod.i, prod.lo.i, prod.hi.i, left.i, right.i, carry.i :
VALOF
SEQ
carry.i := unsign (carry.in)
left.i := unsign (left)
right.i := unsign (right)
prod.i := (left.i * right.i) + carry.i
prod.lo.i := prod.i REM range
prod.hi.i := prod.i / range
RESULT sign (prod.hi.i), sign (prod.lo.i)
:
```

LONGDIV divides an unsigned double length number by an unsigned single length number. The function produces an unsigned single length quotient and an unsigned single length remainder. An overflow will occur if the quotient is not representable as an unsigned single length number. The function becomes invalid if the divisor is equal to zero.

The action of the function is defined as follows:

```
INT, INT FUNCTION LONGDIV (VAL INT dividend.hi, dividend.lo, divisor)

-- Divides (unsigned) dividend.hi and dividend.lo by divisor.
-- Returns two results the first is the quotient and the second is the remainder.

INTEGER divisor.i, dividend.i, hi, lo, quot.i, rem.i :
VALOF
SEQ
hi := unsign (dividend.hi)
lo := unsign (dividend.lo)
divisor.i := unsign (divisor)
dividend.i := (hi * range) + lo
quot.i := dividend.i / divisor.i
rem.i := dividend.i REM divisor.i
-- overflow may occur in the following conversion of quot.i
-- resulting in an invalid process
RESULT sign (quot.i), sign (rem.i)
:
```

SHIFTRIGHT performs a right shift on a double length quantity. The function must be called with the number of places in range, otherwise the implementation can produce unexpected effects.

i.e. $0 \leq \text{places} \leq 2 \times \text{bitsperword}$

The action of the function is defined as follows:

```
INT, INT FUNCTION SHIFTRIGHT (VAL INT hi.in, lo.in, places)

-- Shifts the value in hi.in and lo.in right by the given number of places.
-- Bits shifted in are set to zero.
-- Returns the result as two integers most significant word first.
```

```
INT hi.out, lo.out :
VALOF
IF
  (places < 0) OR (places > (two*bitsperword))
SEQ
  hi.out := 0
  lo.out := 0
TRUE
INTEGER operand, result, hi, lo :
SEQ
  hi := unsign (hi.in)
  lo := unsign (lo.in)
  operand := (hi << bitsperword) + lo
  result := operand >> places
  lo := result /\ wordmask
  hi := result >> bitsperword
  hi.out := sign (hi)
  lo.out := sign (lo)
RESULT hi.out, lo.out
:
```

SHIFTLEFT performs a left shift on a double length quantity. The function must be called with the number of places in range, otherwise the implementation can produce unexpected effects.

i.e. $0 \leq \text{places} \leq 2 \times \text{bitsperword}$

The action of the function is defined as follows:

```

INT, INT FUNCTION SHIFTLEFT (VAL INT hi.in, lo.in, places)

-- Shifts the value in hi.in and lo.in left by the given number of places.
-- Bits shifted in are set to zero.
-- Returns the result as two integers most significant word first.

VALOF
  IF
    (places < 0) OR (places > (two*bitsperword))
    SEQ
      hi.out := 0
      lo.out := 0
  TRUE
    INTEGER operand, result, hi, lo :
    SEQ
      hi := unsign (hi.in)
      lo := unsign (lo.in)
      operand := (hi << bitsperword) + lo
      result := operand << places
      lo := result /\ wordmask
      hi := result >> bitsperword
      hi.out := sign (hi)
      lo.out := sign (lo)
  RESULT hi.out, lo.out
:

```

NORMALISE normalises a double length quantity. No overflow can occur.

The action of the function is defined as follows :

```

INT, INT, INT FUNCTION NORMALISE (VAL INT hi.in, lo.in)

-- Shifts the value in hi.in and lo.in left until the highest bit is set.
-- The function returns three integer results
-- The first returns the number of places shifted.
-- The second and third return the result as two integers with the most significant word first;
-- If the input value was zero, the first result is 2*bitsperword.

INT places, hi.out, lo.out :
VALOF
IF
  (hi.in = 0) AND (lo.in = 0)
    places := INT (two*bitsperword)
  TRUE
    VAL msb IS one << ((two*bitsperword) - one) :
    INTEGER operand, hi, lo :
    SEQ
      lo := unsign (lo.in)
      hi := unsign (hi.in)
      operand := (hi << bitsperword) + lo
      places := 0
      WHILE (operand /\ msb) = 0
        SEQ
          operand := operand << one
          places := places + 1
          hi := operand / range
          lo := operand REM range
          hi.out := sign (hi)
          lo.out := sign (lo)
      RESULT places, hi.out, lo.out
:

```

10.10.3 Arithmetic shifts

ASHIFTRIGHT performs an arithmetic right shift, shifting in and maintaining the sign bit. The function must be called with the number of places in range, otherwise the implementation can produce unexpected effects.

i.e. `places` \leq `bitsperword`

No overflow can occur.

N.B the result of this function is NOT the same as division by a power of two.

e.g. $-1/2 = 0$
ASHIFTRIGHT ($-1, 1$) = -1

The action of the function is defined as follows:

-- Shifts the value in `operand` right by the given number of `places`.
-- The status of the high bit is maintained

```

INT FUNCTION ASHIFTRIGHT (VAL INT operand, places) IS
  INT( INTEGER(operand) >> places ) :

```

ASHIFTLEFT performs an arithmetic left shift. The function is invalid if significant bits are shifted out, signalling an overflow. The function must be called with the number of places in range, otherwise the implementation can produce unexpected effects.

i.e. `0 <= places <= bitsperword`

N.B the result of this function is the same as multiplication by a power of two.

The action of the function is defined as follows:

```
INT FUNCTION ASHIFTLEFT (VAL INT argument, places)

-- Shifts the value in argument left by the given number of places.
-- Bits shifted in are set to zero.

INTEGER result.i :
VALOF
  result.i := INTEGER(argument) << places
  -- overflow may occur in the following conversion
  -- resulting in an invalid process
RESULT INT result.i
:
```

10.10.4 Word rotation

ROTATERIGHT rotates a word right. Bits shifted out of the word on the right, re-enter the word on the left. The function must be called with the number of places in range, otherwise the implementation can produce unexpected effects.

i.e. `0 <= places <= bitsperword`

No overflow can occur.

The action of the function is defined as follows:

```
INT FUNCTION ROTATERIGHT (VAL INT argument, places)

-- Rotates the value in argument by the given number of places.

INTEGER high, low, argument.i :
VALOF
  SEQ
    argument.i := unsign(argument)
    argument.i := (argument.i * range) >> places
    high := argument.i / range
    low := argument.i REM range
  RESULT INT(high \/ low)
:
```

ROTATELEFT rotates a word left. Bits shifted out of the word on the left, re-enter the word on the right. The function must be called with the number of places in range, otherwise the implementation can produce unexpected effects.

i.e. $0 \leq \text{places} \leq \text{bitsperword}$

The action of the function is defined as follows:

```
INT FUNCTION ROTATELEFT (VAL INT argument, places)
```

```
-- Rotates the value in argument by the given number of places.
```

```
INTEGER high, low, argument.i :
```

```
VALOF
```

```
SEQ
```

```
    argument.i := unsign(argument)
```

```
    argument.i := argument.i << places
```

```
    high := argument.i / range
```

```
    low := argument.i REM range
```

```
    RESULT INT(high \& low)
```

```
:
```

11 Exploiting concurrency: a ray tracing example

11.1 Introduction

The INMOS transputer [1] is a family of VLSI microcomputers with processor, memory and communication links for direct connection to other transputers on a single chip, (Figure 11.1). Concurrent systems can be constructed from a collection of transputers which operate concurrently and communicate through links. To provide maximum speed with minimum hardware the transputer uses point-to-point serial communication links.

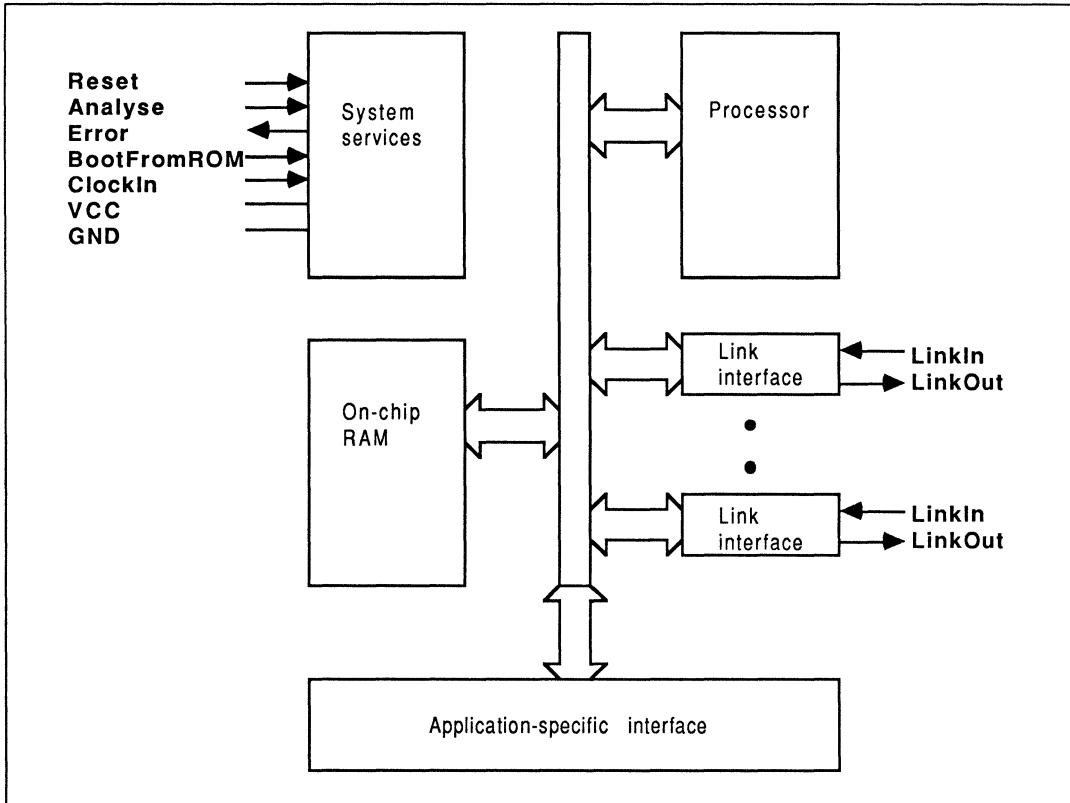


Figure 11.1 Transputer architecture

The first transputer available was the IMS T414, a 32-bit microprocessor with a throughput of 10 MIPS (million instructions per second). It has 2 kilobytes of fast (50ns cycle) on-chip static RAM and four INMOS serial links. The 32-bit multiplexed address/data bus allows up to 4 gigabytes of external memory to be accessed. The IMS T800 transputer is compatible with the T414 but includes floating-point hardware and 4K of internal RAM.

This chapter describes the implementation of a computer graphics program on an array of transputers. The technique used to distribute the work among the transputers is known as a processor farm and is independent of the application. The same approach is suitable for any algorithm which can be subdivided into independent subproblems. For example, another graphics program, the Mandelbrot set, has been distributed in the same way as well as a financial forecasting program and a simulation of metal deposition. The entire program is written in occam [2], a language designed to simplify the programming of concurrent systems. Again, however, the main part of the program could have been written in any suitable language such as C or FORTRAN. Only those parts of the program which deal explicitly with concurrency and the distribution of

work are easier to describe in occam.

The graphics program described here was written to provide a demonstration of the performance obtainable by using large numbers of transputers. We used a technique known as ray tracing which can generate very realistic images but requires massive amounts of computer power. This is an ideal application for transputers as the calculations for each pixel (picture element) on the screen are independent of one another and so can be done in parallel on separate processors. In addition, the complexity of the task means that the time spent calculating is much greater than that spent passing data between processors.

The completed program has two important properties. Firstly, processing speed is directly proportional to the number of transputers used. Virtually any desired performance can be simply obtained by the addition of more transputers. The second feature, which came about as a side-effect of the program structure, is that the system is remarkably robust. Individual transputers can be removed from the system, while the program is running, and the system will continue to function although with reduced performance and possibly some loss of data.

11.2 Logical architecture

11.2.1 Ray tracing

The basic ray tracing algorithm used is that described by Turner Whitted in his classic paper [3]. A brief description of the technique is given here.

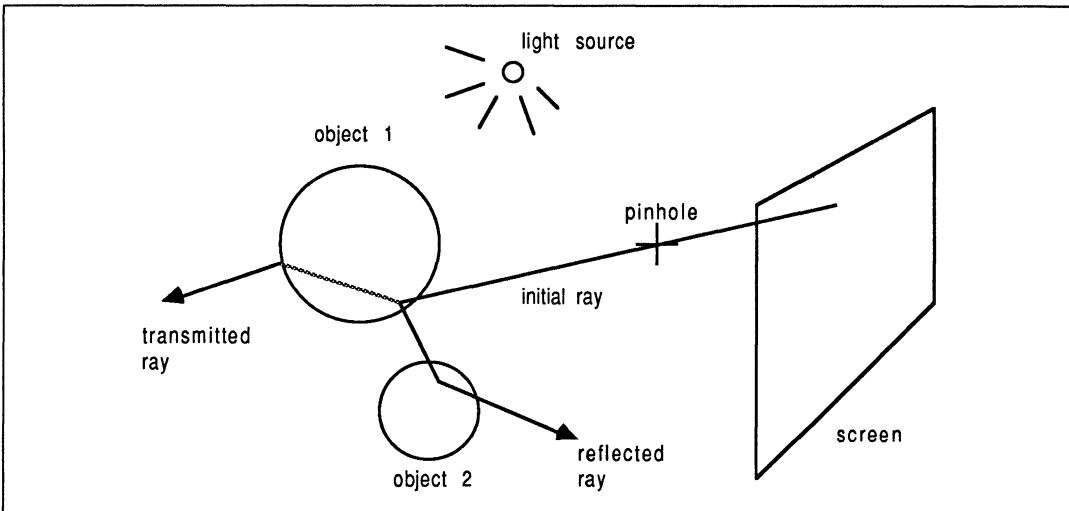


Figure 11.2 Ray tracing

The colour and intensity of each pixel on the screen is determined by calculating the path of a ray projected from the screen through a pinhole, (see Figure 11.2). This ray is tested for intersection with each object in the world model by solving the equation of the line and the surface of the object. This is the reason spheres are so common in ray tracing programs; they are simple to intersect. When the closest point of intersection is found the ray will be reflected and several new rays may be produced. If the object is transparent then a ray is generated which passes through the object, its path modified by the laws of refraction. The effects of shadow casting are handled by sending rays from the point of intersection towards each light source in turn. If this ray intersects an object which is nearer than the light source then this will cast a shadow on the first object. A recurring problem in computer graphics is aliasing, which appears as coarse steps in the image. This is caused by undersampling of the image and can be reduced by increasing the sampling frequency, i.e. tracing several rays for each point on the screen. The number of extra rays traced can be reduced by only oversampling when aliasing is likely to be most objectionable, for instance where there is a sharp change in

intensity at the boundary of objects.

In this way a tree of rays, and the surfaces with which they have intersected, is generated for each pixel. The final colour of the image at this point is calculated by traversing the tree and applying a shading model at each node. This model uses the intensity and positions of the various light sources and the coefficients of reflection and transmission for the objects intersected to determine the intensity of the pixel.

Our implementation of this algorithm, in its simplest form, is not particularly efficient. The time taken to render a scene increases exponentially with the number of objects and light sources as each ray has to be tested for intersection with every object and then a ray fired at every light source to test for shadowing. This shadow ray then has to be tested against every object. Also all calculations are done in floating-point for simplicity which traditionally imposes a considerable performance penalty. It has been estimated that a straightforward ray tracing program like this will spend over 75% of the time performing tests for intersection, so performance is very dependent on the speed of floating-point operations. The T414 has some extra instructions to provide microcode support for floating-point operations and the T800's on-chip FPU enables floating-point operations to be performed at about the same speed as integer operations.

There are many ways in which the basic ray tracing algorithm can be improved (e.g. by the use of space subdivision or bounding volumes around objects) so many implementations could well be faster on a single processor. However, a more sophisticated implementation would also benefit from the use of multiple transputers.

11.2.2 Introducing concurrency

The calculations performed for each pixel on the screen are completely independent so they can be performed in any order and on any number of processors. One way of distributing the work to a number of processors is shown in Figure 11.3.

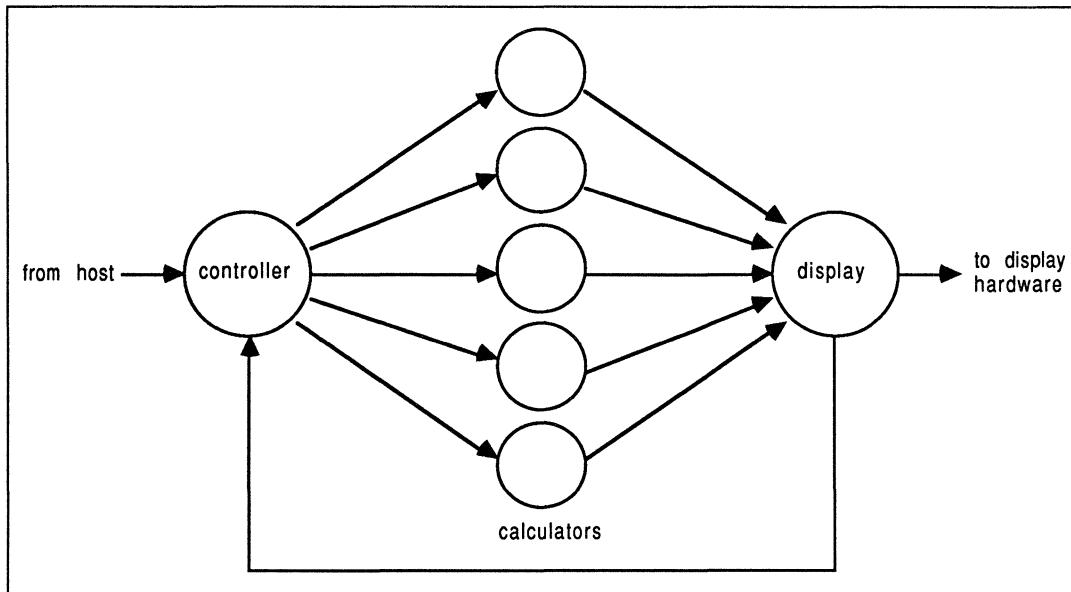


Figure 11.3 Logical architecture

This requires three different processes running concurrently on one, or more, processors: a controller which interfaces with the user or host computer to provide a description of the scene being viewed and allocates work to processors; an intersect and shading calculator, which can be replicated any number of times, to render the pixels; and a display process which collects the results from each rendering process and drives the graphic display. It can be seen that this structure is not related to the ray tracing algorithm and is, in fact,

suitable for any problem which can be broken into independent subproblems. A system like this in which a controller farms out work to a number of application-specific processes has become known as a processor farm.

Every calculating process is first given the description of the scene and then processing work can be allocated by the controller which gives each calculator pixels to evaluate. When the calculations have been completed the results are passed out to the display process. The display process then informs the controller that there is now a free processor and another pixel is sent out for evaluation. The amount of computation required varies from pixel to pixel and this method automatically balances the load amongst the processors and ensures they are all kept busy.

An interesting idea here is that the pixels do not need to be generated in sequence and, if they are generated in some pseudo-random order, a good impression of the final picture can be obtained well before every pixel has been evaluated. This could be particularly useful in a computer-aided design system where the user wishes to generate different views of an object in rapid succession.

11.3 Physical architecture

11.3.1 General description

It appears, at first sight, that the above architecture cannot be mapped directly onto a network of transputers because of the fixed number of links available. However, it is very simple to arrange for the controller to communicate with any transputer in a network by passing messages through the intervening transputers. For simplicity, the ray tracing program was mapped onto a linear array of transputers as shown in Figure 11.4. Each transputer link implements two Occam channels, one in each direction, so this mapping only uses two of the four links available on a T414 or T800.

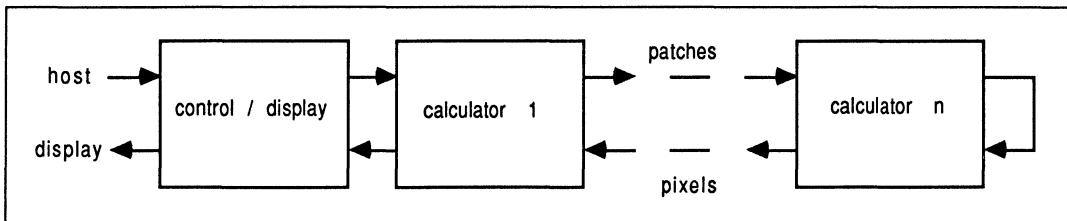


Figure 11.4 Physical architecture

Here the control and display processes are executed in parallel on one transputer and the rest of the transputers do the intersection and shading calculations. In fact the first transputer also does these calculations and the same, parameterised, program is loaded on to every transputer. However, it is simpler to view the system as shown above. This method of mapping processes onto transputers requires that each transputer also executes routing processes. These pass commands and data along the array from the controlling process and pass results back for display. This implies some sort of command protocol for identifying the nature and destination of data. This is simplified by using a linear connection of transputers; the routing process on each transputer only needs to decide whether a message is to be accepted locally or passed on to be dealt with elsewhere. A different array structure (e.g. a 2-D array or a hypercube) could reduce the distance that messages have to pass and increase the bandwidth of data through the controller but at the cost of a slightly more complex protocol and routing process.

A few important points need to be made. Firstly, the work involved in designing and implementing this protocol is trivial compared to that required for the actual ray tracing algorithm and this will be true for any realistic program. Secondly, although two extra processes are being executed in parallel with the main ray tracing process, they actually consume very little processor time. Transputer processes are descheduled whilst waiting for communications to take place and so do not use the processor. They are automatically rescheduled, by the scheduling hardware, when the communication is complete. Also, external communication is done by the autonomous link DMA engines which can transfer data independently of, and concurrently with, the processor. This implies that the processing resource used by the communication depends more on the number of communications than the amount of data transmitted in each message.

The occam description of this transputer configuration has a constant defining the number of transputers in the network, which is all that needs to be changed if the size of the network is changed.

11.3.2 The control/display transputer

There are two processes executed by the control/display transputer, (see Figure 11.5). The first of these, **sendPatches**, interfaces to the host computer to receive the description of the scene being modelled and other commands. It passes the world model out to all the other transputers and then sends out requests for pixels to be evaluated. Square areas of the screen, 'patches', rather than individual pixels, are given to each transputer to enable 'slices' or blocks of data to be transmitted. A slice communication transmits an array of data as a single operation. As there is the same processor overhead for setting up the links to transmit a single byte as for a million bytes, this makes the most efficient use of the transputer link engines. It also allows the processor to continue calculating at very nearly full speed while the communication takes place, with only occasional interruptions to manage the routing processes.

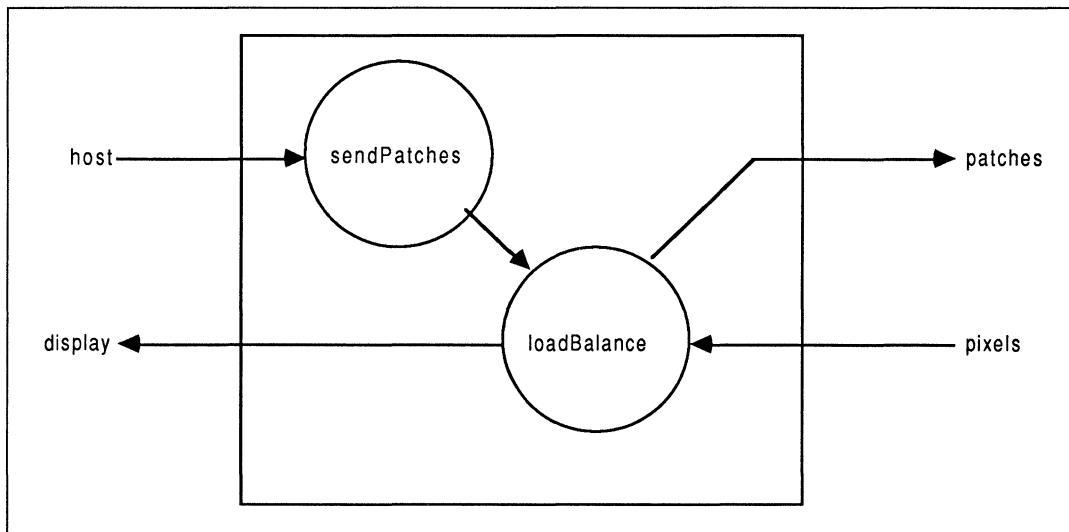


Figure 11.5 Processes running on control transputer

The other process, called **loadBalance**, coordinates the sending of data to the other transputers and the display of the generated pixels. If there are n transputers then **loadBalance** initially passes on $2n$ pixel patch requests from the process **sendPatches**. It then waits until a result is returned before handing out another request. So this process acts like a valve, only allowing work to be passed out when there are transputers able to accept it. Each of the n calculating transputers can accept two patch requests as described below.

11.3.3 The calculating transputers

The work on each of these transputers is organised as three processes shown in Figure 11.6. The most important of these is **render** which is sent patches to evaluate via the **throughput** process. The **render** process is a completely sequential piece of code and could be written in any standard programming language which supports communication over OCCAM channels. It does all the calculations to find intersections, build the tree of rays and then traverse this tree to get the final pixel value. When all the pixels in the patch are evaluated then the pixels are passed out to the **feedback** process and another patch is requested from **throughput**. The **feedback** process multiplexes the local results and those received from other transputers and passes them back towards the display transputer. This process is very simple, using an OCCAM **ALT** construct to wait for an input from either of the two channels.

The task of the **throughput** processes is to route patch requests through the pipeline to a free processor,

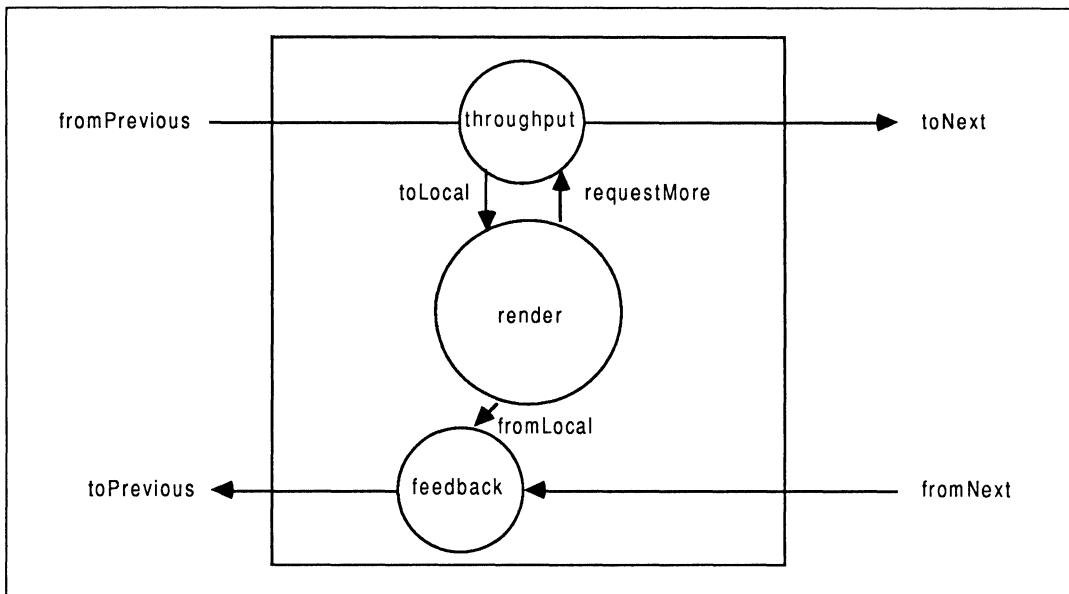


Figure 11.6 Processes running on the calculating transputers

i.e. one that is able to accept a pixel patch for evaluation. Patches can be routed to the next processor; routed to the local **render** process or buffered for local processing later. Initially each processor starts in the state **busy = FALSE** (not currently processing a patch) and **buffered = FALSE**. Patches are routed by **throughput** according to these state variables: if not **busy** then the patch is sent to the **render** process; if not **busy** and not **buffered** the patch is saved for later processing; otherwise the patch is passed on for processing elsewhere. Each processor therefore accepts two patches at startup, the first is passed immediately to **render** for evaluation and the second is held until needed. Any further patches received are passed on to be evaluated elsewhere until the processor becomes free again. After the first patch has been completed by the **render** process it sends a request to **throughput** for another. This is shown in the simplified piece of OCCAM below:

```

BOOL busy, buffered, running :
BYTE byte :
[3]INT patch, buffered.patch :
SEQ
  -- initialise state variables
  busy := FALSE
  buffered := FALSE
  running := TRUE
  WHILE running
    ALT
      -- a request for another patch from render
      requestMore ? byte
      IF
        -- we have some work buffered, pass it on
        buffered
        SEQ
          toLocal ! buffered.patch
          buffered := FALSE
        -- else indicate that the renderer is free
        TRUE
        busy := FALSE
      END
    END
  END
END
  
```

```

-- a message from the pipeline
fromPrevious ? CASE
  -- if it is a patch...
  rt.render; patch
  IF
    -- this processor not busy, pass to render
    NOT busy
    SEQ
      toLocal ! patch
      busy := TRUE
    -- if can't handle it here, pass to next transputer
    busy AND buffered
      toNext ! patch
    -- save patch for later processing
    busy AND (NOT buffered)
    SEQ
      buffered.patch := patch
      buffered := TRUE
  -- the terminate message
  rt.stop
  running := FALSE

```

Provided that the time taken to render a pair of patches is greater than the time before **throughput** receives a new patch, the render process is always kept busy. This provides distributed control of work allocation; each processor simply passes on any work that it cannot handle to be done elsewhere. It doesn't need to know where the work will be done or any other details of the system configuration. Because no more work requests are sent out than can be handled, the last processor in the network will never find itself with work to pass on to a non-existent processor.

11.4 Maximising performance

The processing speed of the system is directly related to the number of transputers used; ten transputers perform ten times faster than one. A number of factors contribute to this aspect of the system.

The work is given to the transputers in large chunks which require only three words of data (the X and Y coordinates and size of the patch) to specify the position of all the pixels in the patch. If the work were distributed on a pixel-by-pixel basis then two words of data would be required for every pixel. This would mean a much larger ratio of communication to processing.

Use of slice communication for data means there is less processor overhead per byte sent and allows a greater amount of concurrency between the link engines and the processor. Allocating the work in chunks made this even more important as entire patches of pixels were returned to the control/display transputer as a single communication.

The message routing processes are run at high priority to ensure that an incoming message can be examined and forwarded immediately it is received. The input guards of the **ALT** constructs in these processes are also carefully ordered in priority to ensure that patches are returned to the control processor as quickly as possible.

As well as holding an item of work in **throughput**, software buffers were added to any channels which communicate via a transputer link. These decouple the communication taking place via the link from the processes using the channel, thus allowing more overlap between processing and link communication. Channel buffers are frequently used, and easy to implement in OCCAM.

These issues and others, such as efficient use of on-chip RAM, are discussed in more detail in another INMOS technical note [4].

The performance of the system has been measured with up to eighty IMS T414-15 transputers and the results

are summarised below. These times were measured by the ray tracing system itself using the low-priority transputer timer which has a resolution of 64 microseconds. The image generated consisted of a simple scene containing four spheres and a single light source at a screen resolution of 256×256 . The time taken with each number of transputers was averaged over four runs. The processing speed in the table below is the number of pixels generated per second, linearity was calculated as **relative speed / transputers * 100**.

transputers	speed	relative speed	linearity %
1	164.0	1.00	100.0
2	327.6	2.00	99.9
4	654.0	3.99	99.7
8	1296.4	7.91	98.8
16	2601.6	15.87	99.2
32	5189.5	31.65	98.9
64	10300.0	63.15	98.7
80	12500.0	76.37	95.5

The ray tracer has also been run on T800 processors showing a factor of about 6 or 7 speed improvement due to the on-chip floating-point unit.

11.5 Fault tolerance of the system

It should be possible to exploit the number of processors in a multi-transputer system to introduce a degree of redundancy. The system described above is already remarkably robust. If a transputer fails then the system will progressively deadlock only if the neighbour, on the controller side, attempts to communicate with it. This is unlikely to occur, however, because results are passed back to the display by the shortest route, and new pixel patches are not sent out until results are returned. If a transputer is stopped while it is actually communicating, or between sending out results and being given its next patch of pixels then the system will deadlock. Otherwise, apart from the loss of processing power of the transputers on the far side of the fault, and the associated data, the system continues to operate.

In order to make the system more robust it must be possible to detect when a failure has occurred. This can be done by using a timeout on all communications. Secondly it must be possible to ensure that, even if a communication does fail, all the input and output processes will terminate. As this cannot be achieved directly in Occam, INMOS provide a number of predefined procedures which perform the desired functions. These allow an input or output to be attempted within a time limit, and recovery from a failed communication. They are described more fully in [5]. The use of these procedures means that failure of a transputer can be detected by its neighbour. The controlling transputer could then be informed and so take action to recover or regenerate the lost data.

Detection of the failure of a transputer implies that facilities could be added to allow the defective transputer to be bypassed. This can be done with no extra hardware as shown in Figure 11.7. If a transputer decides that its neighbour has failed then it switches to the other link to communicate with the next transputer along. Alternatively, if boards with more than one transputer are used (for example the IMS B003) it may be better to arrange the link connections so that an entire board is bypassed if a failure is detected. Obviously, this will not be sufficient if two adjacent transputers or boards were to fail, but this unlikely event could be catered for with extra hardware to allow link connections to be switched externally thus allowing any number of devices to be bypassed.

11.6 References

- 1 *The Transputer Databook*, INMOS Ltd, 1989.
- 2 *occam Reference Manual*, INMOS Ltd.
- 3 *An Improved Illumination Model for Shaded Display*, Turner Whitted, Communications of the ACM, pp. 343-349, June 1980, 23(6).
- 4 *Performance Maximisation*, INMOS Ltd, Technical note 17.

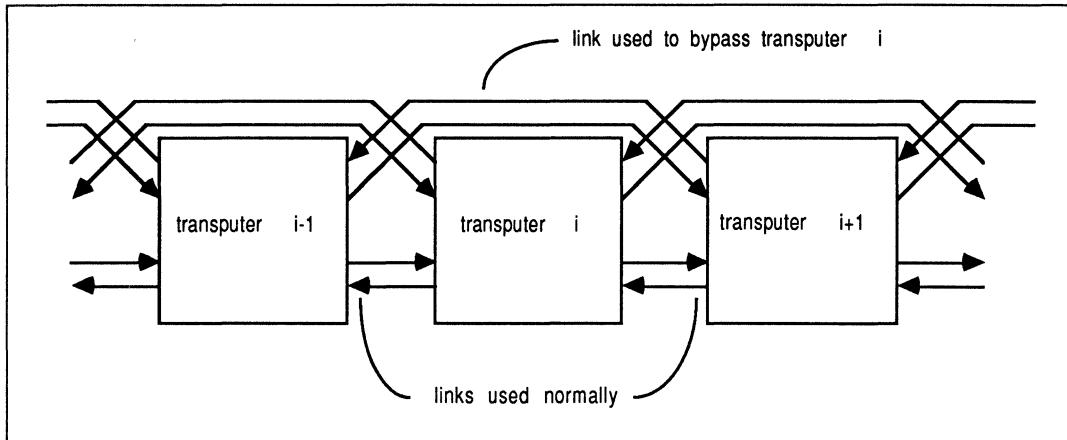


Figure 11.7 Bypassing a failed transputer

5 Extraordinary use of transputer links, INMOS Ltd, Technical note 1.

11.7 Note on the ray tracing program

The OCCAM language enables a system to be described as a collection of concurrent processes which communicate with one another, and with the outside world, via channels. OCCAM programs are built from three primitive processes: assignment (**variable** := **expression**); input (**channel** ? **variable**); and output (**channel** ! **expression**).

Each OCCAM channel provides a one-way communication path between two concurrent processes. Communication is synchronised and unbuffered. The primitive processes can be combined to form constructs which are themselves processes and can be used as components of another construct. Conventional sequential programs can be expressed by combining processes with the sequential constructs **SEQ**, **IF** and **WHILE**. Concurrent programs are expressed using channel communication, the parallel construct **PAR** and the alternative construct **ALT**. An alternative process may be ready for input from a number of channels; input is taken from the first of the channels to become ready.

Below is an outline of the OCCAM program for each transputer, and the description of the entire transputer system. The procedures all have several more parameters (such as screen size, maximum number of reflections, etc.) but, for simplicity, only the essential outline is given here.

In order to pass the various types of message (e.g. object definitions, patch requests and pixel values) around the system a variant protocol was used:

```
PROTOCOL trace.p
CASE
  rt.stop
  rt.done
  rt.render;    [3]INT          -- rt.render; x; y; patchSize
  rt.data;      INT; INT::[]INT -- rt.data; type; data
  rt.pixels;    INT; INT; INT::[]INT -- rt.pixels; x; y; n::data
  rt.message;   INT::[]INT     -- rt.message; n::chars
:
```

Each message then consists of: the message tag followed by the arguments. For example a 16×16 patch is sent as: `out ! c.render; [x; y; 16]`.

The code running on the control/display transputer is:

```
PROC control (CHAN OF trace.p fromHost, toDisplay,
              toCalculators, pixelsIn)

... definition of sendPatches procedure
... definition of loadBalance procedure

CHAN OF trace.p data :
PAR
  sendPatches (fromHost, data)
  loadBalance (data, toCalculators, pixelsIn, toDisplay)
:
:
```

Each of the calculating transputers runs the following code:

```
PROC calculate (CHAN OF trace.p fromPrev, toNext, fromNext, toPrev)

... throughput procedure
... render
... feedback

CHAN OF trace.p toLocal, fromLocal, requestMore :
PRI PAR
  -- run these at high priority for
  -- fastest response to messages
PAR
  throughput (fromPrev, toNext, toPrev, toLocal, requestMore)
  feedback (fromLocal, fromNext, toPrev)

  -- and this one at low priority
  render (toLocal, fromLocal, requestMore)
:
:
```

The system description is as follows:

```
... define constants for the link addresses
VAL number.transputers IS 42 :
VAL last IS number.transputers - 1 :

CHAN OF trace.p host, display, loopback :
[number.transputers]CHAN OF trace.p forward, return :

PLACED PAR

-- processor 0 is the control/display processor
PROCESSOR 0 T4
PLACE host      AT link0in : -- data from host
PLACE display   AT link2out : -- to display
PLACE forward[0] AT link1out : -- patches out
PLACE return[0]  AT linklin : -- pixel values back

control (host, display, forward[0], return[0])

-- the main body of the pipeline of calculators
PLACED PAR i = 1 FOR number.transputers - 2
PROCESSOR i T8
PLACE forward[i]  AT link0in : -- patches in
PLACE return[i]   AT link1out : -- pixels out
PLACE forward[i+1] AT link1out : -- patches out
PLACE return[i+1] AT link0in : -- pixels in

calculate (forward[i], forward[i+1], return[i+1], return[i])

-- the last transputer is a special case as it
-- has no one else to talk to. The fact that the
-- channel 'loopback' is not placed means that
-- an internal ("soft") channel will be created.
-- In fact this channel is never used but is
-- required as a parameter.

PROCESSOR last T8
PLACE forward[last] AT link0in :
PLACE return[last]  AT link0out :

calculate (forward[last], loopback, loopback, return[last])
```

Further information on the program is available from the Central Applications Group at INMOS Ltd in Bristol.

12 High-performance graphics with the IMS T800

12.1 Introduction

This chapter examines some applications of the IMS T800 floating-point transputer in high-performance graphics systems. Firstly there is a brief introduction to some of the basic techniques and terminology used in computer graphics. This includes comments on implementation and processing requirements.

Section 12.3 provides an overview of transputer, and specifically IMS T800, architecture. This concentrates on the aspects of the device which make it particularly suitable for using in parallel graphics systems. There is also a brief description of the Occam language, designed for programming highly parallel systems. This part concludes with a summary of how the IMS T800 meets the requirements of a modern graphics system.

The next section describes in some detail how the computing performance of the floating-point processor is obtained. It uses, as an example, a procedure which forms one of the key routines in all our graphics demonstrations.

Finally two particular applications are described in detail. These are the INMOS distributed Z-buffer, a near real-time multiprocessor solution to the hidden surface problem, and the INMOS multi-player flight simulator, a real-time interactive combat simulator. Both programs have been implemented on standard INMOS transputer evaluation boards with no custom hardware design and written entirely in a high-level programming language.

12.2 Computer graphics techniques

Computer-generated images, and in particular interactive graphics, is one of the fastest growing and most important application areas for high-performance computing systems. Some common applications are computer-aided design (CAD), simulation and medical imaging. These allow the user rapidly to see the effects of, for example, a design change on the appearance or behaviour of an object; or to view a large amount of data (for example a three-dimensional scan of a human body) in an understandable form.

There are a number of common requirements for these systems. Firstly the system must be fast, both to generate an image and to respond to input from the user. Secondly the displayed images must be realistic, or at least readily comprehensible to the user. This will usually mean that objects can be viewed with correct perspective, with natural shading and possibly shadowing, and that the way in which one part of the scene obscures another (the 'hidden surface problem') is correctly represented. For interactive systems response speed is an important factor to maintain realism and usability.

A brief introduction to some of the techniques and terminology used in this chapter is given below. A good introduction to interactive computer graphics can be found in [1].

12.2.1 Modelling objects

In order to render or generate images of an object some way of modelling the object in the computer is needed. A convenient primitive to use as the basis of modelling objects is the polyhedron. By increasing the number of faces the shape of any solid object can be approximated, although at the cost of having more data to manipulate. An arbitrary polyhedron can be modelled by defining its faces; each of these faces is then a polygon which can be defined by an ordered list of vertex coordinates.

Each polygon will have other attributes associated with it, such as colour and orientation. The orientation is represented by a line or vector perpendicular to the surface. This is called the surface normal and can be calculated from the coordinates of three vertices. The surface normal is closely related to another attribute, the plane equation of the face. A plane is represented by four numbers (a, b, c, d) so that $ax + by + cz + d = 0$ is true only if the point $[x \ y \ z]$ lies in the plane. If a point does not lie in the plane then the sign of the expression $ax + by + cz + d$ indicates which side of the plane the point is located on. By convention, points in front of the plane have positive values of $ax + by + cz + d$. The components of the normal vector are given by the plane equation; the vector is $[a \ b \ c]$. The plane equation and normal vector are very important for visibility and shading calculations.

12.2.2 Transformation

Geometric transformations play an important role in generating images of three-dimensional scenes. They are used (a) to express the location and orientation of objects relative to one another and (b) to achieve the effect of different viewing positions and directions. Finally a perspective transformation is used to project the three-dimensional scene on to a two-dimensional display screen.

Transformations are implemented as matrices which are used to multiply a set of coordinates to give the transformed coordinates. All rotations, translations and other transformations to be performed on data are combined into a single matrix which can then be applied to each point being transformed. Transformations may be nested, like subroutine calls, so that parts of a model can be moved independently but still take on the global movement of the model or the viewpoint.

The homogeneous coordinate system

The coordinates of points are represented using what are known as homogeneous coordinates. Any point in three-dimensional space can be mapped to a point in four-dimensional homogeneous space. The fourth coordinate, w , is simply a scaling factor so a point with the homogeneous coordinates $[x \ y \ z \ w]$ is represented in three-dimensional space as $[x/w \ y/w \ z/w]$. This representation simplifies many calculations and, in particular, means that the division required by perspective transformation can be done after clipping when there may be many fewer points to process.

The value of w is arbitrary as long as x , y , and z are scaled by the same amount. Generally when converting from 3-D to homogeneous coordinates it is simplest to make $w = 1$ so no multiplication of x , y and z is necessary. After being transformed the value of w may have changed so at some point the x , y , and z coordinates must be divided by w . This can be done when scaling to physical screen coordinates.

The transformation matrices used are 4×4 matrices for the transformation of homogeneous coordinates and are designed to have the desired effect on the point in ordinary three-dimensional space. When implemented on a computer, coordinates and transforms will generally use floating-point representation for maximum accuracy and dynamic range.

Translation

Translation, or movement of a point in space, is simply achieved by adding the distance to be moved in each axis to the corresponding coordinate:

$$\begin{aligned}x' &= x + t_x \\y' &= y + t_y \\z' &= z + t_z\end{aligned}$$

where t_x , t_y and t_z are the distances moved in x , y and z respectively. This can be represented as a matrix multiplication:

$$[x' \ y' \ z' \ w'] = [x \ y \ z \ w] \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ t_x & t_y & t_z & 1 \end{bmatrix}$$

Rotation

Three-dimensional rotations can be quite complex. The simplest form is rotating a point about an axis which passes through the origin of the coordinate system, and is aligned with a coordinate axis. For example, rotation about the z axis by an angle of θ is written as:

$$\begin{aligned}x' &= x \cos \theta + y \sin \theta \\y' &= -x \sin \theta + y \cos \theta\end{aligned}$$

This can be represented as a matrix multiplication as shown:

$$\begin{bmatrix} x' & y' & z' & w' \end{bmatrix} = \begin{bmatrix} x & y & z & w \end{bmatrix} \begin{bmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

To perform rotations about an arbitrary point it is necessary to translate the point to the origin, perform the rotation and then translate the point back to its original position. Rotations about axes which are not aligned with the coordinate system can be performed by concatenating a number of simpler rotations.

Concatenation

The successive application of any number of transforms can be achieved with a single transformation matrix, the concatenation of the sequence. Suppose two transformations M_1 and M_2 are to be applied to successively to the point v . First v is transformed into v' by M_1 , this is then transformed into v'' by M_2 :

$$v' = v M_1 \quad (12.1)$$

$$v'' = v' M_2 \quad (12.2)$$

Substituting Equation 12.1 into Equation 12.2 gives:

$$v'' = (v M_1) M_2 = v (M_1 M_2)$$

Therefore the concatenation of a sequence of transformations is simply the product of the individual transform matrices. Note that, because matrix multiplication does not commute, the order of application of the transformations must be preserved.

Perspective projection

The most realistic way of displaying three-dimensional objects on a two-dimensional screen is the perspective projection. There is a simple transformation that distorts objects so that, when viewed with parallel projection (orthographically), they appear in perspective. This defines a viewing volume, a truncated pyramid, within which objects are visible (see Figure 12.1). This transformation preserves the flatness of planes and the straightness of lines and simplifies the clipping process that follows. The perspective transform uses three parameters: the size of the virtual screen on to which the image is projected; the distance from the viewing position to this screen; and the distance to the furthest visible point. The result of the perspective transform is to normalise all coordinates so that values range between -1 and $+1$, the centre of the image is at point $(x, y) = (0, 0)$. To display these on a real device the coordinates must be scaled by the screen resolution of the display.

The perspective transform used in the programs discussed in this document is based on that in Sutherland and Hodgman [2].

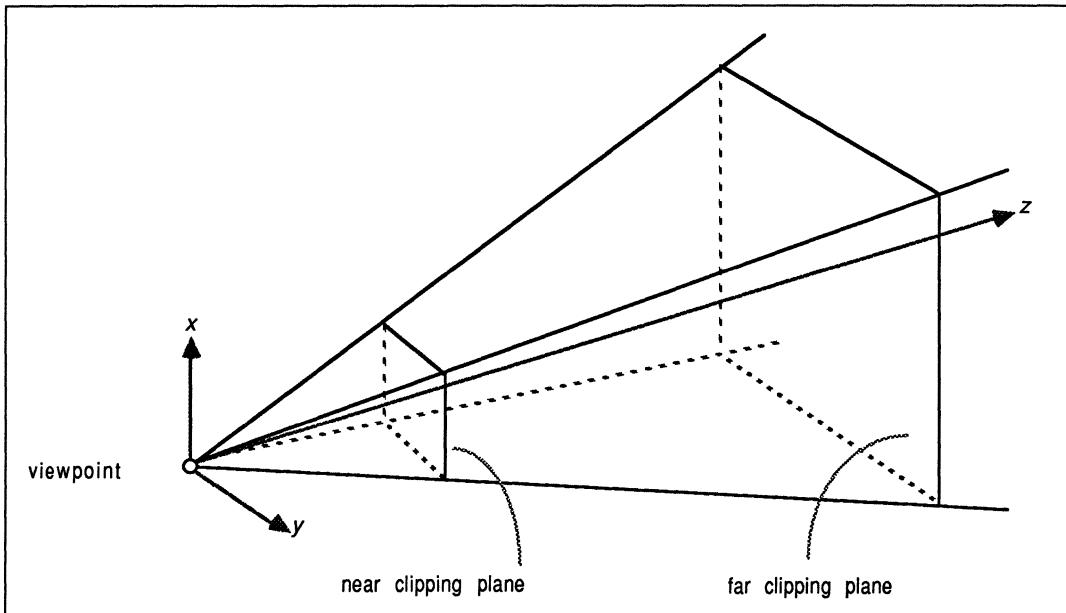


Figure 12.1 Viewing objects in perspective

12.2.3 Scan conversion

Raster displays are the most commonly used output device for computer graphics systems. They represent an image as a rectangular array of dots or 'pixels'. The image to be displayed is stored in a 'frame buffer', an area of memory where each location maps on to one pixel. The main advantages of raster displays are low cost and their ability to display solid areas of colour as easily as text and lines.

In order to display objects which are represented as a number of polygons it is necessary to scan convert the polygons. This involves finding all the pixels that lie inside the polygon boundaries and assigning them the appropriate colour. A shading model is used to calculate the colour of each pixel.

A number of techniques have been developed for scan conversion. These generally take advantage of 'coherence'; the fact that the visibility and colour of adjacent pixels is usually very similar, and there are only abrupt changes at polygon boundaries. This allows incremental methods using only integer arithmetic to be used.

12.2.4 Shading

To generate realistic images it is necessary to assign the correct colours to the various parts of the model. This means shading the objects to represent lighting conditions. The apparent colour of a surface is dependent on the nature of the surface (its colour, texture, etc.), the direction of the light source and the viewing angle. A realistic shading model may require a large amount of floating-point arithmetic to multiply together the vectors representing surface orientation (the surface normal), direction of the light source, etc.

Where objects are represented as a number of polygons, the faceted appearance can be reduced by using a smooth shading model. There are two simple and reasonably effective techniques. Gouraud shading simply interpolates the surface colour across each polygon. This can, however, introduce a number of anomalies for example, in the shape of highlights and the way shading changes in moving sequences. Many of these problems can be relieved by using a technique developed by Phong but at the expense of increased calculation. Phong shading interpolates the surface normals across the polygons and re-applies the shading model at each pixel.

12.2.5 Clipping

Clipping is necessary to remove points which lie outside the viewing volume and to truncate lines which extend beyond the boundaries. Clipping can be done more simply after the perspective transformation. However, clipping in the z axis must be done before the division by depth which the full perspective projection requires as this destroys the sign information that determines whether a point is in front of or behind the viewer. Points with a negative value of z are behind the viewer.

Clipping to the x and y coordinates need only be performed to screen resolution. This has led to many clever, although not always simple, techniques using fast integer arithmetic to clip lines quickly. The availability of fast floating-point hardware means that more straightforward methods can be used.

The use of homogeneous coordinates and the perspective projection simplifies clipping. Because the points can be viewed in parallel projection x and y values which are inside the viewing pyramid are in the range -1 to 1 and z values are in the range 0 to 1 . The use of scaled, homogeneous coordinates means that the tests that have to be applied are:

$$-w \leq x \leq w$$

$$-w \leq y \leq w$$

$$0 \leq z \leq w$$

These limits correspond to the six bounding planes of the truncated viewing pyramid. A fast polygon clipping algorithm is described in [2].

12.2.6 Hidden surface removal

In order to generate realistic images it is important to remove from an image those parts of solid objects which are hidden. In real life these would be obscured by the opaque material of the object. In computer graphics the visibility of every point must be explicitly calculated.

Hidden surface algorithms are classified as either object-space or image-space. An object-space algorithm uses the geometrical relationships between the objects to determine the visibility of the various parts and so will normally require at least some floating-point arithmetic. An image-space method works at the resolution of the display device and determines what is visible at each pixel. This can be done most efficiently using integer arithmetic. The computation time of object-space techniques tends to grow with the total number of objects in the scene whereas image-space computation will tend to grow with the complexity of the displayed image.

There is also a trade-off between speed, complexity and memory usage. For example the Z-buffer technique described in section 12.5 is a very simple, reasonably fast image-space algorithm but requires a large amount of working memory. It uses an array of integers, the same size as the frame buffer, to store the depth at each pixel.

The BSP-tree used in the INMOS flight simulator (section 12.6) is an object-space algorithm which is efficient in memory usage, but uses floating-point arithmetic to determine the ordering of polygons. Its performance depends on the availability of a fast floating-point processor. It is also not completely general: in its simplest form it can only be applied to rigid objects constructed from non-intersecting polygons.

12.3 The IMS T800 transputer

The IMS T800 is the latest member of the INMOS transputer family [3]. It integrates a 32-bit 10 MIP processor (CPU), four serial communication links, 4 Kbytes of RAM and a floating-point unit (FPU) on a single chip. An external memory interface allows access to a total memory of 4 gigabytes.

The transputer family has been designed for the efficient implementation of high-level language compilers. Transputers can be programmed in sequential languages such as C, PASCAL and FORTRAN (compilers for

which are available from INMOS). However, the occam language (see section 12.3.4) allows the programmer to fully exploit the facilities for concurrency and communication provided by the transputer architecture.

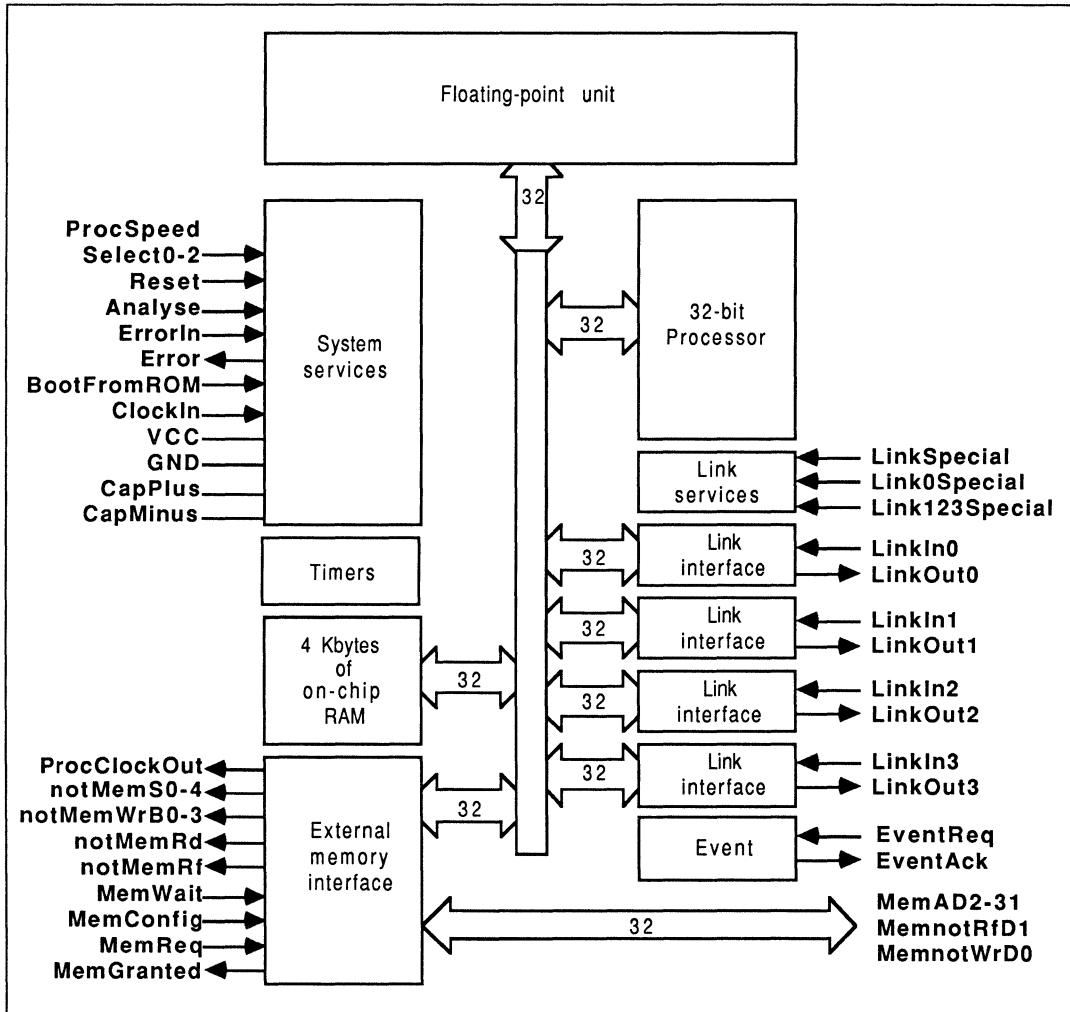


Figure 12.2 IMS T800 block diagram

The on-chip memory is not a cache, but part of the transputer's total address space. It can be thought of as replacing the register set found on conventional processors, operating as a very fast access data area, but can also act as program store for small pieces of code.

12.3.1 Serial links

The four serial links on the IMS T800 allow it to communicate with other transputers. Each serial link provides a data rate of 1.7 Mbytes per second unidirectionally, or 2.35 Mbytes per second when operating bidirectionally.

Since the links are autonomous DMA engines, the processor is free to perform computation concurrently with link communication. With all four links receiving simultaneously, the maximum data rate into an IMS T800 is 6.8 Mbytes per second. This allows a graphics card based round a single IMS T800 to act as an image sink,

accepting byte-wide pixels down its serial links directly into video RAM. This is the architecture used in the INMOS distributed Z-buffer (section 12.5) and in the INMOS flight simulator (section 12.6).

12.3.2 On-chip floating-point unit

The IMS T800 FPU is a coprocessor integrated on the same chip as the CPU, and can operate concurrently with the CPU. The FPU performs floating-point arithmetic on single- and double-length (32- and 64-bit) quantities to IEEE 754. The concurrent operation allows floating-point computation and address calculation to fully overlap, giving a realistically achievable performance of 1.5 Mflops (4 million Whetstones[4] / second) on the 20 MHz part, 2.25 Mflops (6 million Whetstones / second) at 30 Mhz.

12.3.3 2-D block move instructions

Among the new instructions in the IMS T800 are those for graphics support. The IMS T800 has a set of microcoded two-dimensional block move instructions which allow it to perform cut-and-paste operations on irregularly shaped objects at full memory bandwidth. The three **MOVE2D** operations are:

- | | |
|----------------------|---------------------------------------|
| MOVE2DALL | which copies an entire area of memory |
| MOVE2DZERO | which copies only zero bytes |
| MOVE2DNONZERO | which copies only non-zero bytes |

The use of these instructions is described more fully elsewhere [5].

12.3.4 The occam programming language

The occam language enables a system to be described as a collection of concurrent processes which communicate with one another, and with the outside world, via communication channels. occam programs are built from three primitive processes:

variable := expression	assignment
channel ? variable	input
channel ! expression	output

Each occam channel provides a one-way communication path between two concurrent processes. Communication is synchronised and unbuffered. The primitive processes can be combined to form constructs which are themselves processes and can be used as components of another construct. Conventional sequential programs can be expressed by combining processes with the sequential constructs **SEQ**, **IF**, **CASE** and **WHILE**.

Concurrent programs are expressed using the parallel construct **PAR**, the alternative construct **ALT** and channel communication. **PAR** is used to run any number of processes in parallel and these can communicate with one another via communication channels. The alternative construct allows a process to wait for input from any number of input channels. Input is taken from the first of these channels to become ready and the associated process is executed.

This chapter contains some short program examples, including a few written in occam. These should be readily understandable but, if necessary, a full definition of the occam language can be found in the occam reference manual [6].

12.3.5 Meeting computer graphics requirements

Computer graphics have always required large amounts of computing power. As users become more demanding in their requirements for higher resolution, more colours and faster response from graphics-based systems, more and more processing speed and i/o bandwidth are required.

Graphics applications can require huge amounts of floating-point maths for performing transformations, spline curve interpolation and evaluating complex shading models. Realistic images may contain many thousands of primitives to be manipulated and displayed. Some of the most impressive computer images have been

produced using ray tracing, a very expensive computer graphics technique. The implementation of a multi-processor ray tracing program using transputers is described in [7].

For desktop publishing, very high quality fonts are required, which must be manipulated at high speed if the feeling of user interaction is to be maintained. For digital compositing and 'paintbox' type applications, large irregular shapes must be moved around on screen at high speed, without annoying jerks and hops as the processor strains to keep up with the user.

High-quality printed output may use a laser printer, a very high-resolution output device. Typical modern laser printers produce images with 300 – 400 dots per inch on A3 or A4 size paper. A bitmap at this resolution requires up to 4 Mbytes of data. As colour laser printers become available the memory requirements increase dramatically.

Finally, real-time graphics work demands very high bandwidth to the display device — a modest 16 frames per second on a $512 \times 512 \times 8$ bit pixel display requires the transfer of 4 Mbytes of data to the display each second. This is easily met by the four links on a single IMS T800. As frame rates and screen resolutions continue to increase so does the performance required from a graphics system. Multiple IMS T800s could be connected to a common frame store, using video RAMs, to provide even greater bandwidth to the display. The hardware aspects of transputer-based graphics systems are discussed in [8].

The major requirements of the ideal graphics processor then are: high-speed floating-point performance; high-speed text manipulation and 2-D cut/paste operations (actually the same operation but on different scales); fast movement of large quantities of data; and high bandwidth in and out of the processor.

Although not specifically a graphics device, the IMS T800 transputer fulfils all the above requirements — massive compute power, a large linear address space, high i/o bandwidth and instruction level support for pixel graphics operations.

12.4 3-D transformation on the IMS T800

One of the main uses for a floating-point processor in a computer graphics system is for calculating 3-D transformations. This will include both generating a transformation matrix and applying this transformation to sets of coordinates.

Here, a four-element vector is multiplied by a 4×4 matrix, to give a four-element result:

$$[x' \ y' \ z' \ w'] = [x \ y \ z \ w] \begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix}$$

This can be expanded as:

$$x' = ax + ey + iz + mw$$

$$y' = bx + fy + jz + nw$$

$$z' = cx + gy + kz + ow$$

$$w' = dx + hy + lz + pw$$

Hence to multiply the vector by the matrix requires 28 floating-point operations (16 multiplications, 12 additions) which pipelines very efficiently on the IMS T800. The following OCCAM procedure multiplies the vector by the matrix, storing the result:

```

PROC vectorProdMatrix ([4]REAL32 result,
                      VAL [4]REAL32 vec,
                      VAL [4][4]REAL32 matrix)

  VAL X IS 0 :
  VAL Y IS 1 :
  VAL Z IS 2 :
  VAL W IS 3 :

  SEQ
    result[X] := (vec[X]*matrix[0][X]) + ((vec[Y]*matrix[1][X]) +
                                              ((vec[Z]*matrix[2][X]) + ((vec[W]*matrix[3][X]))))

    result[Y] := (vec[X]*matrix[0][Y]) + ((vec[Y]*matrix[1][Y]) +
                                              ((vec[Z]*matrix[2][Y]) + ((vec[W]*matrix[3][Y]))))

    result[Z] := (vec[X]*matrix[0][Z]) + ((vec[Y]*matrix[1][Z]) +
                                              ((vec[Z]*matrix[2][Z]) + ((vec[W]*matrix[3][Z]))))

    result[W] := (vec[X]*matrix[0][W]) + ((vec[Y]*matrix[1][W]) +
                                              ((vec[Z]*matrix[2][W]) + ((vec[W]*matrix[3][W]))))
  :

```

Analysing the statement:

```

result[X] := (vec[X]*matrix[0][X]) + ((vec[Y]*matrix[1][X]) +
                                         ((vec[Z]*matrix[2][X]) + ((vec[W]*matrix[3][X]))))

```

it can be seen that all vector offsets are constant and will be folded out by the compiler into very short instruction sequences. Furthermore all floating-point operations are fully overlapped with subsequent address calculations.

The statement compiles into only 27 instructions, most of which are only a single byte. The details of the transputer instruction set are given in [9] and the implementation of the FPU in [10].

The instruction sequence generated by this expression is:

(1)

```

ldl 2          -- load local variable 2 (address of vec)
ldnlp 2        -- compute address of vec[Z]
ldl 3          -- load address of matrix
ldnlp 8        -- compute address of matrix[2][X]
fpldnlnsn      -- transfer matrix[2][X] to top of FPU stack

```

(2)

```

fpldnlmulsn    -- transfer vec[Z] to FPU and multiply

```

(3)

```

ldl 2          -- compute address of vec[W]
ldnlp 3        -- compute address of vec[W]
ldl 3          -- compute address of matrix[3][X]
ldnlp 12       -- compute address of matrix[3][X]

```

(4)

```

fpldnlsn      -- transfer matrix[3][W] to FPU
fpldnlmulsn  -- transfer vec[W] to FPU and multiply
fpadd         -- add so top of FPU stack contains
                -- (vec[Z]*matrix[2][X]) + (vec[W]*matrix[3][X])

ldl  2        -- calculate address of vec[Y]
ldnlp 1
ldl  3        -- and address of matrix[1][X]
ldnlp 4

fpldnlsn      -- transfer matrix[1][X] to top of FPU stack
fpldnlmulsn  -- transfer vec[Y] to top of stack and multiply
fpadd         -- add product to previous intermediate result

ldl  2        -- calculate address of vec[X]
ldl  3        -- and address of matrix[0][X]

fpldnlsn      -- transfer matrix[0][X] to FPU
fpldnlmulsn  -- transfer vec[X] to FPU and multiply
fpadd         -- final accumulate, followed by
ldl  1        -- final store to
fpstnlsn     -- result[X]

```

Most FPU operations pop the top two values off the stack to use as operands and then push the result back on to the stack. The stack consists of three registers inside the FPU and nearly all expressions can be compiled so that no temporary memory variables are needed.

The code between (1) and (2) calculates the address of the first two operands and transfers **matrix[2][X]** to the top of the FPU stack. The code between points (2) and (3) loads **vec[Z]** on to the FPU stack and initiates a floating-point multiply. The CPU then executes the code between (3) and (4) which calculates the addresses of the next pair of operands. Meanwhile the FPU continues with its multiplication. Finally the *floating point load non local* instruction at point (4) is executed and a hardware interlock causes the CPU and FPU to synchronise. In this way, the computation of the operand addresses is entirely overlapped with the floating-point multiplication. In the remainder of the expression the FPU is kept busy, never having to wait for the CPU to perform an address calculation, and so achieving its quoted 1.5 MFLOP rating.

The entire vector matrix multiplication operation, including the call to the procedure, takes less than 19 μ s on the IMS T800-20, allowing a single transputer to perform 3-D transformation on over 50 000 points per second. This is important — the example is not a bizarre and meaningless benchmark designed to make the IMS T800 look as fast as possible. It is a genuine piece of application code, and the inner loop of all 3-D transformations.

The efficiency of this piece of code does not depend on it being written in OCCAM. An efficient compiler for any other language can easily obtain similar performance. Neither does the performance depend on constant array subscripts as in this example. The transputer's fast product instruction can be used to calculate the address of an array element and this will still be fully overlapped with the FPU operation. This is true even for two-dimensional arrays with code for range checking the array subscripts. The loops were expanded out in this example to remove jump instructions, which are relatively slow and prevent full overlapping of FPU and CPU operations.

12.5 The INMOS distributed Z-buffer

The Z-buffer is a general solution to the computer graphics hidden surface problem. When presented with the primitives which constitute a scene, the Z-buffer will output the scene as viewed by the observer, with hidden or partially hidden surfaces correctly obscured.

The core of the Z-buffer program is the distributed scan converter, which allows the processes of scan conversion and Z-buffering to be distributed over a number of transputers.

12.5.1 The Z-buffer algorithm

For each pixel on the screen a record is kept, in a depth- or Z-buffer, of the depth of the object at that pixel which lies closest to the observer, and the colour of that pixel is kept in a separate frame buffer. As each new object is scan-converted the depth of each pixel generated is compared with the value currently in the Z-buffer; if this pixel is closer than the previous one at that position then the depth and frame buffers are updated with the values for the pixel.

When all polygons (and other primitives) have been scan converted into the Z-buffer, the frame buffer contains the correct visible surface solution.

In pseudo-code the Z-buffer algorithm is essentially:

```

for each polygon
{
    for each (x,y) on the screen covered by this polygon
    {
        compute z and colour at this (x,y)
        if z < zbuffer[x,y] then
        {
            framebuffer[x,y] := colour
            zbuffer[x,y] := z
        }
    }
}

```

So for each polygon, the ***z*** value and the ***colour*** must be computed at each screen position covered by that polygon. For maximum speed the value of ***z*** and ***colour*** for each pixel is usually computed using only simple integer arithmetic at each step.

12.5.2 Scan conversion

The scan converter discussed here is restricted to convex polygons (polygons with no acute angles and no holes) and spheres.

Scan-converting polygons

The scan converter traverses each polygon from bottom to top, maintaining data for a pair of 'active edges'. These active edges delimit the horizontal extent of the polygon, and this horizontal extent is scanned, to give depth and colour for each pixel covered by the polygon. As it scans up, the polygon values of *x*, *z* and colour are maintained along a left-active edge and a right-active edge. When the scan converter encounters a vertex in one of the active edges, the appropriate set of edge data is updated.

Each active edge has associated slope values, $\frac{dx}{dy}$, $\frac{dz}{dy}$ and $\frac{d\text{colour}}{dy}$. The scan converter computes *x*, *z* and colour for the next scanline (i.e. at *y*+1) by adding on these slope values to the current values of *z* and colour. The scan converter computes $\frac{dz}{dx}$ and $\frac{d\text{colour}}{dx}$ for each horizontal extent, to allow horizontal interpolation of *z* and colour for full Z-buffering. Linear interpolation of colour gives Gouraud shading, a simple and effective smooth shading approximation (compare photographs A and C at the front of the book).

Scan-converting spheres

Polygons can be scanned easily since they are planar, and *z* can be interpolated linearly over planar surfaces. Spheres are not so simple. There are two problems: first, scan-converting the sphere involves determining the projected circular outline of the sphere on the screen; secondly, scanning the region inside the outline to compute *z* and colour at each pixel covered by the sphere. In fact, a sphere in perspective does not always project exactly into a circle, but in general this is a close enough approximation. The spheres code was written to allow complex molecules to be rendered. When displaying molecules, the individual atoms are generally small in relation to the complete image, so the distortion due to circular projection is acceptable.

The projected radius of the sphere is obtained from the perspective calculations. Bresenham's circle algo-

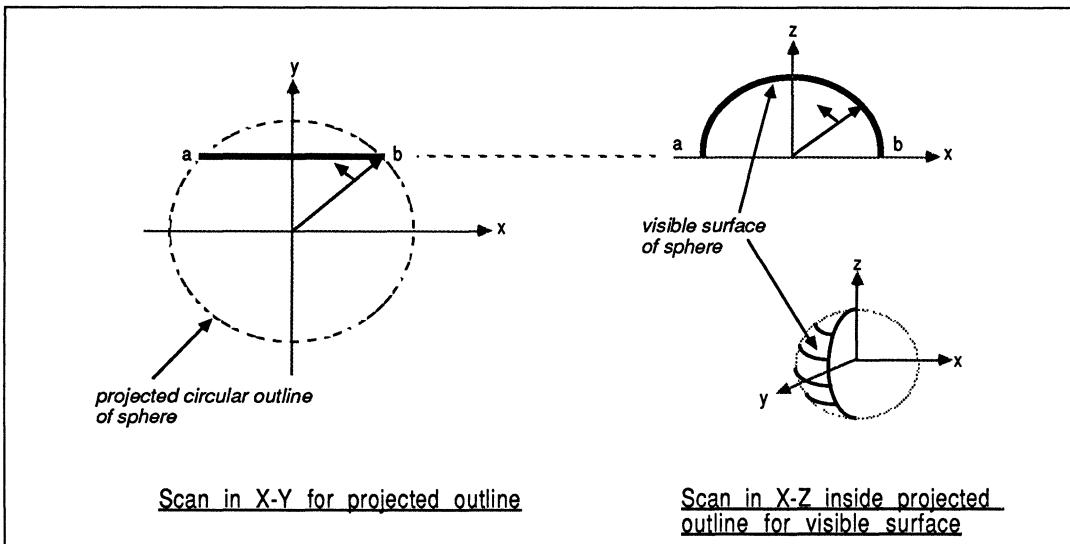


Figure 12.3 Scan conversion of spheres via Bresenham's algorithm

rithm [11] is then used to scan the outline of the projected circle, and is also used at each scanline to scan the sphere in z (Figure 12.3). Exact spherical shading is complex (and therefore slow), requiring lots of maths at each pixel (including square roots), so an approximate shading technique is used as described by Fuchs et al. [12]. The visible hemisphere is shaded as though it were a paraboloid. The resulting shading is smooth and very hard to distinguish from correct spherical shading.

Implementation details

Scan conversion with a DDA

Scan converters are generally implemented using a digital differential analyser (DDA), or a variant of Bresenham's line-drawing algorithm [1, Chapter 2]. The reasoning behind this is that divisions can be avoided, and all addition operations are on integers, improving performance. Tracking an edge with a DDA involves maintaining two items of information about the edge: the current position and the current error term. A step is taken along the 'driving axis', the axis of greatest step. A fixed value is unconditionally added to the error term. When the error term overflows (generally this means when the error becomes positive), a step is taken along the 'driven axis', and a different fixed value is subtracted from the error term.

Here is an example of drawing a line using Bresenham's algorithm — it is assumed that the `deltaX` is greater than `deltaY`, so `x` is the driving axis:

```
e = (2 * deltaY) - deltaX;
for (i = 1; i == deltaX; i++)
{
    plot (x, y);
    e = e + (2 * deltaY);
    if (e > 0)
    {
        y = y + 1;
        e = e - (2 * deltaX);
    }
    x = x + 1;
}
```

Note that a decision must be made at each pixel; the **if** statement means that the processor will execute a conditional jump instruction. The break in instruction pipelining (and subsequent forced instruction fetch) this causes will consume valuable processor cycles.

Scan conversion on transputers

There is an alternative solution for the transputer. Bresenham's algorithm removed division operations because historically this was a prohibitively slow operation. The division was removed at the expense of generality — the slope of the line must be between zero and one. This means that a scan converter, which must have y as the driving axis, still requires at least one division operation and also requires greater complexity in the inner loop.

As a division is now necessary, an alternative approach was looked for. The transputer's designers were sufficiently far-sighted to include fast extended arithmetic operations in the instruction set. Instead of maintaining an error term (which is scaled in terms of **deltaX** and **deltaY**, rather than machine precision) we simply put a 32-bit fraction on the end of the 32-bit integer, and use **longadd** instructions to step along the slope.

The value **slope := deltaX / deltaY** is computed as a signed 64-bit value (32 integer plus 32 fraction bits), and the **if** at every pixel is avoided. Computing the slope to 64 bits consumes about 100 processor cycles ($5 \mu s$) per edge, but simplifying the code in the inner loop makes the fractional version run some 40% faster than the Bresenham version. The code also becomes more readable, as shown in the simplified example below:

```
y := y + slope           -- slope is 64 bits (integer + fraction)
```

is more obvious than:

```
SEQ
  y := y + dyBydx      -- dyBydx is the integer part of the slope
  e = e + (2 * deltaY)
IF
  e > 0                  -- take care of fractional part of slope
    SEQ
      y := y + 1
      e := e - (2 * deltaX)
    TRUE
    SKIP
```

This becomes even more apparent when several variables are being interpolated (i.e. x , z and colour). Note that for x and colour, a 32-bit value for the slope (16 bits integer and 16 bits fraction) would provide sufficient resolution and be faster to compute. However, the advantages of this are outweighed by having to extract the upper 16 bits of the word which contain the desired x and colour values.

The scan conversion of spheres is also done using long arithmetic.

Distributing scan conversion over multiple transputers

A standard scan converter traverses each polygon one scanline at a time. The distributed scan converter running on N transputers traverses each polygon N lines at a time. Each scan converter starts scanning at a different scanline, i.e. at the lowest y -coordinate enclosed by the polygon which can contribute to its subsection of the image.

In effect, each scan converter reconstructs a slightly different 'squashed', but interleaved, copy of the scene. When merged these subimages create the final picture, so the net effect is that the polygon is fully shaded and Z-buffered (Figure 12.4).

This requires careful coding (and a little more computation) to initiate the scan conversion process and to follow corners correctly, but the scan converter distributed on N machines runs (very nearly) N times as fast as on one machine.

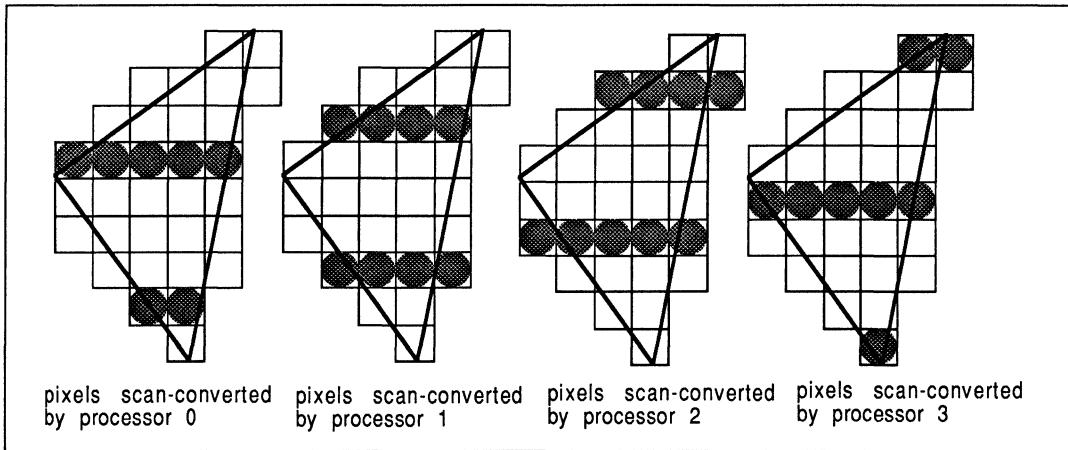


Figure 12.4 Distributed scan conversion

12.5.3 Architecture

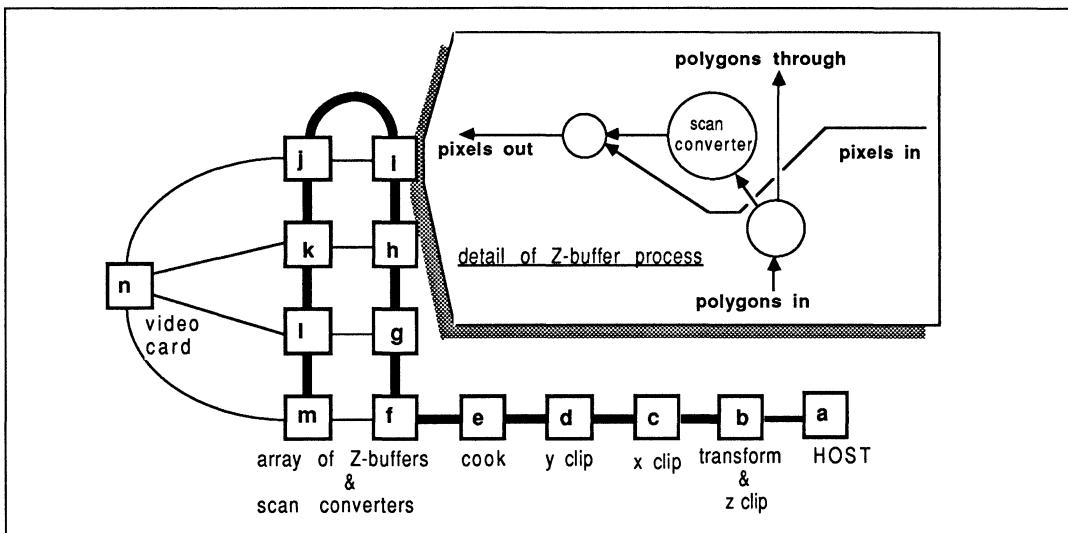


Figure 12.5 Distributed Z-buffer architecture

The architecture of the Z-buffer system (Figure 12.5) is simple, but is flexible and easily extended. An INMOS IMS B004 board (a) is used as a database, file interface and user interface. It sends transformation matrices, polygons and spheres to the geometry system.

The geometry system consists of four transputers on a single IMS B003-2 transputer evaluation board, which has been modified by replacing one of the IMS T414-20s with an IMS T800-20. This transputer (b) performs all the floating-point computation, performing 3-D transformation, z clipping and conversion to screen coordinates. Two IMS T414s (c and d) then perform x and y clipping. A final IMS T414 (e) preprocesses ('cooks') polygons and spheres into a form suitable for the scan converters: polygon vertex format is converted to edge format and edge slopes are computed; coefficients are calculated for the sphere shading equation.

The 'cooker' outputs its processed polygons and spheres to the Z-buffer array (*f* through *m*). Note the link usage — polygons are passed through the emboldened vertical links, independently of the horizontal links which pass pixels to the graphics card (*n*). This separation of polygon flow and pixel flow allows a finished frame to be passed to the graphics card while the next frame is being computed, pipelining work efficiently for animated sequences. This organisation also takes maximum advantage of the autonomous link engines on each transputer. The graphics card used is an IMS B007 evaluation board which has two banks of video memory allowing the next frame to be read in without disturbing the currently displayed image. When the complete frame has been received the two memory banks are swapped by writing to a control register. This must be synchronised with the frame flyback of the display to avoid distracting visual artefacts.

12.5.4 Performance

The Z-buffer is fully interactive, and on our existing models image generation speeds range from over 10 frames per second down to around 1 frame per second.

Performance of the system is sensitive to the number of polygons and to screen coverage per polygon. With small numbers of large polygons, scan conversion time dominates, so a larger number of scan converters gives a linear performance improvement.

The IMS T800 in the geometry system is crucial for images with large numbers of small polygons. In this case screen coverage and hence scan conversion time (per polygon) is low, and transformation time can dominate unless a lot of floating-point performance is available. If the IMS T800 is replaced by an IMS T414, refresh rates can drop by a factor of 15 for a complex model such as that of the IMS T800 package.

Some screen photographs of the output generated by this system are included at the front of this book. The bevelled cubes consist of 112 polygons, 128 points, and were computed at 6.3 frames per second using 8 scan converters. The molecule (54 spheres, 108 points) achieves 2.9 frames per second, but this drops dramatically if the screen coverage is increased, since the computation per pixel is higher for spheres than for polygons. Note that the individual atoms intersect correctly, and that the lighting conditions are locally modelled — the highlight is in different positions on different atoms. The Starship (596 polygons, 943 points) is displayed at 3.1 frames per second; no surface normal information is yet available for this model, so it is flat shaded. The IMS T800 package (1254 polygons, 1584 points) refreshes at between 1.8 and 1.4 frames per second.

12.6 The INMOS multi-player flight simulator

The INMOS flight simulator came from the need to demonstrate the real-time graphics capabilities of the transputer family. Although the Z-buffer is much faster than any other yet implemented on microprocessors (rather than custom hardware), it is still not fast enough to implement the vision system of a flight simulator, even when running with thirty-two scan converters. This is due to the per pixel calculation involved in Z-buffering — a 'greater than' comparison is required at every pixel covered by each polygon. An alternative hidden surface algorithm without this overhead is required for the flight simulator.

12.6.1 Requirements

The primary requirement of the flight simulator was that it be fast. It should be able to sustain 17 frames per second, the bandwidth limit into the IMS B007 graphics card, when shading a reasonable number of polygons — say 200 to 300. It should have low latency, i.e. the time from user input to visual feedback should be no more than three, preferably only two frame times. It should also use only a small number of transputers to implement a four-player system.

12.6.2 Implementation details

The distributed polygon shader

The core of the flight simulator is a distributed polygon shader, similar in design to the scan converter in the Z-buffer. It is optimised for flat shading of polygons and does not include the Z-buffer. This reduces the amount of computation and means that a fast block move operation can be used to shade the horizontal

regions between polygon edges. It can be arranged that the block move copies the value defining the colour from on-chip memory so a 32-bit word can be copied (in other words, four pixels can be shaded) every $n+1$ machine cycles, where n is the number of machine cycles required to access off-chip memory.

When coded in this way, a 20MHz transputer with single wait-state (4 cycle) external memory can shade polygons at a rate of 16 million 8-bit pixels per second, or 62.5 nanoseconds per pixel. Four transputers can therefore shade at up to 64 million pixels per second, only 15.6 nanoseconds per pixel. With this high polygon shading speed it becomes possible to display a reasonable number (over 200 'average size') polygons at 17 frames per second, a very high number for a software implementation with no custom hardware. Using four transputers allowed the use of the INMOS IMS B003-2 transputer evaluation board, so no new hardware design was necessary.

Geometry system

From the previous figures quoted for transformation time, the IMS T800 has processing power to spare, it can transform 200 quadrilaterals (800 points) in less than one sixtieth of a second. Three more transputers are used in the geometry system — another IMS T800 for z clipping (often called hither-and-yon clipping) and conversion to screen coordinates, and two IMS T414s for clipping in x and y . Clipping in x and y are performed in screen space, so integer maths is sufficient. The geometry system now consists of four transputers, so again an IMS B003-2 is used, but this time slightly modified (two IMS T414s replaced with IMS T800s).

At this stage the importance of pin compatibility between the IMS T414 and IMS T800 cannot be over emphasised — it allows high floating-point performance to be injected into a multiprocessor system just where it is required, allowing performance tuning simply by removing one transputer from a socket and plugging in another.

This is a very fast polygon shader and geometry system, all that is required is a hidden surface algorithm which outputs its solution in polygon form to implement the entire vision system of the flight simulator.

BSP-trees

The BSP-tree [13] is a recursive data structure which implicitly holds all possible hidden surface solutions for the object it represents. Each node of the BSP-tree contains a polygon and pointers to front and back subtrees. The front subtree contains all polygons in front of the node polygon, the back subtree contains those behind the node polygon. The notion of 'in front-ness' is determined by substitution of the current viewing position into the plane equation of the polygon.

By traversing the BSP-tree in an order determined solely by the viewing position, the polygons are passed to the distributed polygon shader in reverse z order, so that nearer surfaces are painted after (and hence obscure) more distant surfaces, giving the correct hidden surface solution.

The following algorithm is used to perform BSP-tree traversal:

```
traverseTree (tree)
{
    if (tree is empty)
        return
    else
    {
        if (view point in front of rootPolygon)
        {
            traverseTree (tree->back);
            displayPolygon (tree->rootPolygon);
            traverseTree (tree->front);
        }
        else
        {
            traverseTree (tree->front);
            displayPolygon (tree->rootPolygon);
            traverseTree (tree->back);
        }
    }
}
```

In some applications this procedure can be optimised by not painting backfacing polygons. This is useful if there are only closed objects in the model, for example a cube has six faces but only three of these are visible at any time. In the flight simulator each polygon has a flag to indicate whether it should be painted when the viewpoint is behind it. This allows rotor blades, for example, to be implemented as a single polygon while allowing backface elimination on the body of the helicopter.

This process is recursive — our traverser is implemented in occam which does not allow recursive procedure definitions, so a state machine is constructed. Further details of implementing recursive data structures and procedures in occam programs can be found in another INMOS technical note [14]. The state machine maintains two variables, the current node in the tree, and the current action being performed. These nodes and actions are explicitly stacked as the tree is traversed. Here is an outline of the state machine in occam:

```
SEQ
-- initialise
push (NIL, a.terminate)
action := a.testPosition
node   := rootNode
WHILE action <> a.terminate
CASE action

    a.testPosition
        -- test whether we are in front of
        -- or behind the current polygon
        IF
            node = NIL
                -- end of subtree
                pop (node, action)
            inFront (node, viewPoint)
                -- in front of current polygon
            SEQ
                push (node, a.traverseFront)
                node := tree[node + backSubTree]
            TRUE
                -- behind current polygon
            SEQ
                push (node, a.traverseBack)
                node := tree[node + frontSubTree]
```

```

a.traverseFront
  -- output current polygon
  -- then traverse front subtree
SEQ
  outputPoly (node)
  action := a.testPosition
  node   := tree[node + frontSubTree]

a.traverseBack
  -- output current polygon
  -- then traverse back subtree
SEQ
  outputPoly (node)
  action := a.testPosition
  node   := tree[node + backSubTree]

```

Only half a dozen floating-point instructions are required to determine which subtree to traverse first at any node, so the BSP-tree traverser was incorporated into the same transputer as the 3-D transformation, leaving run-time still dominated by polygon painting time.

BSP-trees are used to determine polygon visibility within each object in the simulator (e.g. aeroplanes, helicopters, teapots, buildings), and a simple bounding box test in z is used to determine the relative z ordering of objects. This means that the system will not correctly render objects when they intersect. However, if this condition occurs in the flight simulator it implies that the objects have collided.

12.6.3 Architecture

The vision system of the flight simulator is as illustrated in Figure 12.6. A geometry system consisting of four transputers performs: (a) BSP-tree traversal and 3-D transformation; (b) z clipping and conversion to screen coordinates; (c) y clipping; and (d) x clipping.

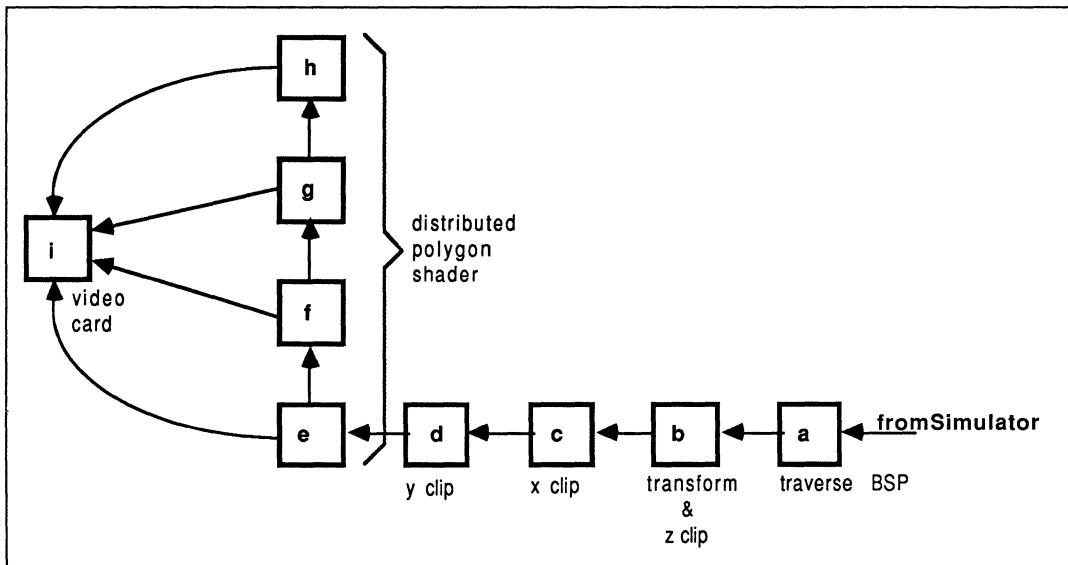


Figure 12.6 Flight simulator vision system

Four transputers (**e**, **f**, **g** and **h**) perform distributed polygon shading and a graphics card operates as a pixel sink (**i**). The processor in the graphics card would normally be idle, the transputer simply waiting for images to appear down its links. This is a waste of a good processor, so more functionality is added. The graphics

card now implements a head-up display showing an artificial horizon, air speed, altitude, bearing, radar with enemy positions and missile fuel readings. All of these make extensive use of the IMS T800's 2-D move instructions.

The simulator itself runs on a single transputer with the vision system connected to one link, and has been designed to allow many simulators to be connected in a ring (Figure 12.7). This allows a number of players to take part in a combat simulation, each player seeing the others through his simulated cockpit window.

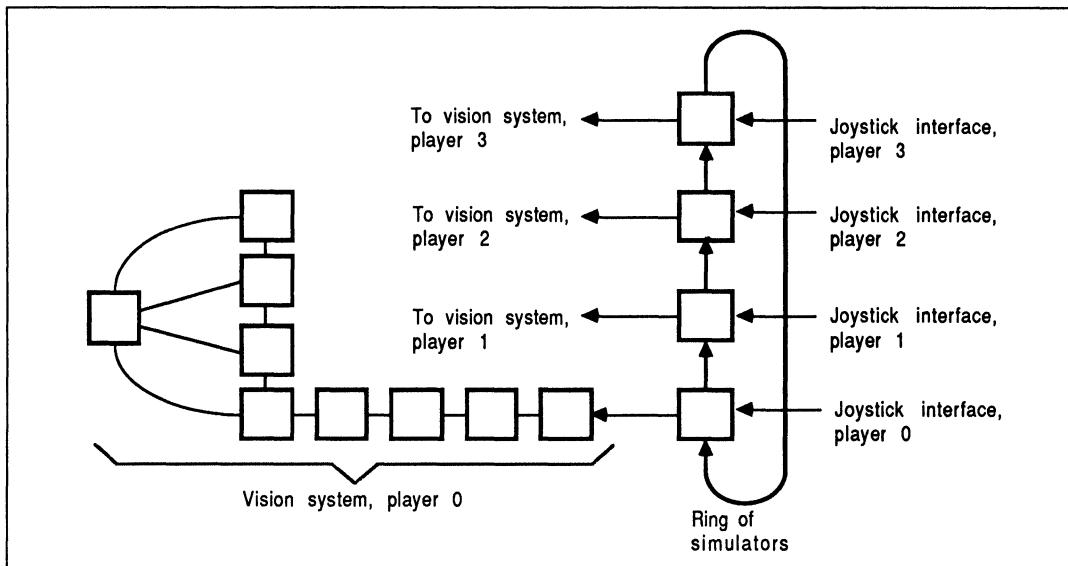


Figure 12.7 Full four-player simulator

At SIGGRAPH '87 INMOS demonstrated a four-player combat simulator, and members of the public were invited to try and shoot down INMOS application engineers. The whole system (i.e. four entire flight simulators) was housed in a pair of INMOS card cages; taking up only 13 double-extended eurocard slots and less than five cubic feet. In the course of a 10 hour combat session more than a terabyte of data (i.e. over a thousand gigabytes) will flow through a four-player simulator.

The implementation of the flight simulator is described in greater detail in another INMOS technical note [15].

12.6.4 Performance

The flight simulator performs as well as anticipated — it consistently achieves a refresh rate of 17 frames per second. The main limiting factor is the need to synchronise the updates to the graphics display with frame flyback. Frame rates approaching the theoretical maximum of 27 frames per second could be achieved by having more buffering in the graphics display hardware. This would allow image data to be received asynchronously with frame flyback. If desired even higher frame rates can be obtained by using more than one transputer in the display system.

Looking at the flight simulator screen, the images only come to life when animated at 17 frames per second. The impression of flight is uncanny, despite the simplistic polyhedral design of the aircraft.

12.7 Conclusions

The IMS T800 offers all the features required for high-performance computer graphics. It is a very high-performance microprocessor capable of being used in large numbers to form extremely powerful multiprocessor computers, with a few well-chosen instructions for computer graphics support.

The IMS T800s 2-D block manipulation instructions make it an ideal candidate for the next generation of high resolution full-colour workstations, and for future generations of colour laser printer controllers.

The IMS T800 has sufficient floating-point performance for any application. If more than 1.5 Mflops are required then use more transputers. Thirty-two IMS T800-20s offer the computational equivalent of current vector super-computers (48 consistently achievable Mflops), and take up only 56 square inches of PCB area (i.e. they will fit on an IBM PC plug-in card).

12.8 References

- 1 *Principles of interactive computer graphics*, William M Newman and Robert F Sproull, McGraw Hill.
- 2 *Reentrant polygon clipping*, Ivan E Sutherland and Gary W Hodgman, CACM 17(1), January 1974.
- 3 *The Transputer Databook*, INMOS Ltd, 1989.
- 4 *Lies, damned lies and benchmarks*, INMOS Ltd, Technical Note 27.
- 5 *Notes on graphics support and performance improvements on the IMS T800*, INMOS Ltd, Technical Note 26.
- 6 *occam 2 Reference Manual*, INMOS Ltd, Prentice Hall 1988.
- 7 *Exploiting concurrency: a ray tracing example*, INMOS Ltd, Technical Note 7.
- 8 *A transputer based distributed graphics display*, INMOS Ltd, Technical Note 46.
- 9 *The transputer instruction set: a compiler writers guide*, INMOS Ltd, Prentice Hall 1988.
- 10 *IMS T800 architecture*, INMOS Ltd, Technical Note 6.
- 11 *A linear algorithm for incremental digital display of circular arcs*, J E Bresenham, CACM 20(2):100-106, February 1977.
- 12 *Fast spheres, shadows, textures, transparencies and image enhancements in Pixel-Planes*, Fuchs, Goldfeather, Hultquist, Spach, Austin, Brooks, Eyles and Poulton, Computer Graphics 19(3), July 1985 (Proc. SIGGRAPH 85).
- 13 *Near real-time shaded display of rigid objects*, Fuchs, Abram and Grant, Computer Graphics 17(3), July 1983 (Proc. SIGGRAPH 83).
- 14 *Data structures and recursion in occam*, INMOS Ltd, Technical Note 38.
- 15 *The INMOS flight simulator*, INMOS Ltd, Technical Note 36.

13 A transputer based multi-user flight simulator

13.1 Introduction

Recently, one of the most popular applications for computer graphics has surely been simulation systems, for example flight simulators. The aim of these systems is to generate some scenario in sufficient detail, and with enough realism to give the operator the impression that the exercise is happening in the real world, and not in a simulated one.

Systems like these require huge amounts of computation, and very high-performance display systems to achieve such realism. In fact, most current systems are implemented in hardware to get the super-computer performance required. The major attraction with these systems is the interaction the user has with the environment (flight simulators allow the pilots to push their aircraft to their limits, without endangering themselves and others). The biggest setback is their inability to allow the environment to react with the user (the computer can rotate radar dishes etc., but complex movements like other aircraft cannot be done without huge amounts of extra processing).

This chapter describes the implementation of a multi-user flight simulator system using transputers and occam. The design allows any number of single-user simulators, each a small supercomputer in its own right, to be linked together to allow interaction between systems. By having a number of other pilots controlling the other objects in the simulation, a trainee pilot can be subjected to more complex scenarios than those that could be programmed into a computer.

The program described is written entirely in occam, and the hardware used in the implementation consists of a number of transputer variants (T212, T414, and T800), all running on standard INMOS transputer evaluation boards. These boards are connected using the INMOS links, allowing complex systems like this to built with relative ease (approximately 10 minutes to wire up a four-player implementation of the simulator).

More details of the transputer and occam can be found in [1], [2] The joystick module is the only custom hardware used. This board is described in the **user interface** section of this note.

13.2 Flight simulators

Simulators (for aircraft, cars, etc.) are becoming increasingly popular applications for computer graphics. Top-of-the-range flight simulators can be used to train pilots for any situations that might occur (freak weather conditions, instrumentation failure, surprise ambush from mischievous mountain ranges, slight reduction in the number of wings, etc.), without endangering pilot, passengers, or crew.

A simulator system must be able to portray the outside world in sufficient detail (both graphically on the windshield, and numerically using instruments), and possibly simulate the motion of the vehicle using a motion platform, such that the pilot feels that he/she is really flying the aircraft. Such a system consists of a number of very high-performance subsystems, such as a display system, motion controller, and a database system that can maintain a model of the world (and any objects that may appear in it), and environment data (rain, clouds, etc.) that must be accessed during the simulation. All this data must be displayed fast (and realistically) enough to give the impression of real flight. To get a frame rate of 20 to 30 frames per second, most of the work is currently done in hardware, which explains the high cost of this sort of system.

With a multi-user simulator, each node must be able to access the data and display it in accordance to the current position of the craft at that node. Also needed is a knowledge of where all other users in the system are, so they can also be displayed (if visible).

This implementation of a flight simulator allows any number of users, each with their own simulation engine, to interact. Each user gets a view of the world as if looking through the cockpit window. The world is made of polygons, and these are displayed at a rate of approximately 17 frames per second (at about 200 polygons per frame).

This frame rate is limited by the design of the graphics board used in the current system. To avoid visual artifacts it is necessary to wait for frame fly-back before updating the display. As there are only two display banks on the IMS B007 graphics board, this has the effect of holding up link communications with the shader

processors for up to one frame time (**1/50th sec**). If the wait for frame fly-back is removed, the frame update rate is increased to approximately 22 frames/sec, as the buffered image can be displayed as soon as it is received. An enhanced graphics board is currently being designed at INMOS, which has up to four frame buffers, allowing higher frame rates (as the $n+1$ th frame can be read whilst the n th frame is waiting to be displayed) **WITH** frame fly-back.

All users are connected in a ring. Any part of the world that needs to be distributed is passed around to each user in turn, who can read it, modify it, or ignore it, then pass it on to the next user. Objects can be dynamically added to this network (such as missiles that have been fired), and taken out when finished with. The ring architecture allows any number of users to be included in the system, and the software has been written with this feature in mind.

13.3 Architecture

The architecture for a single-user system is shown in figure 13.1. The system has been subdivided into the most logical processes that occur in a simulator, e.g. the core simulator, 3-D transformation, clipping, etc. Note that this is the software model for such a system, and not the hardware implementation, which is described in the **hardware implementation** section.

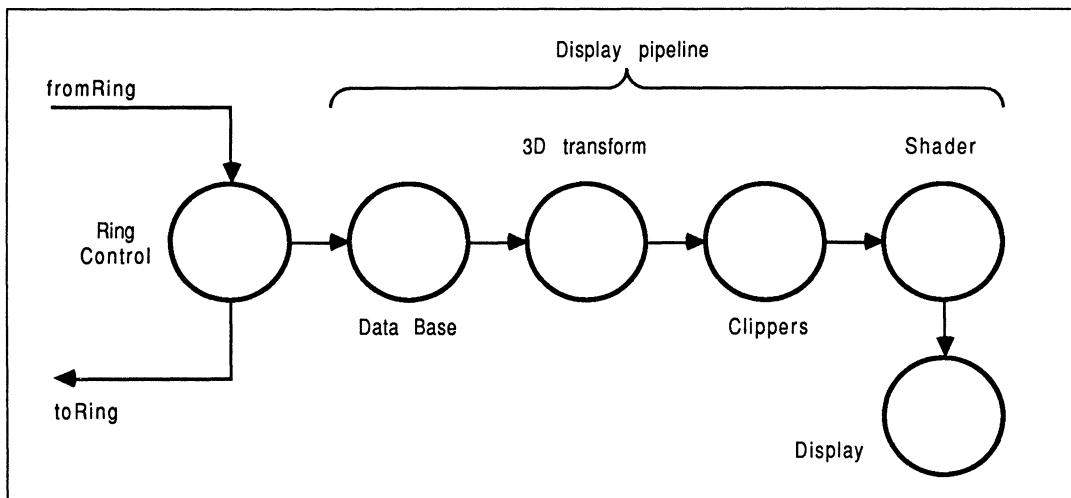


Figure 13.1 Architecture for one player

13.3.1 An overview

Before explaining in detail the various processes that are used in the simulator, a brief overview of the system, and some of the terminology used, is required.

The system has been designed to allow any number of flight simulators, such as the one shown on figure 13.1, to be connected in a ring. Messages are passed around the ring defining the position, and orientation of objects in the simulation. Figure 13.2 shows an example of four such systems connected in a ring.

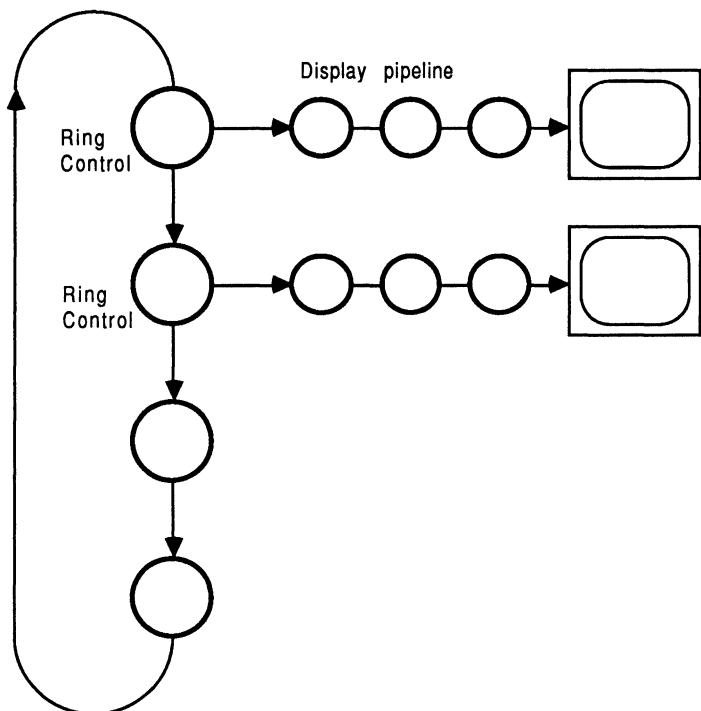


Figure 13.2 The ring architecture

The **ring Control** process handles all ring communications, and interaction with the pilot controls. From these information sources, a description of the next visible scene for this user can be derived.

Objects are described as a set of polygons, and these polygons are stored in a large database. To display an object, these polygons must be output such that the near faces of the object obscure those further away. The **data base** process takes care of this **hidden surface** algorithm.

The polygons described in the database are not suitable to be written directly to the screen. Each polygon must undergo a transformation to convert it to a displayable form. In the following sections, reference will be made to **model**, **world**, **eye** and **screen** coordinate systems. Figure 13.3 shows an example of how these different coordinate systems relate. Converting between coordinate systems requires the coordinates (of a polygon, for example) to be transformed using matrix multiplications [3], [4]. The transformations can be rotations about an axis, scaling, and translation along an axis. For example, the transform from **model** to **world** space requires a 90-degree rotation about the Y-model axis (see Figure 13.3).

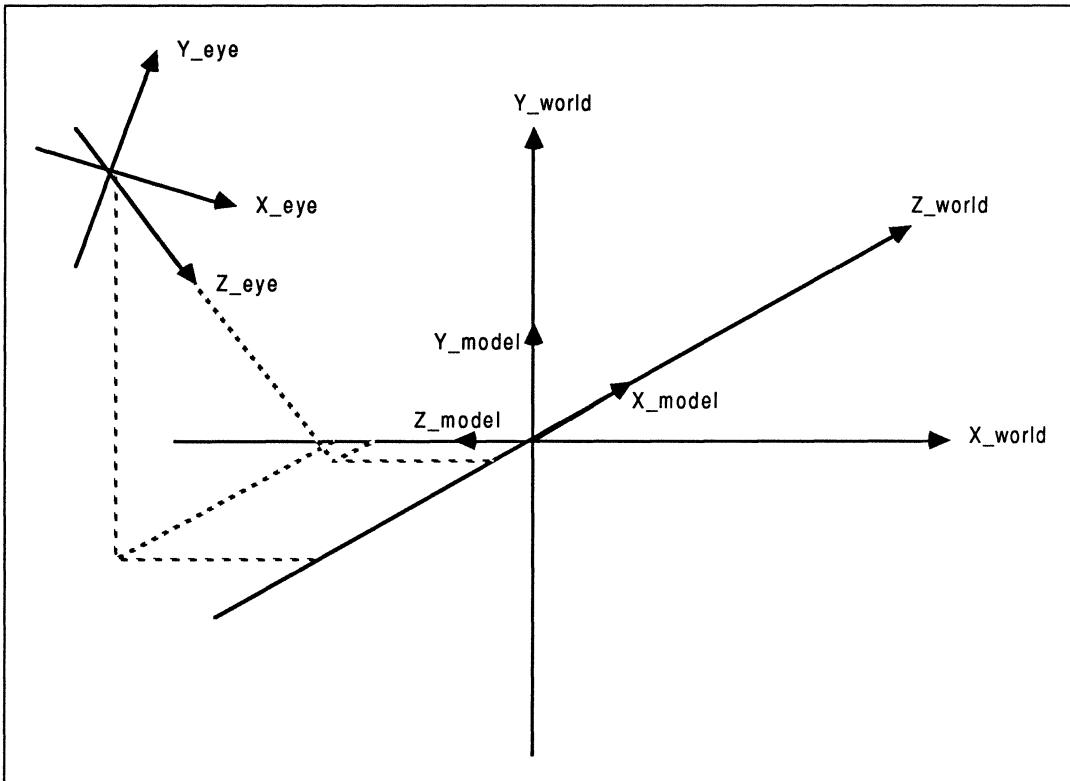


Figure 13.3 The coordinate systems

Matrices may be concatenated, allowing a single point to undergo a number of transformations with only a single matrix multiplication. The order in which the transformations are applied to a point is determined by the order in which the transformation matrices are concatenated. By reversing this order, a reverse transformation (from **screen** to **model** coordinate systems, for example) can be generated.

The conversion from **model** to **screen** coordinate systems is performed by the **transformation** process. Matrices that define the transformations to be performed are generated by the **ring Control** process, and are used to transform the polygons that are output from the database.

Before shading a polygon, a clipping operation must be performed to remove any parts of that polygon that may not be visible. As well as clipping to the screen boundaries, the polygons must also be clipped to the z coordinate viewing boundaries (often referred to as the 'hither' and 'yon' clipping planes) to remove parts of polygons that may be behind the viewer, or beyond the horizon. Before clipping to the screen boundaries, the perspective calculations are performed. This effectively scales the 2-D polygons according to their distance from the view point.

Many shading algorithms can be used to generate the final display. The simplest is flat shading, where each polygon is filled with a designated colour. More complex shading algorithms take the position of light sources into account, and so require more complex calculations to be performed at each pixel to determine the colour and intensity at that point. Shadows, highlights and reflections could also be included, but require vast amounts of extra processing.

In the following sections, an understanding of both **occam** and the transputer is assumed.

13.4 Implementation

13.4.1 The ring Control process

The ring Control process controls the operations on all data on the ring. Figure 13.4 shows the internal configuration of this process.

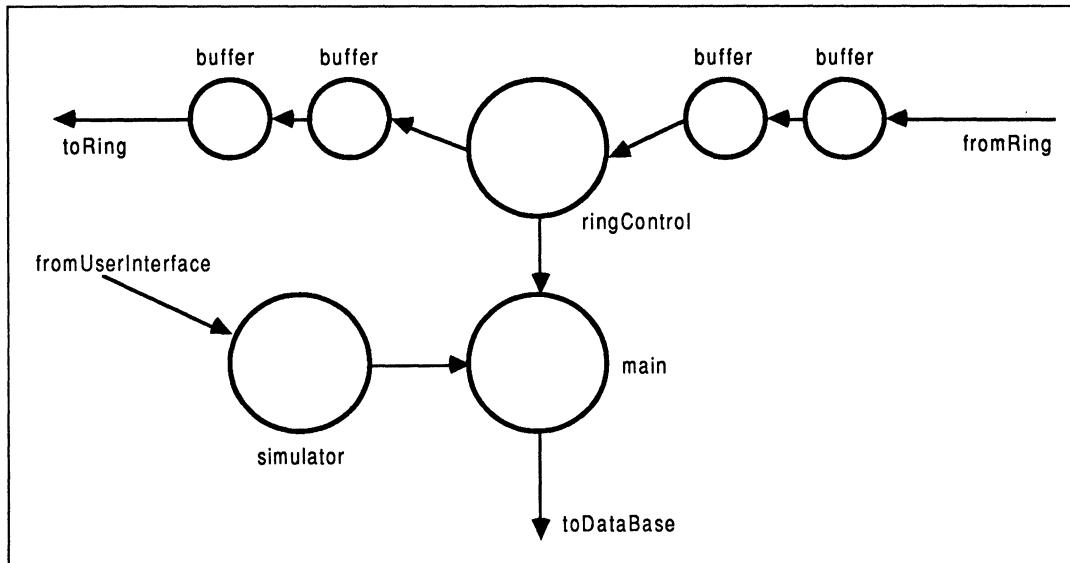


Figure 13.4 The ring controller

The ring Controller

The interplayer communications are implemented as a ring-based architecture. This allows any number of users to be incorporated into the system.

For a true multi-user system, each single-user simulator must have access to the same database. As shared memory is not supported, the systems must communicate by passing messages over OCCAM channels [2]. The database required for a simple flight simulator is quite large, and passing this as a message between many small flight simulators would severely limit the performance of a system. One feature with such a system that can be exploited is that a very small proportion of the database actually changes during the operation of the simulator. The ground remains static, buildings tend not to move very often, and so the communications can be reduced to passing information regarding dynamic objects within the system, i.e. the aircraft controlled by the other users.

Messages passed around the ring are object descriptors (such as type, position, and other attributes). By keeping the descriptor as a block, the transfers of the block around the ring can be done 'for free' by the block move engines in the transputer links. Double-buffering techniques (described in [5]) increase system performance by allowing the processing of the current descriptor whilst outputting the previous one and inputting the next.

The **ringController** process (and associated buffers) handle all ring traffic. The buffers allow the number of messages on the ring to change dynamically (deadlocks could occur if all links used in the ring were active (busy transmitting or receiving), and a new object (such as a missile) was added).

A number of simple decisions are made on the objects as the **ringController** receives them. These are:

```
[blockSize]INT inBlock, outBlock :
SEQ
    in ? inBlock
    WHILE running
        SEQ
            IF
                opponent_aircraft
                PAR
                    toMain ! inBlock
                    outBlock := inBlock -- (1)
                missile
                SEQ
                    toMain ! inBlock
                    fromMain ? outBlock -- (2)
                owners_aircraft
                SEQ
                    toMain ! inBlock
                    fromMain ? outBlock -- (3)
            PAR
            in ? inBlock
            out ! outBlock
```

- 1 The descriptor is sent to the **main** process, and can be also be passed on to the ring.
- 2 The result from the main process could be the missile attributes being altered to indicate the opponents missile has hit this (the testing process) aircraft, or the missile has been removed from the ring (in which case, the output to the ring is not performed).
- 3 The result is the new owner descriptor (current position etc.).

The simulation process

The simulation process controls the interaction with the player (via a joystick interface), and adjusts the position and rotation of objects belonging to that player (such as missiles and the aircraft the player is flying). There are up to three objects to keep track of (one aircraft and two missiles), and any of these can be under the control of the player. The others fly 'blind' (i.e. along the course on which they were fired). When a missile is under control, the aircraft continues flying in the direction it was moving before the missile was taken under control.

```
SEQ
    ... get data from user interface
    ... decode to give any changes in direction etc
    IF
        driving_missile
        ... alter missile course, move others -- (1)
    TRUE
        ... alter aircraft course, move others -- (1)
    ... get status from main -- (2)
IF
    collision -- (2)
    ... change state to explode
TRUE
    SKIP
PAR
    ... output the player descriptor
    ... set up head-up display (HUD) -- (3)
    ... send HUD (also marks end of frame)
```

- 1 Each object is stored as current position and delta movement for the next frame. When controlling a certain object via the joystick interface, the stored delta values are overwritten with those derived from the joystick inputs. Here, any environment simulations, such as gravity and atmospheric conditions such as wind, can also be included.

Transformation matrices are generated to describe the positions and orientation of these objects, and the view point, and these matrices are passed to the **main** process.

- 2 The **main** process (see below) will, at this point, have completed the reading of data from the ring, and will have a list of objects for output, together with information giving the result of the collision detection, and an array of coordinates to be used in the map. This data is taken as being a request for the object descriptor generated in (1).

The value of the collision tag is used to alter the object attributes to signal an explosion (which selects an explosion database to be displayed). During the explosion cycle, inputs from the user are ignored, and the aircraft is forced into a fixed sequence of moves before restarting. The coordinate array passed is used in the head-up display.

- 3 The head-up display is sent as a set of high-level commands to the display engine. The attitude of the aircraft (pitch, roll, yaw, etc.) is converted to a sequence of line draw commands (which build up an artificial horizon display), the map is a set of plot pixel commands, and the instruments are defined as circle commands. These commands are built into a display list, which is transferred as a single block through the display pipeline to the display process.

Missile descriptors pass around the ring until the missile has hit something (either another aircraft or the ground), or has run out of fuel. The 'hit' detection is done in the **main** process below, and that process will request a particular missile be updated or removed from the ring. The current implementation allows for up to two missiles per person to be active (flying) at any time. This is limited only by the available ring bandwidth. Future topologies should allow for more objects to be present in the system.

The main process

The **main** process (the frame generator) takes its inputs from the **ringController** and **simulation** processes, and calculates which of the objects can be seen. The information from the **simulation** process determines the location (in world space) of the player's screen, and this information is used to translate all other objects to the **eye_space** of the player.

Any objects that can be seen are z-sorted to give a list of object descriptors, which are output (in reverse depth order) to the pipeline to be converted into polygons and shaded. At this stage, only a description of the object (e.g. type, location, rotation) is needed. It is expanded on in later stages of the pipe.

Within the z-sort routine, collision detection and trivial rejection are also performed. As there is no master in the system, all user processes do their own collision detection (i.e. they detect if **they** have been hit). If a hit has occurred, messages are passed back around the ring to the owner of the other object in the collision, who will take the appropriate action.

Both collision detection and trivial rejection are done using bounding boxes, which undergo a simple transformation to put the bounding box in eye space. If the bounding box can be clipped from view, the object is rejected (and is not inserted in the list). If two bounding boxes intersect, a collision has occurred, and the other object owner is informed that a collision has taken place. Because all players do their own collision detection, a test of all boxes interacting with all other boxes does not have to be performed. The test is simply sort each object as it arrives from the ring, and test the closest object with the bounding box of the player. As the test is done when the descriptor is read from the **ringController** process, the **hit** flag can be set in the object attributes before it is sent back to the ring.

The end of a frame is signalled by the **main** process reading the object descriptor of its owner from the ring (this is sent out at the start of the frame, and its return signals that all other descriptors have passed through). At this point, the list of descriptors is sent to the traverser process (in reverse z order) for the hidden surface algorithm to be executed.

```

SEQ
PAR
... get ring data, z-sort, collision detect -- (1)
... output head-up display, ground -- (2)
PAR
... output sorted descriptor list -- (3)
... get new player data from movement process -- (4)

```

- 1 For each item received from the ring, do a trivial reject and collision detect (if the object is closest to the player), and set **hit** flag if collision has occurred. If the object belongs to this player (i.e. a missile), then request the new missile descriptor from the simulation process. The descriptor is sent back to the **ringController** to be inserted into the ring.
- 2 Whilst reading in the next frame data, we can keep the display pipeline busy by sending the head-up display list to the graphics engine, followed by the end-of-frame marker, and then getting the **traverser** to output the ground details for the next frame.
- 3 For each object in the list, three items must be passed to the display pipeline. First, the transformation matrix (from model to screen space, including perspective transform), is passed on to the **transform** process. The viewpoint (which is transformed to be in the **model** coordinate system) is passed to the **traverser** process, as is the model type. The last two items are used to select and output a particular portion of the database.
- 4 Pass the map data (and result of the collision detection) to the **movement** process, and receive the descriptor for next frame.

13.4.2 The Data Base manager

The models that are used in the simulator are stored as a tree of polygons. A version of the Binary Space Partitioning (BSP) algorithm [6] is used to determine which polygons are visible (and in what order) from any viewpoint.

The algorithm is quite simple. A polygon lies on a plane, defined by the equation:

$$Ax + By + Cz + D = 0$$

where A, B, C, D are constants (calculated from three coplanar vertices of the parent polygon). If a point [x,y,z] satisfies the above equation, that point is said to lie on that plane. However, if the result is negative, the point is said to lie behind the plane, and if the result is positive, the point lies in front of the plane.

BSP trees store the polygons in a recursive manner, with a polygon at each node of a binary tree. Each node points to a subtree of polygons that lie in front of the parent polygon, and a subtree of polygons that lie behind the parent.

Building the BSP tree

Building the BSP tree is a recursive operation. Starting with a list of the polygons that make up a model, we set the root node of the tree to be the first polygon in the list. The plane on which this root node lies is calculated (the A, B, C, D constants are stored in the record for that node, for use during the tree traversal at run-time), and all subsequent polygons in the list are tested to see whether they lie in front or behind the root node.

After testing all polygons in the main list, the **in_front** and **behind** pointers of the root then point to two sublists of polygons. Each sublist is then recursively traversed, until a binary tree, with a single polygon per node, is created.

The test to derive whether a polygon lies in front or behind the plane involves entering each polygon vertex into the plane equation. If all vertices lie in front of the plane, the polygon is in front of the plane, and it is added to the end of the **in_front** sublist of the root. If the polygon lies behind, it is added to the **behind**

sublist (when the plane and the polygon are coplanar, the polygon can be added to either list). Note that if the polygon is split by the plane (some vertices are in front, some are behind), the polygon must be divided into two subpolygons, which are inserted into either sublist.

A graphical example is shown in Figure 13.5 (we will work in two dimensions, but it is simple to extend the principles to three dimensions). At the start (13.5a), the list contains five polygons **a**, **b**, **c**, **d**, **e** (the arrows are used to show where **in_front** is for each polygon). Polygon **a** is chosen to be the root, and Figure 13.5b shows the result of the tests performed on the other four polygons. If the algorithm traverses the **in_front** sublist, then the **behind** sublist (as done by the **combine** routine in the description below), Figures 13.5c shows the results as each sublist is traversed.

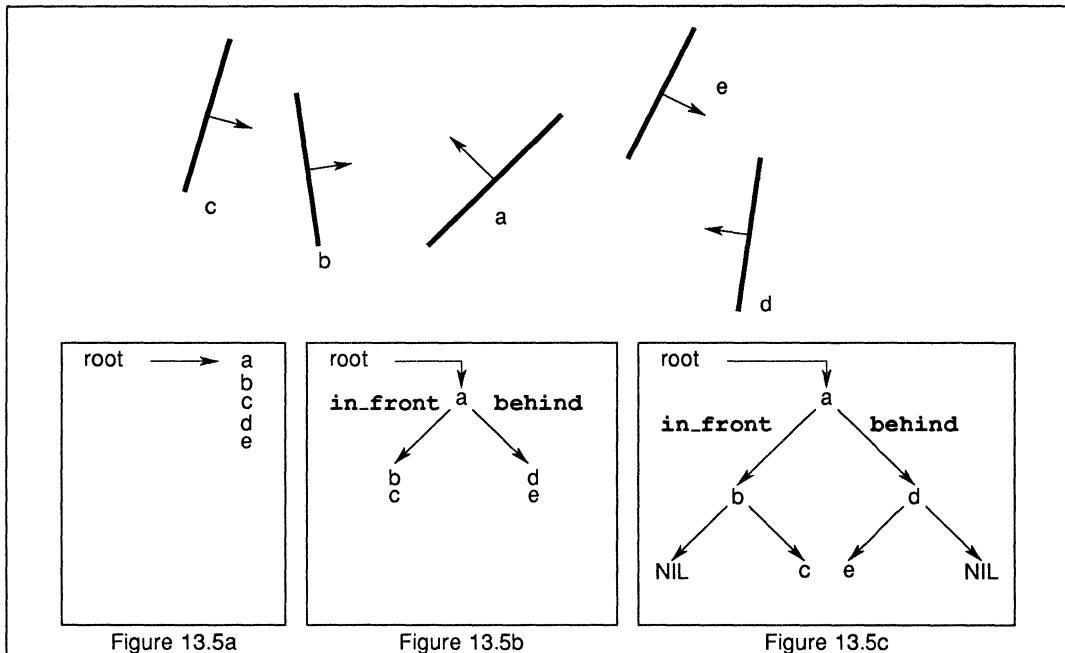


Figure 13.5 Building a binary-space partitioned tree

Here is a piece of pseudo-code describing the algorithm [6]:

```

PROC make_tree (poly_list)
    returns (BSP_tree) ;

if (poly_list is EMPTY)
    return (NULL_tree)

else
{
    root := select (poly_list) ;
    back_list := NULL ;
    front_list := NULL ;
    foreach (polygon in poly_list)
        if (polygon is not the root)
        {
            if (polygon in front of root)
                Addlist (polygon, front_list) ;
            else if (polygon is behind root)
                Addlist (polygon, back_list) ;
            else
            {
                split_poly (polygon, root,
                            front_part, back_part) ;
                Addlist (front_part, front_list) ;
                Addlist (back_part, back_list) ;
            }
        }
    return (combine_tree (make_tree (front_list)),
            root, (make_tree (back_list)) ) ;
}
END

```

combine_tree links the root to the sub trees.

This procedure will generate a BSP tree from a list of polygons. The node chosen for the root strongly determines the order in which the polygons are stored in the tree. In the simulator, it was necessary to have certain polygons at leaf nodes of the model tree, and so the tree was constructed manually. This is not an easy task, but writing a routine to build the tree , following certain constraints as to the location of arbitrary polygons, was not thought possible in the time allowed.

The BSP trees are static for all the models, and so can be constructed during the initialisation phase of the simulator.

Traversing the BSP tree

Traversing the tree is a recursive operation. Here is a piece of pseudo-code describing the algorithm:

```
traverseTree (tree)
  if (tree is empty) return
  else
    {
      if (view point in front of root polygon)
        {
          traverseTree ( tree -> back )
          displayPolygon ( tree -> rootPolygon )
          traverseTree ( tree -> front )
        }
      else
        {
          traverseTree ( tree -> front )
          displayPolygon ( tree -> rootPolygon )
          traverseTree ( tree -> back )
        }
    }
```

The result of this operation is that the polygons in the tree are always displayed in a back-to-front order, i.e. the furthest polygon from the viewpoint is output first. In this way, the correct hidden surface solution is achieved for all possible viewpoints.

A state machine is required to simulate this recursive operation when using OCCAM. The state machine has two variables, the current node in the tree, and the current action being performed. Nodes and actions are stacked as the state machine traverses the tree.

```
push (NIL, a.terminate)
action := a.testPosition
node := rootNode
WHILE action <> a.terminate
  IF
    action = a.testPosition
    IF
      node = NIL
      pop (node, action)
      inFront (node, viewPoint)
      SEQ
        push (node, a.traverseFront)
        node := tree [ node + backSubTree]
    TRUE
    SEQ
      push (node, a.traverseBack)
      node := tree [ node + frontSubTree]
  action = a.traverseFront
  SEQ
    outputPoly (node)
    action := a.testPosition
    node := tree [ node + frontSubTree]
  action = a.traverseBack
  SEQ
    outputPoly (node)           -- (1)
    action := a.testPosition
    node := tree [ node + backSubTree]
```

¹ In some cases, it is not necessary to draw this node, as the definition of being 'behind' a polygon means that the polygon is facing away from the viewer, and so should be obscured by polygons

facing the viewer. For example, a cube has six faces, but it is only possible to see a maximum of three. The other three are **backfacing** polygons. These backfacing polygons need not be drawn.

For space considerations, we have a boolean tag in the record for each polygon which enables 'backface rejection' on specific polygons. Wings of an aircraft can be described as a single polygon, displayed no matter where the viewpoint is, but the box defining the bulk of an aircraft body can be forced to reject backfacing polygons.

To output a model, the **traverser** process reads a viewpoint and model type from the **main** process (above). The model type selects the particular tree to be output, the viewpoint determining the order in which the polygons are output.

13.4.3 The transformation process

The transformation process takes polygons (lists of points), and transforms these points from the model coordinate system to the screen co-ordinates

Three-dimensional transformation (scaling, translation and rotation) is performed using matrix multiplication [3]. Matrix multiplication can be implemented very efficiently on the T800 (see [4] for a more detailed discussion). Whilst the FPU is calculating the product (for example) of two matrix elements, the integer processor can be calculating the address where the result must be stored. By overlapping the floating-point calculations with the index calculations (done by the compiler, incidentally), a coordinate transform:

```
[4]    REAL32 a, c :
[4][4]  REAL32 transform :
..
.
SEQ
  matrixMult (c, a, transform) -- does (c := a * transform)
```

can be done in approximately 19 microseconds. This gives a peak transform rate of over 52 000 points per second.

The transformation process accepts a transformation matrix from the pipeline. All subsequent polygons are then transformed with that matrix until a new matrix is received.

13.4.4 Clipping

Before the polygons can be displayed, they must be clipped to the viewing boundaries. Clipping in the z-plane removes the parts of polygons that are behind the viewpoint, and also polygons that are beyond the horizon. Perspective calculations are then performed (scaling the polygons with respect to their distance from the viewpoint). Finally, the polygons are clipped to the screen boundaries.

Both the z-clip and perspective calculations require floating-point calculations. However, the screen is addressed as an integer device, and so the x- and y-clip operations can be performed in integer form. This eliminates the need for a more expensive floating-point unit in the later clipping stages.

Figure 13.6 shows the structure of the clipping processes. The output of the final y-clipper is a stream of polygons ready to be drawn on the screen.

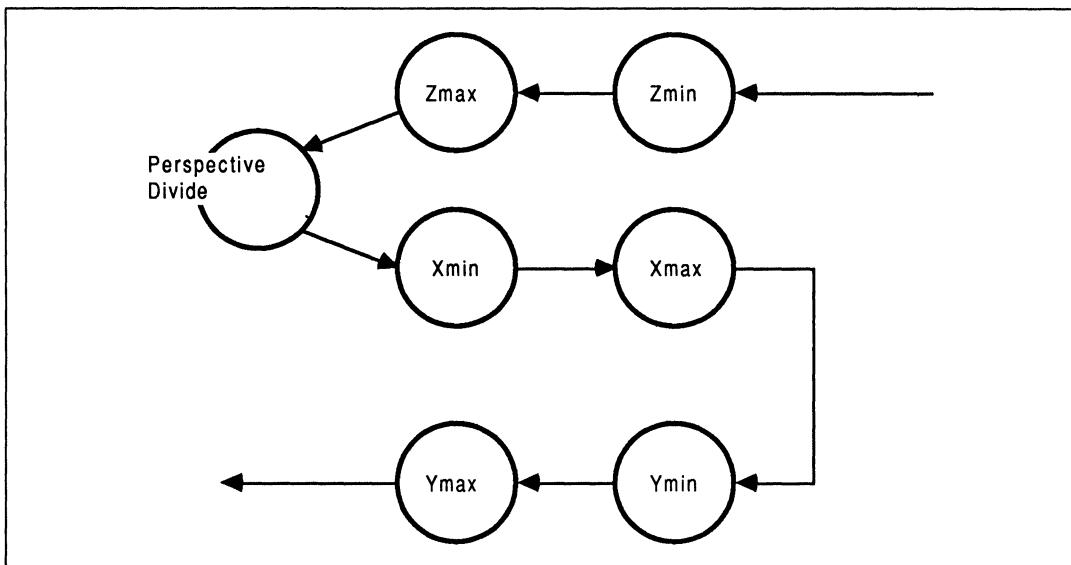


Figure 13.6 The x,y,z clipping architecture

13.4.5 Shading

For the polygon filling, the screen is split into a number of subscreens, each handled by its own shader (Figure 13.7 shows the internal structure of one such shader). In the current implementation, there are four subscreens, each handled by a transputer. Each polygon that survives the clipping process is passed to all shading processors, which shade their part of the polygon.

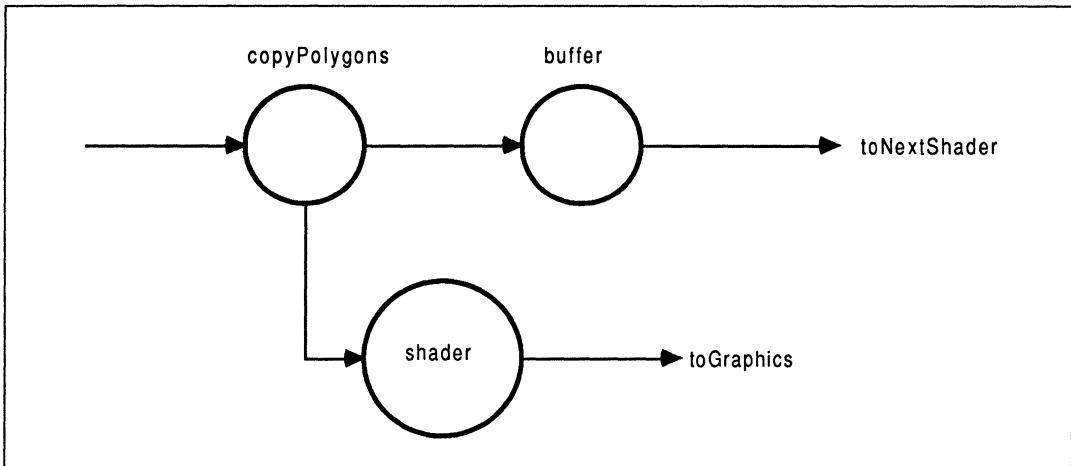


Figure 13.7 Internal structure of shader

Each polygon shading process shades every fourth line of the polygon. The operation starts at the bottom vertex (smallest y-coordinate) of the polygon. Here, two vectors are set up, which define the rate of climb along the two edges which meet at that vertex. This vector defines the step in x that will occur for every step in y . The y step is four pixels (for four shading processors). After initialising these vectors, shading

is a matter of taking a step in y , calculating the new (x,y) locations for along the edges, then joining those points with a horizontal line (the colour of which is defined in the polygon descriptor). As each of the polygon vertices is encountered, the vector defining the appropriate x step is recalculated.

Polygons are flat shaded (for simplicity). Shading the horizontal line is simply a matter of block moving data from on-chip RAM into the buffer used to generate the subscreen. As the shading processor may see many lines within one polygon, an array (stored in the on-chip RAM) is initialised to the polygon colour as the polygon is read in, and this is used for every line fill until a new polygon is received.

When implemented on a T414-20 with single wait-state external memory (200 nanosecond cycle), a single shader can fill polygons at a rate of 16 million pixels (8 bits per pixel) per second, or 62 nanoseconds per pixel. Therefore, four shading processors can shade at a rate of 64 million pixels per second, only 15.6 nanoseconds per pixel. If more performance is required, more shading processors can be added.

At the end of each frame, the subscreen is transferred to the graphics process for display. Again, full use of double buffering of the links and processor is used, to allow the last frame's worth of image to be transmitted to the display engine, while the start of the next frame is being computed. The output format is simply the y coordinate of each line, followed by the 512 bytes that make up that line. The 512 byte line is transferred as a single block, increasing efficiency.

13.4.6 The display

The display card (Figure 13.8) accepts complete subscreens from the shader processes, and transfers them (using the INMOS links) directly into screen RAM.

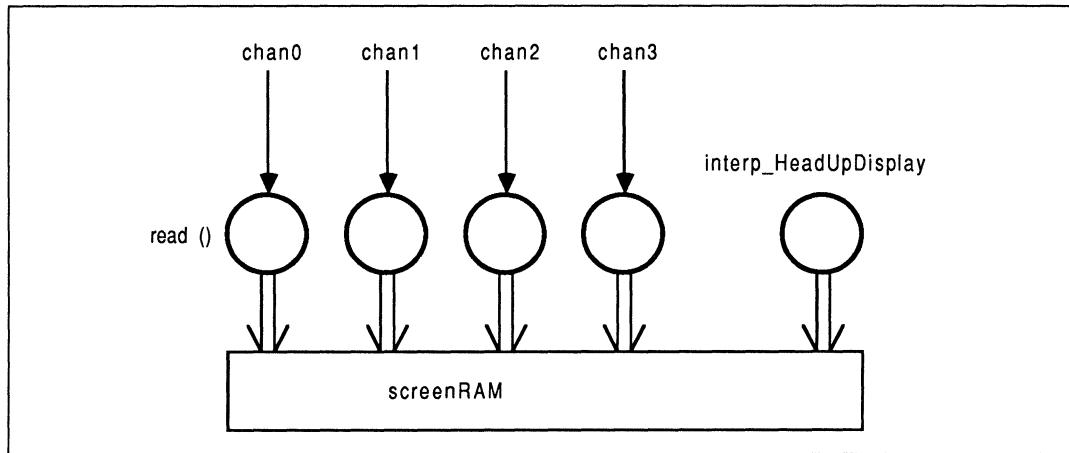


Figure 13.8 The display process

The graphics card (IMS B007 transputer evaluation board) used has a single IMS T800, thus giving four links into the display process. The four input channels `chan0`, `1`, `2`, `3` are mapped on to the hardware links, allowing complete subscreen to be read directly into the screen RAM independent of the processor. Hardware double buffering on the IMS B007 allows one screen of data to be read in while another screen is being displayed, so screen update is invisible. At the end of frame mark, the screens are flipped over.

Also passed at the end of frame is extra information which is used to generate a head-up display (for the next frame). This information can be processed while the next frame data is being read in.

```

PLACE chan0 AT link0in :
PLACE chan1 AT link1in :
PLACE chan2 AT link2in :
PLACE chan3 AT link3in :

WHILE running
SEQ
  in ? headUpDisplay
  ... input bottom_128_lines
PRI PAR
  ... input top_384_lines -- at high priority
  interp_HeadUpDisplay () -- using bottom 128 lines
  flipScreens ()

```

The data from the polygon shaders is input from all four links simultaneously.

```

PRI PAR
{{{
  PAR
    read (chan0) -- read from link0
    read (chan1)
    read (chan2)
    read (chan3)
}}}
interp_HeadUpDisplay ()

```

The **read** processes are started at highest priority, and will be descheduled as each link starts to operate. Once all high-priority processes have been descheduled, the processor, is free to run the low-priority **interp_HeadUpDisplay** process. These processes can be run in parallel, as there are effectively operating on two separate arrays, i.e. the bottom 128 lines and the top 384 lines of the screen.

The head-up display information is written to the screen using the extra graphics instructions [7] of the IMS T800. The **move2Dnonzero** command will transfer all 'non-zero' bytes of an array, giving the effect of an overlay operation.

13.4.7 User interface

The interface to the user joysticks is implemented using IMS C011 link adaptors. These devices convert byte-wide parallel data to the INMOS Link format, and vice versa.

The joysticks used simply present 6 bits of information to the link adaptor. This device is wired such that a message byte, sent from the controlling transputer, will trigger the input half of the C011 to sample, and transmit the current joystick value. This value is decoded to find which switches were active at that time. A circuit diagram for the joystick interface is shown in Figure 13.9.

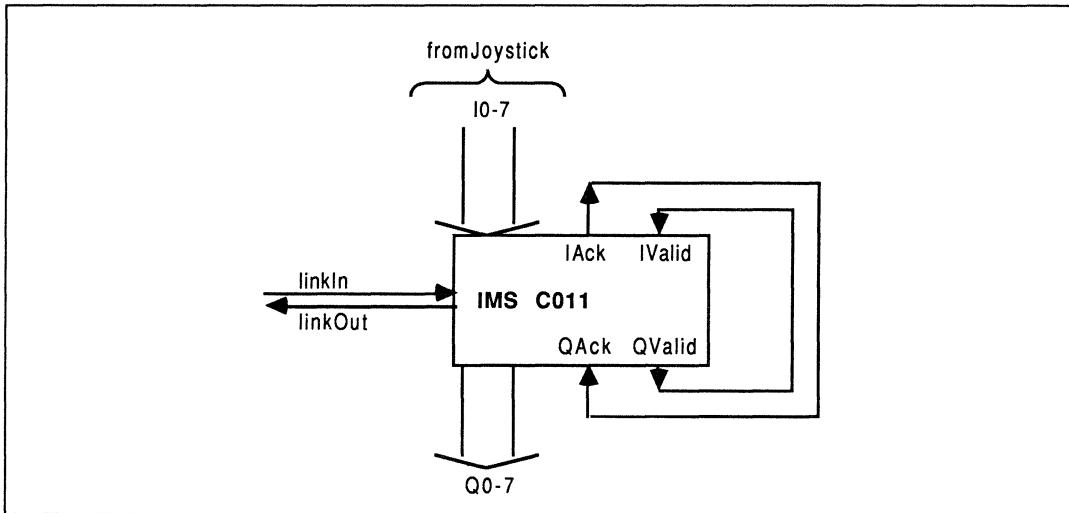


Figure 13.9 The joystick interface module

13.4.8 The hardware implementation

The hardware used in the demonstration system is shown in Figure 13.10, along with the mapping of the processes described earlier.

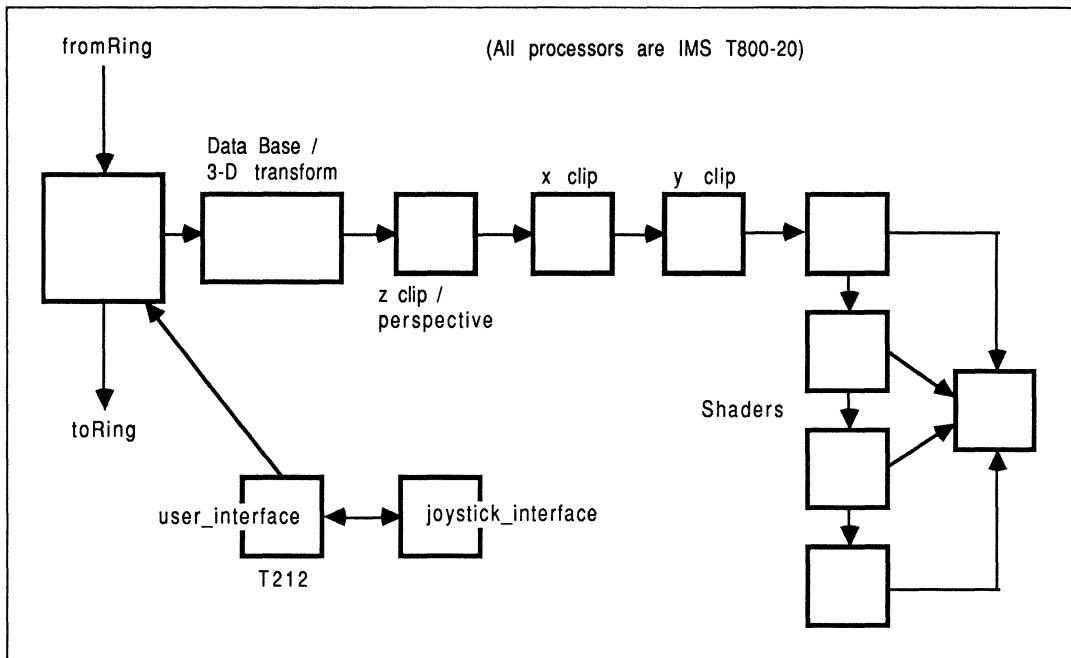


Figure 13.10 Hardware implementation for a single-user system

The front end of the system (the ring controller and transforming processes) are floating-point intensive, and

are PLACED on to IMS T800 processors. Other processes, such as the x- and y-clippers do not use the FPU of the IMS T800, but take advantage of the higher link bandwidth available (all links are run at 20 Mbits per second, and use overlapped acknowledge packets).

As well as interfacing to the joystick modules, the IMS T212 processors run an autopilot process, which cuts in if the joysticks are not touched for a certain time.

The minimum hardware for a system is a single-user simulator as shown in Figure 13.10. Larger systems can consist of a mixture of these full simulators, and a cut-down version with no display pipeline, connected in a ring (see Figure 13.11). The cut-down simulator runs an aircraft under autopilot control, giving the interactive users something to shoot at !

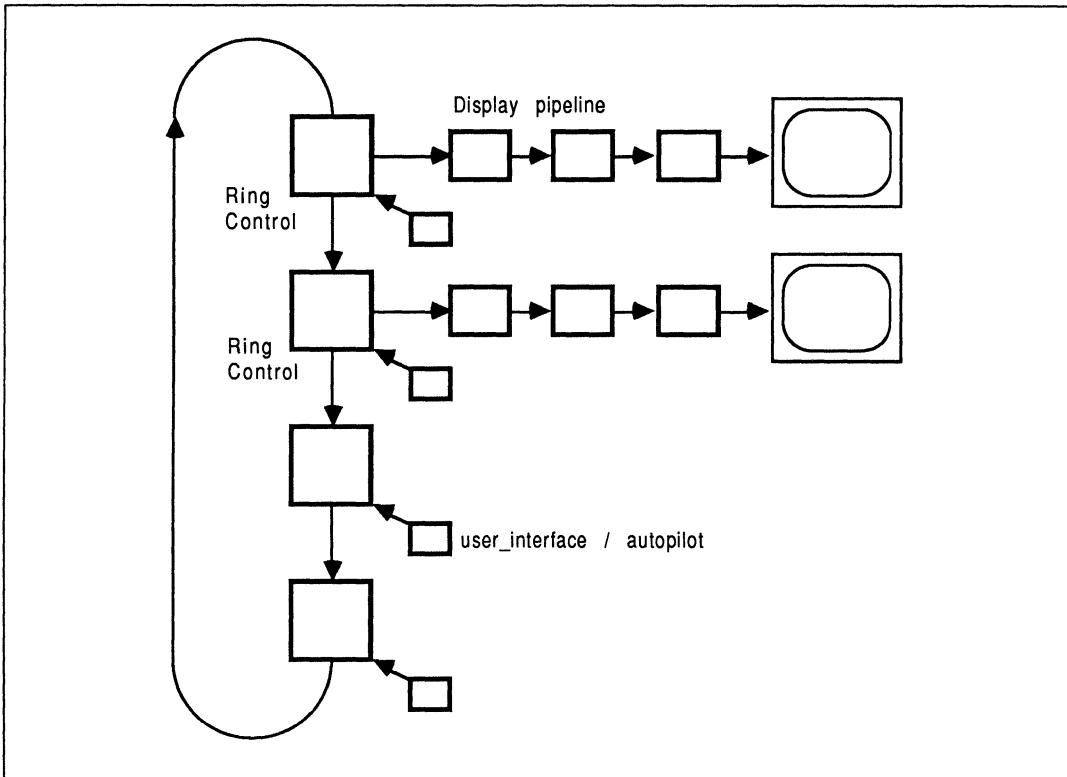


Figure 13.11 A two-player, two-autopilot example

13.5 Conclusions

An implementation of a multi-user, interactive flight simulator, using occam and transputers has been described. The system hardware is standard INMOS transputer evaluation boards, and all the software (written using the INMOS transputer development system) was written in under three weeks. We believe this is a record for such a system.

Future upgrades to the system include 3-D terrain mapping, better shading models, and more realistic flight characteristics. As extra features are added, more transputers can be added into the system to cope with the extra processing required.

13.6 References

- 1 *The Transputer Databook*, INMOS Ltd, 1989.
- 2 *Occam 2 Reference Manual*, INMOS Ltd, Prentice Hall 1988.
- 3 *Principals of interactive computer graphics*, William M Newman and Robert F Sproull, McGraw Hill.
- 4 *High performance graphics with the IMS T800*, Inmos Ltd, Technical Note 37.
- 5 *Performance Maximisation*, INMOS Ltd, Technical Note 17.
- 6 *Near Real-Time Shaded Display of Rigid Objects*, Henry Fuchs, Gregory D Abram, Eric D Grant, Computer Graphics (Vol 17, No 3) July 1983.
- 7 *Notes on Graphics Support and Performance Improvements on the IMS T800*, INMOS Ltd, Technical Note 26.

14 Porting SPICE to the INMOS IMS T800 transputer

14.1 Introduction

This document describes work carried out by INMOS Bristol to port the public-domain circuit simulator program SPICE to the INMOS IMS T800 transputer [1]. The document concentrates on the issues of porting the application, but also includes some background information on the application, on transputers, and some performance information. Methods of increasing the performance of SPICE are also outlined.

It is hoped that the experiences described in this document are of value to others attempting to port existing applications onto transputers. For additional information on the general subject of porting software to transputers, the interested reader is directed to [2].

We chose to port SPICE as an example because it is in the public domain, is widely used within the electronic engineering community, and because it is a highly floating-point intensive application. It is written in FORTRAN, and the results we have obtained show that a single transputer is a high-performance sequential processor in its own right. As such it can be used to accelerate the performance of any conventional application.

However, the transputer is specifically designed to allow multiple processors to be used for a single application, and with a small amount of work many conventional applications can be modified to use a number of processors. The availability of a FORTRAN 77 compiler¹ for the transputer allows most scientific applications to be ported in this manner.

14.2 Background on SPICE

SPICE is a circuit simulator program, written in FORTRAN-77, which is widely used in the electronic design community. It was written at the University of California, Berkeley, by L. Nagel, E. Cohen, and R. Newton with contributions from many others [3, 4]. Berkeley have continually updated the program over the years, and have released successive versions into the public-domain. This work is based on version SPICE 2G.6.

SPICE simulates the behaviour of electrical circuits, at the level of voltages and currents in the circuit, rather than at the logical behaviour level. To execute the SPICE program, a data input file is supplied by the user. This file contains a description of an electrical circuit to be simulated, as a node connection list for the circuit. It also includes electronic device model parameters, and operating specifications for the simulation (e.g. time, temporal resolution, required outputs, etc.).

The output from SPICE takes the form of tables of figures, or character-based graphical plots.

Partly because SPICE is so large and demanding, and partly because of its early origins, it is generally run on large main-frame or mini computer facilities.

SPICE is not an *interactive application*, so it is well suited to being run as a background task in a batch job. However, as SPICE is very computationally intensive, especially in its usage of floating-point numbers and matrix operations, it can consume a large proportion of a multi-user machines processing power, unless it is run at a very low priority.

More recently SPICE has been run on workstations dedicated to supporting a single engineer, but the users machine is again fully occupied whilst a SPICE job is in progress.

The FORTRAN compiler for the transputer allows SPICE to be easily run on a separate processor from the users' other tasks. In particular, the T800 has an on-chip floating point processor which is rated at 1.5 MFLOPS on the 20 MHz part [5]. The transputer concept is discussed in the next section.

¹PC-hosted transputer FORTRAN compiler, version 1.1, Part IMS D713C

14.3 Background on transputers

14.3.1 Transputers

The INMOS transputer consists of a high-performance processor, on-chip RAM, and inter-processor communication links, all on a single chip of silicon. Figure 14.1 shows an example of the transputer family, the IMS T800 transputer.

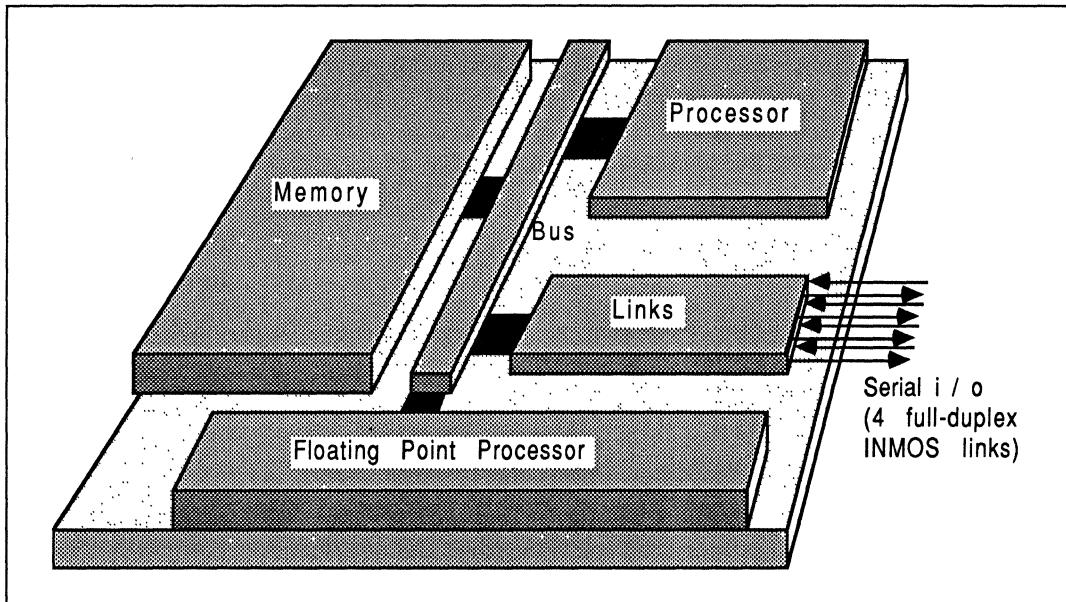


Figure 14.1 Block diagram of an IMS T800 transputer

The IMS T800 integrates a 32-bit micro-processor, a 64-bit floating point unit, four standard 20M bits/sec transputer communications links, 4 Kbytes of on-chip RAM, a memory interface and peripheral interfacing on a single CMOS chip. The floating point unit performs floating point operations concurrently with the CPU, and operates on single and double length (32 bit and 64 bit) items to the ANSI/IEEE 754-1985 floating point arithmetic standard. The concurrent operation allows floating point computation and address calculation to fully overlap, giving a realistically achievable performance of 1.5 MFlops (4 million Whetstones / second) on the 20 MHz part [5].

The on-chip RAM is part of the transputer's address space, and allows critical routines and data to be accessed in a single machine cycle. The on-chip RAM can be thought of as replacing the register set found on conventional micro processors. The inter-processor links are controlled by autonomous DMA engines, and permit any number of transputers to be connected together in arbitrary networks, allowing extra processing power to be injected into a system very easily. The external memory interface allows linear access to a total memory space of 4 gigabytes.

Transputers can be programmed in conventional sequential languages such as C, Pascal, and FORTRAN. The occam language is supported to allow the development of complex parallel programs across multiple transputers. However, sequential sections of code written in C, Pascal or FORTRAN can be included in an occam program.

For further information on the transputer family, the reader is directed to [1].

The T800 is especially relevant in connection with SPICE, because of its floating point performance and the ability to interface to large amounts of external memory.

14.3.2 The transputer / host relationship

The transputer is normally employed as an addition to an existing computer, referred to as the host. Through the host, the transputer application can receive the services of a file store, a screen, and a keyboard. Presently, the host computer can be an IBM PC or compatible, a NEC PC, a DEC MicroVAX II, or a Sun-3. Also available are software tools to allow VAX development for transputer systems. For a more thorough guide to product availability, please refer to [6].

The transputer communicates with the host down a single INMOS link. A program, called a server, executes on the host at the same time as the program on the transputer network is run. All communications between the application running on the transputer and the host services (like screen, keyboard, and filing resources) take the form of messages, which are always initiated by the transputer system.

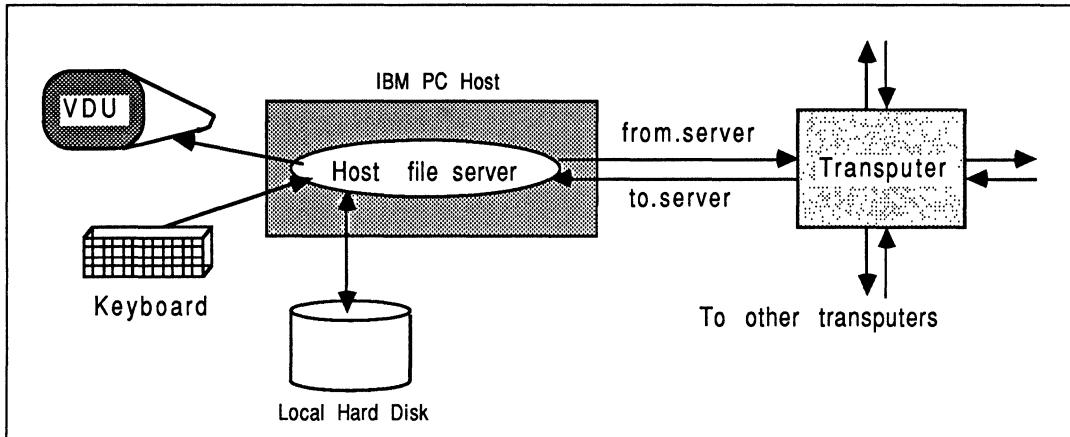


Figure 14.2 The transputer / host relationship

The transputer connected to the host by means of a link adapter is known as the *root transputer*. Figure 14.2 shows the root transputer of a transputer network. All other transputers in the network (if there are any) are connected, using INMOS links, to the root transputer.

14.3.3 SPICE and the transputer

A single process, such as SPICE, is run on a single transputer in the same way that a single process would be run on any other microprocessor. Using the development tools, the single conventional program is actually run as a process within a standard *harness*, which is used to establish the correct workspaces and provide access to the screen, keyboard, and filing facilities on the host. The communication channels defined in the harness are then mapped directly onto the hardware links of the transputer, allowing the application to execute.

14.3.4 Multiple tasks on one or many transputers

Using the tools supplied with the transputer FORTRAN compiler, it is possible to sub-divide a single conventional program into a set of OCCAM [7] processes. The OCCAM multi-process model for transputers is defined by the CSP model of communicating processes [8]. A system can be described in terms of a collection of concurrent processes which communicate with each other and with the outside world. Processes are connected together using synchronized, un-buffered, point-to-point, uni-directional communication channels. An example of this is shown in Figure 14.3, where each circle represents a process, and each arrow represents a communications channel. At this stage, there is no implied or rigidly defined mapping between the software processes and the actual hardware.

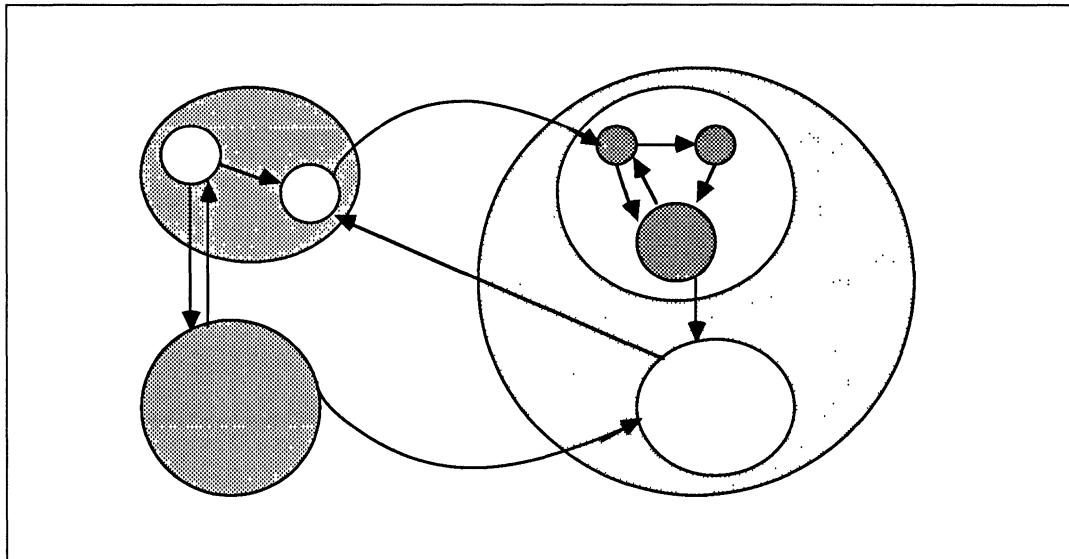


Figure 14.3 OCCAM processes and their communication channels

Some ways in which conventional programs such as SPICE can be distributed across a number of transputers are discussed in later sections of this document. The concepts of CSP and OCCAM are only required to distribute multiple processes onto one or more processors. Note that there is no need to be familiar with OCCAM in order to be able to directly port any conventional program, such as SPICE, to a single transputer.

14.4 The transputer implementation of FORTRAN

SPICE is written in FORTRAN-77. The transputer FORTRAN compiler is based on the ANSI FORTRAN-77 standard, as defined in ANSI X3.9-1978. Extensions to the language have been provided as a transition aid from other FORTRAN dialects. A full description of this compiler can be found in [9].

The T800 transputer has 4 Kbytes of single-cycle on-chip RAM (50ns access time on a 20 MHz part). The on-chip RAM is usually at least four times faster than the external memory provided with most transputer boards. The fastest external memory supported by the transputer is three-cycle, with most boards using four- or five-cycle memory, although using external RAM will *not* make programs run three to five times slower.

The next two sections describe how the FORTRAN compiler allows the on-chip RAM to be used, in terms of stack and code storage. The discussions are appropriate to any of the INMOS scientific language compilers.

14.4.1 Placement of the run-time stack

The user can select to place the run-time stack either in on-chip RAM or in external memory.

If the *whole* of the stack for a program can be accommodated within 2 Kbytes, then it can be placed on-chip on either the T414 or the T800. The general heap storage is then placed in external memory. This is the default assumed by the standard OCCAM harness.

If the size of the stack is expected to be larger than 2 Kbytes, then it must be placed off-chip, and the application is therefore run with all workspace off-chip. This is the manner in which SPICE is run. The parameter `-:o 1`, supplied to the host server at run-time, specifies that all workspace is to go off-chip. Note that no action is required at compile-time to specify the location of the stack. This facility should be used while developing a program, for which one is uncertain of the requirements in terms of stack size.

14.4.2 Placement of the code

The other half of on-chip RAM on the T800 (2 Kbytes) is reserved for code storage. The ordering of the files to link is critical for the performance of the program, because code placement on a processor is determined by the linking order of the binary object files. On the T800, files specified at the beginning of the link operation will be loaded into the 2 Kbytes of on-chip RAM that is not reserved for the variable stack. Programs will therefore run faster if small, speed-critical routines are placed at the beginning of the list of files to be linked, and the OCCAM harness is placed at the end.

It is not possible to have the whole of on-chip memory on the T800 exclusively as a stack or code area. It is also not possible to have part of the stack on-chip and part of it off-chip. This is due to the implementation of the development tools. Note that on the T414, there is no internal RAM available for code storage with the scientific language compilers.

These restrictions on the specification of the scientific-language compilers were adopted for the following reasons. Studies showed that in the event of a trade-off in the use of on-chip memory between code and data, it is generally more efficient to permit some data to be placed on-chip (in the stack) rather than only having application code on-chip. This is due to the high density of transputer machine code, and the transputer's hardware instruction pre-fetch mechanism. Therefore, to provide a development system that could be used on both the T800 and the T414 transputer, both transputers can have their internal RAM used as a variable stack (2 Kbytes in each case), but only the T800 can additionally accommodate some code.

14.4.3 Use of stack space

Besides deciding whether to place the stack on or off-chip, the user can choose to place the local scalar variables of active subroutines on the heap or on the stack. Placing them on the heap guarantees that their values remain unchanged between calls to the same subroutines, but at a cost of a measured 20% performance penalty for the SPICE application.

By default, local scalar variables are placed on the stack. They are placed on the heap by using the /S compiler option. To the authors' knowledge, SPICE does not rely on local variables retaining their values between subroutine calls, so they were placed on the stack.

In addition to local variables, the stack space of FORTRAN programs is also used for SUBROUTINE calls (5 words per level of calling), storing arguments, and run-time library workspace (about 40 words).

14.5 Porting SPICE

This chapter describes how to port SPICE to run on a single T800 transputer. Details of how to run compilers, linkers, and other software tools are not included, as these are readily available in the appropriate software documentation, and in [10].

There are around 130 source routines in SPICE, which fall into four categories with respect to porting to the transputer :

14.5.1 Routines needing no modification

The following routines require no modification for use with the V1.1 transputer FORTRAN compiler:

ALFNUM	ALIAS	AVLM16	AVLM4
AVLM8	CDIV	CLRMEM	CMEYER
CMULT	COMPRS	CRUNCH	DMPMEM
ERRMEM	EVTERM	EXTMEM	FETLIM
GETCJE	GETLIN	GETM16	GETM4
GETM8	GETMX	KEYSRC	LIMVDS
MAGPHS	MEMADJ	MEMORY	MEMPTR
MOSEQ1	MOSEQ2	MOSEQ3	MOSQ2

MOSQ3	MQSPOF	NXTCHR	NXTEVN
NXTMEM	NXTPWR	PNJLIM	PTRMEM
RELMEM	SCALE	SIMPMM	SIZMEM
SHLSRT	UNDEFI	XXOR	

14.5.2 Routines that set the size of VALUE in a COMMON block

In porting SPICE to any new machine it is necessary to choose a value for the size of the COMMON array called VALUE, which is used to manage the dynamic data requirements of the program. On machines that support virtual memory it is usual to set this to a very large size, and leave the memory management system to page the data in and out of memory. However, the transputer does not support virtual memory, but a range of transputer boards are available with different physical memory sizes. It is therefore convenient to be able to easily change the size of VALUE in all the routines that reference it.

This would be most easily achieved by using an include mechanism, but unfortunately version V1.1 of the transputer FORTRAN compiler does not support included files. We used a simple DCL procedure and the EDT editor on the VAX to overcome this problem and to automate these changes in a single batch file, involving a search and replace operation for each of the listed files.

From the point of view of porting SPICE the following line:

```
COMMON /BLANK/ VALUE(.....)
```

must be edited so that the array VALUE has a size suitable for the hardware to be used. For example, for a 2 MByte board, the array would consist of 150000 elements each of 8 bytes, which therefore occupies 1200 Kbytes of store. See section 14.5.5 for guidance in the calculation of appropriate sizes for this array.

Here are the routines requiring this treatment:

ACAN	ACASOL	ACDCMP	ACLOAD
ACSL	ADDELT1	ADDNAM1	ALTER
ASOL	BJT	CARD	CODGEN
COMCOF	CPYTB4	CPYTB8	DCDCMP
DCOP	DCSOL	DCTRAN1	DINIT
DIODE	DISTO	DMPMAT	ELPRNT1
ERRCHK	EVPOLY	EXTNAM	FIND1
FNDNAM	FOURAN	GETNOD1	INDXX
INTGR8	ITER8	JFET	LNKREF
LOAD	MATLOC	MATPTR	MODCHK
MOSCAP	MOSFET	NAMTAB1	NEWNOD1
NLCSRC	NODSTR1	NOISE	NTRPL8
OUTDEF1	OUTNAM1	OVTPVT	PLOT
PUTNOD1	READIN1	REORDR	RESERV
ROOT1	RUNCON1	SENCAL	SETMEM
SETPLT	SETPRN	SETUP	SORSTP
SORUPD1	SSTF	SUBCKT	SUBNAM
SWAPIJ	TERR	TITLE1	TMPUPD
TOPCHK1	TRUNC	UPDATE	

14.5.3 Routines often supplied in assembler

The following routines are often supplied in assembler. Equivalent FORTRAN routines are given in section 14.10, and we have compiled these to obtain the performance information given in this document.

COPY16	COPY4	COPY8
ZERO16	ZERO4	ZERO8
MOVE		

The block move instructions of the transputer could be used to write efficient assembler routine equivalents — refer to section 14.13 for some suggestions.

14.5.4 Other routines to be modified

The following routines require modification, either due to machine dependency, unsupported language extensions, or compiler limitations :

ROOT This is the entry routine to SPICE. It is machine dependent, with routines to handle date and times, file opening and closing etc. Section 14.11 gives the list of changes made to ROOT, as found by VAX DIFFERENCES.

TITLE VAX specific dates use LOGICAL*1, which is a FORTRAN extension not supported by the transputer FORTRAN compiler. This has been removed from TITLE, as shown in section 14.12.

SECOND This routine calls machine specific system routines concerned with timing. We simply return zero, although a full implementation could be provided using the transputer's **TIMER** facility.

LOCF This is the only routine that the authors of SPICE intended to be machine dependent. It returns the physical address of any variable passed to it. On the VAX, LOCF calls a system routine called LOC. We have implemented an equivalent routine to LOC as an in-line transputer assembler insert in a C routine :

```
int loc(a)
int *a;
{
    asm {
        ldl 2; /* return first parameter, which is */
                /* the address of the parameter (a) */
    }
}
```

This is compiled using the transputer C compiler². The source must be compiled separately for the T414 and the T800 transputers to satisfy the processor type requirements imposed by the linker. The compiled code produced is compatible with that from the FORTRAN compiler, and is linked without problem. This is because the INMOS scientific language compiler set permits different parts of the *same* task (process) to be written in different languages, and to interact using the normal mechanisms of procedure / function calling.

It is also possible to write multi-process, mixed-language applications (like the ones described in [10]), in which each process is written in one of the supported scientific languages. These processes then communicate with each other using OCCAM channels.

ERRMEM uses an Octal output specifier which is an extension not supported in transputer FORTRAN V1.1. We changed it to use a Hex output specifier, as it was only used in connection with an error message.

MOSFET exceeded a compiler implementation limit, giving a *Dictionary Table Overflow* error. We overcame this problem by replacing some unused variables at the end of COMMON block declarations with arrays containing the same number of variables. This reduced the number of declared variables to a compilable size. This problem should be fixed on the next release of the compiler.

²PC-hosted transputer C compiler, version 1.3, Part IMS D711C

14.5.5 Calculating the FORTRAN VALUE array size

The FORTRAN VALUE array, mentioned previously, occupies the largest amount of workspace in SPICE. The larger this array, the bigger the simulations that can be run. The size of this array is set to occupy about 300 Kbytes less than the amount of memory available on the board once the SPICE code has been loaded.

For example, an executable SPICE file takes about 500 Kbytes of memory, leaving 1500 Kbytes on a 2 Mbyte board (such as an IMS B004 with T800 [6]). Each element of the VALUE array occupies 8 bytes, so if the array was sized at 150000 elements, this would occupy 1200 Kbytes of store. So, the FORTRAN sources would all have the array dimensioned as 150000:

```
COMMON /BLANK/ VALUE(150000)
```

This is illustrated in Figure 14.4. The T800 internal on-chip RAM and the first 500 Kbytes are used for the SPICE executable code. The VALUE array occupies the next 1200 Kbytes of memory, with some additional FORTRAN workspace shown. The standard OCCAM harness will automatically reserve sufficient workspace for SPICE, but if the user is intent on writing their own harness, then at least 1400 Kbytes of workspace must be reserved for SPICE. If an application has insufficient workspace, it will fail to operate.

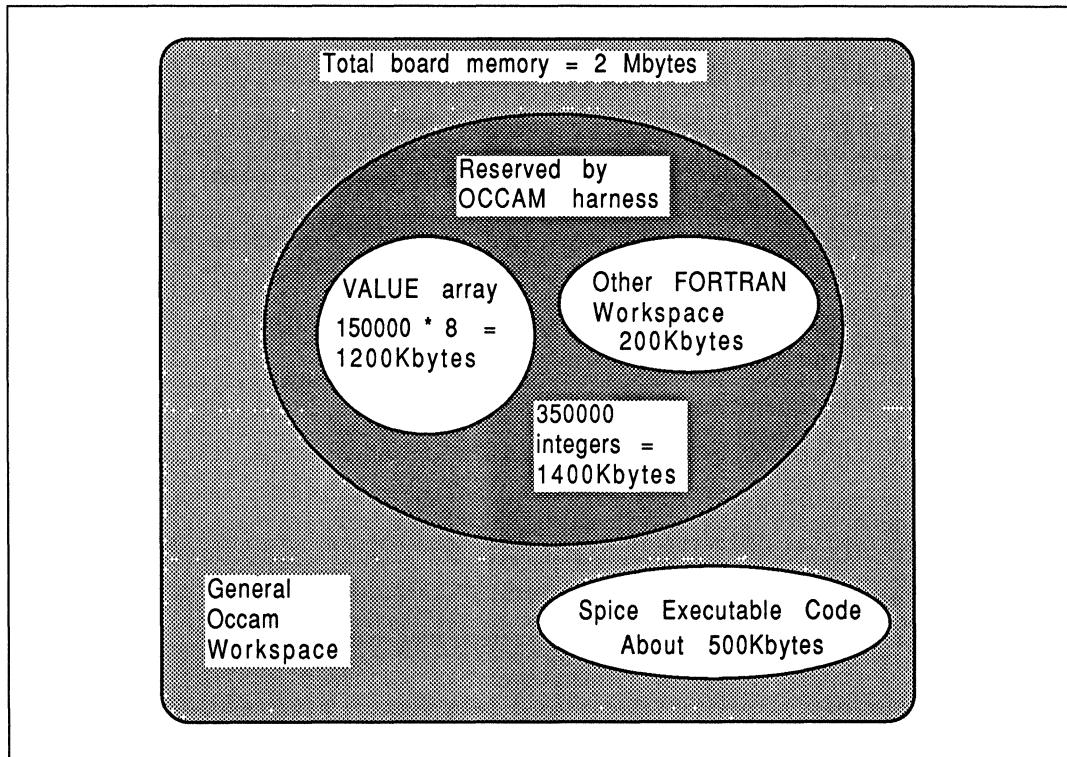


Figure 14.4 SPICE memory usage for a 2 Mbyte board

As another example, on an IMS B405 TRAM board (8 Mbytes of RAM and a T800 transputer [6]), there is 7500 Kbytes of store available after SPICE has loaded. An array of 900000 floating point values occupies 7200 Kbytes of store (8 bytes per element). Therefore, the FORTRAN source code would have the array dimensioned at 900000 :

```
COMMON /BLANK/ VALUE(900000)
```

14.5.6 Problems with long or large simulations

SPICE uses the array VALUE to store all of its simulation data structures, and to store all the simulation output during simulation. This means that large amounts of data can be accumulated during a simulation and this is normally coped with by virtual memory. The requested user output is transferred to the output file at the end of the simulation.

This can give problems for large circuits on machines not supporting virtual memory, where little space is left in which to store simulation results.

The problem can be solved by making changes to the routine DCTRAN, to cause the required voltages and currents to be dumped to a file rather than to the array VALUE. These changes require a detailed understanding of the internal operation of SPICE, and are beyond the scope of this document.

14.6 Performance information

14.6.1 Performance comparisons

The following table gives an indication of the performance of some randomly selected SPICE input decks when run on a variety of different machines. The first benchmark involves a simple resistor network, the second simulates an inverter circuit. The third benchmark represents a clock distribution network, and the fourth is a sense amplifier circuit. Comparisons were made between a Sun-3 (with and without a 68881 numerics co-processor), a VAX 11/785 with FPA³, and the T800 transputer.

The timings were obtained by averaging the time taken for the same job to be run several times on each machine in an attempt to isolate non-computational factors such as fluctuations in speed of disk access, I/O bandwidth, and CPU loading peaks. Note that the output files from all machines were identical. The timings represent the actual CPU time used, and are given in seconds.

Machine	Resist	Invert	Corclk	SenseAmp
Sun-3/160C	0.20	19.40	356.90	1855.50
Sun-3 + 68881	0.30	4.60	44.20	266.70
VAX 11/785 + FPA	0.38	4.51	30.22	141.55
IMS T800	1.48	5.17	23.72	153.64

The transputer timings do *not* include the time taken to boot the transputer with SPICE — they are pure execution times. The boot time for SPICE is around 15 seconds, depending on the host computer, but once booted it can rerun instantly without any re-load penalties.

The high usage of floating point arithmetic in SPICE lends itself much better to the T800 transputer than the T414. The T414 requires almost 75 Kbytes of floating point software support, and SPICE ran about ten times slower than the T800 on the same jobs. Even without a floating point processor, the T414 is still faster than a Sun-3 without a floating point processor.

Note that for extremely small simulations, the simulation time on the transputer is dominated by file transfer times from the host PC.

14.6.2 Additional performance improvements to SPICE on a T800

The performance described in the previous section is typical of that which can be achieved using near-standard INMOS products. However, it is possible to obtain higher performance using the following techniques :

Faster memory and shorter cycle times

The figures quoted above are for a T800-20 transputer with a 4-cycle memory interface. This compares with the IMS B004 evaluation board which is supplied with a T414 and a 5-cycle memory interface. The transputer can support a 3-cycle memory interface, which would reduce the run-times quoted above by an order of 25%.

³FPA – Floating Point Accelerator

Sample T800-25 components are now available, and using these with a shorter memory cycle time would increase performance by a further 25%.

Optimum linkage strategy

To make best use of the existing hardware without modifying the application code, software tools can be used to ensure optimal utilization of the T800's on-chip RAM.

Code placement is determined by the linking order of the binary object files per processor. On the T800, files specified at the beginning of the link operation will be loaded into the 2 Kbytes of on-chip RAM that is not used for the variable stack. Programs will therefore run faster if small, speed-critical routines are placed at the beginning of the link list, and the harness is placed at the end. One can use profiling techniques⁴ to establish the routines which consume most CPU time, and place these in on-chip RAM.

Rewriting critical routines in assembler

As mentioned earlier, the SPICE routines to move, copy, and initialize blocks of memory are often coded in assembler. The T800 has special instructions for performing block operations of this type. By encoding the **move**, **copy** and **zero** subroutine sets into C and then transputer assembler (to ensure the use of specific transputer instructions), a 5% to 10% performance increase was observed. This is shown in more detail in section 14.13

Program profiling can also be used to establish the relative benefits in converting specific routines into C or transputer assembler code.

14.7 Multiple transputer SPICE

The work described so far in this technical note has demonstrated that a single transputer is a powerful processor in its own right. However, the transputer was specifically designed to allow many of them to be used to solve a single problem.

There are two ways in which multiple transputers can be applied to SPICE. The first is to increase the performance of SPICE by modifying the program to run parts of it across multiple transputers. The second is to increase the throughput of a series of SPICE tasks by means of a *processor farm*, in which many copies of SPICE are run simultaneously, each on its own transputer.

14.7.1 Ways of running SPICE on multiple transputers

There are two compute intensive tasks performed by SPICE in a simulation. The first is to set up a matrix to be solved, which involves calculating the current through every element in the circuit from the model equations for the device. The second is to solve the matrix to obtain an improved estimate of the unknown voltages and currents in the circuit. Both of these operations have to be performed many times at each timestep in the simulation, and there are usually thousands of timesteps in each simulation.

For relatively small circuits, of less than around 100 nodes, the model evaluation dominates the simulation time. For larger circuits the matrix evaluation dominates, because the solution of large sparse matrices takes time of order $O(n^{1.2})$ to $O(n^{1.4})$, where n is the number of nodes in the circuit.

The distribution of the solution of matrix equations across multiple processors is currently a subject of much research [11, 12, 13]. It appears that by using the latency present in most large circuits, and by applying relaxation and partitioning techniques, it is possible to achieve significant performance increases by using multiple processors to solve large circuit simulation problems.

In the remainder of this section we discuss a simple technique which can be applied to SPICE to allow many transputers to be used to speed up the evaluation of the device models. This approach has already been experimentally applied to INMOS's in-house circuit simulator.

⁴For example, as in the UNIX environment's PROFIL command.

SPICE maintains a list of the devices whose currents have to be evaluated, as shown in Figure 14.5.

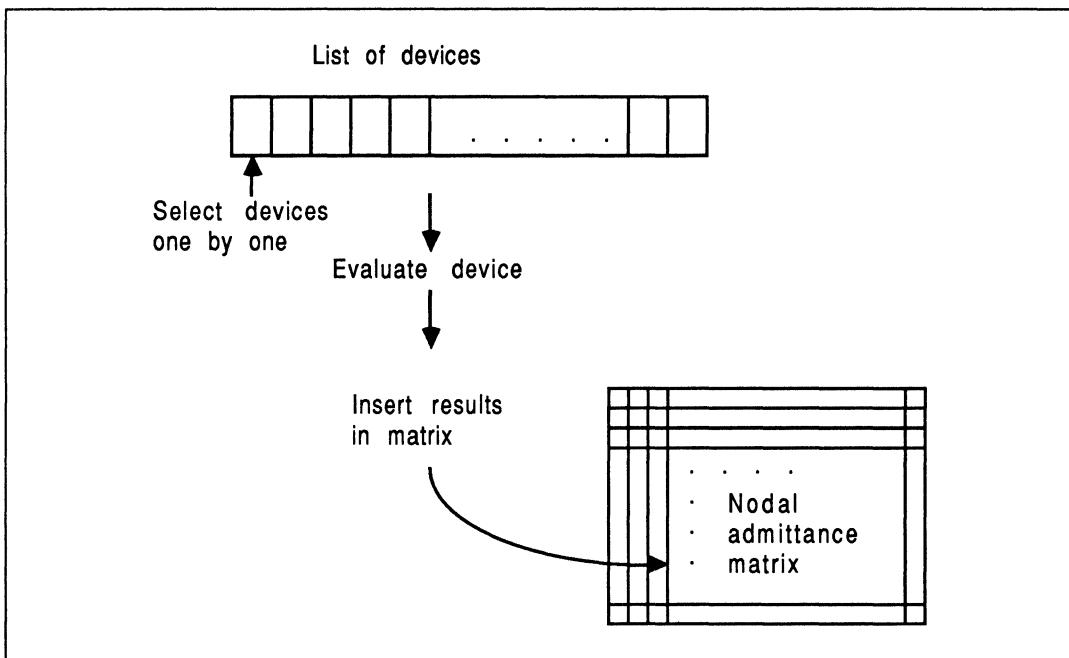


Figure 14.5 SPICE's device evaluation mechanism

For each device its terminal voltages are passed as parameters to a routine which evaluates the current through the device (together with terminal to terminal capacitances and current derivatives with respect to terminal voltages). The calculated values are used to fill in the appropriate locations in the nodal admittance matrix. When this has been completed for all devices, the matrix is solved to obtain the next estimate of the node voltages in the circuit.

From the point of view of the present discussion it is important to notice that all the model evaluations are completely independent of each other. It does not matter in which order they are performed, or even if they are performed in parallel on separate machines !

This fact can be used to distribute the model evaluations across a number of processors. Figure 14.6 shows how the concurrent model evaluations are handled, with respect to data movements in the system, by forming work packets for a farm of model evaluators. This figure can be usefully compared with Figure 14.5 shown previously.

In hardware terms, one method of implementing this mechanism on a number of transputers is shown in Figure 14.7.

The core of SPICE is run on a single transputer, and the model evaluation routines are placed on a number of other transputers. SPICE is modified so that for each device to be evaluated it sends a message to the farm of model evaluation processors, specifying which device is to be evaluated and what its terminal voltages are. A free processor will accept the message, evaluate the model equations, and pass back a message containing the required results to the core transputer.

However, the core transputer is programmed so that it does not wait for this message to return, but continues to transmit requests for other devices to be evaluated. When a results message arrives back at the core processor it inserts the results in the appropriate location in the nodal admittance matrix as before. Again, once all devices have been evaluated the single central transputer must solve the matrix, and the model evaluation processors will be idle during this period.

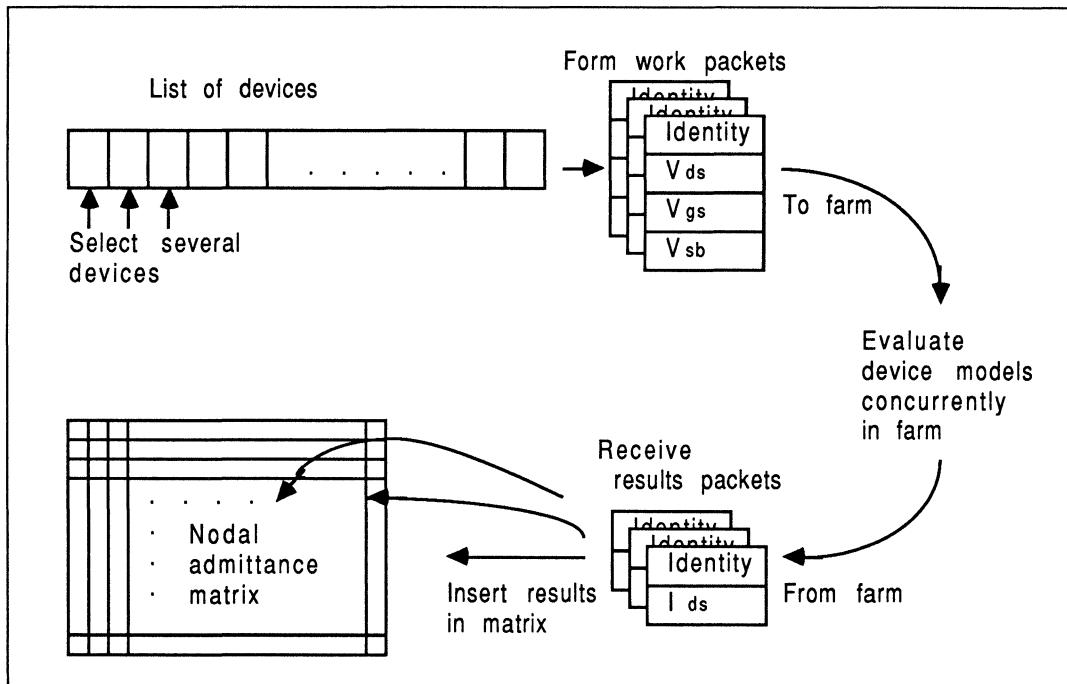


Figure 14.6 Data movements required to distribute the model evaluations

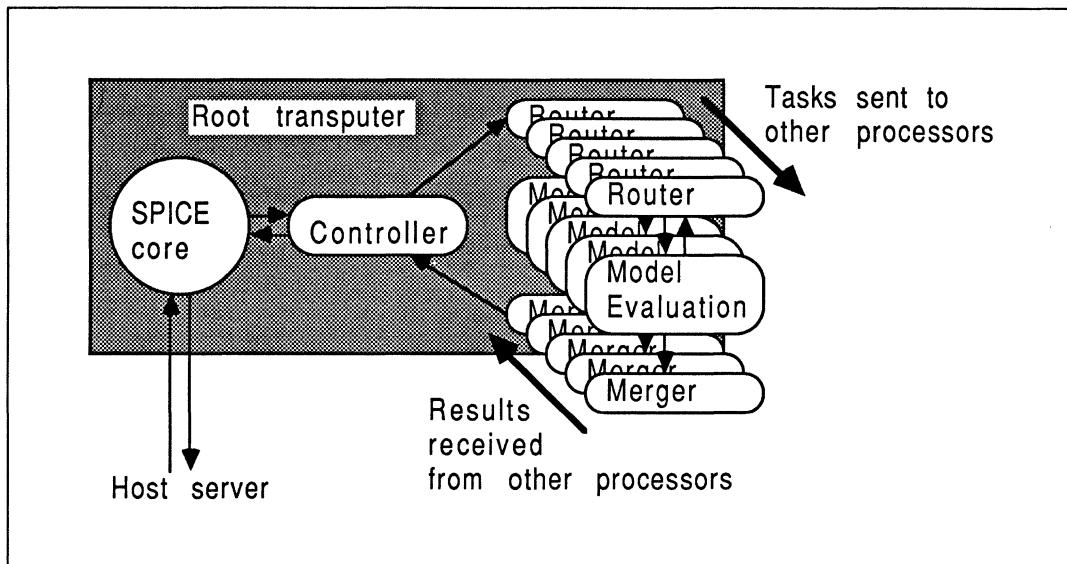


Figure 14.7 One way of farming the model evaluations

For small circuits of less than 100 nodes the model evaluations typically take 75% of the simulation time [14]. Therefore, halving the time taken for model evaluations by adding two extra transputers, should theoretically

make the simulation run 38% faster. However, in practice another copy of the model evaluation process would be run on the core transputer, which would, at best, allow the model evaluation time to be reduced by a factor of three. Therefore we would expect to roughly halve the simulation time by using three transputers instead of one. From a circuit designers point of view this is a very worthwhile improvement.

Note that using an infinite number of transputers just to solve the model equations can at best only cut the simulation time by a factor of 4, as all the remainder to the simulation is still running on the single core machine. This illustrates that it is important to identify those sections of the code where parallelism is available, and to concentrate on applying the correct number of processors to exploiting this potential parallelism, without getting to the point that the remaining sequential sections of code completely dominate the CPU time.

14.7.2 A multiple SPICE farm

Farming is a technique which can be applied to almost all existing applications, where the same program is run on a *farm* of processors, each one working on an independent set of data.

Some additional processes, usually written in Occam are used to control and regulate the distribution of work within the farm. The techniques of farming are explained more fully elsewhere [15, 16, 17, 18, 2], though the general composition of a processor farm is shown below in figure 14.8.

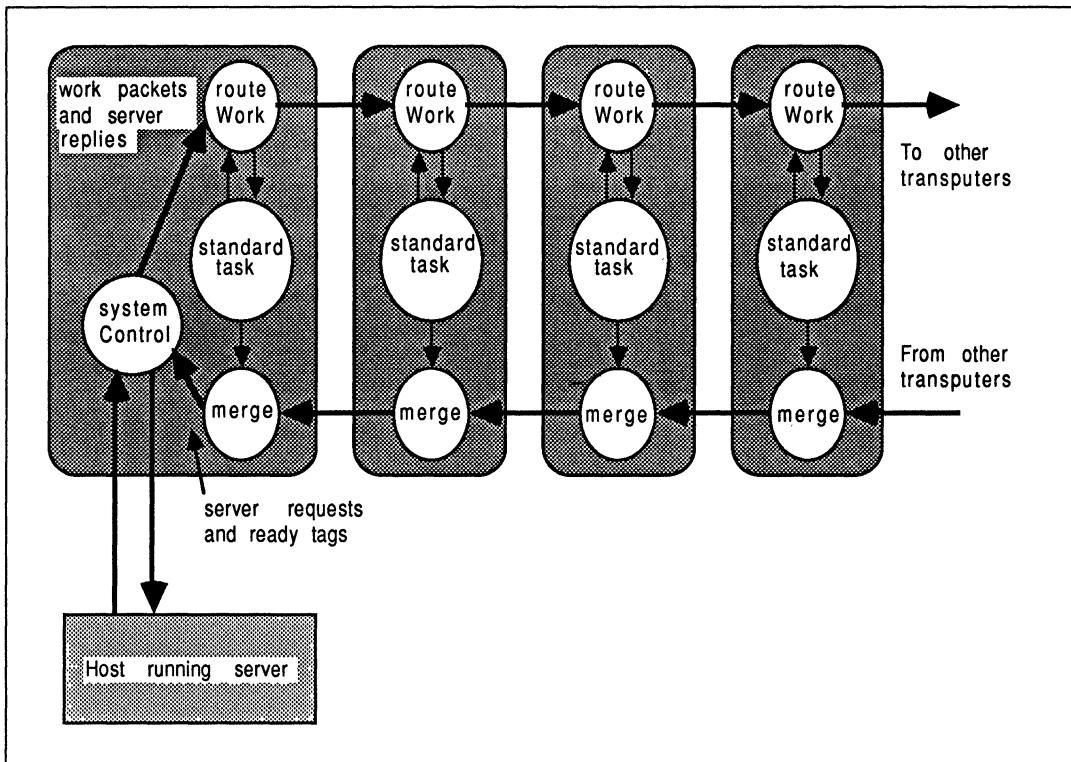


Figure 14.8 A general pipeline processor farm

Concurrent processes handle the tasks of work distribution, results collection, and performing the program itself. A system control process on the root transputer acts as an overall manager. Referring to Figure 14.8, the farm is controlled by the **systemControl** process, work is routed into the farm by the **routeWork** process, and results are collected by the **merge** process. In this case, the SPICE application is inserted in the place of the **standardTask** process.

Although the diagram shows a pipeline farm, the farm can have any connected topology. A pipeline is particularly easy to implement on the INMOS TRAM motherboards [6], as a suitable pipe is hard-wired into the motherboard, and requires no additional hardware or software configuration.

The list of tasks to be executed by the farm is made available to the `systemControl` process, and the transputers are fed with tasks until there are no more left. We used a small file on the host computer to specify the tasks to be performed. Although this approach operates several identical concurrent SPICE applications, the time taken by any given SPICE job is not reduced.

It would be possible to combine the techniques of this and the previous section, to arrive at a system where each farm 'worker' was itself a number of transputers. In this case, each farm worker still executes identical code to its' neighbours, but this code is distributed over a number of transputers.

14.7.3 A networked SPICE farm example

As an example of a SPICE farm, we have constructed a tree-like farm to relieve VAX CPU overhead, by offloading SPICE jobs from a VAX to a PC-hosted server which returns SPICE output files to the VAX. The server was written to communicate via DECnet, allowing the farm to become a remote processing engine for a VAX network. This is shown in Figure 14.9.

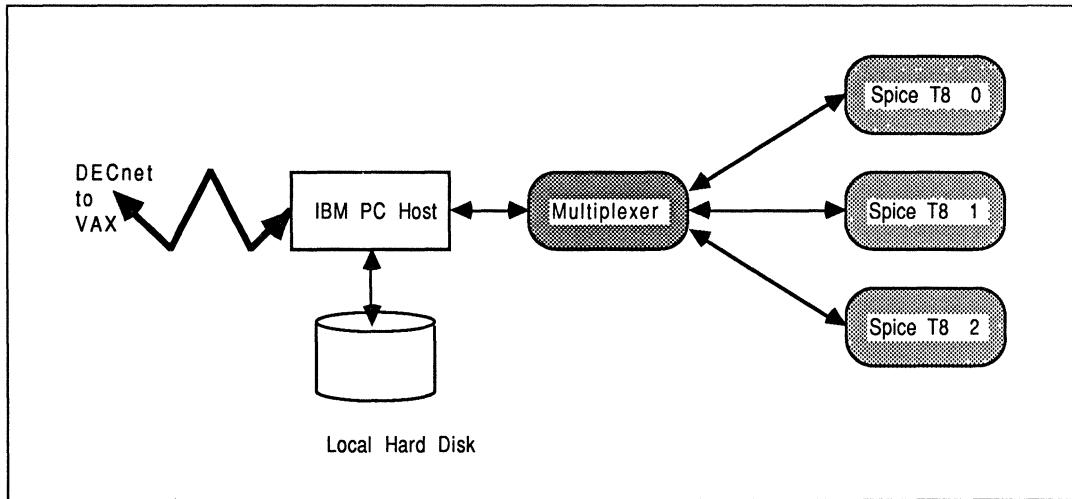


Figure 14.9 A four processor SPICE system

In this implementation, the PC server program was used to control the farm, receiving incoming task requests over DECnet. An occam multiplexer was used to correctly interleave SPICE accesses to the host facilities. Although a tree structure rather than a pipeline was used, the farming principle is the similar to that discussed in the previous section. A full discussion of our implementation of this DECnet-hosted farm server, as shown in Figure 14.9, can be found in [19].

14.8 Summary

This technical note has demonstrated that existing programs can be easily ported to run on the transputer. Very little modification was required to SPICE, which is a large and demanding application, to allow it to compile and run on the transputer.

The floating point performance of the IMS T800 allowed a pure FORTRAN version of SPICE to run more than one and a half times faster than on a Sun-3 with 68881 coprocessor, and nearly as fast as a version with assembler code routines running on a VAX 11/785 with floating point accelerator. By coding the same

memory management routines in transputer assembler as those coded in assembler on the VAX, we obtained a performance equal to that of the VAX 11/785.

We have discussed how a single conventional program can be distributed over a number of processors, and illustrated some of the techniques that can be used to make use of potential parallelism in an application. We have also shown how a farm of transputers can be used as a cost effective way of offloading CPU intensive tasks from mainframe and mini-computers.

14.9 References

- 1 *The Transputer Databook*, INMOS Limited, 1989.
- 2 *Issues in Application porting and farming*, Technical Note 53, INMOS Limited.
- 3 *SPICE2: A computer program to simulate semiconductor circuits*, L. W. Nagel, Memorandum No. ERL-M520, University of California, Berkeley. May, 1975
- 4 *Program Reference for SPICE2*, E. Cohen, Memorandum No. ERL-M592, University of California, Berkeley. June, 1976
- 5 *Lies, damned lies, and benchmarks*, Technical Note 27, INMOS Limited.
- 6 *INMOS Spectrum*, (Contains a brief description of INMOS products) INMOS Limited.
- 7 *occam 2 Reference Manual*, INMOS Limited, Prentice Hall.
- 8 *Communicating Sequential Processes*, C. A. R. Hoare, Prentice Hall 1985.
- 9 *3L FORTRAN Reference Manual* (compiler version 1.1), INMOS Limited
- 10 *Using the occam toolset*, Technical Note 55, INMOS Limited.
- 11 *Relaxation techniques for the simulation of VLSI circuits*, J. K. White and A. Sangiovanni-Vincentelli, Kluwer Academic Publishers, 1987
- 12 *A Pipelined Event-driven Mixed-mode Simulator*, Clive M. Dyson and Alan H. Gray, IEEE Int. Conf on Computer-Aided Design, pp 488-491 Santa Clara, California, 1987
- 13 *CINNAMON : Coupled integration and nodal analysis of MOS networks*, L. M. Vidigal, S. R. Nassif and S. W. Director, 23rd Design Automation Conference, pp 179-198, 1986
- 14 *The Simulation of Large-scale Integrated Circuits*, A. Richard Newton, Memorandum No. ERL-M78-52, University of California, Berkeley. July, 1978
- 15 *Exploiting concurrency; A Ray tracing Example*, Technical Note 7, INMOS Limited.
- 16 *Program design for concurrent systems*, Technical Note 5, INMOS Limited.
- 17 *Performance Maximization*, Technical Note 17, INMOS Limited.
- 18 *Communicating Process Computers*, Technical Note 22, INMOS Limited.
- 19 *A transputer farm accelerator for networked computing facilities*, Technical Note 54, INMOS Limited.
- 20 *Transputer instruction set: a compiler-writer's guide*, INMOS Limited, Prentice Hall

14.10 Routines for copy, zero and move

This section lists the FORTRAN equivalents for the copy, zero, and move routines.

```

SUBROUTINE COPY4(IFROM, ITO, NWORDS)
IMPLICIT DOUBLE PRECISION (A-H,O-Z)
C
DIMENSION IFROM(1), ITO(1)
C THIS ROUTINE COPIES A BLOCK OF #NWORDS# WORDS (OF THE APPROPRIATE
C TYPE) FROM THE ARRAY #FROM# TO THE ARRAY #TO#. IT DETERMINES FROM
C WHICH END OF THE BLOCK TO TRANSFER FIRST, TO PREVENT OVER-STORES WHICH
C MIGHT OVER-WRITE THE DATA.
C
IF (NWORDS.EQ.0) RETURN
IF (LOCF(IFROM(1)).LT.LOCF(ITO(1))) GO TO 20
C... LOC() RETURNS AS ITS VALUE THE ADDRESS OF ITS ARGUMENT
DO 10 I=1,NWORDS
ITO(I)=IFROM(I)
10 CONTINUE
RETURN
C
20 CONTINUE
DO 40 I=NWORDS,1,-1
ITO(I)=IFROM(I)
40 CONTINUE
RETURN
END

SUBROUTINE COPY8(RFROM, RTO, NWORDS)
IMPLICIT DOUBLE PRECISION (A-H,O-Z)
C
DIMENSION RFROM(1), RTO(1)
IF (NWORDS.EQ.0) RETURN
IF (LOCF(RFROM(1)).LT.LOCF(RTO(1))) GO TO 120
DO 110 I=1,NWORDS
RTO(I)=RFROM(I)
110 CONTINUE
RETURN
C
120 CONTINUE
DO 140 I=NWORDS,1,-1
RTO(I)=RFROM(I)
140 CONTINUE
RETURN
END

SUBROUTINE COPY16(CFROM, CTO, NWORDS)
IMPLICIT DOUBLE PRECISION (A-H,O-Z)
C
COMPLEX CFROM(1), CTO(1)
IF (NWORDS.EQ.0) RETURN
IF (LOCF(CFROM(1)).LT.LOCF(CTO(1))) GO TO 220
DO 210 I=1,NWORDS
CTO(I)=CFROM(I)
210 CONTINUE
RETURN
C
220 CONTINUE
DO 240 I=NWORDS,1,-1
CTO(I)=CFROM(I)
240 CONTINUE
RETURN
END

SUBROUTINE ZERO4(IARRAY, LENGTH)
IMPLICIT DOUBLE PRECISION (A-H,O-Z)

```

```
C      DIMENSION IARRAY(1)
C      THIS ROUTINE ZEROES THE MEMORY LOCATIONS INDICATED BY ARRAY(1)
C      THROUGH ARRAY(LENGTH) .
C
C      IF (LENGTH.EQ.0) RETURN
C      DO 10 I=1, LENGTH
C          IARRAY(I)=0
C
10   CONTINUE
      RETURN
      END

      SUBROUTINE ZERO8(ARRAY, LENGTH)
      IMPLICIT DOUBLE PRECISION (A-H, O-Z)
C
C      DIMENSION ARRAY(1)
C      THIS ROUTINE ZEROES THE MEMORY LOCATIONS INDICATED BY ARRAY(1)
C      THROUGH ARRAY(LENGTH) .
C
C      IF (LENGTH.EQ.0) RETURN
C      DO 10 I=1, LENGTH
C          ARRAY(I)=0.0D0
C
10   CONTINUE
      RETURN
      END

      SUBROUTINE ZERO16(CARRAY, LENGTH)
      IMPLICIT DOUBLE PRECISION (A-H, O-Z)
      COMPLEX CARRAY(1)

C
C      THIS ROUTINE ZEROES THE MEMORY LOCATIONS INDICATED BY ARRAY(1)
C      THROUGH ARRAY(LENGTH) .
C
C      IF (LENGTH.EQ.0) RETURN
C      DO 10 I=1, LENGTH
C          CARRAY(I)=CMPLX(0.0E0, 0.0E0)
C
10   CONTINUE
      RETURN
      END

      SUBROUTINE MOVE(A, I, B, J, N)
      CHARACTER*1 A(1), B(1)

C
C      THIS ROUTINE MOVES N CHARACTERS FROM CHARACTER ARRAY B TO CHARAC-
C      TER ARRAY A, BEGINNING WITH THE J*TH AND I*TH CHARACTER POSITIONS,
C      RESPECTIVELY.
C
C      IF (N.EQ.0) RETURN
C      DO 10 K=1, N
C          A(I+K-1)=B(J+K-1)
C
10   CONTINUE
      RETURN
      END
```

14.11 Changes to ROOT found by VAX DIFFERENCES

This section is concerned with changes made to ROOT to allow compilation with the V1.1 transputer FORTRAN compiler. The changes mostly concern the VAX-specific handling of files, dates, and times. The first part of each result shows the source as used on the VAX, and the second part shows the source as used on the transputer.

```
*****
File [.VAX]ROOT.FOR;2
157      COMMON /VMSDAT/ BDATE
158      LOGICAL*1 BDATE(9)
159      CHARACTER*64 FILNAM
160      C
File [.TX]ROOT.FOR;2
158      C      COMMON /VMSDAT/ BDATE
159      C      LOGICAL*1 BDATE(9)
160      C      CHARACTER*64 FILNAM
161      C
*****
File [.VAX]ROOT.FOR;2
169      C      ****
File [.TX]ROOT.FOR;2
170      DATA AHDRCMD / 8H00-00-00 /
171      C      ****
*****
File [.VAX]ROOT.FOR;2
185      CALL TODALF(ATIME)
186      CALL DATE(BDATE)
187      BOLTZ=1.3806226D-23
File [.TX]ROOT.FOR;2
187      C      CALL TODALF(ATIME)
188      ATIME=AHDRCMD
189      C      CALL DATE(BDATE)
190      BOLTZ=1.3806226D-23
*****
File [.VAX]ROOT.FOR;2
202      TYPE 1
203      1 FORMAT(' INPUT FILE: '$)
204      ACCEPT 2,FILNAM
205      2 FORMAT(A)
206      OPEN (UNIT=5,NAME=FILNAM,TYPE='OLD')
207      TYPE 3
208      3 FORMAT(' OUTPUT FILE: '$)
209      ACCEPT 2,FILNAM
210      OPEN (UNIT=6,NAME=FILNAM,TYPE='NEW')
211      C
File [.TX]ROOT.FOR;2
205      C      TYPE 1
206      C      1 FORMAT(' INPUT FILE: '$)
207      C      ACCEPT 2,FILNAM
208      C      2 FORMAT(A)
209      C      OPEN (UNIT=5,NAME=FILNAM,TYPE='OLD')
210      C      TYPE 3
211      C      3 FORMAT(' OUTPUT FILE: '$)
212      C      ACCEPT 2,FILNAM
213      C      OPEN (UNIT=6,NAME=FILNAM,TYPE='NEW')
214      OPEN (UNIT=5,FILE='SPICE.IN',STATUS='OLD')
215      OPEN (UNIT=6,FILE='SPICE.OUT',STATUS='NEW')
216      C
*****
File [.VAX]ROOT.FOR;2
215      CALL TIMRB
216      CALL GETCJE
File [.TX]ROOT.FOR;2
220      C      CALL TIMRB !CMD
221      CALL GETCJE
```

```
*****
File [.VAX]ROOT.FOR;2
 354      CALL TIMRE
 355      ET=TIME2-TIME1
File [.TX]ROOT.FOR;2
 359      C      CALL TIMRE !CMD
 360      ET=TIME2-TIME1
*****
File [.VAX]ROOT.FOR;2
 368      900 IF ((MAXTIM-ITIME).GE.LIMTIM) GO TO 10
 369      WRITE (IOFILE,901)
 370      901 FORMAT('1WARNING: FURTHER ANALYSIS STOPPED DUE TO CPU TIME LIMIT'
 371      1/)
 372      1000 IF(NODATA.NE.0) WRITE(IOFILE,1001)
File [.TX]ROOT.FOR;2
 373      C 900 IF ((MAXTIM-ITIME).GE.LIMTIM) GO TO 10
 374      C      WRITE (IOFILE,901)
 375      C 901 FORMAT('1WARNING: FURTHER ANALYSIS STOPPED DUE TO CPU TIME LIMIT'
 376      C 1/)
 377      1000 IF(NODATA.NE.0) WRITE(IOFILE,1001)
*****
```

14.12 Changes to TITLE found by VAX DIFFERENCES

This section is concerned with changes made to the TITLE routine to permit compilation with the V1.1 transputer FORTRAN compiler. The changes mostly concern the VAX-specific handling of dates and times. The first part of each result shows the source as used on the VAX, and the second part shows the source as used on the transputer.

```
*****
File [.VAX]TITLE.FOR;2
 33      COMMON /VMSDAT/ BDATE
 34      LOGICAL*1 BDATE(9)
 35      C
File [.TX]TITLE.FOR;2
 33      C      COMMON /VMSDAT/ BDATE
 34      C      LOGICAL*1 BDATE(9)
 35      C
*****
File [.VAX]TITLE.FOR;2
 47      WRITE (IOFILE,31) BDATE,APROG,ATIME,(ATITLE(I),I=1,10)
 48      31 FORMAT(1H1,15(1H*),9A1,1X,23(1H*),3A8,23(1H*),A8,15(1H*)//1H0,
 49      1 15A8/)
File [.TX]TITLE.FOR;2
 47      C      WRITE (IOFILE,31) BDATE,APROG,ATIME,(ATITLE(I),I=1,10)
 48      WRITE (IOFILE,31) APROG,ATIME,(ATITLE(I),I=1,10)
 49      C 31 FORMAT(1H1,15(1H*),9A1,1X,23(1H*),3A8,23(1H*),A8,15(1H*)//1H0,
 50      31 FORMAT(1H1,15(1H*),10(1H*),23(1H*),3A8,23(1H*),A8,15(1H*)//1H0,
 51      1 15A8/)
*****
File [.VAX]TITLE.FOR;2
 58      100 WRITE (IOFILE,101) BDATE,APROG,ATIME,(ATITLE(I),I=1,10)
 59      101 FORMAT(1H1,7(1H*),9A1,1X,7(1H*),3A8,7(1H*),A8,5(1H*)//1H0,10A8/)
 60      IF (ICOM.EQ.0) GO TO 110
File [.TX]TITLE.FOR;2
 60      C 100 WRITE (IOFILE,101) BDATE,APROG,ATIME,(ATITLE(I),I=1,10)
 61      100 WRITE (IOFILE,101) APROG,ATIME,(ATITLE(I),I=1,10)
 62      C 101 FORMAT(1H1,7(1H*),9A1,1X,7(1H*),3A8,7(1H*),A8,5(1H*)//1H0,10A8/)
 63      101 FORMAT(1H1,7(1H*),10(1H*),7(1H*),3A8,7(1H*),A8,5(1H*)//1H0,10A8/)
 64      IF (ICOM.EQ.0) GO TO 110
*****
```

14.13 Rewriting routines in transputer assembler

As an example of re-writing a SPICE FORTRAN subroutine in transputer assembler, consider the **move** subroutine.

The FORTRAN for this routine has already been shown in section 14.10. Since the transputer FORTRAN compiler does not permit the inclusion of transputer assembler mnemonics, the first stage is to code and test an equivalent routine written in C. This is shown below :

```
int move (a, i, b, j, n)
char *a, *b;
int *i, *j, *n;
{
    int k;

    if (*n == 0) return;
    for ( k=-1; k < (*n)-1; k++)
        a[(*i)+k] = b[(*j)+k];
}
```

A couple of points need explaining here. Firstly, the parameter passing mechanism implemented in the transputer FORTRAN compiler is to call by reference. Secondly, arrays in FORTRAN (in SPICE) generally start from subscript 1, and those in C start from subscript 0. This accounts for the start and finish values of the index variable **k** shown above.

The V1.3 transputer C compiler allows *limited* transputer assembler inserts, using the **asm** directive. So, the loop construct in the C representation is replaced by an explicit transputer assembler instruction, the **move** instruction. One way of doing this is shown below :

```
int move (a, i, b, j, n)
char *a, *b;
int *i, *j, *n;
{
    int source, dest, len;

    source = b + (*j) - 1;
    dest   = a + (*i) - 1;
    len    = *n;

    asm {
        ldl 0 ; /* source */
        ldl 1 ; /* dest */
        ldl 2 ; /* len */
        move ;
    }
}
```

The **move** instruction is more fully described in [20], but briefly it takes a source, destination, and byte count, and performs a fast memory copy operation. The arguments are easily set up in C, and there is little performance penalty as this is only done once. The C compiler allocates local integer automatic variables in the order they are declared, starting from workspace location 0. Therefore, the instruction **ldl 0** will access the data held in **source**, which is the address of the vector **b**.

By implementing the seven **move**, **copy**, and **zero** routines in C, a 5% to 7% performance increase over the FORTRAN equivalents were observed. This can partly be explained by remembering that the FORTRAN routines had to call C functions to obtain the addresses of the vectors being operated on — this overhead is not incurred here. By implementing the **move** function in the assembler shown above, another 2% increase was observed.

This technique can also be used in other areas.

15 A transputer farm accelerator for networked computing facilities

15.1 Introduction

This technical note describes the use of INMOS transputers as end-application accelerators to a larger computing resource. As well as describing a specific implementation, some general ideas and arguments are discussed.

15.1.1 A modern trend

In contrast to the prominence of centralized computing facilities traditionally associated with large companies and institutions, nowadays the trend is towards desktop personal computers and networked diskless-node workstations.

This trend has been brought about by the decreasing cost and increasing performance of personal computers and networking options, which are being offered by a growing number of manufacturers.

By having desk-top processing power at one's finger tips, users have the response, flexibility and the control they want over the software they use. In addition, they are not as dependent on the loading and reliability (or otherwise) of a centralized machine and its communications network.

As an example of this, it is not at all uncommon to see a centralized cluster of VAX machines, spanned by networks, with various satellite MicroVAXes and DECnet-DOS personal computers. This network is frequently composed of several sub-nets, spanning the geographical distances between the sites of a company. A fairly typical company network is shown below in figure 15.1. In the figure, the geographical dispersion of the computers may extend across several buildings, towns, or countries.

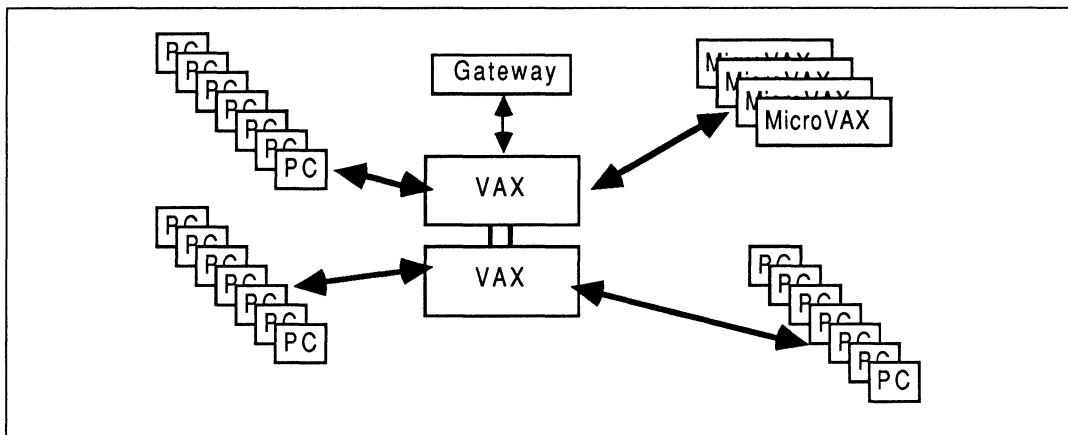


Figure 15.1 A typical computer network

15.1.2 Resolving the loading problem

Despite the proliferation of networked workstations and localized personal-computer processing power, the centralized resources are often overloaded with requests for large amounts of compute-power. Many tasks are just too large for the workstations and personal computers to handle in a reasonable timescale. It only takes a few such compilations and simulations to be concurrently executing on most machines to bring them to a virtual halt. How can this be avoided?

One approach would be to use an organization's existing infrastructure of networks and remote workstations to offload work from the centralized computers and MicroVAXes. This could be achieved by having, at any physical location on a connected network, a PC- or MicroVAX-hosted server, connected to a transputer *farm*

capable of extensive number crunching. In the remainder of this document, the word 'farm' will be used to signify a collection of transputers, all executing the same application but on different data sets.

Such a system could be totally transparent to users of the services that most incapacitate the computing resources. They would invoke the application in exactly the normal way, except that the work would be performed remotely, by transputers, and the results would be returned shortly afterwards. Transputers can offer a previously impossible amount of compute-power in a small box.

A PC-hosted transputer server system can run *existing* applications, *unmodified*, and reduce loading from overworked machines, in a manner that is attractive because of its flexible and infinite expandability. Furthermore, once the application has been ported to a transputer, it is independent of any of the other computer or communications equipment owned.

If you're still interested, read on

15.2 The systems involved

This chapter will discuss the hardware and software systems involved in the implementation to be described, with a view to placing the requirements and demands made of everything in some sort of perspective. The discussion focusses on specific systems, although the arguments are appropriate in a more general sense too.

The items that have to be discussed are the INMOS transputer, the transputer host, the centralized computing resource, and the communications network. The arrangement is as shown below in figure 15.2.

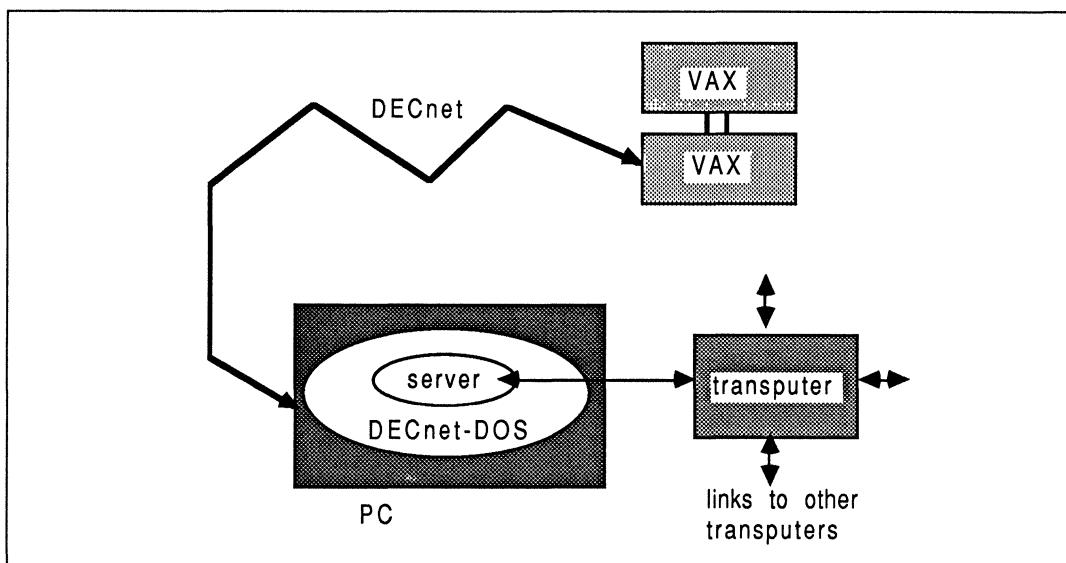


Figure 15.2 From the VAX to the transputer

15.2.1 The INMOS transputer

The INMOS transputer is a high performance micro-processor, offering a CPU, RAM, fast serial links, and various applications-specific facilities on a single chip of silicon. The IMS T800 transputer combines a 32-bit 10 MIPS integer CPU, a 1.5 MFlops 64-bit floating point processor (compliant to the ANSI/IEEE 754-1985 floating point arithmetic standard [1]), four 20 MHz serial links, and four kilobytes of fast single-cycle RAM (50ns access on the 20 MHz part). This technical note will make particular reference to the IMS T800, since the applications described make good use of the in-built floating point processor.

For a proper technical description of the INMOS transputer family, the reader is directed towards [2].

15.2.2 The transputer host

The transputer is normally employed as an addition to an existing computer, known as the host. In the context of this discussion, the host is a personal computer – an IBM PC or compatible. Through the host, the transputer application can receive the services of a file store, a screen, and keyboard.

The selection of the transputer host is important for two main reasons: firstly, it has to be able to accommodate and communicate with transputer hardware; secondly, it has to be capable of appearing as a node on the network to the centralized computing resource. It should also possess some local file store such as a 20 Mbyte Winchester, although our implementation used a virtual disk on the VAX.

The IBM PC and most of its clones fulfil the main requirements.

15.2.3 The existing computing resource

This will typically consist of a centralized or distributed cluster of large mini-computers or main-frame computers, interconnected by one or more networks. The machines may all be of different type, manufacture, and specification.

In this technical note, the existing computing resources are represented by a pair of DEC VAX 11/785 mini-computers, a number of DEC MicroVAX II's, and an Ethernet LAN (Local Area Network). The network involved is in fact more extensive than suggested here, but this document discusses the only relevant part of it.

15.2.4 The communications network

This section gives an overview of the network. In hardware terms, the relevant part of our network was an Ethernet network. An Ethernet network is a finite capacity shared-channel LAN. Sites on the network, called *nodes*, are connected by using vampire taps on a single co-ax cable.

In software terms, we used DECnet software to control the network activity.

DECnet Introduction

The term DECnet refers to a range of software that provides a network interface for Digital Equipment Corporation operating systems. A set of standards called the Digital Network Architecture defines the relationships between the various network components.

DECnet allows multiple computer systems to communicate and share resources within a network. Each computer system, called a node, is connected by some physical communications medium. Tasks that run on different nodes and exchange data are connected by *logical links*. Logical links are temporary software information paths established between two communicating tasks in a DECnet network.

DECnet-DOS is installed on a PC node in the network, on top of the existing MS-DOS operating system. It is said to be a non-routing implementation of the Phase IV Digital Network Architecture.

DECnet concepts

A *client task* is the program that initiates a connect request with another task. The *server task* waits for and accepts/rejects the pending connect request. Client and server¹ tasks communicate through *sockets*. These tasks exchange data over logical links.

Sockets are the basic building blocks for DECnet-DOS task-to-task communication, and are created by tasks for sending and receiving data. They contain information about the status of the logical link connection.

Each system in a DECnet network has a unique node name and address. When initiating a connect request

¹The word server is used here in a different context from the rest of this document

with a remote node, the node is identified by its name or address.

DECnet-DOS allows C and assembly-language programs to use sockets to perform DECnet functions. This allows a user application to communicate with another application running on a different node, using DECnet.

Refer to the DECnet-DOS programmer's reference manual [3] for further information.

15.2.5 How everything fits together

On local storage (or virtual disk) media, the PC will have copies of all the application code that may be required (suitably prepared for execution on the transputer). The relevant piece of application code is initially booted to the transputer network using a special PC file server / loader. The special server monitors DECnet instead of the PC's keyboard. This means that it can accept tasks over the network from the VAX automatically, and act as a completely un-attended autonomous system.

When the user wishes to send a task specification, the VAX software establishes a logical link by means of a connect request procedure. The logical link allows the exchange of data between the VAX and the PC server, because the PC has a unique node name and address on the network. The input data is forwarded to the transputer by the server, and the results are collected afterwards and returned to the originator of the task request.

Depending on the transputer hardware available in the system, several concurrent tasks can be underway at once. It is possible for all these tasks to be different applications entirely. It's up to you, and is easily altered to suit the demands made on the transputer workers.

15.3 A specific implementation

This chapter describes a specific implementation of a DECnet hosted transputer server, undertaken at INMOS in Bristol. The system is extendible, transparent to the user, and involved little change to the application code. It is also simple to maintain from both a hardware *and* a software point of view.

Firstly, an overview of the system is given, then some design issues are discussed. Then, the PC, transputer, and VAX support detail is explored, followed by an outline of how the system was operated.

15.3.1 Overview

INMOS Corporation (Colorado) used a version of SPICE, written in FORTRAN, for intensive circuit simulation on a VAX. To relieve VAX CPU overhead, it was planned to offload SPICE jobs to a server which would simply return the results to the VAX. The system is referred to as MultiSPICE.

The implementation consisted of three concurrently executing copies of SPICE each running on their own T800 transputer, shown in figure 15.3.

This level of parallelism granularity is at the job-level, in that there was to be no attempt to alter the code of SPICE itself to explicitly run parts of the application in parallel. Speedup can be achieved by means of adding more processors, each running another completely independent version of the application code. The transputer network was hosted by a PC, from which it obtained local filing facilities. For this application, each SPICE worker packs about the same computation power as a VAX 11/785 with FPA support.

Each SPICE worker ran on a B405 TRAM module, offering an IMS T800-20 transputer with 8 Mbytes of RAM. For smaller simulations, the B404 TRAMs, which have 2 Mbytes of RAM with an IMS T800 transputer, were used. The connection to the IBM PC was by means of a single IMS B004 evaluation board with an IMS T414 transputer, which was used for the multiplexer.

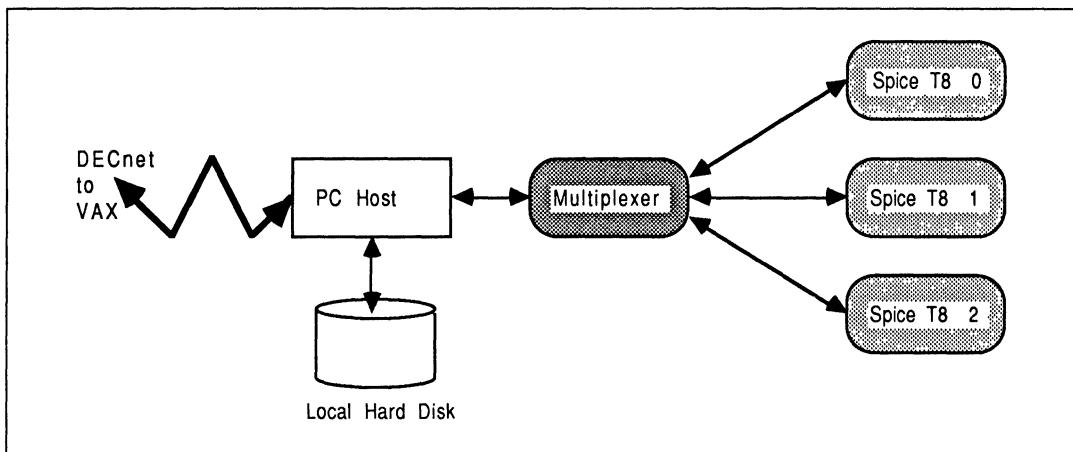


Figure 15.3 A four processor SPICE system

15.3.2 System design notes

Requirements

The system had to be simple to operate and maintain, capable of operating with several transputers in a processor 'farm', capable of integrating additional applications, it had to be extensible, and it had to offer a useful service to the users. In the event of the application crashing, the system should endeavour to recover itself automatically. This had a bearing on the network topologies permitted, and the distribution of support processes in the farm network. The interface to DECnet would be through the PC server, which would be modified to accommodate this requirement.

Some of these requirements are discussed below.

Overall system floorplan and development strategy

Each SPICE program runs on a separate T800 transputer, to obtain the maximum performance. In this and the following discussions, the term 'worker transputer' will be used to indicate a transputer which executes the target application.

Since the server would have to be modified anyway to communicate with DECnet, it was decided to delegate to the server the allocation of tasks to each worker transputer. A multiplexer was written in OCCAM to ensure correct processor interleaving when communicating with the server.

Automated failure recovery and network topology implications

The occam multiplexer sits between all tasks on the worker transputers and the host server, as shown in figure 15.3. As such, it is aware of every inter-communication between any worker and the server. In addition to performing message interleaving between all the workers, it provides timeout facilities to identify if any worker has been 'silent' for greater than some specified time interval. This information can be used by the server to notify the VAX if any jobs fail to complete for some reason. This maintains the reliability and throughput of the system in the event of a partial failure, and allows a graceful degradation of the system. Individual jobs do not see *any* degradation.

The requirement for this capability arises from the fact that the SPICE program does occasionally crash while performing a simulation (not just on transputer systems).

By placing each SPICE on a transputer which is separate from the multiplexer transputer used by other worker transputers, it can be guaranteed that should any SPICE job crash a transputer, the multiplexer can

detect this and take action to complete current jobs and re-boot the network automatically. The *guarantee* of automatic recovery from a crashed task arises from the fact that the multiplexer has a transputer to itself (which excludes parasitic shared-memory problems with rogue processes on other transputers), is written purely in OCCAM and is significantly less complex than SPICE – it will *not* crash.

One consequence of the ‘separate multiplexer per transputer’ approach, as well as the cost, concerns the maximum number of SPICE workers that can be connected without compromising the recovery capability of the system. Because each transputer has four links, it can multiplex three inputs down to one output. So, up to three SPICE processors can be run with one multiplexer.

By cascading multiplexers, additional hardware links become available to accommodate more task processors. As an example, up to three ‘worker multiplexers’ (denoted Wkr Mux in figure 15.4) may be controlled from an ‘intermediate multiplexer’ (denoted Int Mux), and each worker multiplexer can of course accommodate three SPICE tasks. The distinction between the multiplexer types is as follows: a worker multiplexer connects directly to application workers and has the additional role of timeout monitoring to detect ‘dead’ workers; the intermediate multiplexer connects only to OCCAM worker multiplexers and does *not* require to perform timeout detection.

This design of point-to-point communication allows for more T800s to be controlled if necessary, and also serves to contain any ‘failed’ SPICE jobs, as discussed above. The multiplexers can be connected into quite complex configurations, providing that no implementation limits of the host, its operating system, or the server are exceeded (for example, there may be a host operating system limit on the maximum number of files that can be open at once). Some possibilities are outlined in figure 15.4.

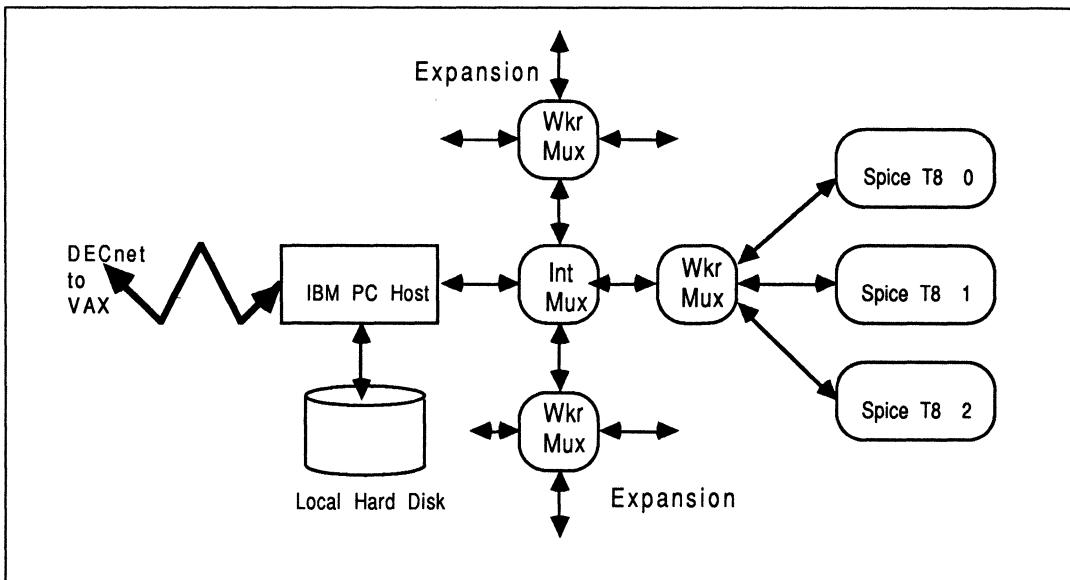


Figure 15.4 Multiplexer connectivity example

Had a pipeline-based topology been employed, this would have necessitated a multiplexer *and* SPICE to reside on each transputer. While this is straight-forward to arrange, it is conceivable (although improbable) that should a SPICE task fail, this sharing of hardware by a rogue process and a healthy one might prevent the multiplexer from sending a failed signal to the host server, which would thereby prevent the network from automatically rebooting.

In most practical systems, however, this requirement for failure recovery can be satisfied to an almost equivalent level without the need for a transputer per multiplexer — this is useful in keeping the cost of the system realistic. The method we adopted is only one of several options we could have chosen.

15.3.3 PC support

The PC support is provided by modifications to the standard file server / loader program, called AFserver. These modifications allow the server to control bi-directional communications between the PC and the VAX, via DECnet.

An outline of the PC server

The PC server has two important functions to perform. Firstly, it must communicate with the transputer board, which will send requests for work. Secondly, it must communicate with DECnet and transceive work requests and results between the PC file-store and the VAX.

The original PC server, called AFserver, is written in C. It consists of a small collection of functions which allow communication between the transputer system and the PC host. Normal server functions include file access and stream management etc. This communication is implemented using an INMOS link-adaptor, which interfaces a transputer link to the host PC's bus.

The small collection of routines provided by the server are grouped together in a flexible way, facilitating 'hooks' for adding additional commands into the body of the server. In different situations, where a lot of application code has to remain on the PC host, the source of the AFserver could be built into the PC-part of the application [4].

The transputer and the host conform to a master / slave relationship. The transputer is the master, implying that all commands, which form part of the so-called 'AFserver protocol', are initiated from the transputer system. The function that decodes the command coming from the INMOS link-adaptor (connected to the transputer system) is called `read_link()`. This is outlined below:

```
void read_link ()
/* Read a message coming down the link. */
{
    if (read_integer (&command))
    {
        switch (command)
        {
            case TERMINATE_CMD:
                filer_close ();
                write_integer (F_OK);
                terminate_server (T_TERMINATED);
                break;

            . . .

            default:
                terminate_server (T_ILLEGAL_COMMAND);
        };
    }
}
```

It should be apparent how additional cases can be added to the AFserver protocol to accommodate the user's specific requirements, by providing further alternatives in the `switch` statement.

The design of the server, in connection with the protocols involved in managing interactions with the transputer on the one hand, and interactions with DECnet on the other hand, is now discussed.

Server extensions

It was decided to change the transputer's AFserver protocol as little as possible, and make the PC server contain all the DECnet accessing software (although this has the effect of making the PC server less general-purpose). Only two additional commands were added to the AFserver's protocol. These were:

- **FinishedTask.Cmd** : Sent to the server from a SPICE worker transputer to identify a completed task and request more work. Parameters identify the current task completed, and the name of the new task to be started is returned.
- **Failed.Cmd** : Sent to the server by the occam multiplexer, following an extended period of inactivity by a SPICE worker. There are no parameters for this command.

The operation of the SPICE system, and it's use of these commands is now outlined.

System operation

The **FinishedTask.Cmd** is sent to the server from an IMS T800 transputer running a SPICE task. When idle, each SPICE sends a **FinishedTask.Cmd** command with a null filename every second. If a SPICE has just completed a task, it sends the name of the completed task as a parameter to **FinishedTask.Cmd**. When the server receives a **FinishedTask.Cmd**, it checks the filename parameter. If the filename was a valid one, the server copies the output file to the VAX. Then, regardless of the filename parameter, the server polls DECnet (on the listening socket) to see if any new connect requests have been received. If so, the connect request is accepted, the logical link is attached to the data socket, and transfer can occur. The new file name is read, the file is copied onto local disk, and the local file name is sent to the available SPICE worker.

The server initially allocates a socket on which to listen for incoming connect requests (for task number 242 — see the implementation notes in the next section). It maintains a local list of file names, their corresponding VAX destinations, and their data socket numbers. This is because SPICE tasks can originate from any of several VAXes on the network, from different users, and from different directories, so the local filename is insufficient information to allow the results to be returned to the sender.

The **Failed.Cmd**, is sent to the server when an occam multiplexer believes that a SPICE simulation has failed. After receiving this command, the server does not accept any new connect requests. The server doesn't know which SPICE task has failed until the other tasks finish or fail, because it does not maintain a correspondance list of transputers executing specific tasks. When all current tasks complete, or fail, the server reboots the transputer network and sends messages down the remaining logical links to explain the failure to the VAX users. The whole network has to be rebooted because of the use of the standard SubSystem ports on the evaluation boards — it is not easy to reboot individual transputers in the network.

The server only polls DECnet when there is a transputer available to do work. The main reason for doing this was to improve the file server response for the transputer system, since polling DECnet is time consuming and only needs to be done when a SPICE processor is waiting for a task filename.

Implementation of the new server commands

This section discusses in some detail the C extensions written for the AFserver.

The `read_link()` function is shown here with the two additional hooks to implement the extra AFserver protocol tags, called `FINISHEDTASK_CMD` and `FAILED_CMD`:

```
void read_link ()
/* Read a message coming down the link. */
{
    if (read_integer (&command))
    {
        switch (command)
        {
            case TERMINATE_CMD:
                filer_close ();
                write_integer (F_OK);
                terminate_server (T_TERMINATED);
                break;
            .
            .
            /* new AFserver protocol commands for SPICE farm */
            case FINISHEDTASK_CMD :
                finished_task();
                break;
            case FAILED_CMD :
                failed();
                break;
            default:
                terminate_server (T_ILLEGAL_COMMAND);
        }
    };
}
```

Consider the `FINISHEDTASK_CMD` first.

Implementing the FINISHEDTASK_CMD command

The **FINISHEDTASK_CMD** is sent by a SPICE application when it is available to do work. It is processed by the set of functions shown in figure 15.5. This figure also shows the hierarchy of these functions.

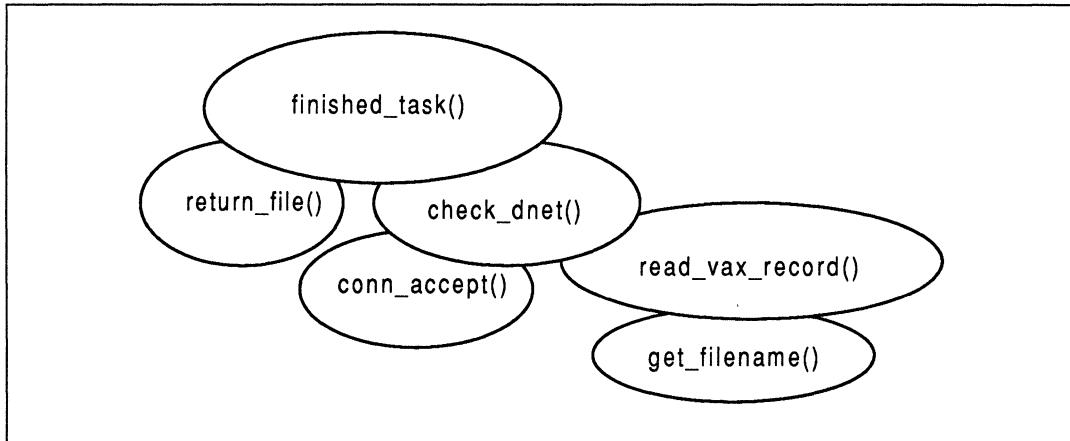


Figure 15.5 The FINISHEDTASK_CMD function hierarchy

When the server receives the **FINISHEDTASK_CMD**, it knows that there could be an output file ready to send to the VAX. If so, then it sends the file to the VAX using the **return_file()** function. Next, it polls DECnet to see if there are any connection requests. This is outlined in the **finished_task()** function :

```

void      finished_task()
/* Just received FINISHEDTASK_CMD from worker multiplexer, */
/* so send output file back to the VAX and check for a new */
/* input file to be processed by the transputer farm. */
{
    int block_size;
    char buffer [RECORD_LENGTH + 1];

    if (read_record (&block_size, buffer))
    {
        buffer [block_size] = '\0';
        /* buffer is the name of the finished task */
        if (block_size > 0)
            return_file(buffer, block_size);
        if (aborted)
        {
            write_record(0, "");
            write_integer(OPERATIONFAILED_ERR);
        }
        else
        {
            strcpy(buffer, check_dnet());
            /* about to write filename info to transputer */
            write_record(strlen(buffer), buffer);
            write_integer(F_OK);
        }
    }
}
  
```

Notice the use of the **aborted** flag in the above function, which is set as part of the **FAILED_CMD** handler. If the **aborted** flag is set, then the network will shortly reboot so no further polling of DECnet is entertained.

If the system is still allowed to poll DECnet, then it does so using the `check_dnet()` function :

```

char    *check_dnet()
/* Check DECNet for incoming connect requests, */
/* returns pointer to a filename or NULL          */
{
    struct timeval
    {
        long      tv_sec;      /* seconds */
        long      tv_usec;     /* and microseconds */
    } tim;
    unsigned long      read;
    int      nfds;
    int      i;
    int      ready_bits;

    tim.tv_sec = 0;
    tim.tv_usec = 25; /* check for activity for 25 microsec */
    read = 1<<sock_no;
    nfds = sock_no + 1;

    if (keyb_input())
        net_err(NULL, 0);
        /* check active sockets for input */
    if (select(nfds, &read, 0, 0, &tim) > 0)
        conn_accept();
        /* conn request received, wait for fnam*/
    read = in_use_mask;
    nfds = MAX_SLAVES + 3;
    ready_bits = select(nfds, &read, 0, 0, &tim);
    if (ready_bits > 0)
        return( read_vax_record( read));

    if (ready_bits < 0)
        net_err("Cannot select data sockets:\n", errno);

    return( NULL );
}

```

The `check_dnet()` function polls DECnet and listens for connect requests. If it receives a connect request, it uses low level DECnet socket interfacing commands in `conn_accept` to establish a data logical link over a new data socket. The routine uses a number of global variables, most of which are concerned with managing the available/used DECnet sockets. `check_dnet()` uses `read_vax_record()` to read one of the sockets specified, display the record on the console, and return a pointer to valid local filename. This is done using `get_filename()` which performs NFT (Network File Transfer) commands to copy the input file from the VAX to the PC.

Implementing the FAILED_CMD command

The `FAILED_CMD` is received by the server from an OCCAM multiplexer, rather than from a SPICE application — how could a SPICE application know it had failed if it was, itself, out of control? It is used to set a global flag in the server called `aborted` to prevent any further polling of DECnet for new tasks.

```

void    failed()
/* Just received a FAILED_CMD from worker multiplexer, */
/* so set aborted flag to prevent further DECnet testing */
{
    write_integer(F_OK);
    aborted = TRUE;
    active_task_count-- ;
}

```

Initializing and closing down

There is, of course, a lot of additional code required to initialize DECnet and to close things down in an orderly manner.

For example, the following initialization sequence uses DECnet-DOS socket interface calls, and is called as part of the booting sequence:

```
void      decnet_init()
{
    /* Initialise the DECNet side of things,
     * - return the socket number of the listening socket */
    slaves_init();
    if (system("NCP SET KNOWN LINKS STATE OFF"))
        net_err("NCP Call failed:\n",errno);
    printf("\n\t Inmos PC Server, Version 2.0\n\n");

    /* open a DECnet socket */
    if ((sock_no = socket( AF_DECnet, SOCK_SEQPACKET, 0 )) < 0)
        net_err("Socket allocation failed:\n",errno);
    /* bind an object num to the socket */
    bzero( &socket_char, sizeof( socket_char));
    socket_char.sdn_family = AF_DECnet;
    socket_char.sdn_objnum = 242;
    /* 242 is DECnet server task number */

    if (bind( sock_no, &socket_char, sizeof( socket_char)) < 0)
        net_err("Bind to socket failed:\n",errno);
    /* listen for connect requests */
    if (listen( sock_no, backlog) < 0)
        net_err("Listen failed:\n",errno);
    return;
}
```

The following function is used to close all active sockets before the transputer system is rebooted after FAILED_CMD.

```
void      remove_socks()
{
    /* Shut down all sockets still active, and          */
    /* tell Vax user that his job has failed          */
    int      i;
    char    msg[MAX_BUF_SIZE];
    char    command_buff[MAX_BUF_SIZE];

    for (i=1; i<MAX_SOCK; i++)
        if (file_names[i].file_name != NULL)
        {
            strcpy(msg,"Abnormal Completion for file : ");
            strcat(msg,file_names[i].file_name );
            strcat(msg,"\n");
            swrite(i,msg,strlen(msg));
            sclose(i);
            strcpy( command_buff, "NFT COPY ");
            strcat( command_buff, file_names[ i ].file_name );
            strcat( command_buff, ".OUT ");
            strcat( command_buff, file_names[ i ].full_spec);
            strcat( command_buff, ".OUT >>NFT.LOG");
            system( command_buff );
        }
}
```

Hopefully, the above functions give some appreciation of the work involved in this part of the project — about 30k (source size) of specially-written C was required to interface to the DECnet-DOS software at the PC end. This amounts to more than half of the source size of the original AFserver. [3] gives useful examples and guidance for doing this type of work.

Once the appropriate network communications software exists for the environment, the stages to incorporate it into the AFserver software are trivial.

15.3.4 Transputer support

In arranging for an application to be incorporated into an ‘autonomous worker’ environment, there are two options concerning the amount of transputer support required. These are directly related to the mechanism of how new tasks are allocated to workers within the farm.

- New work tasks can be explicitly requested in the non-OCCAM application code itself, and their dispensement can be controlled from the server. This results in minimal transputer-resident support software, because it removes the need for a transputer-resident farm controller (task allocator). A simple server-protocol multiplexer is sufficient to interleave work requests to the server, and the server is extended to cope with additional protocols to handle work assignment in the farm and DECnet interfacing.
- A set of farm controller processes, written in OCCAM, can be used to receive work tasks (from the DECnet server) and allocate them amongst available worker processors. This approach is the more general-purpose of the two, because it is completely host-independent, and it obviates the need to modify the application code or the host server-transputer protocols. The application does not need to know it is in an autonomous working environment, or that it may be one of several running on the same transputer network.

In both cases, a process on a transputer is responsible for requesting more work from the server. In the first instance, the application itself directly asks the server for work. In the second case, an available worker application asks the transputer-resident farm controller for work, and the farm controller then asks the server. In both cases, once the server receives a request for work, it would check DECnet for any pending requests.

In our implementation, the first of these two options was selected. The application was slightly modified to ‘ask for work’ from the server, and the server assumes the responsibility of dispensing tasks.

Modifications to the application

Like SPICE itself, the modifications were written in FORTRAN, and placed *around* the ‘root’ part of the application. The modifications simply concerned the requesting for work (using the extended AFserver protocol defined earlier), and the establishment of data input and output file names for each simulation to run. It is an obvious requirement that since each SPICE worker is served from the same file store, the local file names being processed concurrently must not clash with any others. This is handled by the server.

The message-passing routines provided by the run-time libraries supplied with all the scientific-language compilers are used to communicate with the PC server, using the newly defined protocol. These protocols handle the requests for new work, and have already been explained at the host server end. Here, at the application end, the standard FORTRAN message passing routines called **CHANINMESSAGE()**, **CHANOUTMESSAGE()**, **CHANOUTBYTE()**, and **CHANOUTWORD()** are used [6], as shown overleaf in our implementation for the SPICE farm.

```
C *****  
C * Communication with DECnet file Server *  
C *****  
C SUBROUTINE ReadInteger( N )  
    INTEGER Tag  
    CALL CHANINMESSAGE( 1, Tag, 1 )  
    CALL CHANINMESSAGE( 1, N, 4 )  
    RETURN  
END  
  
SUBROUTINE WriteInteger( N )  
    INTEGER Int32Value  
    PARAMETER (Int32Value = 4)  
    CALL CHANOUTBYTE( Int32Value, 1 )  
    CALL CHANOUTWORD( N, 1 )  
    RETURN  
END  
  
SUBROUTINE ReadRecord( Len, Record )  
    INTEGER NilRecordValue, Record32Value  
    INTEGER Tag  
    PARAMETER (NilRecordValue = 8, Record32Value = 12)  
    DATA Tag / 0 /  
    CALL CHANINMESSAGE( 1, Tag, 1 )  
    IF (Tag.EQ.NilRecordValue) THEN  
        Len = 0  
    ELSE  
        CALL CHANINMESSAGE( 1, Len, 4 )  
        IF (Len.GT.0) CALL CHANINMESSAGE( 1, Record, Len )  
    END IF  
    RETURN  
END  
  
SUBROUTINE WriteRecord( Len, Record )  
    INTEGER NilRecordValue, Record32Value  
    PARAMETER (NilRecordValue = 8, Record32Value = 12)  
    IF (Len.EQ.0) THEN  
        CALL CHANOUTBYTE( NilRecordValue, 1 )  
    ELSE  
        CALL CHANOUTBYTE( Record32Value, 1 )  
        CALL CHANOUTWORD( Len, 1 )  
        CALL CHANOUTMESSAGE( 1, Record, Len )  
    END IF  
    RETURN  
END  
  
SUBROUTINE FinishedTask( SizeOldTaskName, OldTaskName,  
1                         SizeNewTaskName, NewTaskName,  
2                         Result )  
    INTEGER FinishedTaskCmd  
    PARAMETER (FinishedTaskCmd = 127 )  
    CALL WriteInteger( FinishedTaskCmd )  
    CALL WriteRecord( SizeOldTaskName, OldTaskName )  
    CALL ReadRecord( SizeNewTaskName, NewTaskName )  
    CALL ReadInteger( Result )  
    RETURN  
END
```

The subroutine **FinishedTask()** makes use of one of the additional tags to the AFserver protocol, called **FinishedTaskCmd**. It is used in the main top-level part of the application as follows:

```

PROGRAM SPICE
IMPLICIT NONE

INTEGER MaxFileNameSize
INTEGER SizeOldTaskName, SizeNewTaskName
INTEGER OneSecond
INTEGER Result
PARAMETER (OneSecond = 120000, MaxFileNameSize = 20)
CHARACTER*(MaxFileNameSize) OldTaskName, NewTaskName
CHARACTER*(MaxFileNameSize) SpiceIn, SpiceOut

5   CALL Delay( OneSecond )

CALL FinishedTask( 0, OldTaskName,
1           SizeNewTaskName, NewTaskName,
2           Result )
20  IF (SizeNewTaskName.EQ.0) GO TO 5

SpiceIn = NewTaskName( 1: SizeNewTaskName ) // '.in'
SpiceOut = NewTaskName( 1: SizeNewTaskName ) // '.out'
OPEN (UNIT=5,FILE= SpiceIn,STATUS='OLD')
OPEN (UNIT=6,FILE= SpiceOut,STATUS='NEW')

CALL SpicRoot()

CLOSE (UNIT=5)
CLOSE (UNIT=6)
SizeOldTaskName = SizeNewTaskName
OldTaskName      = NewTaskName
CALL FinishedTask( SizeOldTaskName, OldTaskName,
1           SizeNewTaskName, NewTaskName,
2           Result )
2   GO TO 20
END

```

The line **CALL SpicRoot()** is the new call to the main SPICE application. Since the application has been delegated the responsibility of requesting more work, it has been made into a non-terminating work request loop. This means that once MultiSPICE is running, it will accept work continuously. Within this non-terminating work request loop is a small delay, which is used to prevent an available worker from pestering the server continuously in cases where there is no *new* work, but where other workers may be busy. In [4], a general-purpose farming technique, which does not involve modifications to the application, is presented.

The OCCAM harness

A SPICE worker is encapsulated by a small amount of occam, known as the harness. The harness is required to ensure that the FORTRAN application receives access to the server for filing, screen, and keyboard facilities. For a description of the OCCAM language, developed to express and exploit the parallelism offered by the INMOS transputer, the reader is directed towards [5]. For information concerning the content and creation of an occam harness for non-OCCAM programs, please refer to [6].

Due to MultiSPICE being an autonomous computing engine, none of the software is written to self-terminate. In this situation, either the application code or the harness must never terminate. Due to an earlier decision to modify the application to request work, the application itself was made non-terminating. This allowed the standard occam harness, as supplied with the scientific-language compilers and the D705A occam-2 toolset, to be used for each SPICE worker. All this work is still relevant in the context of the D705B occam-2 toolset. To stop MultiSPICE requires deliberate and specific user interaction on the host PC. Refer to section 15.3.6 for operation details.

The OCCAM multiplexers

Running several SPICE jobs concurrently requires that their accesses to the host PC be dynamically interleaved. This is most easily done in occam by having a single transputer that talks down the single channel to the AFserver on the PC.

A worker multiplexer sits between the IMS T800 transputers running the application code, and the server on the PC. It provides message interleaving and time-out services for each application transputer. The timeout is determined empirically. The multiplexer should sit on its own transputer, so as to preserve the crash-recovery capability of the MultiSPICE system. Its general structure is illustrated below:

```

WHILE TRUE
  BYTE Tag :
  SEQ
    ALT
      (NOT WorkerFail[0]) & FromWorker0 ? Tag
        ActOnMessage( FromWorker0, ToWorker0, 0, Tag )
      (NOT WorkerFail[0]) &
        Clock ? AFTER LastInput[0] PLUS InactiveDelay
        ActOnTimeOut( 0 )

      (NOT WorkerFail[1]) & FromWorker1 ? Tag
        ActOnMessage( FromWorker1, ToWorker1, 1, Tag )
      (NOT WorkerFail[1]) &
        Clock ? AFTER LastInput[1] PLUS InactiveDelay
        ActOnTimeOut( 1 )

      (NOT WorkerFail[2]) & FromWorker2 ? Tag
        ActOnMessage( FromWorker2, ToWorker2, 2, Tag )
      (NOT WorkerFail[2]) &
        Clock ? AFTER LastInput[2] PLUS InactiveDelay
        ActOnTimeOut( 2 )
  
```

This code fragment has the effect of allowing the first SPICE worker requiring access to the server, exclusive use of the server for a single AFserver protocol transaction. It also allows for inactivity timeout monitoring on any worker, and prevents workers previously identified as 'dead' from further servicing by the system.

Not shown here is the use of OCCAM's **PRI ALTs**, which can be used to ensure fairness of servicing worker requests for all the participating workers.

The **ActOnTimeOut()** procedure is responsible for the origination of the **Failed.Cmd**. The **ActOnMessage()** procedure uses the **InputOrFail.t** and **OutputOrFail.t** communications procedures [7], allowing controlled recovery from failure of transputer link input / output.

The parameter **InactiveDelay** is designed to trap a crashed simulation. The nature of SPICE is such that it has a high computation to communication ratio. This means that relatively long periods of time can elapse between communication bursts and any observable link activity to the host server. OCCAM allows simple handling of timeout issues in comparison to other high- and low-level languages. OCCAM has constructs to allow the reading of the transputer's timers, and to cause delays until certain periods have elapsed. The transputer has two timers, one accessible during high priority execution, and the other accessible during low priority execution. In high priority execution, the timer tick once every micro-second. In low priority, the timer ticks once every 64 micro-seconds.

The **InactiveDelay** is set to correspond to a time interval of around an hour or so. Its value is specified in the OCCAM configuration description for the system, and passed in to each processor as parameter. This allows its value to be changed easily without recompiling anything. The timeout period is calculated to be larger than the longest time taken by the largest simulation intended to run on MultiSPICE.

The simple design of the multiplexer software means that while a **FinishedTask.Cmd** request from a SPICE processor is being serviced, the server blocks any of the other transputers from receiving or sending data to / from the PC. While this avoids routing overheads within the SPICE array (because there is no

need to pass source and destination information with the message and explicitly route it within the farm), it also means that SPICE applications can remain unserviced for several minutes while file transfers over the Ethernet are taking place. This is not as severe a problem as it might seem, because only one device can have access to the file store at once anyway.

The multiplexer software can be compiled for execution on the 16-bit T212/T222 transputers, or for the 32-bit T414/T425 transputers – there is no real need to use a T800 here.

The afore-mentioned intermediate multiplexer connects *only* to occam worker multiplexers and does *not* require to perform timeout detection. This is because the worker multiplexers, by virtue of their design, will always be capable of identifying inactivity problems with any of their applications. Therefore the intermediate multiplexer need only have the capability to through-route timeout messages to the server.

15.3.5 VAX support

The VAX is the main central computing resource in this system. The PC has a unique node name and address on DECnet. When the user wishes to send a task specification, the VAX establishes a logical link which allows the exchange of data between the VAX and the PC server.

Two DCL² command files on the VAX were written to arrange for the SPICE input file on the VAX to be sent to the PC node on DECnet. This input file is then sent to an available transputer by the server.

The first command file, called SPICE.COM, receives from the user the name of the SPICE input deck to send to the PC server, the destination node, and the password. It then spawns a subprocess to do the actual data exchanges. Our implementation of SPICE.COM is shown below.

```
$ ! Command file used to talk to IBM PC SPICE Server
$ ! P1 is node name of target PC
$ ! Prompts for users password and list of filenames (1 per line)
$ ! A subprocess is spawned for each filename given.
$ ! All subprocess messages are displayed on the screen.
$
$ set NoOn
$ on control_c then goto cleanup
$ IF P1 .EQS. "" THEN Inquire P1 "Node"
$ Set ter/noecho
$ Inquire Password
$ Set ter/echo
$Loop:
$ inquire record "Filename"
$ if record .EQS. "" then goto loop
$ File_spec = F$Parse(record)
$ File_spec = File spec - ""
$ Node = F$Logical("SYS$NODE")
$ Colon = F$Locate(":",Node)
$ Node = F$Extract(0,Colon,Node)'
$ Node = Node - ""
$ User = F$Getjpi("", "USERNAME")
$ Space = F$Locate(" ",User)
$ User = F$Extract(0,Space,User)
$ spawn /nowait -
@subproc 'P1' 'Node' 'User' 'Password' 'File_Spec'
$ goto loop
$
$cleanup:
$ Del/sym Password
$Exit
$
```

²DCL – Digital Command Language

The subprocess spawned by SPICE.COM opens a logical link to the PC (a DECnet node) and specifies the 'task number' to run at the PC. The task number specified is 242. Since the PC can only run one task at a time (the server), it has to be running this task before the VAX attempts to talk to it.

Here is the code for our implementation of the SUBPROC.COM DCL file :

```
$ !Called by SPICE.COM to communicate with PC Server
$ 
$ set NoOn
$ open/read/write link  'P1':::"242="
$ write link  "'P2'"'"P3'  "'P4'"":'"P5'
$ read link record
$ Write sys$output "'f$getjpi("", "PRCNAM")'"
$ write sys$output record
$ close link
```

The SUBPROC.COM file operates at a much lower-level than the SPICE.COM file that spawns it. It uses non-transparent DECnet commands. The choice of a task number of 242 was completely arbitrary, but mainly because the first and ultimate choice, 42, was already in use by DEC. The selected identifier 242 must correspond to the task number in the special PC server, otherwise communication between the PC server and the VAX will not be possible.

15.3.6 Operating the system

MultiSPICE receives jobs over DECnet from the VAX. It is important to run the PC server to boot the transputer network before any attempt at the VAX is made to send work, otherwise the system is likely to fail to establish communication before a VAX timeout takes effect.

Running MultiSPICE at the PC end

On the PC, the special server was called SERVER.EXE. It understands the same command-line parameters as the INMOS AFserver, so to boot the transputer system with the SPICE workers and multiplexer file, called **spicfarm.bt**, the following MS-DOS command could be used:

```
server -:b spicfarm.bt
```

It may be necessary to perform some one-off set-up commands concerning the PC and DECnet, for example, ensuring that there are sufficient 'file links'³ available over the network.

MultiSPICE will now load, execute, and wait for incoming jobs. Our version was tested and operated with up to three SPICE Applications executing concurrently. The system as it stands will not accept job assignments from MS-DOS : only requests from DECnet are recognised.

Running MultiSPICE at the VAX end

The VAX is responsible for sending jobs to MultiSPICE, and retrieving the results. To run the VAX DCL command file, called SPICE, one could invoke the command by typing @**SPICE**.

The command file prompts the user for the name of the node on DECnet that MultiSPICE can be found, a password to allow access to the file on the VAX, and the filename of the job to be simulated. Everything else happens in the background without the user being aware of anything exciting happening, until a message is displayed on the terminal screen describing the final status of the job. The job is either successful or aborted.

As files are being transferred, the support software at both the VAX and the PC ends issues messages to indicate the current activities. For example, the VAX shows the task completion status message, and the PC shows file transfer messages. These are also written into a log file called NFT.LOG for post-mortem debugging, by redirecting the output of the DECnet-DOS NFT command.

³This term is used in the DECnet context, rather than a transputer links context

15.4 Other considerations

15.4.1 Implementation guidelines

Tools required

To reproduce any of the work described in this document, the OCCAM-2 toolset is required to create and manipulate the transputer components. A transputer compiler for the non-OCCAM application will also be required. To modify the AFserver, which is supplied in C source form on the PC, a (Microsoft) C compiler for the host computer is needed. This will allow the compiled server module to be linked with the C library supplied with DECnet-DOS.

Suitable applications

INMOS provide scientific language compilers for C, Pascal, and FORTRAN. The INMOS development systems allow applications written in mixtures of these languages, including OCCAM to be easily executed on a transputer system. The range of INMOS' scientific language compilers is growing constantly — please refer to [8] for current product availability.

The applications should preferably be batch-like in nature, i.e. they take an input file, perform some compute-intensive operations, and produce output files, without user interaction or screen access.

Especially appropriate are applications in which the ratio of 'computation to communication' is high. This means that the overheads in sending the input data to the transputer, and receiving the results back over the network, are low in comparison to the amount of computation that is to be performed on the said input data. Typical applications that fall into the category include simulation packages (chemical, thermal, dynamic, electrical), technical modelling packages, compilers, and text formatting packages (eg *T_EX*, troff, PostScript processors etc).

The more interactive an application, the less suitable it is for the type of implementation which is described here. This is due to the latency and overheads in transporting the interactive commands and replies, between the user at one end of the network, and the transputer server at the other end of the network. Network latency is concerned with the delay before processing starts, between the user invoking the command on his / her terminal, and the transputer worker starting on the job. It consists of the time taken to get the input data sent from the VAX to the transputer host. There is an additional small delay due to the time taken by the transputer to read the input work task from the local file store. The response times normally associated with real-time interaction may be unacceptable given these overheads.

At times of heavy loading, the network latency will increase and the transfer rates will correspondingly drop. This will depend on the nature of the network. In non-deterministic Local Area Network's (LAN's) like Ethernet, one can observe almost order of magnitude fluctuations in response time, depending on the instantaneous system loading. So remember, use only applications with a high ratio of compute-time to communicate-time.

Implementation strategy

To implement a remote transputer server of the form described in this document, the first stage is to get a single un-modified version of the application running on a transputer board. This may involve making small changes to the application in order to get it through the scientific language compilers that INMOS provide. This is not because of any particular deficiency in the INMOS scientific-language compilers, but rather because many applications tend to make use of non-standard language-extensions provided on their native environment compilers.

The result of this is software that can be used to obtain performance measurements of the application running on a transputer. [4] discusses some application porting issues.

The next stage would be to replicate the application over a number of transputer workers, using techniques described previously or in [4]. Alternatively, one may wish to have only a single transputer worker in the system.

Next, modify the server to communicate with the network, DECnet in this case. The DECnet-DOS program-

mers reference manual [3] is invaluable here, giving examples of how to establish two-way communication between any two nodes on the network. If you are using a different network, there will be a corresponding technical reference manual. Test the server with a small stub of Occam or C (for example), on the transputer, to be certain that something on the transputer network can request and receive the correct information over the network.

Finally, combine the modified server with the *real* application, and everything should operate correctly. If this is successful, then you can go live!

Timescales

Timescales for a project such as this are difficult to estimate. Many factors are involved. For example, depending on the application's use of non-standard language features, there may be effort required to re-implement these parts of the application in a standard manner before the INMOS scientific-language compilers will accept the source input.

The time and effort to make a suitable server will depend on the available documentation describing the interfacing and protocols between the PC server and your network. In the case of DECnet, the examples given in this technical note should be of use.

15.4.2 Multiple task farms

If a farm is created which has several different tasks running, each on their own transputer, then it would be necessary for each job request to be accompanied by some means of identifying the task. In such situations, a need for additional hardware to allow individual transputers to be reset/loaded could be identified.

If the code for that task is not currently loaded onto the farm, then it must be fetched from local file store and loaded into the appropriate transputer. A discussion of how to organize such a system, and how it might be implemented, is given in [4].

15.4.3 Receiving work from DOS rather than DECnet

Work done in February 1988 by the INMOS Central Applications Group produced a version of MultiSPICE which accepts jobs in a batch fashion from MS-DOS, and can run an arbitrary number of concurrent SPICES by using a pipeline of tasks run on B404 TRAMs on a B008 motherboard [8]. (B404's have an IMS T800 and 2 Mbytes of dynamic RAM on a size 2 TRAM module). The system was more general purpose than the one described in this technical note, for two main reasons. Firstly, the SPICE application was *not* modified. Secondly, the task allocation was controlled from processes executing on the transputer array rather than the on the host, which makes the software more portable and more applicable in different host relationships. [4] discusses this approach in detail.

15.4.4 Network monitoring software

Our network had a traffic-monitoring system program which could detect periods of inactivity on 'open connections', and log-off users or applications that had not responded with the VAX for a certain time.

One of the main reasons for choosing SPICE as a candidate for a dedicated remote application server is its good ratio of computation to communication. SPICE in particular produces all its output in one go at the end of each simulation, and hence will tend to communicate over DECnet in bursts, separated by (possibly) extended periods of inactivity, rather than in any continuous fashion.

This temporal distribution of activity (as far as the VAX is concerned) caused a few headaches initially, resulting in the transputer workers getting logged off before the results were produced. Once the network monitor had been instructed correctly, there were no more unintentional detachments.

15.4.5 Other transputer hosts

More recently, a number of manufacturers have produced a range of transputer-based boards for use with a DEC MicroVAX II. If your network includes MicroVAXes, this might be a preferable route to follow compared to using a PC host for the transputer board(s). Factors of cost, available driving software, and network performance would have to be examined — as well as that of technical challenge!

The scope of this work would also be appropriate in connection with Sun's NFS⁴ environment.

15.4.6 Is it worth it? — Weighing up the pros and cons of using transputers

In an effort to establish the suitability of a hosted transputer server to reduce the loading on existing computer facilities, the following discussion may be relevant.

To implement the system described requires a node on the network to host a transputer motherboard. Such a node could take the form of a personal computer, with an Ethernet card. The personal computer can accommodate several transputer modules, all of which can be performing the same or different tasks. As a rough guide, a single B404 transputer module [8], consisting of a 32-bit IMS T800 transputer [2] and 2 Mbytes of dynamic RAM, can give the same performance as a VAX 11-785 with floating point accelerator hardware, when executing non-modified non-Occam code [9].

If the application is written in a largely standard dialect of a supported-language, then it is unlikely that there will be major problems in 'porting' the application onto a single transputer, in a timescale that could be measured in days. Several such applications can then be run together in a farm. The best results will come from applications that are highly compute-intensive, perform only limited file access and perform no user-interaction.

The personal computer host can also be used as a normal PC at times when it is not used by the network users. The effort of running your software on a transputer means that the software can now execute on any machine hosting a transputer product ; given a suitable server. This opens up useful portability opportunities that were previously not feasible. The unparalleled inter-connectivity of the INMOS transputer means that once the application is running on a single transputer, one can explore the possibilities for further performance increases by re-structuring parts of the software and by using additional hardware. Not least of course is the impressive performance one would obtain even when using only one transputer.

Everything reduces down to the question 'How much does CPU time cost me on my existing computing facilities?'. This cost has to include factors for equipment maintenance contracts, spares, upgrades etc. What could that time have been better spent on doing? Just think ... with a modest amount of effort, all the benefits discussed previously become a reality. So, 'is it worth it?'. Yes.

15.5 Summary and conclusions

The transputer farm accelerator described in this document has proved to be a powerful, reliable and cost-effective dedicated application accelerator. For example, in general, a single transputer is more than one and a half times faster than the Sun with 68881 (especially on larger jobs), and on-par with VAX 11/785 (with FPA) performance [9].

Another advantage of this system is that the node being used to host the transputer need not always be used as a server in this way. It can still be used as a normal networked / standalone PC, running normal PC software, and indeed, running non-networked transputer software on a demand basis.

As well as the cross-host application portability that the transputer offers, there is also great flexibility and extensibility at the system design level. More and more power and capability can be added easily at any stage, by interconnecting different combinations of transputers and memory. By using the INMOS module motherboards and TRAM modules, one can 'pick and match' the appropriate performance and memory requirements for each stage of a progressive and well defined upgrade path.

⁴NFS is a network file system developed by Sun Microsystems to allow machines of different types to share files.

In addition, this work could be extended to cover different network communication software such as Suns' NFS, by altering only the code that ran on the host — the transputer part wouldn't even require re-compilation.

It's this modular and flexible extensibility that promotes the transputer as a candidate for dedicated 'grow as you do' transparent job-level application acceleration. This technical note has discussed only one scenario to which transputers can be applied — that of using a network to offload work from over-used resources. There are many other scenarios possible; limited only by imagination.

15.6 References

- 1 *Transputer Instruction Set — A compiler writer's guide*, INMOS Limited, Prentice Hall.
- 2 *The Transputer Databook*, INMOS Limited, 1989.
- 3 *DECnet-DOS V1.1 Programmer's Reference Manual*, 'Example Socket Interface Calling Sequence', Section 4.4.1, pp 4–6.
- 4 *Issues in Application Porting and Farming*, Technical Note 53, INMOS Limited.
- 5 *occam 2 Reference Manual*, INMOS Limited, Prentice Hall.
- 6 *Using the Occam toolsets with non-Occam applications*, Technical Note 55, INMOS Limited.
- 7 *Extraordinary use of transputer links*, Technical Note 1, INMOS Limited.
- 8 *INMOS Spectrum*, (contains a brief description of INMOS products), INMOS Limited.
- 9 *Porting SPICE to the INMOS IMS T800 transputer*, Technical Note 52, INMOS Limited.



FIRST EDITION 1989

The Transputer Applications Notebook - Architecture and Software is a compilation of technical notes written by INMOS technologists. The compilation describes an approach to VLSI Architecture based on communicating processes and includes the following topics:

- Communicating processes and OCCAM**
- The transputer implementation of OCCAM**
- Communicating process computers**
- Compiling OCCAM into silicon**
- The development of OCCAM 2**
- IMS T800 architecture**
- The role of OCCAM in the design of the IMS T800**
- Simpler real-time programming with the transputer**
- Long arithmetic on the transputer**
- Exploiting concurrency: a ray tracing example**
- High performance graphics with the IMS T800**
- The transputer-based multi-user flight simulator**
- Porting SPICE to the IMS T800**
- A transputer farm accelerator for networked computing facilities**

Additional information concerning the use of transputer products can be found in another collection of INMOS technical notes, entitled The Transputer Applications Notebook - Systems and Performance.

