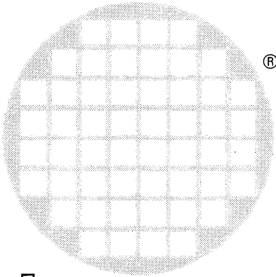


THE T9000 TRANSPUTER INSTRUCTION SET MANUAL

1st edition 1993

INMOS is a member of the SGS–THOMSON Microelectronics Group

INMOS transputer databook series

Transputer Databook
Military and Space Transputer Databook
Transputer Development and *i*q Systems Databook
Transputer Applications Notebook: Architecture and Software
Transputer Applications Notebook: Systems and Performance
T9000 Transputer Hardware Reference Manual
T9000 Transputer Instruction Set Manual
T9000 Development Tools – Preliminary Datasheets

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 Limited 1993

, inmos, IMS, occam and DS-Link are trademarks of INMOS Limited.

SGS-THOMSON MICROELECTRONICS is a registered trademark of the SGS-THOMSON Microelectronics Group.

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

INMOS document number: 72 TRN 240 01

ORDER CODE:DBT9000UMST/1

Printed in Italy

Contents overview

Contents iii

List of tables ix

1 Introduction 1

2 Notation, conventions and terminology 3

3 An overview 5

4 Addressing and data representation 11

5 Registers, status bits and control bits 15

6 Instruction representation 21

7 Sequential operations 27

8 Concurrent processes 65

9 Protection and memory management 97

10 The trap mechanism 107

11 Floating-point instructions 125

12 Channels 151

13 Process state 189

14 Debugging mechanisms 207

15 Cache instructions 211

A T9000 instruction set reference guide 215

B T9000 instruction set sorted by op-code 455

Instruction index 463

Index 465

Contents

List of tables ix

1 Introduction 1

2 Notation, conventions and terminology 3

3 An overview 5

3.1 Processes 5
3.1.1 The occam process model 5
3.1.2 Implementation of processes 6
3.2 Communication 7
3.3 Traps 7
3.4 Configuration of control system 8
3.5 Instructions and pipelining 9

4 Addressing and data representation 11

4.1 Word address and byte selector 11
4.2 Ordering of information 11
4.3 Words, objects and signed integers 11
4.4 Unaligned address detection 12

5 Registers, status bits and control bits 15

5.1 Machine registers 15
5.1.1 State registers 15
5.1.2 Other machine registers 17
5.2 Process status and control bits 18
5.3 The process descriptor and its associated register fields 19

6 Instruction representation 21

6.1 Instruction encoding 21
6.1.1 An instruction component 21
6.1.2 The instruction data value and prefixing 21
6.1.3 Primary Instructions 22
6.1.4 Secondary instructions 22
6.1.5 Summary of encoding 23
6.2 Generating prefix sequences 23
6.2.1 Prefixing a constant 23
6.2.2 Evaluating minimal symbol offsets 24

7 Sequential operations 27

7.1 Registers 27
7.2 Local variables and constants, and stack operations 27
7.3 Integer stack evaluation 30

7.3.1 Loading operands 32
7.3.2 Tables of constants 33
7.3.3 Single length signed integer arithmetic 34
7.3.4 Single length modulo integer arithmetic 34
7.3.5 Unary minus 35
7.3.6 Fractional arithmetic 35
7.3.7 Bitwise logic and shifts 35
7.4 Non-local variables 36
7.5 Arrays and subscripts 36
7.5.1 Counting bytes and words 36
7.5.2 Forming addresses 37
7.5.3 Arrays 38
7.5.4 Transferring array elements 39
7.6 Multiple assignment 39
7.7 Comparisons and jumps 40
7.7.1 Representation of true and false 41
7.7.2 Comparisons 41
7.7.3 Implementation of languages with different representations of true and false 41
7.7.4 Boolean negation 41
7.7.5 Jump and conditional jump 42
7.7.6 Evaluation of boolean expressions 42
7.7.7 Conditional transfer of control 43
7.7.8 Compiling CASE statements 44
7.8 Long arithmetic and shifts 46
7.8.1 Multiple length addition and subtraction 46
7.8.2 Multiple length multiplication and division 47
7.8.3 Multiple length shifts 47
7.8.4 Normalizing 49
7.9 Object length conversion 49
7.9.1 Conversion between 8/16-bit object and word representations 49
7.9.2 Conversion between single word and double word representations 50
7.9.3 General conversion between N-bit object and word representations 50
7.10 Replication 51
7.11 Procedures 52
7.11.1 Adjusting workspace 52
7.11.2 Call and return 52
7.11.3 Use of (Wptr+0) 53
7.11.4 Loading parameters 53
7.11.5 The static chain 53
7.11.6 Other calling techniques 54
7.11.7 Other workspace allocation techniques 56
7.12 Functions 56
7.12.1 Calling a function 57
7.12.2 Single result functions 57
7.13 Error checking instructions 57
7.14 Device access instructions 59
7.15 Specialist instructions 61
7.15.1 Two dimensional block move 61
7.15.2 Bit manipulation and CRC evaluation 63

8 **Concurrent processes** **65**

8.1 Workspace 65
8.1.1 Process workspace data structure 65
8.1.2 Size of workspace 65
8.2 Scheduling and priority 66
8.2.1 The current process, the null process, and scheduling lists 66
8.2.2 Descheduling 67
8.2.3 Rescheduling after communication 67
8.2.4 Clocks and timeslicing 68
8.2.5 Priorities and interruption 68
8.2.6 Scheduling/descheduling of L-processes 69
8.3 Initiation and termination of processes 69
8.3.1 Scheduling parallel processes 70
8.3.2 Other scheduling instructions 73
8.4 Channel communication, synchronization and data-transfer 73
8.4.1 Channels 73
8.4.2 Synchronization 74
8.4.3 Communication 75
8.4.4 Implementation of channels 79
8.5 Time 83
8.5.1 Past and future 83
8.5.2 Reading the clock 84
8.5.3 Timer input 84
8.5.4 Timer lists 85
8.6 Semaphores 85
8.7 Alternative input 86
8.7.1 The occam ALT construct 86
8.7.2 The 'alternative sequence' 87
8.7.3 Execution of the alternative sequence 89
8.7.4 Compiling an ALT statement 91
8.7.5 Trapping degenerate alternatives 92
8.7.6 Replicated ALT 92
8.7.7 PRI ALT 93
8.8 Resource channels 93
8.8.1 The client-server model 93
8.8.2 Resource mechanism and data structures 95

9 **Protection and memory management** **97**

9.1 The mechanism 97
9.2 Instruction protection – privileged instructions 97
9.3 Address translation, memory protection, and stack extension 98
9.4 Regions 99
9.5 Region descriptors 101
9.6 Registers 102
9.7 Data structures 102
9.7.1 P-state data structure (PDS) 103
9.7.2 Region descriptor data structure (RDDS) 104
9.8 Instructions 105

10 The trap mechanism 107

10.1 The trap-handler 107
10.1.1 The THDS (trap-handler data structure) 107
10.1.2 Sharing a trap-handler data structure 110
10.1.3 The null trap-handler 110
10.2 State storage and retrieval when a trap is taken 110
10.3 Trap causes and signalling of errors 113
10.3.1 Trap causes 113
10.3.2 Signalling of errors 116
10.3.3 Null trap causes 121
10.4 Instructions 122

11 Floating-point instructions 125

11.1 IEEE floating-point arithmetic 125
11.2 The implementation of IEEE floating-point arithmetic on the IMS T9000 125
11.2.1 Formats 125
11.2.2 Floating-point operations 125
11.2.3 Exceptions 126
11.2.4 Not-a-Number representations (NaNs) 126
11.2.5 Implementation of underflow 127
11.3 Floating-point stack 127
11.4 Loading and storing floating-point values 128
11.4.1 Loading 128
11.4.2 Storing 129
11.5 Compiling floating-point expressions 130
11.6 Floating-point rounding mode 131
11.7 Floating-point arithmetic instructions 131
11.7.1 Dyadic operations 131
11.7.2 Monadic operations 132
11.8 Remainder and range instructions 133
11.9 Comparisons 136
11.9.1 Comparison instructions 136
11.9.2 Implementation of IEEE comparisons 137
11.9.3 Some anomalies 139
11.10 Class analysis 139
11.11 Type conversion 139
11.11.1 REAL to REAL conversions 140
11.11.2 REAL to INT conversions 140
11.11.3 INT to REAL conversions 141
11.12 Floating-point state 143
11.12.1 Floating-point status word 143
11.12.2 Saving the floating-point state 143
11.12.3 Instructions for saving and loading floating-point state 144
11.13 Exception handling mechanism 145
11.13.1 State delivered by floating-point exception – Implementing an IEEE (trap) handler 146
11.13.2 Some anomalies 147
11.14 Implementation of NaNs 148

12 Channels 151
12.1 Compilation and configuration of channels – an overview 151
12.2 External channels 152
12.2.1 Virtual channels 152
12.2.2 Byte-stream channels 155
12.2.3 Event channels 155
12.3 Channel states and modes of operation 157
12.3.1 Normal channel states 157
12.3.2 Resource channel states 157
12.3.3 Virtual and event channel activation modes 157
12.4 Channel configuration and mapping 157
12.4.1 Configuration register instructions 159
12.4.2 Configuration registers used for memory mapping 160
12.4.3 Virtual link mapping functions 160
12.4.4 Packet header labelling 164
12.5 Other configuration registers for setting up links and VCP 165
12.6 Setting up the virtual link control blocks 166
12.6.1 Instructions for setting up a VLCB 166
12.6.2 Null header 169
12.6.3 An example 169
12.7 Resource channels 173
12.7.1 Implementation of internal resource channels 173
12.7.2 Implementation of external resource channels 173
12.7.3 Reverse channel 174
12.7.4 Instructions for setting and using the resource mechanism 175
12.7.5 Usage of resource channels 178
12.8 Resetting and stopping a channel 182
12.8.1 Dealing with a communication failure 182
12.8.2 Recovering the use of a virtual channel which is in operation 184
12.9 Channel instructions according to usage 185
12.9.1 Instructions that can be applied to all channels 185
12.9.2 Instructions that can be applied to resource channels 186
12.9.3 Instructions that can be applied to external channels 186
12.9.4 Instructions that can be applied to virtual channels 188

13 Process state 189
13.1 Context switching 189
13.2 Partial context switch – descheduling and trapping 189
13.2.1 Descheduling and execution of the next process 189
13.2.2 Trapping 191
13.2.3 Instructions that are used to store and retrieve additional state 196
13.3 Full context switch – interruption 198
13.4 Restarting an interrupted process 200
13.5 Enabling and disabling interruption and timeslicing, and forcing a timeslice 202
13.6 Scheduling list and timer list queue manipulation 203

14 Debugging mechanisms 207
14.1 Breakpoints 207

14.2 Single-stepping ... 207
14.2.1 Single-stepping a P-process ... 207
14.2.2 Single-stepping an L-process ... 208
14.2.3 Early 'single-step' trap ... 208
14.3 Watchpoints ... 208
14.3.1 Watchpointing a P-process ... 209
14.3.2 Watchpointing an L-process ... 209
14.4 Single-stepping and watchpointing an L-process – some special considerations ... 209

15 Cache instructions ... 211
15.1 Workspace cache ... 212
15.2 Main Cache ... 212
15.3 Instructions ... 213

A T9000 instruction set reference guide ... 215
A.1 Introduction ... 215
A.1.1 Instruction name ... 215
A.1.2 Code ... 215
A.1.3 Description ... 215
A.1.4 Definition ... 216
A.1.5 Error signals ... 216
A.1.6 Comments ... 217
A.2 Notation ... 217
A.2.1 The transputer state ... 217
A.2.2 General ... 218
A.2.3 Undefined values ... 218
A.2.4 Data types ... 218
A.2.5 Representing memory ... 219
A.2.6 The configuration subsystem ... 220
A.2.7 Constants ... 220
A.2.8 Operators ... 224
A.2.9 Functions ... 227
A.2.10 Conditions to instructions ... 227
A.3 Instruction set definitions ... 229

B T9000 instruction set sorted by op-code ... 455
B.1 Primary functions ... 455
B.2 Secondary functions ... 455
B.2.1 Instructions encoded without using prefix ... 455
B.2.2 Instructions encoded using prefix ... 456
B.2.3 Instructions encoded using negative prefix ... 460

Instruction index ... 463

Index ... 465

List of tables

Table 3.1 Definition of errors signalled by the IMS T9000 8
Table 5.1 Registers that define the machine state – state registers 16
Table 5.2 Extra registers that define the P-process state – state registers 17
Table 5.3 Register group names 17
Table 5.4 Other machine registers 18
Table 5.5 Process status and control bits 19
Table 6.1 Prefixing instruction components 21
Table 6.2 Primary instructions 22
Table 7.1 Instructions that can be used for loading, storing and manipulating the integer stack 28
Table 7.2 Register loading sequences 33
Table 7.3 Single length signed integer arithmetic instructions 34
Table 7.4 Single length modulo integer arithmetic instructions 34
Table 7.5 Special instruction for fixed point arithmetic fractional multiply 35
Table 7.6 Bitwise logic and shift instructions 35
Table 7.7 Non-local load and store instructions 36
Table 7.8 Instructions that provide processor wordlength characteristics 36
Table 7.9 Subscript addressing instructions 37
Table 7.10 instruction that performs block transfer 39
Table 7.11 Comparisons and conditional behavior instructions 41
Table 7.12 Long arithmetic instructions 46
Table 7.13 Long arithmetic instructions 47
Table 7.14 Long shift instructions 48
Table 7.15 Instruction for normalizing double length value 49
Table 7.16 Instructions used for converting between bytes, 16-bit objects and words 49
Table 7.17 Instructions for conversion between single word and double word representation 50
Table 7.18 Instructions for conversion between word and N-bit object representation 50
Table 7.19 Conditional replication instruction 51
Table 7.20 Loop end data structure 51
Table 7.21 Instructions for implementing procedures 52
Table 7.22 Instructions that may explicitly signal IntegerError 58
Table 7.23 Device access instructions 60
Table 7.24 Two dimensional block transfer instructions 61
Table 7.25 Instructions that perform bit manipulation and CRC evaluation 63
Table 8.1 Word offsets and names for data slots in a L-process workspace 65
Table 8.2 List of descheduling points 67
Table 8.3 Instructions which relate to the timer and timeslicing 68
Table 8.4 Timeslicing points 68
Table 8.5 List of interruptible instructions 69
Table 8.6 Instructions for starting and terminating processes 70
Table 8.7 Parallel process data structure 70
Table 8.8 Fixed length i/o operation instructions 76
Table 8.9 Variable length i/o operation instructions 77

Table 8.10 Single byte transfer instructions 79
Table 8.11 Instructions which use the on-chip clocks 83
Table 8.12 Word offsets and names for data slots in a semaphore data structure 86
Table 8.13 Semaphore operation instructions 86
Table 8.14 Instructions required to implement an alternative sequence 87
Table 8.15 Resource data structure (RDS) 96
Table 8.16 Resource channel data structure 96
Table 9.1 Region addresses 99
Table 9.2 Registers used by protection mechanism 102
Table 9.3 P-state data structure (or PDS) 103
Table 9.4 Region descriptor data structure 105
Table 9.5 Instructions used for switching to and from protected mode 105
Table 10.1 Trap-handler data structure (or THDS) 108
Table 10.2 General trap causes and their abbreviations 113
Table 10.3 Trap causes that can occur simultaneously 114
Table 10.4 Bit mapping of trap reason delivered in Areg 115
Table 10.5 Possible trap reasons as loaded into Areg when a is taken 116
Table 10.6 Definition of errors signalled by the IMS T9000 117
Table 10.7 Flags and the error signals that cause them to be set 118
Table 10.8 Error trap enable bits and error signals that cause a trap when bits are set 118
Table 10.9 Effect of signalling errors 119
Table 10.10 Possible error types as loaded into Breg when a trap is taken 120
Table 10.11 Instructions used in conjunction with trap-handling 122
Table 11.1 Signals raised when an IEEE exceptional condition is detected 126
Table 11.2 Instructions which are used to directly manipulate the floating-point stack 128
Table 11.3 Floating-point load instructions 128
Table 11.4 Instructions for loading floating-point zero 129
Table 11.5 Floating-point store instructions 129
Table 11.6 Rounding mode setting instructions 131
Table 11.7 Arithmetic instructions with two floating-point operands 131
Table 11.8 Floating-point load and operate instructions 132
Table 11.9 Arithmetic instructions with one floating-point operand 132
Table 11.10 Signals raised and result of fpsqrt instruction 133
Table 11.11 Floating-point remainder and range reduction instructions 133
Table 11.12 Signals raised and result of fprem instruction 134
Table 11.13 Signals raised and result of fprange instruction 135
Table 11.14 Floating-point comparison instructions 136
Table 11.15 Results of comparison instructions for all possible relations 137
Table 11.16 Signals raised by comparison instructions, for various operand conditions 137
Table 11.17 Code sequences that should be used to implement IEEE comparisons 138
Table 11.18 Class analysis instructions 139
Table 11.19 Real to real conversion instructions 140
Table 11.20 Real to integer conversion instructions 140

Table 11.21	Integer to real conversion instructions	141
Table 11.22	The format of the floating-point status word	143
Table 11.23	The floating-point rounding mode field values	143
Table 11.24	The floating-point type field values	143
Table 11.25	Floating-point state instructions	144
Table 11.26	Floating-point state data structure	145
Table 11.27	Effect of signals raised due to detection of exceptional conditions	145
Table 11.28	Exceptional conditions and error types	146
Table 11.29	Quiet NaNs generated when FPInvalidOp is signalled but no trap is taken	148
Table 11.30	Behavior of monadic floating-point operations for NaN input	149
Table 11.31	Behavior of dyadic floating-point operations for NaN inputs	149
Table 12.1	Configuration instructions	159
Table 12.2	Bit fields in registers VCPlink0–3Mode	165
Table 12.3	Bit fields in the VCPcommand register	165
Table 12.4	Instructions used for setting up and manipulating the VLCB	166
Table 12.5	Meaning of value loaded into Creg by readbfr	168
Table 12.6	Instructions used for resource mechanism	176
Table 12.7	Bit settings for virtual/event channel status in Areg	187
Table 12.8	Bit settings for byte-stream channel status in Areg	188
Table 13.1	State register values when a process is descheduled	190
Table 13.2	Register values after loading state for execution of process	190
Table 13.3	State register values before and after an L-process trap	191
Table 13.4	State register values before and after a tret instruction	192
Table 13.5	State register values before and after a supervisor trap	193
Table 13.6	State register values before and after a goprot instruction	194
Table 13.7	Instructions for storing/retrieving extra state	196
Table 13.8	Block move data structure	197
Table 13.9	Shadow register instructions	198
Table 13.10	Instructions for restarting interrupted processes	201
Table 13.11	Timeslice and interrupt instructions	202
Table 13.12	Instructions used to manipulate scheduling and timer lists	204
Table 15.1	Cache instructions	213
Table A.1	Loop end data structure	220
Table A.2	Word offsets and names for data slots in a L-process workspace	220
Table A.3	Parallel process data structure	220
Table A.4	Word offsets and names for data slots in a semaphore data structure	221
Table A.5	Resource data structure (RDS)	221
Table A.6	Resource channel data structure	221
Table A.7	P-state data structure (or PDS)	221
Table A.8	Region descriptor data structure	222
Table A.9	Trap-handler data structure (or THDS)	222
Table A.10	Floating-point state data structure	222
Table A.11	Block move data structure	223

Table A.12 Constants used in the instruction descriptions 223
Table A.13 Constants used within the T9000 224
Table A.14 Device identity values 224
Table A.15 Operators used in the instruction descriptions 226
Table B.1 Instructions encoded as primary functions 455
Table B.2 Instructions encoded without using prefix 456
Table B.3 Instructions encoded using prefix 460
Table B.4 Instructions encoded using negative prefix 461

1 Introduction

This book describes each instruction in the IMS T9000 instruction set and explains the context within which that instruction is used. It is essentially divided into two parts: a narrative that introduces each instruction within a logical group of instructions, and a reference section that gives a code-like specification of each instruction with a cross reference to the narrative.

[For details of the T9000 products and development tools, refer to *The T9000 Hardware Reference Manual*.]

This is a useful document to all transputer users, but it is aimed in particular at the following.

- the high-level programmer who wants to use low-level code inserts to enhance the performance of his code
- the compiler writer
- the operating system or run-time kernel writer
- the writer of high-level debugging tools
- the writer of run-time support libraries
- the writer of bootstrap code

The high-level programmer in a language such as C might find that, either his particular compiler is limited in certain respects, or a particular piece of code is time critical, and may hence need to write a low-level code sequence. Provided that the compiler/linker system that he is using enables him to write instruction level sequences into his program, he can overcome such difficulties. For this though he needs a good understanding of the capabilities and range of the entire instruction set.

A compiler writer needs to understand the exact action of each instruction in order to write the code generation part of his program.

The writer of a run-time kernel needs to understand the transputer's scheduling mechanism and the instructions which enable him to implement his own scheduling/interrupt scheme.

Similarly the writer of run time support libraries may need to write time critical code at assembly level.

The writer of a debugging tool needs to be able to control low-level context switching (e.g. for implementing breakpoints/single-step) and needs to be able to access and manipulate certain registers and data structures.

Bootstrap code must usually be very compact code and requires some machine specific instructions which are not available in high-level programming languages. For these reasons, a bootstrap program is written at assembly code level and the writer of the program needs to be familiar with the processor's instruction set.

In the narrative part of the book, the art of programming the IMS T9000 is considered subject by subject. Each subject introduces certain instructions. This provides the reader with the purpose for the instructions as they are introduced and describes any environmental issues, such as data structures and register/data-structure pre-conditions. Some of these subjects are of interest to all IMS T9000 users while some are of specific interest to certain categories of reader.

Chapters 3 to 8 should be read by all transputer programmers who are likely to need to know anything about the instruction set. The basic concepts of the IMS T9000 are introduced, including: addressing, instruction representation, processes, registers, communication, and the instructions which are essential for sequential and concurrent programming. These chapters may also be of interest to high-level programmers or system designers who would like a general background on how the IMS T9000 works.

Chapters 9 and 10 are concerned with running code under protection, implementing a memory management scheme, and handling errors or unexpected behavior. They are of primary interest to operating system writers.

Chapter 11 describes the support for floating-point arithmetic and is of interest to compiler writers, for implementing mathematical run-time library support, and to programmers who are required to write IEEE floating-point exception-handlers.

Chapter 12 discusses how channels are used to communicate between transputer processes. In particular, it describes virtual channels, event channels and resource channels. This information is needed if writing a program to configure a network of transputers (a configurer).

Chapter 13 overviews the various context switches and associated state storage and retrieval mechanisms. These include traps, high-priority process interruption (and return), descheduling and timeslicing. Instructions are described which enable process queues to be manipulated and interrupts/timeslices to be enabled/disabled.

Chapter 14 discusses the various mechanisms available for monitoring the behavior of a process as it runs, including: breakpointing, watchpointing, single-stepping. The instructions and mechanisms here are of interest to the programmer implementing debugging tools.

Chapter 15 provides a brief overview of the memory architecture and describes the instructions that can be used to invalidate and flush the main cache.

Appendix A is the instruction reference section. It lists all the IMS T9000 instructions in alphabetical order. For each instruction, there is a short English description, a pseudo-code description, and list of pre-conditions, a list of any conditions which may be set by the instruction, and cross reference to the page(s) in the narrative section of the book where the instruction is introduced.

Appendix B provides a tabulated list of all the instructions in ascending order of operation codes. This list is useful for disassembly of instruction code. That is, it provides the user with the information needed to convert from a hex code sequence to an instruction code sequence.

2 Notation, conventions and terminology

This chapter introduces and explains some of the conventions, terminology and special notations used throughout the book.

Instructions

The mnemonic and full name for each IMS T9000 instruction are given when the instruction is first introduced. All succeeding references to instructions use the mnemonic which is written in italic font and lower case – e.g. *stl n*.

Registers

The registers referenced in this text are those described in section 5.1. Usually a reference to a register uses its textual description. For example, **ThReg** is referred to as the 'trap-handler register'. Sometimes however this would be unwieldy and the shortened name is given (e.g. **Areg** is usually used in preference to the integer stack A-register). Registers are written in bold font with an initial capital letter.

Undefined values

In the definition of some instructions the values left in certain registers are said to be undefined. This means that those values are implementation dependent, and are not even guaranteed to be consistent within an implementation. No application should attempt to make use of the value that any version of a transputer implementation happens to provide.

Constants

Some special constant names are used throughout the book. A full list of these is given in tables A.12 and A.13 (appendix A). They are written in italic font – e.g. *BytesPerWord*.

Bits

Where a particular bit value is established by the machine, the text usually explicitly states the value to which it is set. For example 'the trap-handler in use bit is set to 1'. Occasionally however, where this becomes cumbersome, the convention is used that the word 'set' implies 'set to 1' and the words 'reset' or 'cleared' imply 'set to 0'.

Data structures

Some instructions use data structures for storage and retrieval of data. Data structures are introduced in the text as required, but the entire set of data structures is duplicated in appendix A. The following explains the conventions used to describe a data structure.

A data structure within the context of this document is a contiguous block of store that comprises a number of 'slots'. A slot normally contains a machine word. The address of the structure is a pointer to a location in the vicinity of this block. Each slot has a defined word offset from that address. In a description of the data structure, a name for each slot is provided as well as an offset. Slots are then referred to by name. Slot names are written in bold font.

For example, the process workspace data structure is presented in section 5.3. Given this, the phrase 'the **pw.Iptr** slot of the process workspace data structure' refers to the slot addressed by the process workspace (data structure) offset by –1 word (i.e. the process workspace byte address minus 4).

Program notation

The occam language is used in this book both as a 'source language' to represent program constructs and program fragments to be compiled, and as a 'meta-language' to represent algorithms to produce compiled code and other examples. These two uses of occam are distinguished by the use of an italic font for meta-language occam as in

x := a + b

and a teletype font for source language occam as in

```
x := a + b
```

A detailed knowledge of occam is not required. Unfamiliar constructs and operations are explained before use. However, an important semantic difference between occam and most other high level languages, is that statements may be executed sequentially or concurrently. This is shown syntactically by use of SEQ and PAR constructs.

For example, the two statements in

```
SEQ
  a := a + 1
  b := a + b
```

are executed sequentially. This is implicitly assumed in most high level programming languages, in which a group of statements (without jumps) are executed in the order in which they are encountered. This compares with the occam code

```
PAR
  a := a + 1
  b := b + 1
```

in which the two statements are executed concurrently.

More generally

```
SEQ
  S_pred
  PAR
    P_1
    P_2
    ...
    P_n
  S_succ
```

means that the processes which are enclosed by the PAR construct (P_1, P_2 ... P_n) are run concurrently, but as a whole are run sequentially between the execution of S_{pred} and S_{succ}. In practice the processes may not be able to execute simultaneously (unless there is more than one CPU), and so a looser specification of the PAR construct is: none of the processes within the construct may start until the predecessor process (S_{pred}) has been executed, and successor process (S_{succ}) may not execute until all the processes within the construct have terminated. Note that this does not say anything about when or where these processes are executed.

Note that an italic font is used for 'meta-variables' inside source language – e.g. the processes P_i in the above.

The source language occam is the occam 2 language as defined in the *occam 2 Reference Manual*. The meta-language occam is based on occam 2 with some restrictions removed and extensions added to enable certain algorithms to be expressed more simply.

Footnotes

Footnotes are used occasionally to provide more detail and background information to avoid breaking the flow of the main text.

Where there is just a single reference to a footnote, the footnote is positioned at the bottom of the page on which that reference is made. It is referenced using a symbol superscript such as: †, ‡, †, ‡.

Where there is more than one reference to a footnote, the footnote is positioned at the end of the chapter in which the references are made. It is referenced using a numeric superscript.

3 An overview

This chapter gives an overview of the IMS T9000 and introduces some of the concepts used throughout the book. No detailed explanation is given because each of these topics is discussed in later chapters.

The IMS T9000 transputer is a 32-bit microprocessor with on-chip hardware support for floating-point arithmetic. As such a single unit can be used as a very fast sequential microprocessor. However, there are a number of advanced features which characterize the IMS T9000.

- A hardware scheduling mechanism means that the processor can appear to run many processes concurrently. At a high level, this can be achieved either with an operating system kernel, or with a language which has a model of concurrency (e.g. occam, parallel C).
- On-chip serial communication engines (data links) make it easy for connected IMS T9000 processors to communicate data and implement true concurrency.
- A separate communications processor multiplexes messages across these serial links. Any number of messages can be simultaneously transmitted to any number of external destination processes, and the means by which this is achieved is transparent to the programmer.
- An easy to use trap-handling mechanism provides for error detection and debugging, and supports IEEE floating-point exception handling.
- There is a workspace cache which allows very fast (single-cycle) access to local variables.
- There is on-chip memory which can be used as cache, fast static RAM or a combination of the two. This gives accelerated access to non-local code or data.
- A pair of control links is provided for transfer of control and boot information. These are entirely independent of the data links, hence enhancing system reliability.
- An internal pipelined architecture means that instruction sequences can be grouped so that several operations can occur simultaneously.

This chapter introduces some of these features and the concepts that utilize them.

3.1 Processes

This section firstly describes the occam process model, and secondly introduces the IMS T9000 implementation of processes.

A single transputer can efficiently implement the occam process model, by sharing processor time between concurrent processes. Also a network of transputers can be used to implement the occam process model with improved performance, by running communicating processes on different transputers.

Note that although the transputer has been carefully designed to implement the occam model of processes, the user is not restricted to this process model. For example communication between processes on the same IMS T9000 tranputer can be achieved via shared variables protected by semaphores.

3.1.1 The occam process model

In the occam process model, a process starts, performs a number of actions, and then either stops or terminates successfully. Each action is either an assignment, an input or an output. An assignment sets the value of a variable, an input receives a value from a channel, and an output sends a value to a channel. The variable set by an assignment should not be accessible to any other process — the only method of transferring information from one process to another should be by using a channel.

At any time between it starting and terminating successfully a process may be ready to communicate on one or more of its channels. Each channel provides one way communication between two processes.

Communication is synchronized. If a channel is used for input in one process and output in another then communication takes place when both processes are ready. The inputting and outputting processes then proceed with the value output being copied from the outputting process to the inputting process.

Externally a process may be seen as being a 'black box' that, after starting, may or may not wish to communicate along one or more of its channels until it terminates successfully. In order to perform the task it is designed to achieve, a correctly functioning process normally communicates data with the processes connected to it, and then terminates successfully. However, a process can indefinitely fail to communicate. This failure of communication can be due to internal deadlock (where all internal processes are waiting to communicate with each other), internal livelock (where internal processes are only communicating with themselves and never communicate with the outside world) or due to the process ceasing to execute without terminating successfully (in occam this is the STOP process).

The internal state of a process is not visible to the outside world and all interactions with the process occur via channel communication. This process model removes the problems associated with variable sharing.

3.1.2 Implementation of processes

A single IMS T9000 can implement the occam model of a process, and can simulate concurrent execution of a number of such processes.

At any point in time a transputer can only be executing a single process. This is referred to as the 'current' process. The current process usually executes for a finite period of time. It ceases to be the current process if it is descheduled (i.e. stops executing).

The following may cause the process to deschedule.

- Execution has reached a point where the process must wait for action by another process (e.g. a communication or semaphore).
- Execution has reached a point where the process must wait for a timer.
- The process has had its fair share of execution time and must temporarily give way to another process. When this occurs, the process is said to have been 'timesliced'.
- The process has executed an instruction which explicitly forces the current process to be stopped or terminated, or forces another process to start executing.

Sometimes the current process must give way temporarily to a more urgent (higher priority) process which requires execution. This is referred to as an 'interrupt'.

Because more than one process may be ready to execute, the transputer provides scheduling lists. Whenever a process becomes ready to execute, it is placed at the end of a scheduling list. When this occurs, the process is said to be 'scheduled'. The processor automatically timeslices the execution of scheduled processes, and so over a period of time, this gives the appearance of more than one process running simultaneously. The set of processes that is either executing, interrupted or on a scheduling list, is referred to as the 'active set'.

Two scheduling lists are maintained by the transputer: a list of low priority processes and a list of high priority processes. When the processor deschedules the current process, it loads the state of the process at the front of the high priority scheduling list, or if there are no high priority processes scheduled, it loads the state of the process at the front of the low priority scheduling list. It then starts to execute that process. No low priority process can be executed until the high priority list is empty.

By default, whenever there is a process on the high priority list, any currently executing low priority process is interrupted. However a low priority process can explicitly disable interruption and timeslicing on the IMS T9000. A high priority process cannot be timesliced or interrupted.

The processor can monitor a process for certain error conditions, debug conditions etc. and can optionally cause a context switch or cause the machine to halt. When this occurs, the process is said to have taken a 'trap'.

A T9000 process can run in trusted mode or in protected mode. In trusted mode, it is called an L-process, and in protected mode, it is called a P-process. A process is always an L-process when it is loaded for execution. It may become a P-process while it is executing, but always reverts to an L-process before it is descheduled. An L-process can implement local trap-handling, by having a trap-handler associated with it. The trap-handler is discussed later, but amongst other things, it contains status and control bits for the L-processes which use it.

The capacity to run a process under protection (in protected mode) enables the programmer to write-protect and/or execute-protect regions of memory. When a process is running in protected mode, the L-process that invoked the mode is referred to as the 'supervisor' (or 'stub') of the P-process which is currently executing. The protected mode also provides memory mapping, by mapping logical addresses to a physical address. Some of the IMS T9000 instructions cannot be executed under protection. These are called 'privileged instructions'.

3.2 Communication

Processes communicate via channels. A channel is a unidirectional point-to-point medium of communication, which may be between processes running on the same transputer (internal), between processes on different transputers (external), or between a process on a transputer and an external device. Internal communication is achieved via memory transfer. External communication is achieved via special purpose on-chip hardware, known as (data) links.

The instructions used for communication are the same, regardless of whether the channel is internal or external; the channel address is used by the processor to determine the action to be performed. This allows a procedure to be compiled without knowledge of whether its parameter channels are implemented by memory locations or by external links.

Channel communication incorporates synchronization and data-transfer. Synchronization ensures that data-transfer takes place only when both the inputting and outputting processes are ready. When one of the two processes wants to communicate (using an input or output instruction), it must wait until the second process is also ready.

There are three synchronization mechanisms available when using channels on the IMS T9000.

- Simple synchronization – This is the simplest form of synchronization. The first process to attempt communication (input or output) is descheduled and the processor starts to execute the next process on the scheduling list. When the second process becomes ready (performs an output or input), the channel is ready for data-transfer.
- Alternative synchronization – For processes that want to select an input communication from several possibilities, the IMS T9000 provides a series of instructions known as the 'alternative sequence'. In this mechanism one of several possible communication channels is selected for data-transfer.
- Resource synchronization – For a client-server model, the IMS T9000 provides a resource mechanism, whereby certain channels can be set to 'resource mode'. A queue of clients can thus be associated with each resource (server). The mechanism ensures that channel communications to a particular server are selected one at a time. When a channel is selected, it is ready for data-transfer. A channel which can be set to resource mode is called a 'resource channel'.

When communication has completed, the processor ensures that both communicating processes are scheduled.

For processes running on the same processor, there is also an n-valued semaphore mechanism provided on the IMS T9000 .

3.3 Traps

A trap is an unexpected change of flow of execution, which can occur in any process, due to the occurrence of various conditions. The user may specify for some of these conditions whether or not a trap is taken.

A trap from an L-process normally results in a context switch to a trap-handler, and a trap from a P-process results in a context switch to its supervisor.

The conditions that can cause a trap include:

- debug conditions – The IMS T9000 provides facilities to insert breakpoints in code, employ a single-step mode, and set up watchpointing for access to a specified area of memory.
- errors and IEEE floating-point (fp) exceptional conditions
- system call – A trap may be forced by explicitly execution of an instruction (*syscall*).
- timeslice – A timeslice causes a trap when running under protection.

Detection of these conditions facilitates the implementation of error-handlers, debugging tools, IEEE exception-handlers, operating systems etc.

The errors (and floating-point exceptions) which can be signalled are shown in table 3.1. All of these 'error signals' cause a trap unless this facility is disabled by the user. Some of them set flags in the status register if a trap is not taken. This is discussed fully in chapter 10.

error signal	brief description
IntegerOverflow	integer overflow or integer divide-by-zero
IntegerError	integer error other than *IntegerOverflow* – e.g. explicitly checked or explicitly set error, misuse of channel
Unalign	address of instruction operand is not aligned to the correct boundary
IllegalInstruction	attempt to execute an illegal instruction
PrivInstruction	attempt to execute a privileged instruction in protected mode
AccessViolation	attempt to access a memory protected or non-existent address
FPError	floating-point 'error'
FPInvalidOp	IEEE floating-point 'invalid operation'
FPDivideByZero	IEEE floating-point 'divide by zero'
FPOverflow	IEEE floating-point 'overflow'
FPUnderflow	IEEE floating-point 'underflow'
FPInexact	IEEE floating-point 'inexact result'

Table 3.1 Definition of errors signalled by the IMS T9000

Associated with an L-process is a 'trap-handler pointer'. This is either the address of a trap-handler data structure (THDS) or is a null pointer. A THDS may be shared with a number of other L-processes and contains: status information for the processes that use it, control information which specifies the conditions that cause a trap, and identification of the trap-handler code which is executed when a trap occurs from one of these processes.

Associated with a P-process is a pointer to a P-state data structure (PDS). This contains: status information for the P-process and control information which specifies the conditions that cause a trap.

3.4 Configuration of control system

There are two additional links on the IMS T9000, that are used to send and receive hardware control information. These 'control links' make it possible for a controlling processor to control and monitor a series of IMS T9000-family devices in a subsystem connected by the control links. This control network is entirely independent of the data network which is connected by data links, and thus guarantees the integrity of the control system.

There is a set of commands that is used for transfer of control information. It includes commands to: reset a processor, read or write data to a specified address, and transmit boot-code. When one of these commands is sent on a control link, it is preceded by header information to identify the destination processor.

A full description of the IMS T9000 control system and the associated command set is given in *The T9000 Hardware Reference Manual*.

3.5 Instructions and pipelining

The CPU of the IMS T9000 has a superscalar pipelined micro-architecture, which enables concurrent execution of instructions on a single transputer. A group of instructions can be issued to this pipeline on every transputer cycle, and since a large proportion of the IMS T9000 instructions require only one cycle to complete execution, this means that on average several instructions are executed in each cycle.

There is also a mechanism (grouper) which examines the sequence of instructions and divides these into optimal groups for concurrent execution. This 'grouper' takes account of which operations can occur concurrently in the pipeline.

The existence of the grouper means that the programmer need not have any knowledge of the pipelined architecture. It is entirely transparent to the user. He does not have to specify the groups or the order in which they are executed.

4 Addressing and data representation

The IMS T9000 transputer is a 32-bit word machine, with byte addressing and a 4 Gbyte address space. This chapter explains how data is loaded from and stored into that address space, explains how signed arithmetic is represented, and defines the arithmetic significance to ordering of data items.

4.1 Word address and byte selector

A machine address is a single word of data which identifies a byte in memory – i.e. a byte address. It comprises two parts, a word address and a byte selector. The byte selector occupies the two least significant bits of the word; the word address the thirty most significant bits. An address is treated as a signed value, the range of which starts at the most negative integer and continues, through zero, to the most positive integer. This enables the standard comparison functions to be used on pointer (address) values in the same way that they are used on numerical values.

Certain values can never be used as pointers because they represent reserved addresses at the bottom of memory space. They are reserved for use by the processor and initialization. In this text, names are used to represent these and other values (e.g. *NotProcess.p*, *Disabling.p*). A full list of names and values of constants used in this book is given in tables A.13 and A.12 (appendix A).

4.2 Ordering of information

The transputer is 'little-endian' — i.e. less significant data is always held in lower addresses. This applies to bits in bytes, bytes in words and words in memory. Hence, in a word of data representing an integer, one byte is more significant than another, if its byte selector is the larger of the two. Figure 4.1 shows the ordering of bytes in words and memory for the IMS T9000. Note that this ordering is compatible with Intel processors, but not Motorola or SPARC.

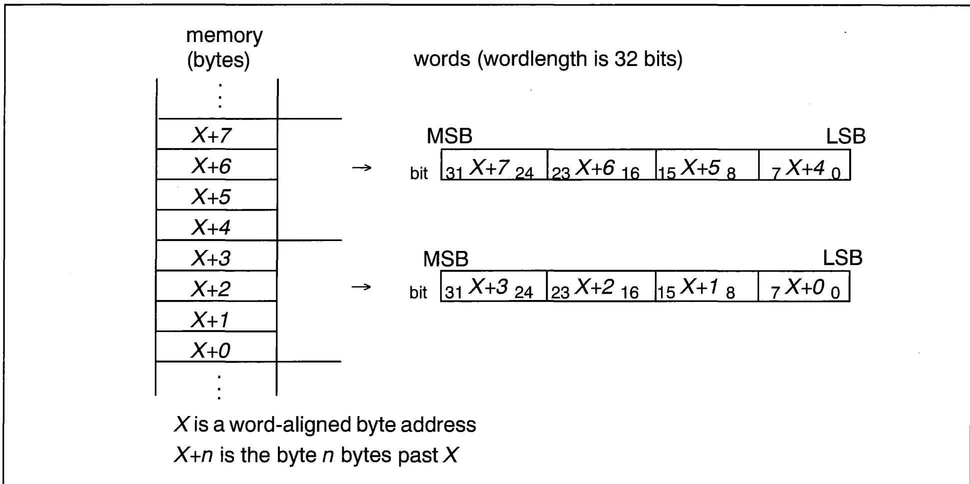


Figure 4.1 Bytes in memory and words

4.3 Words, objects and signed integers

Most instructions that involve fetching data from or storing data into memory, use word aligned addresses (i.e. bits 1 and 0 are set to 0) and load or store four contiguous bytes. However, there are some instructions that can manipulate part of the bit pattern in a word, and a few that use double words.

A data item that is represented in two contiguous bytes, is referred to as a16-bit object. This can be stored, either in the least significant 16-bits of a word location, or in the most significant 16 bits, hence addresses of such locations are16-bit aligned (i.e. bit 0 is set to 0).

A data item that is represented in in two contiguous words, is referred to as a 64-bit object or a double word.

Similarly, a data item represented in a single byte is sometimes referred to as an 8-bit object.

Signed integers and sign extension

A signed integer is stored in twos-complement format and may be represented by an N-bit object. Most commonly a signed integer is represented by a single word (32-bit object), but as explained, it may be stored, for example, in a 64-bit object, a 16-bit object, or an 8-bit object. In each of these formats, all the bits within the object contain useful information.

Consider the example shown in figure 4.2, which shows how the value –10 is stored in a 32-bit register, firstly as an 8-bit object and secondly as a 32-bit object. Observe that bits 31 to 8 are meaningful for a 32-bit object but not for an 8-bit object. These bits are set to 1 in the 32-bit object to preserve the negative sign of the integer being represented.

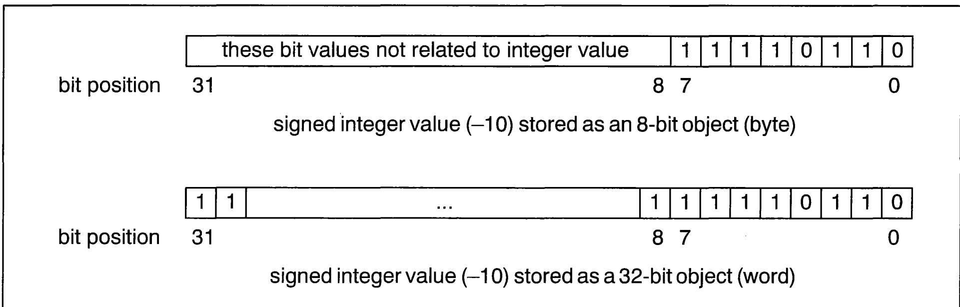


Figure 4.2 Storing a signed integer in different length objects

The length of the object that stores a signed integer can be increased (i.e. the object size can be increased). This operation is known as 'sign extension'. The extra bits that are allocated for the larger object, are meaningful to the value of the signed integer. They must therefore be set to the appropriate value. The value for all these extra bits is in fact the same as the value of the most significant bit – i.e. the sign bit – of the smaller object. The IMS T9000 provides instructions that sign extend to 32 bits and 64 bits. These are described in chapter 7.

4.4 Unaligned address detection

Instructions that address words or 16-bit objects expect the addresses to be aligned on 4-byte and 2-byte boundaries respectively. This requires bits 0 and 1 to be set to 0 for an address operand of a word instruction or bit 0 to be set to 0 for an address operand of a 16-bit object instruction. If the processor detects a bad address alignment while executing an instruction, then it signals *Unalign*.

The user can specify the action that the processor takes when *Unalign* is signalled: either

- a trap is taken, or
- the address is re-aligned by setting the bottom bit to '0' for a 16-bit object, or the bottom two bits to '0' for a word.

See chapter 10 for details on trap enabling.

The word instructions to which this applies are:–

ldnl	*stnl*	*chantype*	*devlw*	*devsw*
disc	*disg*	*enbc*	*enbg*	*endp*
erdsq	*gajw*	*goprot*	*grant*	*in*
initvlcb	*insphdr*	*irdsq*	*ldchstatus*	*ldresptr*
ldshadow	*lend*	*mkrc*	*out*	*outbyte*
outword	*readbfr*	*readhdr*	*resetch*	*restart*
selth	*setchmode*	*sethdr*	*signal*	*startp*
stmove2dinit	*stopch*	*stresptr*	*stshadow*	*swapbfr*
unmkrc	*vin*	*vout*	*wait*	*writehdr*
fpb32tor64	*fpi32tor32*	*fpi32tor64*	*fpldall*	*fpldnladddb*
fpldnladdsn	*fpldnldb*	*fpldnldbi*	*fpldnlmuldb*	*fpldnlmulsn*
fpldnlsn	*fpldnlsni*	*fpstall*	*fpstnldb*	*fpstnli32*
fpstnlsn				

The16-bit object instructions to which this applies are:–

devls	*devss*	*ls*	*lsx*	*ss*

5 Registers, status bits and control bits

5.1 Machine registers

This section introduces the IMS T9000 registers. The registers listed here are those visible to the programmer. Firstly the set of registers known as state registers are presented and discussed. These fully define the state of the executing process. Secondly the other registers of interest to the programmer, are presented. The function of each register is detailed later in the book as each register is first encountered in the context of a machine instruction.

5.1.1 State registers

The state of an L-process at any instant is defined by the contents of the machine registers listed in table 5.1. For a P-process the state also includes the contents of the registers listed in table 5.2. These registers may be referred to as 'state registers' to distinguish them from registers that are not part of the process state. The 'register' column gives the abbreviated name of the register. The 'full name / description' column provides the full textual name which is usually used when referencing a register in this manual; and where unclear, a brief description of the information contained in this register. The 'shadow register' column states the abbreviated name of the associated shadow register (explained below).

register	full name / description process modes	shadow register
StatusReg	status register	**StatusReg.sh**
WdescReg	workspace descriptor register – contains the process descriptor of the currently executing process	**WdescReg.sh**
IptrReg	instruction pointer register – pointer to next instruction to be executed	**IptrReg.sh**
Areg	integer stack register A	**Areg.sh**
Breg	integer stack register B	**Breg.sh**
Creg	integer stack register C	**Creg.sh**
ThReg†	trap-handler register – pointer to the current THDS (trap-handler data structure)	**ThReg.sh**
FPstatusReg	floating-point status register – indicates type of value in each FP register and the current rounding mode	**FPstatusReg.sh**
FPAreg	floating-point stack register A	**FPAreg.sh**
FPBreg	floating-point stack register B	**FPBreg.sh**
FPCreg	floating-point stack register C	**FPCreg.sh**
BMreg0	2D block move control register 0	**BMreg0.sh**
BMreg1	2D block move control register 1	**BMreg1.sh**
BMreg2	2D block move control register 2	**BMreg2.sh**
WlReg	watchpoint lower bound register	**WlReg.sh**
WuReg	watchpoint upper bound register	**WuReg.sh**
Ereg‡	internal register	**Ereg.sh**
Xreg‡	internal register	**Xreg.sh**
EptrReg	error pointer register	**EptrReg.sh**

† contains trap-handler pointer of supervisor when executing a P-process

‡ **Ereg** and **Xreg** are internal registers which the processor uses to store temporary values during the execution of some instructions. The processor saves their contents when an instruction is interrupted and restores the registers to their previous state when it is restarted. The actual data values are of no use to the programmer.

Table 5.1 Registers that define the machine state – state registers

RegionReg0	region descriptor register 0 – contains the region descriptor for protection region 0	**RegionReg0.sh**
RegionReg1	region descriptor register 1 – contains the region descriptor for protection region 1	**RegionReg1.sh**
RegionReg2	region descriptor register 2 – contains the region descriptor for protection region 2	**RegionReg2.sh**
RegionReg3	region descriptor register 3 – contains the region descriptor for protection region 3	**RegionReg3.sh**
PstateReg	protected state register – pointer to the current PDS (P-state data structure)	**PstateReg.sh**
WdescStubReg	workspace descriptor stub register – contains the process descriptor of the supervisor	**WdescStubReg.sh**

Table 5.2 Extra registers that define the P-process state – state registers

Hence depending on the mode in which the process is currently executing, the process state is defined by a slightly different set of registers. For example the region descriptor registers are listed in table 5.2 because their contents may be valid while a P-process is executing, but their contents are irrelevant when an L-process is executing.

Register grouping

It is convenient for use now and elsewhere in this manual to refer to certain related registers by a group name. These names are intended to be intuitive, but table 5.3 clarifies.

register group name	**registers include in group**
integer stack (registers)	**Areg**, **Breg**, **Creg**
floating-point stack (registers)	**FPAreg**, **FPBreg**, **FPCreg**
stack (registers)	integer stack registers, floating-point stack registers
floating-point registers	floating-point stack registers, **FPstatusReg**
block move registers	**BMreg0**, **BMreg1**, **BMreg2**
watchpoint registers	**WlReg**, **WuReg**
region descriptor registers	**RegionReg0**, **RegionReg1**, **RegionReg2**, **RegionReg3**
internal registers	**Ereg**, **Xreg**

Table 5.3 Register group names

Shadow registers

When a high priority process interrupts a low priority process, the state of the currently executing process needs to be saved. For this purpose, a 'shadow register' is provided for each state register. On interrupt, the content of each register is copied into its shadow. On return from an interrupt the state of the interrupted process is copied back from the shadow registers. The content of the shadow registers is therefore only valid during an interrupt. A high priority process may manipulate the shadow registers with the instructions *ldshadow* and *stshadow*. Details of these are provided in section 13.3.

5.1.2 Other machine registers

There are several other registers which the programmer needs to know about. These are not part of the process state, and hence are not state registers and do not have shadow registers. They are presented in table 5.4.

register	full name / description
FptrReg0	high priority front pointer register – contains pointer to first process on the high priority scheduling list
FptrReg1	low priority front pointer register – contains pointer to first process on the low priority scheduling list
BptrReg0	high priority back pointer register – contains pointer to last process on the high priority scheduling list
BptrReg1	low priority back pointer register – contains pointer to last process on the low priority scheduling list
ClockReg0	high priority clock register – contains current value of high priority clock
ClockReg1	low priority clock register – contains current value of low priority clock
TptrReg0	high priority timer list pointer register – contains pointer to the first process on the high priority timer list
TptrReg1	low priority timer list pointer register – contains pointer to the first process on the low priority timer list
TnextReg0	high priority alarm register – contains the time of the next process on the high priority timer queue
TnextReg1	low priority alarm register – contains the time of the next process on the low priority timer queue

Table 5.4 Other machine registers

5.2 Process status and control bits

The status register (**StatusReg**) contains status and control information for the current process. The 32-bit word held in the status register comprises 'status bits' ('flags') and 'control bits'. Status bits describe current state, such as the mode of operation (protected/unprotected) and any errors which may have occurred. Control bits specify future behavior which may occur, such as trapping and timeslicing.

The process status and control bits are shown in table 5.5. The meaning and function of each bit is detailed later in the book when each bit is first encountered in the context of a machine instruction. Bits within the status register that are not shown in table 5.5, have reserved use.

- Prior to executing an L-process, the processor loads these bits from the THDS (trap-handler data structure) for that process into the status register. When this process is descheduled or trapped, the processor writes these bits back from the status register into the THDS. Hence these bits are local to L-processes which are associated with a particular trap-handler.
- The situation is similar for a P-process, but in this case, the bits are loaded from and written to the PDS (P-state data structure). The status and control bits here are local to P-processes which share a particular supervisor.

bit number	status bit name	full name / description
2	**sb.FPErrorFlag**	floating-point error flag – indicates, when set, that a floating-point error has occurred
3	**sb.FPErrorTeBit**	floating-point error trap enable bit – specifies, when set, that if a floating-point error occurs, the current process will be trapped
6	**sb.IntOvFlag**	integer overflow flag
7	**sb.IntOvTeBit**	integer overflow trap enable bit
8	**sb.FPInOpFlag**	floating-point invalid operation flag
9	**sb.FPInOpTeBit**	floating-point invalid operation trap enable bit
10	**sb.FPDivByZeroFlag**	floating-point divide by zero flag
11	**sb.FPDivByZeroTeBit**	floating-point divide by zero trap enable bit
12	**sb.FPOvFlag**	floating-point overflow flag
13	**sb.FPOvTeBit**	floating-point overflow trap enable bit
14	**sb.FPUndFlag**	floating-point underflow flag
15	**sb.FPUndTeBit**	floating-point underflow trap enable bit
16	**sb.FPInexFlag**	floating-point inexact result flag
17	**sb.FPInexTeBit**	floating-point inexact result trap enable bit
19	**sb.UnalignTeBit**	unaligned address trap enable bit
25	**sb.TimesliceDisabledBit**	timeslice disable bit – specifies, when set, that timeslicing cannot occur
26	**sb.StepBit**	single-stepping trap enable bit – specifies, when set, that a single-step trap will be taken at the end of the current instruction
27	**sb.IsPprocessBit**	protection bit – indicates, when set, that the processor is running under protection
29	**sb.WtchPntEnbl**	watchpoint trap enable bit – specifies, when set, that watchpointing is enabled
30	**sb.WtchPntPend**	watchpoint trap pending flag – indicates, when set that a watchpoint trap will be taken at the end of the current instruction

Table 5.5 Process status and control bits

Sometimes in this manual it is stated that under certain circumstances, the status register is loaded with the 'default control word'. This means that all bits in the status register are set to 0 with the exception of bit 7, **sb.IntOvTeBit**, which is set to 1. In particular, the default control word is loaded when an L-process executes with a null trap-handler.

5.3 The process descriptor and its associated register fields

In order to identify a process completely it is necessary to know: its workspace address (in which the byte selector is always 0), and its priority (high or low). This information is contained in the process descriptor. The process descriptor of the currently executing process is held in the workspace descriptor register (**WdescReg**).

The workspace descriptor register is formed from a pointer to the process workspace or-ed with the priority indicator at bit 0. Often it is required to use or update only some part of the workspace descriptor register, so two 'register fields': **Wptr** and **Priority** are defined so that the following invariants are obeyed.

$$\textbf{Wptr} = \textbf{WdescReg} \wedge \textbf{(NOT 3)}$$

$$\textbf{Priority} = \textbf{WdescReg} \wedge \textbf{1}$$

Wptr points to the current process workspace, which is always word-aligned. **Priority** is the priority of the currently executing process where the value 1 indicates low priority and 0 indicates high priority. Bit 1 is always set to 0†.

workspace address	0	priority
31 ... 2	1	0

Figure 5.1 Constituents of a process descriptor

† In preliminary revisions of the IMS T9000 (*viz* versions α and β), bit 1 may be set to 1.

6 Instruction representation

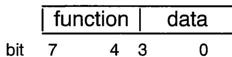
The transputer encoding is designed so that the most commonly executed instructions occupy the least number of bytes. This chapter describes the encoding mechanism and explains how it achieves this.

A sequence of single byte 'instruction components' is used to encode an instruction. The IMS T9000 interprets this sequence at the instruction fetch stage of execution. Most users (working at the level of microprocessor assembly language) need not be aware of the existence of instruction components and do not need to think about the encoding. The first section (6.1) has been included to provide a background. The following section (6.2) need only concern the reader that wants to implement a code generator.

6.1 Instruction encoding

6.1.1 An instruction component

Each instruction component is one byte long, and is 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 used to build an 'instruction data value'.



The representation provides for sixteen instruction components (one for each function), each with a data field ranging from 0 to 15.

There are three categories of instruction component. Firstly there are those that specify the instruction directly in the function field. These are used to implement 'primary instructions'. Secondly there are the instruction components that are used to extend the instruction data value – this process of extension is referred to as 'prefixing'. Thirdly there is the instruction component *operate* (*opr*) which specifies the instruction indirectly using the 'instruction data value'. *opr* is used to implement 'secondary instructions'.

6.1.2 The instruction data value and prefixing

The data field of an instruction component is used to create an 'instruction data value'. Primary instructions interpret the instruction data value as the operand of the instruction. Secondary instructions interpret it as the operation code for the instruction itself.

The instruction data value is a signed integer that is represented as a 32-bit word. For each new instruction sequence, the initial value of this integer is zero. Since there are only 4 bits in the data field of a single instruction component, it is only possible for most instruction components to initially assign an instruction data value in the range 0 to 15. However two instruction components are used to extend the range of the instruction data value. Hence one or more prefixing components may be needed to create the correct instruction data value. These are shown in table 6.1 and explained below.

mnemonic	name
pfix	prefix
nfix	negative prefix

Table 6.1 Prefixing instruction components

All instruction components initially load the four data bits into the least significant four bits of the instruction data value.

pfix loads its four data bits into the instruction data value, and then shifts this value up four places. *nfix* is similar, except that it complements the instruction data value† before shifting it up. Consequently, a sequence of one or more prefixes can be included to extend the value. Instruction data values in the range –256 to 255 can be represented using one prefix instruction.

† Note that it inverts *all* 32 bits of the instruction data value.

When the processor encounters an instruction component other than *pfix* or *nfix*, it loads the data field into the instruction data value but doesn't shift it because the instruction encoding is now complete and the instruction can be executed. When the processor is ready to fetch the next instruction component, it starts to create a new instruction data value.

6.1.3 Primary Instructions

Research has shown that computers spend most of the time executing instructions such as: instructions to load and store from a small number of 'local' variables, instructions to add and compare with small constants, and instructions to jump to or call other parts of the program. For efficiency therefore, the transputer encodes these directly as primary instructions using the function field of an instruction component.

Thirteen of the instruction components are used to encode the most important operations performed by any computer executing a high level language. These are used (in conjunction with zero or more prefixes) to implement the primary instructions. Primary instructions interpret the instruction data value as an operand for the instruction. The mnemonic for a primary instruction will therefore normally include a this operand – *n* – when referenced.

The mnemonics and names for the primary instructions are listed in table 6.2. Their usage and behavior are given in chapter 7.

mnemonic	**name**
adc n	add constant
ajw n	adjust workspace
call n	call
cj n	conditional jump
eqc n	equals constant
j n	jump
ldc n	load constant
ldl n	load local
ldlp n	load local pointer
ldnl n	load non-local
ldnlp n	load non-local pointer
stl n	store local
stnl n	store non-local

Table 6.2 Primary instructions

6.1.4 Secondary instructions

The transputer encodes all other instructions (secondary instructions) indirectly using the instruction data value.

mnemonic	**name**
opr	operate

The remaining instruction component – *opr* – causes the instruction data value to be interpreted as the operation code of the instruction to be executed. This selects an operation to be performed on the values held in the integer or floating-point stacks. This allows a further 16 operations to be encoded in a single byte instruction. However the prefix instructions can be used to extend the instruction data value, allowing any number of operations to be performed.

Secondary instructions do not have an operand specified by the encoding, because the instruction data value has been used to specify the operation.

To ensure that programs are represented as compactly as possible, the operations are encoded in such a way that the most frequent secondary instructions are represented without using prefix instructions.

6.1.5 Summary of encoding

The encoding mechanism has important consequences.

- Firstly, it simplifies language compilation, by providing a completely uniform way of allowing a primary instruction to take an operand of any size up to the processor word-length.
- Secondly, it allows these operands to be represented in a form independent of the word-length of the processor.
- Thirdly, it enables any number of secondary instructions to be implemented.

The following provides some simple examples of encoding:–

- The instruction *ldc 17* is encoded with the sequence

  pfix 1; ldc 1

- The instruction *add* is encoded by

  opr 5

- The instruction *and* is encoded by

  opr 46

  which is in turn encoded with the sequence

  pfix 2; opr 14

To aid clarity and brevity, prefix sequences and the use of *opr* are not explicitly shown in this guide. Each instruction is represented by a mnemonic, and for primary instructions an item of data, which stands for the appropriate instruction component sequence. Hence in the above examples, these are just shown as: *ldc 17*, *add*, and *and*. (Also, where appropriate, an expression may be placed in a code sequence to represent the code needed to evaluate that expression.)

6.2 Generating prefix sequences

Generating a prefix sequence to create an instruction data value is extremely tedious — especially when the value is negative. Prefixing is intended to be performed by a compiler (or assembler). Prefixing by hand is not advised!

Normally a value can be loaded into the instruction data value by a variety of different prefix sequences. It is important to use the shortest possible sequence as this enhances both code compaction and execution speed. The best method of optimizing object code so as to minimize the number of prefix instructions needed is shown below.

6.2.1 Prefixing a constant

The algorithm to generate a constant instruction data value *e* for a function *op* is described by the following recursive function.

prefix(op, e) = IF

$e < 16 \ AND \ e \geq 0$

$$
\begin{array}{l}
\quad op(e) \\
e \geq 16 \\
\quad prefix(pfix, e \gg 4);\ op(e \wedge \#F) \\
e < 0 \\
\quad prefix(nfix, (\sim e) \gg 4);\ op(e \wedge \#F)
\end{array}
$$

where *op(e)* is the instruction component with function code *op* and data field *e*, $\sim$ is a bitwise NOT, and $\gg$ is a logical shift right.

6.2.2 Evaluating minimal symbol offsets

Several primary instructions have an operand that is an offset between the current value of the instruction pointer and some other part of the code. Generating the optimal prefix sequence to create the instruction data value for one of these instructions is more complicated. This is because two, or more, instructions with offset operands can interlock so that the minimal prefix sequences for each instruction is dependent on the prefixing sequences used for the others.

For example consider the interlocking jumps below which can be prefixed in two distinct ways. The instructions *j* and *cj* are respectively *jump* and *conditional jump*. These are explained in more detail later. The sequence

cj +16; j –257

can be coded as

pfix 1; cj 0; pfix 1; nfix 0; j 15

but this can be optimized to be

cj 15; nfix 15; j 1

which is the encoding for the sequence

cj +15; j –255

This is because when the two offsets are reduced, their prefixing sequences take 1 byte less so that the two interlocking jumps will still transfer control to the same instructions as before. This compaction of non-optimal prefix sequences is difficult to perform and a better method is to slowly build up the prefix sequences so that the optimal solution is achieved. The following algorithm performs this.

1. Associate with each jump instruction or offset load an 'estimate' of the number of bytes required to code it and initially set them all to 0.
2. Evaluate all jump and load offsets under the current assumptions of the size of prefix sequences to the jumps and offset loads
3. For each jump or load offset set the number of bytes needed to the number in the shortest sequence that will build up the current offset.†
4. If any change was made to the number of bytes required then go back to 2 otherwise the code has reached a stable state.

The stable state that is achieved will be the optimal state.

Steps 2 and 3 can be combined so that the number of bytes required by each jump is updated as the offset is calculated. This does mean that if an estimate is increased then some previously calculated offsets may

† Where the code being analyzed has alignment directives, then it is possible that this algorithm will not reach a stable state. One solution to this, is to allow the algorithm to increase the instruction size but not allow it to reduce the size. This is achieved by modifying stage 3 to choose the larger of: the currently calculated length, and the previously calculated length. This approach does not always lead to minimal sized code, but it guarantees termination of the algorithm.

have been invalidated, but step 4 forces another loop to be performed when those offsets can be corrected.

By initially setting the estimated size of offsets to zero, all jumps whose destination is the next instruction are optimized out.

Knowledge of the structure of code generated by the compiler allows this process to be performed on individual blocks of code rather than on the whole program. For example it is often possible to optimize the prefixing in the code for the sub-components of a programming language construct before the code for the construct is optimized. When optimizing the construct it is known that the sub-components are already optimal so they can be considered as an unshrinkable block of code.

This algorithm may not be efficient for long sections of code whose underlying structure is not known. If no knowledge of the structure is available (e.g. in an assembler), all the code must be processed at once. In this case a code shrinking algorithm where in step one the initial number of bytes is set to twice the number of bytes per word is used. The prefix sequences then shrink on each iteration of the loop. 1 or 2 iterations produce fairly good code although this method will not always produce optimal code as it will not correctly prefix the pathological example given above.

7 Sequential operations

This chapter introduces and describes the instructions that perform sequential operations within a process – i.e. instructions that do not depend on or control any other processes. It also describes the integer stack architecture, and explains how expressions can be evaluated on this stack. (For details on the floating-point stack and associated instructions, refer to chapter 11.)

7.1 Registers

The following registers are referred to in sequential operations

IptrReg	contains a pointer to next instruction to be executed – the instruction pointer
Areg	integer stack register A
Breg	integer stack register B
Creg	integer stack register C

as well as the following register field

Wptr	contains a pointer to current process workspace – the workspace pointer

Wptr is used as a base from which the local variables of a process can be addressed (see section 5.3).

Integer stack

Areg, **Breg** and **Creg** are organized as a three word stack. Instructions that load **Areg**, push **Breg** into **Creg** and **Areg** into **Breg**. Instructions that store **Areg**, pop **Breg** into **Areg** and **Creg** into **Breg**, leaving **Creg** undefined. The effects of this are shown in figures 7.1 and 7.2.

Before			**After**		
		push x onto stack			
Areg	=	*a*	**Areg**	=	*x*
Breg	=	*b*	**Breg**	=	*a*
Creg	=	*c*	**Creg**	=	*b*

Figure 7.1 Effect of pushing value onto integer stack

Before			**After**		
					a popped off stack
Areg	=	*a*	**Areg**	=	*b*
Breg	=	*b*	**Breg**	=	*c*
Creg	=	*c*	**Creg**	=	*undefined*

Figure 7.2 Effect of popping value from integer stack

7.2 Local variables and constants, and stack operations

The instructions shown in table 7.1 include: instructions for loading and storing local workspace values (local variables), instructions for loading and storing bytes and 16-bit objects, and instructions for manipulating the values in the integer stack.

mnemonic	name
ldc n	load constant
ldl n	load local
stl n	store local
ldlp n	load local pointer
lb	load byte
lbx	load byte and sign extend
sb	store byte
ls	load sixteen
lsx	load sixteen and sign extend
ss	store sixteen
rev	reverse
pop	pop processor stack
dup	duplicate top of stack

Table 7.1 Instructions that can be used for loading, storing and manipulating the integer stack

The most common operations performed by a program are loading and storing one of a small number of variables, and loading small literal values. The *ldc n* instruction pushes the operand† *n* onto the integer stack. This enables values between 0 and 15 to be loaded into the integer stack using a single byte instruction.

The *ldl n*, *stl n* and *ldlp n* instructions all address words in memory relative to the workspace pointer **Wptr**. The first 16 locations can be identified using a single byte instruction. A local variable held in workspace location *n* can be pushed onto the integer stack by

ldl n

and the address of that variable can be pushed by

ldlp n

The value of the variable can be set to a value popped from the stack by

stl n

Note that for the purposes of this text, *ldl X* denotes loading the value from a local variable X. Similarly *ldlp X* denotes loading the address of a local variable X, and *stl X* denotes storing a value into a local variable X. In all three cases, the operand X can be can be interpreted as the appropriate constant offset from **Wptr** that will enable these primary instructions to locate the actual stored location of a local variable X.

lb and *lbx* load the byte at the address in **Areg,** into the integer stack. *lb* replaces the address in **Areg** with the byte stored at that address, treating it as as an unsigned integer by setting the twenty-four most significant bits in **Areg** to 0. *lbx* is similar to *lb*, but treats the byte as a signed integer in twos-complement format, and hence sign extends the representation by setting the twenty-four most significant bits in **Areg** to the same value as the most significant bit of the byte (see section 4.3). **Breg** and **Creg** are unaffected by these operations.

sb writes the least significant byte in **Breg** to the location addressed by **Areg**. It pops **Creg** up into **Areg,** leaving **Breg** and **Creg** undefined.

† Where 'operand' is used in the text, this refers to the instruction data value, which is constructed as described in section 6.1.2. In summary, for an operand between 0 and 15, this can be coded into the data field of a single instruction component (one byte). Operands outside this range can be coded using *pfix* or *nfix* beforehand.

ls and *lsx* load the 16-bit object at the address in **Areg,** into the integer stack. *ls* replaces the address in **Areg** with the 16-bit object stored at that address, treating it as as an unsigned integer by setting the sixteen most significant bits in **Areg** to 0. *lsx* is similar to *ls*, but treats the 16-bit object as a signed integer in twos-complement format, and hence sign extends the representation by setting the sixteen most significant bits in **Areg** to the same value as the most significant bit of the 16-bit object. **Breg** and **Creg** are unaffected by these operations.

ss writes the 16-bit object in **Breg** (two least significant bytes) to the location addressed by **Areg.** It pops **Creg** up into **Areg**, leaving **Breg** and **Creg** undefined.

rev swaps the contents of **Areg** and **Breg**.

pop forces the integer stack to be popped as shown in figure 7.2.

dup takes a copy of the content of **Areg** and pushes this onto the integer stack, hence leaving two identical values in **Areg** and **Breg**.

Single word and byte assignment

Single words, 16-bit objects, and bytes may be assigned using the load and store instructions.

Word assignment

If *x* and *y* are both single word variables and *e* is a word valued expression then compiled code for word assignments

x := *y* = *ldl y; stl x*
x := *e* = *e; stl x*

Byte assignment

If *a* and *b* are both single byte variables and *e* is a byte valued expression then compiled code for byte assignments are

b := *a* = *address(a); lb; address(b); sb*
b := *e* = *e; address(b); sb*

where *address(variable)* is discussed below.

Address calculation

In the previous example, the function *address(variable)* is introduced. This will be used in subsequent examples to represent the instruction sequence required to push onto the integer stack, the address of the memory location used to store *variable*. How this function is compiled depends on a number of things such as: the locality of the variable (local/non-local), the size of the variable, and the packing of the variable. If a word variable is in local workspace, then its address can be loaded into the stack using

e.g. *address(variable)* = *ldlp variable*

This can also be used to obtain the address of local byte variables if each byte variable is word-aligned. But if byte variables are packed contiguously in memory, then their addresses can be obtained in the same way as addresses of byte array elements. This and similar addressing issues are discussed in section 7.5.2.

Use of (Wptr+0)

The location (**Wptr**+0) is used as a temporary store by certain instructions. These are

outword, *outbyte*, *altwt*, *taltwt*, *disc*, *dist*, *disg*, *diss*, and *altend*.

If any of these instructions are being used, then local variables should be allocated from (**Wptr**+1) rather than (**Wptr**+0). See also section 8.1.1.

7.3 Integer stack evaluation

Integer expression evaluation and address calculation is performed using the integer stack. (The floating-point stack can be used for floating-point arithmetic – see chapter 11.) For example, the evaluation of operators with two integer operands is performed by instructions that operate on the values of **Areg** and **Breg**. The result is left in **Areg**, and **Creg** is popped into **Breg** leaving **Creg** undefined.

A compiler loads a constant expression *C* using

ldc C

(or by loading from a constant table – see section 7.3.2). It loads an expression consisting of a single local variable using

ldl x

Methods for loading non-local variables, array elements and function calls are given in later sections.

Evaluation of expressions sometimes requires the use of temporary variables in the process workspace, but the number of these can be minimized by careful choice of the evaluation order.

Let *depth(e)* be the number of stack locations needed for the evaluation of expression *e*, defined by

```
depth(constant)        =   1
depth(variable)        =   1
depth(function call)   =   'infinite'
depth(e1 op e2)        =   IF
                             depth(e1) > depth(e2)
                               depth(e1)
                             depth(e1) < depth(e2)
                               depth(e2)
                             TRUE
                               depth(e1) + 1
```

That is, if the depth required for each expression is the same, then one extra stack location is required to store the result of the first expression, while the second expression is being evaluated. If the stack requirements for each expression are different, then the total stack requirement is the larger stack requirement of the individual expressions. This is only the case if care is taken over the order of evaluation. Note that *'infinite'* should be taken as meaning greater than any finite depth – because a function call does not preserve values on the stack (for further explanation, see section 7.12.2).

Let the function *eval(e, r)* evaluate expression *e* where there are *r* registers available to perform the evaluation. Where this expression is an operation on two sub expressions – *e1 op e2* – it is efficiently evaluated by the following algorithm, where *commutes(op)* is *true* if *op* commutes and *false* otherwise, and *max(v1, v2)* is the larger value of *v1* and *v2*.

```
IF
  max( depth(e1), depth(e2) ) < r    — i.e. depth of both expressions is less than 'r' – the
                                     — number of registers available
    IF
      depth(e2) > depth(e1)
        IF
          commutes(op)
            eval( e2, r ); eval( e1, r–1 ); op
          TRUE
            eval( e2, r ); eval( e1, r–1 ); rev; op                                  (1)
      [depth(e2) ≤ depth(e1)]
        eval( e1, r ); eval( e2, r–1 ); op
  max( depth(e1), depth(e2) ) ≥ r
    IF
      depth(e2) ≥ depth(e1)
        IF
          depth(e1) ≥ r    — i.e. both depths ≥ r
            eval( e2, r ); stl temp; eval( e1, r ); ldl temp; op
          TRUE    — i.e. (depth(e1) < r) AND depth(e2) ≥r
            IF
              commutes(op)
                eval( e2, r ); eval( e1, r–1 ); op
              TRUE    i.e. operation doesn't commute
                eval( e2, r ); eval( e1, r–1 ); rev; op
      depth(e2) < depth(e1)
        IF
          depth(e2) ≥ r    — i.e. both depths ≥ r
            IF
              commutes(op)
                eval( e1, r ); stl temp; eval( e2, r ); ldl temp; op
              TRUE
                eval( e2, r ); stl temp; eval( e1, r ); ldl temp; op
          TRUE    — i.e. (depth(e2) < r) AND depth(e1) ≥r
            eval( e1, r ); eval( e2, r–1 ); op
```

where *(I1; I2; . . . ; In)* represents a sequence of instructions.

The justification of this is as follows. If the depth of both expressions is less than the number of registers available (*r*), then there is no need to store the result of the first evaluation in a temporary variable. The deeper expression is evaluated first to ensure that the operation evaluates in the least number of stack registers. If this were not done then the total depth requirement would have to be incremented due to the extra location for storing the result of the first evaluation. If both expression depths are as great as *r*, then a local variable (*temp*) must be used to store the result of the first expression. It is again better to evaluate the expression with the larger depth if possible, because this minimizes the number of local variables required. If only one of the expressions is as great as *r*, then provided that expression is evaluated first, there is no need to store its result in a local variable.

In the cases where a temporary variable *temp* is required to hold the value of the first expression in the evaluation of *e1 op e2*, then that variable can be used as a temporary variable in the evaluation of first expression. Also a temporary variable used in the evaluation of first expression and not used to hold its result can be used in the evaluation of second expression.

The code sequence

(e2; e1; rev; op)

at (1) in the above algorithm, can be optimized further to

(e1; e2; op)

removing the execution of the *rev* instruction. But be aware that the latter uses an extra stack register, and so there is trade-off here between evaluation depth and code size.

7.3.1 Loading operands

The three registers of the integer stack are used to hold operands of instructions, and the first three parameters of procedure calls. Evaluation of an operand or parameter may involve the use of more than one register. Care is needed when evaluating such operands to ensure that the first operand to be loaded is not pushed off the bottom of the integer stack by the evaluation of later operands. The processor does not detect stack overflow.

Three registers are available for loading the first operand, two registers for the second and one for the third. Consequently, the instructions are designed so that **Creg** holds the operand which — on average — is the most complex, and **Areg** the operand which is the least complex.

In some cases, it is necessary to evaluate the **Areg** and **Breg** operands in advance, and to store the results in temporary variables. This can sometimes be avoided using the reverse instruction. The following sequences may be used to load the operands *A*, *B* and *C* into **Areg**, **Breg** and **Creg**.

1 *C; B; A*

2 *C; A; B; rev*

3 *B; C; rev; A*

4 *A; C; rev; B; rev*

The choice of loading sequence, and of which operands should be evaluated in advance is determined by the number of registers required to evaluate each of the operands. In particular, if *C* requires more than two registers it must be loaded before *A* and *B*. If *A* or *B* requires more than two registers it must be evaluated before *C* and may need to be stored in a temporary variable if *C* requires more than two registers.

registers required			temp		load sequence	instructions
C	B	A	b	a		
≤2	1	1			1	C; B; A
	1	2			2	C; A; B; rev
	1	>2			4	A; C; rev; B; rev
	2	1			1	C; B; A
	2	2		*	1	A; stl a; C; B; ldl a
	2	>2		*	1	A; stl a; C; B; ldl a
	>2	1			3	B; C; rev; A
	>2	2		*	3	A; stl a; B; C; rev; ldl a
	>2	>2		*	3	A; stl a; B; C; rev; ldl a
> 2	1	1			1	C; B; A
	1	2			2	C; A; B; rev
	1	>2		*	1	A; stl a; C; B; ldl a
	2	1			1	C; B; A
	2	2		*	1	A; stl a; C; B; ldl a
	2	>2		*	1	A; stl a; C; B; ldl a
	>2	1	*		1	B; stl b; C; ldl b; A
	>2	2	*		2	B; stl b; C; A; ldl b; rev
	>2	>2	*	*	1	A; stl a; B; stl b; C;ldl b; ldl a

Table 7.2 Register loading sequences

Table 7.2 gives the instruction sequences needed for loading three operands into the integer stack, where the number in the 'load sequence' column refers to the list above. The columns labelled 'temp' indicate where a local variable is needed to temporarily save an operand value in the load sequence. The variable 'a' is used to store operand 'A' and the variable 'b' is used to store operand 'B'.

7.3.2 Tables of constants

The transputer instruction set has been optimized so that the loading of small constants can be coded compactly — for example it allows the loading of constants between 0 and 15 to be coded in a single byte. Analysis of programs shows that such small constants occur markedly more frequently than large constants. However when a large constant does need to be loaded the necessary prefix sequence may be long. Other techniques may be more efficient in these cases.

A simple mechanism to increase the code compactness is to use a table of constants. This is implemented by storing all the long constants into a lookup table. The address of this table is held in a local variable which is used to index the array — this table and all its constant entries must be aligned on a word boundary. Then to load the constant from the nth entry in the constant table stored at address *constants_ptr* the following code would be used

ldl constants_ptr; ldnl n

where the instruction *ldnl n* is explained in section 7.4.

This code sequence only takes 2 bytes, provided *constants_ptr* is less than 16 words from the workspace pointer address and there are no more than 16 wordlength constants. At worse it is unlikely to take more than 4 bytes. Hence, if a constant takes 4 or more bytes to load using *ldc* then this sequence often improves code compactness — especially if the constant is used more than once.

7.3.3 Single length signed integer arithmetic

Single length arithmetic with error (overflow†) checking is provided by the operations listed in table 7.3. Each of these instructions signal *IntegerOverflow* (see section 10.3.2) if the operation overflows or a divide by zero is attempted.

mnemonic	name
adc n	add constant
add	add
sub	subtract
mul	multiply
div	divide
rem	remainder

Table 7.3 Single length signed integer arithmetic instructions

The primary instruction *adc* allows a constant value *c* (the instruction operand) to be added to **Areg** by *adc c*. **Breg** and **Creg** are unaffected.

The instruction sequence

ldl X; ldl Y; op

where *op* is one of: *add*, *sub*, *mul*, *div*, *rem*, evaluates the expression

X op Y

i.e. it takes the value in **Breg** as the lefthand operand and the value in **Areg** as the righthand operand, and loads the result into **Areg**. The content of **Creg** is popped into **Breg** leaving **Creg** undefined. Of these, *add* and *mul* are commutative.

If *adc*, *add*, *sub* or *mul* causes overflow, then *IntegerOverflow* is signalled, and the result is truncated so as to fit into 32 bits (but see note 1 at the end of the chapter).

The result of *div* is the integer division rounded towards zero (truncated). If it causes overflow, then *Integer-Overflow* is signalled and the result is undefined. Overflow can occur only if the divisor (**Areg**) is zero, or if the dividend (**Breg**) is *MostNeg* and the divisor is –1.

The result of *rem* is the remainder of the integer division of the two operands. The sign is always the same as the dividend (**Breg**), regardless of the sign of the divisor (**Areg**). If the divisor is 0 then *IntegerOverflow* is signalled and the result is undefined.

7.3.4 Single length modulo integer arithmetic

Single length arithmetic (with overflow ignored) is provided by

mnemonic	name
sum	sum
diff	difference
prod	product

Table 7.4 Single length modulo integer arithmetic instructions

The results of *sum*, *diff* and *prod* are the same as *add*, *sub* and *mul* respectively, but *IntegerOverflow* is never signalled. Of these, *sum* and *prod* are commutative.

† In general, 'overflow' is said to have occurred if the actual result of an operation cannot be represented by its destination type.

7.3.5 Unary minus

The expression *(–e)* can be evaluated with overflow checking by

e; not; adc 1

or

ldc 0; e; sub

The first sequence, using *not*, requires one less stack register than the second [*not* is a bitwise inversion which is fully defined in subsection 7.3.7.]. However the second sequence will execute significantly faster on the IMS T9000.

The expression can be evaluated without overflow checking by

ldc 0; e; diff

7.3.6 Fractional arithmetic

Many applications, such as scientific function evaluation, use fixed point arithmetic. To enable this to be performed efficiently on the IMS T9000, a fractional multiply instruction is included in the instruction set.

mnemonic	**name**
fmul	fractional multiply

Table 7.5 Special instruction for fixed point arithmetic fractional multiply

fmul is a commutative arithmetic operator that interprets **Areg** and **Breg** as fixed point numbers lying in the range $-1 \leq x < 1$. The value associated with each register is 2^{-31} times its signed integer value. *fmul* returns the rounded fixed point product of these values in **Areg** and pops **Creg** up into **Breg**. The rounding is performed in *Round-to-Nearest* mode as in ANSI/IEEE 754–1985 arithmetic.

Attempting (–1) *fmul* (–1) produces an undefined result and signals *IntegerOverflow*, as +1 cannot be represented in this format — this is the only case in which *fmul* can overflow.

7.3.7 Bitwise logic and shifts

Bitwise logic and shift operations are provided by the instructions listed in table 7.6.

mnemonic	**name**
and	and
or	or
xor	exclusive or
not	bitwise not
shl	shift left
shr	shift right

Table 7.6 Bitwise logic and shift instructions

The *not* operation has only one operand that is taken from **Areg**. The result of this, which is a bitwise inversion of all bits in the operand, is loaded into **Areg**, leaving **Breg** and **Creg** unaffected.

and, *or* and *xor* have two operands that are taken from **Areg** and **Breg**. For each, the result, which is a bitwise logical operation on the two operations, is loaded into **Areg**. The data previously held in **Creg**, is loaded into **Breg**, leaving **Creg** undefined. These operations are commutative.

The shift operations (*shl* and *shr*) shift the operand in **Breg** by the number of bits specified by the unsigned integer in **Areg**† and put the result in **Areg**. The vacated bit positions are filled with zero bits. If **Areg** is zero, the result is the initial value of **Breg**. When the value in **Areg** is greater than the number of bits in the object being shifted, the result of the operation is zero. **Areg** can be checked to signal *IntegerError* on such out of range shifts using the *csub0* operation which is described later (section 7.13). The data previously held in **Creg**, is loaded into **Breg**, leaving **Creg** undefined.

7.4 Non-local variables

The 'local' operations *ldl n*, *stl n* and *ldlp n* address words in memory relative to **Wptr**. This is useful for accessing local variables. For accessing non-local variables, a level of indirection is required.

mnemonic	name
ldnl n	load non-local
stnl n	store non-local
ldnlp n	load non-local pointer

Table 7.7 Non-local load and store instructions

The primary instructions listed in table 7.7 perform 'non-local' operations in a similar way to their 'local' counterparts except that they access a word address relative to **Areg** rather than **Wptr**. This base address and the word offset specified by the operand of the instruction (*n*), form the non-local address which specifies the location referenced by these load and store instructions.

ldnlp n loads the non-local address into **Areg**. *ldnl n* loads the content of the location specified by the non-local address into **Areg.** For both these instructions, **Breg** and **Creg** are unaffected. *stnl n* writes the content of **Breg** into the location specified by the non-local address. It also pops **Creg** into **Areg** leaving **Breg** and **Creg** undefined. *ldnl* and *stnl* signal *Unalign* if the address in **Areg** is not word-aligned.

An element in an array is accessed by calculating the offset of the element from the base address of that array. These non-local load and store instructions can therefore be used on non-local data structures.

7.5 Arrays and subscripts

The addressing instructions provide access to items in data structures using short sequences of single byte instructions. They also allow the representation of data structure access to be independent of the wordlength of the processor. (The latter characteristic is not important if writing code specifically for the IMS T9000, but may be of importance when considering portability to transputers that have a different wordlength.)

7.5.1 Counting bytes and words

mnemonic	name
bcnt	byte count
wcnt	word count

Table 7.8 Instructions that provide processor wordlength characteristics

The *bcnt* instruction multiplies **Areg** by the number of bytes in a word (i.e. by 4 for the IMS T9000). It is particularly used for producing the length in bytes of a multiword data item. The result is loaded into **Areg**, leaving **Breg** and **Creg** unaffected.

The *wcnt* instruction enables an address to be decomposed into its component word address and byte selector as defined in section 4.1. *wcnt* takes an address in **Areg** and returns the word address in **Areg**

† Note that the time that the IMS T9000 takes to execute these instruction is constant, whereas on the T2/T4/T8-series transputer, the time is proportional to the value in **Areg**.

and the byte selector in **Breg**, where the word address is sign extended (i.e. bits 31 and 30 are set to the same binary value as bit 29). The value previously held in **Breg**, is pushed into **Creg**.

7.5.2 Forming addresses

mnemonic	name
ldpi	load pointer to instruction
mint	minimum integer
bsub	byte subscript
ssub	sixteen subscript
wsub	word subscript
wsubdb	form double word subscript

Table 7.9 Subscript addressing instructions

The address of a local workspace location is loaded using the *ldlp* instruction (described in section 7.2).

Similarly, the address of a location in the program being executed can be obtained by the *ldpi* operation as follows. The address of the location *x* bytes past the next instruction (which is itself pointed to by the instruction pointer register) can be pushed onto the integer stack by

ldc x; ldpi

For example, the address of a label *L* can be loaded by

```
    ldc (L–M); ldpi
M:
```

where the label *M* is the address of the instruction that follows the *ldpi* instruction. Firstly the offset in bytes from *M* to *L* is loaded into **Areg**. The *ldpi* then uses this offset and the value in the instruction pointer register (which will be the address of label *M*) to load the address of label *L* into **Areg**. This technique is useful for generating relocatable code. **Breg** and **Creg** are unaffected.

The lowest address in memory can be pushed onto the integer stack by *mint*. This is particularly useful for forming the address of a communication link. (It is equivalent to *ldc MostNeg*.)

The *wsub*, *bsub*, *ssub* and *wsubdb* instructions interpret **Areg** as the address of the beginning of a vector of data objects, and **Breg** as an index into that vector. After execution, **Areg** holds the address of the indexed element, and **Creg** is popped into **Breg** leaving **Creg** undefined. The purpose of these instructions is to calculate the address of a selected element of a vector, which begins at the address pointed to by **Areg** and stores its elements in contiguous space from that address. They simplify access to vectors of various sized data elements. For example *wsubdb* makes it easier to access vectors of `REAL64`s and `INT64`s.

bsub is used for vectors comprising bytes (8-bit objects). The operation performed by *bsub* is to add the integer in **Breg** to the address in **Areg** (without overflow checking).

ssub is used for vectors comprising 16-bit objects. The operation performed by *ssub* is to multiply the integer in **Breg** by two and to add this to the address in **Areg** (without overflow checking).

wsub is used for vectors comprising words (32-bit objects). The operation performed by *wsub* is to multiply the integer in **Breg** by four and to add this to the address in **Areg** (without overflow checking).

wsubdb is used for vectors comprising double words (64-bit objects). The operation performed by *wsubdb* is to multiply the integer in **Breg** by eight and to add this to the address in **Areg** (without overflow checking).

The user should also be aware that application of these instructions as described makes the best use of the processor pipeline. For example *bsub* is functionally equivalent to *sum*, but the former will be more

efficiently grouped, in the superscalar architecture of the IMS T9000, when calculating the address of an element in a vector of bytes.

7.5.3 Arrays

Access to a component of an array can be split into two sections. Firstly the address of the component must be constructed, and then the transfer of data to or from that component must be performed.

Evaluating a subscript

Array subscripts can be evaluated efficiently using the *prod* instruction. If array *A* has been declared by

[S_1] . . . [S_n]INT *A*:

where S_i ($i = 1..n$) are the dimensions, then one way of arranging this in store is to have all elements of the array in a contiguous block. Let's choose to have the elements in the rightmost dimension stored adjacently. For example figure 7.3 shows the elements of a particular three dimensional array (`Array`) stored in this way.

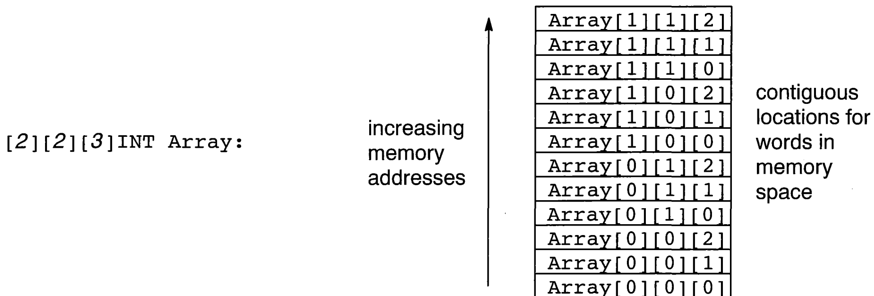


Figure 7.3 A possible method of storing an array of integers

Now if an access is required to such an array,

A[e_1] . . . [e_n]

then the code to evaluate the subscript is

ldc S_2; e_1; *prod;* e_2; *add; ldc* S_3; *prod; . . . ;* e_n; *add*

For example to evaluate the subscript for element `Array[1][0][2]` (where `Array` is declared as in figure 7.3), the code sequence is

ldc 2; 1; prod; 0; add; ldc 3; prod; 2; add

which evaluates to 8, which as can be seen from figure 7.3, is the correct offset from the bottom location of the block.

There is no need for the multiplication to check for overflow as this should be checkable during compilation. Mechanisms for range checking the actual subscripts are given later.

Accessing a word addressed array

Let *Wa_ptr* be a pointer to an array (*Wa*) that starts at a word boundary, and in which all component types are measured in words. Let *e* be a subscript expression. The address of component *e* of *Wa* is

e; Wa_ptr; wsub

or

Wa_ptr; ldnlp e

if *e* is a constant expression.

Accessing a byte addressed array

Similarly, let *Ba_ptr* be a pointer to an array (*Ba*) which may start at any byte location, and in which each component type is measured in bytes. Let *e* be a subscript expression. The address of component *e* of *Ba* is

e; Ba_ptr; bsub

7.5.4 Transferring array elements

An extra function is introduced here for use in this section and section 8.4.3. The function *length(block)* returns the length of *block* in bytes, where *block* is a block of data that may correspond to a variable, a data structure, a channel etc.

Let *Xb* be a variable or expression, the length in bytes of which is given by the value of the expression *b*. Then

length(Xb) = b

Let *Xw* be a variable or expression, the length in words of which is given by the value of the expression *w*. Then

length(Xw) = w; bcnt

If the value of *w* and the wordlength is known by the compiler, then the value can be loaded directly into the integer stack using *ldc n*. For example, for the IMS T9000,

length(Xw) = ldc (w $\times$ *4)*

Block transfer

mnemonic	name
move	move message

Table 7.10 instruction that performs block transfer

Once the address of the array element has been evaluated, its length in bytes is required to enable it to be transferred using the instruction *move*, or using the communication instructions described in section 8.4.3.

Assignment of arrays is achieved with the block move instruction *move*. This interprets **Areg** as an unsigned integer representing the number of bytes to be transferred, **Breg** as the destination address, and **Creg** as the source address. It hence moves **Areg** bytes of data starting at address **Creg** to address **Breg**. All integer stack registers are undefined after execution. This instruction is interruptible (see section 8.2.5).

v1 := *v2* = *address(v2); address(v1); length(v1); move*

where *address(v)* is defined on page 29 and *length(v)* is compiled as described above. The two arrays must not overlap — if they do, the effect of the *move* instruction is not defined. In particular the *move* instruction cannot be used to initialize a region of memory by moving from one location to an overlapping location.

7.6 Multiple assignment

Previous sections have detailed how single assignments to variables, array elements and arrays can be compiled. The compilation of multiple assignments is more complex.

In occam, the multiple assignment

$V_1, \ldots, V_n := E_1, \ldots, E_n$

is defined as being equivalent to

```
T1 temp1:

Tn tempn:
SEQ
  PAR
    temp1 := E1

    tempn := En
  PAR
    V1 := temp1

    Vn := tempn
```

where the parallel separation rules of occam apply so that multiple assignments are restricted to those whose 'expanded' version is a valid occam program. $T_1 \ldots T_n$ are type definitions of the appropriate types.

Because the final assignments are performed as if in a PAR construct they are guaranteed not to interfere — i.e. one assignment cannot affect the destination of another — so that they can be compiled as a sequence of assignments. Hence the multiple assignment can be compiled as

assign($temp_1$, E_1); . . . ; *assign*($temp_n$, E_n);
assign(V_1, $temp_1$); . . . ; *assign*(V_n, $temp_n$)

where

assign(V, E)

represents the compiled code for

V := *E*

as detailed in subsection 7.5.4.

This can be optimized by re-ordering the two assignment sequences to enable registers to be used instead of some of the temporary variables.

For example,

```
a, b := c, d
```

can be optimized from

ldl c; ldl d; stl t_1; *stl* t_2;
ldl t_1; *ldl* t_2; *stl a; stl b*

to

ldl c; ldl d; stl b; stl a

7.7 Comparisons and jumps

Comparisons and conditional behavior are provided by the instructions listed in table 7.11.

mnemonic	name
eqc n	equals constant
diff	difference
gt	greater than
gtu	greater than unsigned
j n	jump
cj n	conditional jump

Table 7.11 Comparisons and conditional behavior instructions

7.7.1 Representation of *true* and *false*

The IMS T9000 uses 0 as *false* and 1 as *true*. These values are generated by predicate operations (for example comparisons).

It is easy to implement programming languages that use a different representation of *true* and *false* as shown in section 7.7.3.

7.7.2 Comparisons

The primary instruction *eqc n* loads **Areg** with a truth value — *true* if **Areg** is initially equal to the instruction operand (*n*), *false* otherwise. **Breg** and **Creg** are unaffected.

diff, *gt* and *gtu* take integer operands in **Areg** and **Breg** and produce a boolean result which is loaded into **Areg**. They also load the value in **Creg** into **Breg**, leaving **Creg** undefined.

diff (introduced in section 7.3.4) can be used as a 'not equals' function. It loads *false* into **Areg** when the values initially held in **Areg** and **Breg** are the same.

The *gt* instruction loads **Areg** with *true* if **Breg** > **Areg**, *false* otherwise.

Similarly *gtu* loads **Areg** with *true* if the *unsigned* value of **Breg** is greater than the *unsigned* value of **Areg**; *false* otherwise.

7.7.3 Implementation of languages with different representations of *true* and *false*

When *true* and *false* are represented by 1 and 0 respectively, these values can be loaded with single byte load constant instructions. However it is also possible to represent *true* by a value other than 1. In particular, using

eqc X; not; adc 1

and

gt; not; adc 1

in place of *eqc X* and *gt* respectively, does not affect the representation of a *false* result, but changes the representation of *true* to –1, which is used in some programming languages.

7.7.4 Boolean negation

The above implementation of the boolean type (section 7.7.1) enables a boolean negation to be represented as follows.

$$\neg(X) \quad = \quad (X;\ eqc\ 0)$$
$$\neg(\neg(X)) \quad = \quad (X;\ eqc\ 0;\ eqc\ 0) \quad = \quad X$$

The symbol '¬' will be used henceforth where a boolean negation is required in a code sequence. This is merely to aid clarity.

7.7.5 Jump and conditional jump

The are two relative jump instructions. Both are primary instructions.

The unconditional jump instruction, *j n*, adds its operand (*n*) to the address of the instruction immediately following it and puts the result into **IptrReg**, thus transferring execution to another part of the program. It leaves the integer and floating-point stacks undefined. This instruction should *never be used with a zero operand* except where a breakpoint is required†.

When executed in an L-process, *j n* allows the current process to deschedule (after the jump) if the current timeslice has been exceeded, ensuring that there is an opportunity to deschedule once each time round a loop. (This is one of two instructions that have this property. Such instructions are referred to as 'timeslicing points'. See section 8.2.4.)

The conditional jump instruction, *cj n*, performs a jump if the value in **Areg** is 0 and does not affect the integer stack; but otherwise pops the value in **Areg** off the integer stack and continues with the next instruction. Consequently *cj n* serves as 'jump if *false*' provided that the language being implemented interprets 0 as *false* (see section 7.7.1).

The *cj* instruction never deschedules the process. The sequence

ldc 0; cj L

can be used in place of

j L

if it is important that descheduling does not occur. This will cause the value 0 to have been pushed onto the integer stack when execution reaches *L*. This 0 value can be removed, if necessary, by making the first instruction after *L* a *pop* which will restore **Areg** and **Breg** to the values they held before the jump — however any value in **Creg** will have been lost.

7.7.6 Evaluation of boolean expressions

The following shows the correspondence between occam expressions and instructions. *X* and *Y* are expressions, and *K* a constant. The symbol '¬' is a boolean negation (see subsection 7.7.4).

TRUE	=	*ldc 1*
FALSE	=	*ldc 0*
NOT *X*	=	*¬(X)*
X = Y	=	*X; Y; diff; eqc 0*
X <> Y	=	*¬(X; Y; diff; eqc 0)*
X = K	=	*X; eqc K*
X <> K	=	*¬(X; eqc K)*
X > Y	=	*X; Y; gt*
X < Y	=	*Y; X; gt*
X >= Y	=	*¬(Y; X; gt)*
X <= Y	=	*¬(X; Y; gt)*

Further optimizations can be made to the 'not equals' comparison when followed by a conditional jump.

X <> Y; cj L	=	*X; Y; diff; cj L*
X <> 0; cj L	=	*X; cj L*

† The instruction *j n* has a dual use. Its normal use is an unconditional jump; but if its operand (*n*) is zero, then it causes a breakpoint to occur, by forcing a trap to be taken. The latter use is detailed in chapter 14.

Evaluation of AND and OR

For evaluation of boolean AND and OR operations, the instruction sequence depends on whether or not strict or non-strict evaluation is used. In occam, evaluation is non-strict and the following short-circuit technique must be used.

$$X \text{ OR } Y \;=\; \neg(\neg(X);\ cj\ L;\ \neg(Y);\ L{:}) \qquad (1)$$
$$X \text{ AND } Y \;=\; X;\ cj\ L;\ Y;\ L{:}$$

The following laws should be applied to the compilation of conditional expressions before code is generated to ensure that the jump is taken as early as possible.

$$\neg(X \text{ AND } Y) \;=\; (\neg X) \text{ OR } (\neg Y) \qquad [= \;\neg(X;\ cj\ L;\ Y;\ L{:})]$$
$$\neg(X \text{ OR } Y) \;=\; (\neg X) \text{ AND } (\neg Y) \qquad [= \;\neg(X);\ cj\ L;\ \neg(Y);\ L{:}]$$
$$(X \text{ OR } Y);\ cj\ L \;=\; (\neg X);\ cj\ M;\ Y;\ cj\ L;\ M{:}$$
$$(X \text{ AND } Y);\ cj\ L \;=\; X;\ cj\ L;\ Y;\ cj\ L$$

In other languages, evaluation of booleans is strict (for example, ADA gives the programmer the choice) and so both expressions in dyadic logical operations may need to be evaluated.

For example where *false* is represented by 0, and *true* is represented by any *fixed* bit pattern other than 0 (e.g. *true* is always 1, or *true* is always –1), then the following transformations apply

$$X \text{ OR } Y \;\Rightarrow\; X \text{ BITOR } Y$$
$$X \text{ AND } Y \;\Rightarrow\; X \text{ BITAND } Y$$

and the bitwise instructions given in section 7.3.7 can be used:

$$X \text{ OR } Y \;=\; X;\ Y;\ or$$
$$X \text{ AND } Y \;=\; X;\ Y;\ and$$

Note that even for some non-strict evaluations, the above sequence may be preferable. Where *Y* is a simple boolean expression such as a local variable, its evaluation does not cause any side-effects, and so it does no harm to implement a non-strict evaluation using a bitwise operation.

7.7.7 Conditional transfer of control

The conditional expressions used in a conditional branch of an IF construct are compiled as follows

```
IF E                 E; cj L;
  P        =>        P; j END;
                  L:
```

where the label *END:* is at the end of the code for the IF construct.

The compilation of a `WHILE` loop is

```
WHILE E           L:   E; cj END;
  P        =>          P; j L
                  END:
```

Note that this loop includes an unconditional jump. The presence of this ensures that rescheduling can take place should the loop continue for longer than a single timeslice.

The compilation of a `REPEAT .. UNTIL` loop is

```
REPEAT                 j K
    P      =>     L:   E; eqc 0; cj END
UNTIL E           K:   P; j L;
                  END:
```

7.7.8 Compiling CASE statements

The CASE statement is a special form of conditional transfer where the transfer is determined by comparing an expression to a number of constants.

When compiling the process

```
CASE x
  ...
```

the expression x is evaluated and stored in a local variable by

x; stl selector

Then each branch of the CASE statement

```
c1, ... , cn
  P
```

can be compiled by

```
    ldl selector; ldc c1; diff; cj L;
    ldl selector; ldc c2; diff; cj L;
    ...
    ldl selector; eqc cn; cj M;
L:  P; j END;
M:
```

where the label *END:* is placed at the end of the CASE statement.

Optimized compilation of CASE

The compilation method given above will produce inefficient code for large CASE statements. To produce more efficient code the following rules can be used.

First build up a set of pairs of selector values and processes, consisting of every selector value in the CASE statement along with its associated process — the process part of each pair can be represented by the offset to the start of the compiled code for that process. Then the following rules can be used.

1 If there are 3 entries or less then use the method as described above.

2 If there are 12 entries or less then use a binary search to limit the number of comparisons required.

3 For more than 12 entries attempt to use a jump table. The offset of the start of each selected process is placed in the table against each selector value. Entries that do not match a selector in the CASE statement must contain the offset of an error handler process. This jump table should be the largest table such that about $\frac{1}{3}$ of the entries are filled. This compilation strategy is then recursively called to handle the two ends. The *gcall* (section 7.11) and *ldpi* (section 7.5.2) instructions, can be used to jump to the selected piece of code.

The choice of 3 or less processes, 12 or less processes and $\frac{1}{3}$ filled table are the values used in current INMOS occam compilers.

Consider compiling the CASE expression

```
CASE X
  c1
    P1

  cn
    Pn
```

where, for brevity, it is assumed that all the case selectors are already in increasing order.

Three entries or less

This case is compiled as

```
IF
  X = c1
    P1
  ⋮
  X = cn
    Pn
```

Four to twelve entries

This case is compiled as

```
IF
  X <= c(n/2)
    IF
      X <= c(n/4)
        ... etc.
      X > c(n/4)
        ... etc.
  X > c(n/2)
    ... etc.
```

Using a jump table

Assume that $c_i \dots c_m$ form a $\frac{1}{3}$ filled jump table. Then the case is compiled as

```
IF
  X < ci
    CASE X
      c1
        P1
      ⋮
      ci-1
        Pi-1
  X > cm
    ... similar
  TRUE
    ... jump table code
```

where *jump table code* is

```
            X; ldc ci; diff; ldc jump_size; prod; ldc (jump_table–M); ldpi
M:          bsub; gcall;
jump_table:
            j case_0; j case_1; ... ; j case_k
ERROR:      ... error code
Li:         ... code for Pi
            ⋮
Lm:         ... code for Pm
```

The code at *jump_table* consists of a sequence of jump instructions which transfer control to the relevant branch *Li* . . . *Lm* or to *ERROR*. The destination, *case_x*, of each of these jumps is *Lj* if c_j is equal to $(c_i + x)$ and is *ERROR* otherwise.

The code at *ERROR* should be the same code as used at the end of an IF statement where all the conditionals have been *false*. The *bsub*, *ldpi* and *gcall* instructions are explained in other sections.

All the jumps in the *jump_table* code must be encoded to the same length (*jump_size* bytes) to enable them to be accessed as a byte array. *nop*, which a single byte instruction that performs no operation, can be used to ensure this where different operands require a different amount of prefixing.

Also note that in the special case where *jump_size* is 1, '*ldc jump_size*; *prod*' can be removed from the sequence, and where *jump_size* is 2, 4 or 8, '*ldc jump_size*; *prod*' can be removed provided *bsub* is replaced with *ssub*, *wsub*, or *wsubdb* respectively.

7.8 Long arithmetic and shifts

7.8.1 Multiple length addition and subtraction

Signed addition and subtraction can be performed on values longer than a word using the instructions shown in table 7.12.

mnemonic	name
lsum	long sum
ldiff	long difference
ladd	long add
lsub	long subtract

Table 7.12 Long arithmetic instructions

The *ladd* and *lsub* instructions are used for the final step of a signed multiple length addition or subtraction. The other steps can be performed using *lsum* and *ldiff*. For all four instructions, there are two single word integer operands held in **Areg** and **Breg**, and a carry (or borrow) operand held in the least significant bit of **Creg** (i.e. $\mathbf{Creg}_{lsb}$ – all other bits in **Creg** are ignored) .

The *lsum* instruction forms (**Breg** + **Areg**) + $\mathbf{Creg}_{lsb}$ leaving the least significant word of the result in **Areg** and the most significant (carry) bit in the least significant bit of **Breg** (all other bits in **Breg** are set to 0). **Creg** becomes undefined.

Similarly, the *ldiff* instruction forms (**Breg** – **Areg**) – $\mathbf{Creg}_{lsb}$ leaving the least significant word of the result in **Areg** and the borrow bit in the least significant bit of **Breg** (all other bits in **Breg** are set to 0). **Creg** becomes undefined.

The *ladd* instruction sets **Areg** to (**Breg** + **Areg**) + $\mathbf{Creg}_{lsb}$. If this instruction causes overflow (i.e. the result is greater than *MostPos* or less than *MostNeg*), it signals *IntegerError*, and the result is truncated so as to fit into 32 bits (but see note 1 at the end of the chapter). **Breg** and **Creg** become undefined.

The *lsub* instruction sets **Areg** to (**Breg** – **Areg**) – $\mathbf{Creg}_{lsb}$. If this instruction causes overflow, it signals *IntegerError*, and the result is truncated so as to fit into 32 bits (but see note 1 at the end of the chapter). **Breg** and **Creg** become undefined.

Addition of two double length signed values with overflow checking can therefore be compiled as follows

ldc 0;
ldl X_{lo}; *ldl* Y_{lo}; *lsum; stl* Z_{lo};
ldl X_{hi}; *ldl* Y_{hi}; *ladd; stl* Z_{hi}

[The subscripts 'lo' and 'hi', used here and in subsequent text, specify the less and more significant word respectively, of the double word variable with which they are associated.]

Subtraction of two double length values without overflow checking is compiled as

ldc 0;
ldl X_{lo}; *ldl* Y_{lo}; *ldiff; stl* Z_{lo};
ldl X_{hi}; *ldl* Y_{hi}; *ldiff; stl* Z_{hi}

with the final borrow left in **Areg**.

7.8.2 Multiple length multiplication and division

The long multiplication and division instructions are

mnemonic	name
lmul	long multiply
ldiv	long divide

Table 7.13 Long arithmetic instructions

The *lmul* instruction multiplies two single word unsigned operands in **Areg** and **Breg**, and adds the single word 'carry' operand in **Creg** to form a double length unsigned result. The most significant (carry) word of the result is left in **Breg**, the least significant in **Areg**. No overflow is possible so an error cannot be signalled by this instruction. Multiplication of a single length unsigned value *X* by a double length unsigned value *Y* can be performed by

> *ldc 0;*
> *ldl X; ldl* Y_{lo}*; lmul; stl* Z_{lo}*;*
> *ldl X; ldl* Y_{hi}*; lmul; stl* Z_{hi}

which leaves the 'carry' in **Areg**.

Double length unsigned multiplication is not quite so obvious. The product of two unsigned double length words *X* and *Y* – i.e.

$$X.Y = (X_{hi}.2^{32} + X_{lo}).(Y_{hi}.2^{32} + Y_{lo}) = (X_{hi}.Y_{hi}).2^{64} + (X_{hi}.Y_{lo} + X_{lo}.Y_{hi}).2^{32} + (X_{lo}.Y_{lo})$$

can be coded as follows.

> *ldc 0;*
> *ldl* X_{lo}*; ldl* Y_{lo}*; lmul; stl* Z_0
> *ldl* X_{lo}*; ldl* Y_{hi}*; lmul; rev; stl* Z_2
> *ldl* X_{hi}*; ldl* Y_{lo}*; lmul; stl* Z_1*;*
> *ldl* X_{hi}*; ldl* Y_{hi}*; lmul; rev; stl* Z_3*;*
> *ldc 0; rev; ldl* Z_2*; lsum; stl* Z_2*;*
> *ldl* Z_3*; sum; stl* Z_3

This gives a quadruple length unsigned result *Z*.

The *ldiv* instruction divides the double length unsigned value held in **Breg** and **Creg** (most significant word in **Creg**) by the single length unsigned value in **Areg**. The quotient is left in **Areg** with the remainder in **Breg**. Overflow occurs if the result cannot be represented as an unsigned single word value and *IntegerOverflow* is signalled. Division of a double length value *X* by a single length value *Y* to produce a double length result *Z* can be performed by

> *ldc 0;*
> *ldl* X_{hi}*; ldl Y; ldiv; stl* Z_{hi}*;*
> *ldl* X_{lo}*; ldl Y; ldiv; stl* Z_{lo}

which leaves the remainder in **Areg**.

Both *lmul* and *ldiv* leave **Creg** undefined.

7.8.3 Multiple length shifts

The long shift instructions are

mnemonic	name
lshl	long shift left
lshr	long shift right

Table 7.14 Long shift instructions

The *lshl* and *lshr* instructions both shift the double length value held in **Breg** and **Creg** (most significant word in **Creg**) . Vacated bit positions are filled with zero bits. The number of bit positions shifted is the unsigned value of **Areg**. If **Areg** is zero, then the result is the unshifted value, and if **Areg** is greater than the number of bits in a double length word, then the result is zero. The result is left in **Areg** and **Breg** (most significant word in **Breg**). (The value of **Areg** can be checked in advance by using the *csub0* instruction – see section 7.13.) Both instructions leave **Creg** undefined.

The value held in a double length variable *X* can be shifted *n* places left by

ldl X_{hi}; *ldl* X_{lo}; *ldl n; lshl; stl* X_{lo}; *stl* X_{hi}

The value held in a double length variable *X* can be shifted *n* places right with the shift length checked by the following code. This will signal *IntegerError* if the shift length is not in the range 0 .. 2 × *BitsPerWord*.

ldl n; ldc (2×BitsPerWord + 1); csub0;
ldl X_{hi}; *ldl* X_{lo}; *ldl n; lshl;*
stl X_{lo}; *stl* X_{hi}

Single length arithmetic shifts

The value held in a single length variable *X* can be arithmetically shifted *n* places right by

ldl X; xdble; ldl n; lshr; stl X

and by *n* places left by

ldl X; xdble; ldl n; lshl; csngl; stl X

where *xdble* and *csngl* are explained in section 7.9.2.

In the first example here, the integer in **Areg** is sign extended to a 64-bit object in **Breg** and **Areg**. When the long shift operation is executed, the extra bits are shifted from **Breg** to **Areg**. This may be used for dividing an integer by a 'power of 2' divisor, because the sign of the dividend is preserved in the result.

The second example can be used similarly for multiplying by a 'power of 2' factor. *csngl* also checks for an out-of-range result – i.e. an integer which is not representable in 32 bits.

Single length rotation

The value held in a single length variable *X* can be rotated *n* places right by

ldl X; ldc 0; ldl n; lshr; or; stl X

and by *n* places left by

ldc 0; ldl X; ldl n; lshl; or; stl X

If the rotate length is not guaranteed to lie in the range $0 \leq n < BitsPerWord$ then the length should be masked with *(BitsPerWord – 1)*, which produces the shift length modulo *BitsPerWord*. This is because the *lshl* or *lshr* will lose the bits in the word being rotated. *BitsPerWord* can be evaluated by

ldc 8; bcnt

bcnt multiplies the value in **Areg** by the number of bytes per word†.

The long shifts can also be used to perform extraction and insertion of bit fields, even where these cross word boundaries in memory.

7.8.4 Normalizing

mnemonic	name
norm	normalize

Table 7.15 Instruction for normalizing double length value

The *norm* instruction normalizes the unsigned double length value in **Areg** and **Breg** (most significant word in **Breg**) . The double length value held in **Areg** and **Breg** is shifted left until the most significant bit of the value is one. The shifted double length value remains in **Areg** and **Breg**. The number of bits shifted is left in **Creg**. If the double length value is initially zero, **Creg** is set to twice the number of bits in a word.

7.9 Object length conversion

Section 4.3 explains that data can be represented in various sized objects. This section describes the instructions that can be used to convert between these representations.

Most of the transputer integer arithmetic instructions operate on signed integers held in the integer stack registers as 32-bit objects, and produce results in this form. A few of the instructions operate on or produce results as 64-bit objects held in two of the 32-bit stack registers. The IMS T9000 therefore provides instructions that allow an integer to be sign extended to 32-bits or 64-bits, as well as instructions that allow a program to check that an integer can be represented in a smaller object than the object in which it is currently represented.

Object length conversion is also important for conversion of high level language data types.

7.9.1 Conversion between 8/16-bit object and word representations

To enable integer representations to be converted between bytes, 16-bit objects, and words, the IMS T9000 provides the instructions shown in table 7.16.

8-bit object		16-bit object	
mnemonic	**name**	**mnemonic**	**name**
xbword	sign extend byte to word	*xsword*	sign extend sixteen to word
cb	check byte	*cs*	check sixteen
cbu	check byte unsigned	*csu*	check sixteen unsigned

Table 7.16 Instructions used for converting between bytes, 16-bit objects and words

xbword extends a signed 8-bit object (byte) in **Areg** to a signed word. It achieves this by examining bit 7 of **Areg** and setting the more significant bits in the word to the same value. The previous values of bits 8 to 32 in **Areg** are overwritten. **Breg** and **Creg** are unaffected by this operation.

xsword extends a signed16-bit object in **Areg** to a signed word. It achieves this by examining bit 15 of **Areg** and setting the more significant bits in the word to the same value. The previous values of bits 16 to 32 in **Areg** are overwritten. **Breg** and **Creg** are unaffected by this operation.

cb examines the value in **Areg** to ensure that it is representable as a signed 8-bit value. If it is greater than or equal to –128 (-2^7) and less than +128 ($+2^7$), then the value is legal. Otherwise *IntegerError* is signalled. All integer stack registers are unaffected by this operation.

† This is not strictly necessary if it is known that the code is to be run on the IMS T9000, since the wordlength is known to be 32 bits. However use of *bcnt* and other wordlength dependent instructions, can be used to make code portable to machines of different wordlength.

cs examines the value in **Areg** to ensure that it is representable as a signed 16-bit value. If it is greater than or equal to –32768 (-2^{15}) and less than +32678 ($+2^{15}$), then the value is legal. Otherwise *IntegerError* is signalled. All integer stack registers are unaffected by this operation.

cbu examines the value in **Areg** to ensure that it is representable as an unsigned 8-bit value. If it is greater than or equal to 0 and less than +256 ($+2^{8}$), then the value is legal. Otherwise *IntegerError* is signalled. All integer stack registers are unaffected by this operation.

csu examines the value in **Areg** to ensure that it is representable as an unsigned 16-bit value. If it is greater than or equal to 0 and less than +65536 ($+2^{16}$), then the value is legal. Otherwise *IntegerError* is signalled. All integer stack registers are unaffected by this operation.

Hence for example, where variables are stored as 16-bit objects in local workspace,

```
a = b + c
```

can be coded as

ldlp b; lsx; ldlp c; lsx; add; cs; ldlp a; ss

where the integers are signed, or

ldlp b; ls; ldlp c; ls; add; csu; ldlp a; ss

if the integers are unsigned.

Note that in these examples, the 16-bit objects are stored on word (32-bit) boundaries making it possible to use *ldlp n* to calculate the object address (see page 29).

7.9.2 Conversion between single word and double word representations

mnemonic	name
xdble	extend to double
csngl	check single

Table 7.17 Instructions for conversion between single word and double word representation

The instruction *xdble* sign extends a single word to a double word – i.e. it changes the representation of a signed integer from a 32-bit object in **Areg** to a 64-bit object in **Areg** and **Breg** (most significant word in **Breg**). It achieves this by examining bit 31 of **Areg** and setting all bits in **Breg** to the same value. The data previously held in **Breg** is pushed into **Creg**.

Conversely, *csngl* reduces the representation of a signed integer from double word in **Areg** and **Breg** to a single word in **Areg**. An *IntegerError* is signalled if the integer value falls outside the range of values representable in a single word. **Creg** is popped into **Breg** leaving **Creg** undefined.

7.9.3 General conversion between N-bit object and word representations

mnemonic	name
xword	extend to word
cword	check word

Table 7.18 Instructions for conversion between word and N-bit object representation

The instruction *xword* sign extends an N-bit object to a word – i.e. it changes the representation of a signed integer from an N-bit object to a 32-bit object (see section 4.3). The instruction *cword* checks that the signed integer represented in a single word can be represented in an N-bit object. For these instructions,

an N-bit object is assumed to be of any length between 1 bit and 32 bits, and assumed to occupy the least significant bits in a word.

The two operands of *xword* are an N-bit object in **Breg** and a length specified in **Areg**. The length of the N-bit object is specified by the bit pattern of the most negative integer representable in the N-bit object – i.e. bit *N–1* is set and all other bits are clear. The representation of a signed integer (currently in **Areg**) can therefore be extended from a 10-bit object to a word by

ldc #200; xword

Similarly, the two operands of *cword* are a signed integer represented in a single word in **Breg**, and a length specified (as above) in **Areg**. The result, left in **Areg**, is the (unchanged) value of **Breg** and *IntegerError* is signalled if the integer cannot be represented in the N-bit object. The following code can be used to determine whether a signed integer (currently in **Areg**) can be represented in a 4-bit object.

ldc #8; cword

Two signed integers *X* and *Y*, represented as 3-bit objects, can be added and checked for overflow by

X; ldc #4; xword;
Y; ldc #4; xword;
add; ldc #4; cword

7.10 Replication

Replicators may be implemented by using the loop end instruction.

mnemonic	name
lend	loop end

Table 7.19 Conditional replication instruction

A loop is controlled by a loop end data structure, the format of which is shown in table 7.20. The **le.Count** slot contains the number of iterations left to perform, and so the user should initialize this to the total number of iterations required. The **le.Index** slot can be used to hold a control variable, and so the user should initialize this (if required) to the value that such a variable should take in the first iteration of the loop.

word offset	slot name	purpose
0	**le.Index**	contains the loop control variable
1	**le.Count**	contains number of iterations left to perform

Table 7.20 Loop end data structure

The *lend* instruction interprets **Breg** as a pointer to the loop end data structure and **Areg** as the number of bytes from the start of the next instruction to the start of the loop. The start of the loop normally will be before the *lend* instruction in memory so the offset for this 'conditional jump' is measured in the opposite direction to the offsets for the other jump instructions (viz. *j n* and *cj n*).

lend signals *Unalign* if the loop end data structure is not word aligned, and signals *AccessViolation* if the data structure is protected. The contents of the integer and floating-point stack registers are undefined after execution.

lend decrements the **le.Count** slot. If the number of iterations remaining (i.e. the value of **le.Count** after the decrement) is less than or equal to zero then execution passes to the next instruction. If the number of iterations remaining is greater than zero, then the processor increments **le.Index** and subtracts **Areg** from **IptrReg**. Note that, like the jump instruction (*j* – section 7.7.5), *lend* is a timeslicing point and causes the process to reschedule if the looping process has exceeded its timeslice, again ensuring that there is an opportunity to timeslice each time round a loop.

As an example take the occam replicated SEQ construct. This is compiled as

```
SEQ i = a FOR n              a; stl i; n; stl i+1;
  P                          ldl i+1; cj END;
               ⇒        L:   P;
                             ldlp i; ldc (END–L); lend;
                        END:
```

Where it is clear that *n* is not zero the following may be used

```
SEQ i = a FOR n              a; stl i; n; stl i+1;
  P                     L:   P;
               ⇒             ldlp i; ldc (END–L); lend;
                        END:
```

The same basic instruction sequence may be used to implement an occam replicated IF, ALT or PAR, and to initialize arrays of channels.

The count of iterations (*n*) to perform should be positive. When the number of iterations is the result of an expression then it may be necessary to add some range checking to cause an error or skip the loop if this evaluates to a negative value. If *n* is negative in the first example above, then the loop executes once before the *lend* instruction causes the loop to end. A negative count value should probably be treated as an error, though this depends on the definition of loops in the language being compiled. To check for a negative *n*, the first line of the above sequence should be amended to

a; stl i; n; mint; csub0; stl i+1;

where *csub0* is explained in section 7.13.

7.11 Procedures

The instructions in table 7.21 are used to implement procedures.

mnemonic	name
call n	call
gcall	general call
ajw n	adjust workspace
gajw	general adjust workspace
ret	return

Table 7.21 Instructions for implementing procedures

7.11.1 Adjusting workspace

The primary instruction *ajw n* adjusts the value of the workspace pointer by the number of words in its operand value – *n*. Workspace should be claimed by using a negative value and released by using a positive value.

7.11.2 Call and return

The primary instruction *call n* adjusts the workspace pointer downwards, allocating four new locations into which it stores the three integer stack registers and the instruction pointer (return address) — the return address is also left in **Areg** by the instruction. The operand to the call – *n* – is added to **IptrReg** to produce the address of the procedure being called.

The state of the workspace after the call instruction is as shown below

		Saved values
Wptr+4	(= old **Wptr**)	
Wptr+3		**Creg**
Wptr+2		**Breg**
Wptr+1		**Areg**
Wptr+0		**IptrReg**

The *ret* instruction restores the **IptrReg** from the **Wptr**+0 slot and adjusts the workspace pointer upwards, to deallocate the four locations. A procedure that requires more workspace will normally include *ajw* instructions to allocate and deallocate space. When the *ret* instruction is executed, the programmer must ensure that

- the **Wptr**+0 holds the return address
- any workspace claimed by the procedure should have been released so that the **Wptr** has returned to the value it held at the start of the procedure.

The *ret* instruction does not affect the integer stack, which can therefore be used to return up to three values to the calling procedure.

7.11.3 Use of (Wptr+0)

The location (**Wptr**+0) is used as a temporary store by certain instructions. These are

outword, *outbyte*, *altwt*, *taltwt*, *disc*, *dist*, *disg*, *diss*, and *altend*.

Any procedure that uses one of these instructions must allocate an extra workspace slot for this use of (**Wptr**+0) so that the return address is not overwritten. Workspace is allocated by the *ajw* instruction (see also section 8.1.1).

7.11.4 Loading parameters

It is convenient to load the first three parameters of the procedure into the integer stack registers, and to arrange the workspace of the calling procedure so that the additional parameters can be stored in locations 0, 1, . . . of the workspace before the procedure is called. In this way, the called procedure will be entered with its parameters stored in consecutive locations starting at workspace location 1. To enable the procedure to access non-local variables the parameters of a procedure should include a link to the environment in which the procedure was declared. This is discussed in section 7.11.5.

7.11.5 The static chain

The scope rules of block structured languages can be implemented using a static chain. This involves passing a single pointer as a parameter whenever a procedure is called. The 'non-local' load, store and pointer operations described in section 7.4, can then be used to access variables declared in an enclosing block.

Variable access via the static chain

Access via the static chain is provided by the *ldnl*, *stnl* and *ldnlp* instructions. Let n be the lexical level of the current procedure, and S_i the offset of the lexical link at level i. Then access to a location x at level $n-1$ is provided by

ldl S_n; *ldnl* x	— to load a variable
ldl S_n; *ldnlp* x	— to load a pointer to a variable
ldl S_n; *stnl* x	— to store a variable

Similarly, access to a location y at level $n-2$ is

ldl S_n; *ldnl* S_{n-1}; *ldnl* y

. . .

etc.

Forming a static link

When a procedure *P* is called, the static link for the call of *P* must be computed. Let *n* be the lexical level of the current procedure, and *m* the lexical level of *P*. If *m = n+1* the new link is computed by *(ldlp x)* with *x* chosen so that *P* can access all of its global variables, channels etc. Otherwise the new link is computed as the value of the link location at level *m*. With S_i as above, this can be obtained by

```
IF
  m = n+1
    ldlp x
  m = n
    ldl Sn
  m = n–1
    ldl Sn; ldnl Sn–1
  m = n–2
    ldl Sn; ldnl Sn–1; ldnl Sn–2
  . . .
  etc.
```

Passing the static link as a parameter

The static link for the called procedure, and the first two parameters are loaded into the integer stack, using a loading sequence as described above. The remaining parameters are each evaluated and stored in workspace locations starting from 0 before calling the procedure with a *call* instruction. In this way the procedure will see the return address at workspace 0, the static link at workspace 1 and the parameters at workspace 2 and onwards.

7.11.6 Other calling techniques

The *gcall* instruction enables any type of procedure call to be constructed as a sequence of instructions. Its only effect is to exchange the **IptrReg** and **Areg** registers. The entry point of the procedure to be called can therefore be computed in the same way as an expression. If necessary, another *gcall* instruction can be used later to return to the calling procedure if the return address, held in **Areg**, is saved on entry to the procedure.

It is possible to compile a procedure so that it can be called using either a *call* or a *gcall* instruction. Both the *call* and *gcall* instructions leave the return instruction pointer in **Areg**. Consequently, if the first instruction in the called procedure is *(stl 0)*, the return instruction pointer will be saved in the appropriate location in the calling workspace. The calling code may then execute either

call relative_address_of_procedure

or the sequence

ajw –4; *stl* 1; *stl* 2; *stl* 3; *absolute_address_of_procedure; gcall*

If using *call*, it is not strictly necessary for the procedure to execute *(stl 0)*, but by executing this one extra instruction, it means that the code does not have to distinguish between the calling methods. The procedure should however assume that there are no useful values in the integer stack after this operation. Note that when using *gcall*, the calling code must first adjust its workspace pointer using *(ajw –4)*, and then explicitly store the first three actual parameters in workspace locations 1, 2 and 3, as this is not be done by *gcall*. The *ret* instruction in the called procedure can then be used in the normal way irrespective of how the procedure was called. However, better ways of dealing with *gcall* are described below.

Efficiency will be improved if all procedures can assume they have been *call*-ed and methods similar to the ones described below are used in cases where a *gcall* is necessary. Combinations of the *call* and *gcall* instructions can be used to provide efficient implementation of procedure parameters, or for runtime linking of separately compiled procedures.

Library linkage

Most high level languages have a library system associated with them. Programs are able to make use of procedures from a library of standard procedures. To prevent the code size becoming too large the li-

brary procedures are not put into the compiled code until it is linked. This involves extracting the relevant library procedures from the libraries and 'linking' all the calls to those procedures in the compiled code to the correct address. Initially it might seem that all the code needs to be scanned for these library calls so that the link address can be instantiated but there is a simple mechanism making use of *call* and *j* to handle this.

Consider the compilation of a program which somewhere includes a call to the library procedure *lib_proc_1*.

Each library call is compiled into a *call* to a 'stub' at the end of the program associated with that library call. The first call to any library procedure will cause the compiler to create a stub for that procedure. A stub is a sequence of bytes into which a short piece of code will be placed by the linker so a sufficient number of bytes need to be reserved for this. So between compilation and linkage the code might look like

```
        ...
        call lib_proc_1_stub;
        ...
lib_proc_1_stub: – n bytes reserved;
        ...
```

When the program is linked the linker inserts the code

```
j offset_to_lib_proc_1_code
```

into the stub. Hence the calls inside the program will transfer control to this stub and the jump will then transfer control to the library procedure. The *j* instruction makes the code relocatable. The process might be timesliced on the *j* instruction but, since the *call* has already stored the integer stack into workspace, this is not important. The parameter passing of the original *call* has been undisturbed so that the return address still points back into the program (and not to the stub) . However since the *j* instruction may be timesliced the value of **Areg** on entry to the library procedure cannot be guaranteed to be the return address. This means that library routines called by this mechanism cannot be written to be *gcall*-able. If this is required then a larger stub that explicitly adjusts the workspace, *gcall*s the library routine and then returns to its call, could be used; but this is more expensive.

In the scheme described above 8 bytes should be reserved for each stub on a 32 bit transputer as the offset could possibly have 32 significant bits needing 7 prefixes before the *j*. (4 bytes are required on a 16 bit transputer.) The final linked code of the example above is

```
        ...;
        call lib_proc_1_stub;
        ...;
lib_proc_1_stub: j offset_to_lib_proc_1;
        ...
```

Procedures as parameters

Calling a procedure that has itself been passed as a parameter needs to be compiled with a *gcall* instruction, because its address cannot be compile time evaluated. Although this *gcall* can be made to look like a *call* by the methods above there is a more efficient way that uses a *call* to set up the parameters. Again this uses a *call* to a program stub. If an invocation of this procedure (which has been passed as a parameter) has itself got n parameters then this can be compiled by invoking a stub using *call*. This *call* should pass n+1 parameters where the last parameter is the address of the procedure. The stub then loads this n+1th parameter into **Areg** and performs a *gcall*. This has the same effect as a normal *call* to the procedure.

For example consider the following fragment of 'C' that uses a pointer to a function (`f_ptr`), to make a dereferenced call.

```
void (*f_ptr)( ... formal list of n parameters ... );
  .
```

```
.
(*f_ptr)( ... actual parameter list ... )
                    /* call of function through 'f_ptr' */
```

Assuming that there are at least 3 formal parameters (i.e. $n \geq 3$), the call may be compiled as

ldl f_ptr;	— where 'f_ptr' contains the absolute address of the function
stl (n–3);	— store as the n+1th parameter – N.B. first 3 parameters
	— are loaded onto the integer stack, 4th parameter
	— is stored at workspace 0 etc.
... load other parameters as normal	
call f_stub	
.	
.	
f_stub: ldl (n+1)	— call has moved the workspace down four slots
gcall	

If there are less than 3 parameters, the function address is passed in the integer stack.

7.11.7 Other workspace allocation techniques

The *gajw* instruction exchanges the contents of **Wptr** and **Areg**, allowing workspaces to be allocated dynamically, and allowing dynamic switching between existing workspaces.

If a process workspace holds a pointer to a new workspace, then

ldl W_{new}*; gajw; stl* W_{old}

changes to the new workspace and stores a pointer to the old workspace. The old workspace can be restored by

ldl W_{old}*; gajw*

In addition, the old workspace can be accessed from the new workspace, using

ldl W_{old}*; ldnl x*
ldl W_{old}*; stnl x*
ldl W_{old}*; ldnlp x*

7.12 Functions

The instructions explained in section 7.11 can also be used to implement a function. Up to 3 results of size less than or equal to the wordlength of the transputer can be returned from a function in the integer stack — the *ret* instruction does not affect the registers. Further results, or results larger than the wordlength, can be returned by passing into the function the addresses of places to store these results as extra parameters.

The occam function is used for purposes of illustration (for simplicity, it is assumed that the first 3 results can be returned in registers):

T_1, ... , T_m FUNCTION F (V_1, ... , V_n)
  local variable declarations
  VALOF
    P
    RESULT E_1, ... , E_m
:

can be compiled as

ajw –local_variables;
P;

$assign(result_4, E_4); \dots ; assign(result_m, E_m)$
$E_3; E_2; E_1;$
ajw local_variables;
ret

where

assign(V,E)

is the code for the assignment

$V := E$

and $result_4, \dots , result_m$ are the addresses of the result stores passed as extra parameters to the function.

One of the loading sequences described earlier may be required if the expressions returned in the registers contain evaluations. Since the values returned by a multiple result function, will be assigned to variables in a multiple assignment which assigns in parallel, it is always possible to evaluate the results in any order. In this way cases can be handled where the results returned in the registers are not the first 3 results.

7.12.1 Calling a function

If the function has more than 3 results, then those results which cannot be passed in the integer stack should be assigned using 'call by reference'. The addresses of these result variables, as well as the function's parameters and the static link, must therefore be passed to the function. As with procedures the first three of these should be loaded into the integer stack before the *call* instruction which automatically stores them in the workspace. The remainder of the 'parameters' passed should be loaded into the workspace before *call* is executed. When the function returns, the results whose addresses were passed will already have been stored so all that remains is to store the (up to) 3 results returned in the integer stack.

For example the function call

$V_1, \dots , V_m := F(E_1, \dots , E_n)$

could be compiled by

E_3*; stl 0; . . . ;* E_n*; stl (n–3);*
genaddr(V_4*); stl (n–2); . . . ; genaddr(*V_m*); stl ((m+n)–6);*
E_2*;* E_1*; static_link; call F;*
stl V_1*; stl* V_2*; stl* V_3

where *genaddr(X)* is the code needed to form the address of *X*. The compiler must have already allocated sufficient workspace for the parameters that are stacked explicitly. For simplicity it has been assumed that $V_1 \dots V_3$ are all local variables whose values can be returned in a register.

7.12.2 Single result functions

In most programming languages, a function that returns a single result can be used in an expression as well as in an assignment.

A common form of function returns a single value contained in a word — the mechanism described above will return this in **Areg**. When compiling expressions (using the algorithm described in section 7.3.1), the depth of such a function call should be taken as being infinite — i.e. deeper than any other form of expression. This is because the function call will always lose any other information in the registers. By giving it infinite depth the expression compilation algorithm will never call a function while another expression result is being held in a register.

7.13 Error checking instructions

The instructions listed in table 7.22 can explicitly check for integer errors. The following text describes these instructions and demonstrates how they may be used. If conditions are such that *IntegerError* is signalled, this causes a trap to be taken. A general treatment of trap-handling is given in chapter 10.

mnemonic	name
csub0	check subscript from 0
ccnt1	check count from 1
cir	check in range
ciru	check in range unsigned

Table 7.22 Instructions that may explicitly signal *IntegerError*

Subscript checking, sign checking and 'signal if *true*'

The *csub0* instruction signals *IntegerError* if the unsigned value of **Breg** is greater than or equal to the unsigned value of **Areg**. The value in **Areg** is popped from the stack by this instruction.

It can be used to check subscript operations. An expression *E* can be checked to signal *IntegerError* if it is greater than or equal to *S* by

E; S; csub0

If *A* is an array of *S* words subscripted from 0, and *E* an expression, then *A* [*E*] can be translated into the range checked access

E; S; csub0; A; wsub; ldnl 0

Note that the *csub0* instruction traps both an overlarge subscript and a negative subscript; since, when considered as unsigned values, all negative values are greater than any positive value.

Similarly *csub0* can test an expression (*E*) for a negative integer value with

E; mint; csub0

The bit pattern for *MostNeg* is 100..0. Since *csub0* treats its operands as unsigned integers, it will in this example signal *IntegerError* for any bit pattern which has the msb (most significant bit) set to 1. In twos complement format, this represents all negative numbers.

It can test a boolean expression for *true* with

E; ldc 1; csub0

This sequence signals *IntegerError* if *E* is any value other than 0; but since 0 is the representation of *false*, it performs a 'signal if *true*' operation on the boolean valued expression *E*. It also signals *IntegerError* if *E* is outside the range of boolean values (i.e. not 0 or 1).

Checking message lengths and 'signal if *false*'

The *ccnt1* instruction signals *IntegerError* if the unsigned value of **Breg** is greater than the unsigned value of **Areg** or is less than 1 (viz. 0). That is, it checks that **Breg** is in the range 1 to **Areg** inclusive. The value in **Areg** is popped from the stack by this instruction.

It can test a boolean expression for *false* with

E; ldc 1; ccnt1

This sequence signals *IntegerError* if *E* is not 1; but since 1 is the representation of *true*, it performs a 'signal if *false*' operation on the boolean valued expression *E*. It also signals *IntegerError* if *E* is outside the range of boolean values (i.e. not 0 or 1).

General range checking

The *cir* instruction signals *IntegerError* if the signed value of **Creg** is greater than the signed value of **Breg** or is less than the signed value of **Areg**. That is, it checks that **Creg** is in the range **Areg** to **Breg** inclusive. The range values in **Areg** and **Breg** are popped from the stack by this instruction.

ciru is similar to *cir*, but uses unsigned comparisons of the values.

Both instructions may be used to check whether the result of an expression is in the correct range before using it further. For example a signed expression *E* can be checked by signalling *IntegerError* if it is not in the range S_{lower} to S_{upper}:–

> *E*; S_{upper}; S_{lower}; *cir*

For example in Pascal, if an array (`A`) is declared as

```
VAR A: ARRAY[ -27:104 ] OF INTEGER;
```

then to load an element of that array (`A[i]`), compile as

ldl i;	— load index
ldc 104; ldc –27; cir;	— check that index is within range of array
ldl a; adc 27; wsub;	— load address of element
ldnl 0	— load element

7.14 Device access instructions

The previously described memory access instructions use the following memory model: a read instruction applied to a particular location should return the last value written to that location by a write instruction. This says nothing about whether the value is actually stored in main memory by the write, nor does it say anything about the temporal order of reading and writing in relation to other instructions.

If a memory address is being handled by one of the main cache banks†, then data written to this address may not get written back to main memory before a subsequent access to the same address. This means, for example, in a sequence where a read is made between two writes to the same location –

ldc A; ldl Location1; stnl 0;	— store constant A into — *Location1*	
ldl Location1; ldnl 0; ldl Location2; stnl 0;	— load content of *Location1* and — store elsewhere (*Location2*)	(2)
ldl B; ldl Location1; stnl 0;	— store constant B into *Location1*	

– that although the read returns the last value written to the location before the read (*A*), the main memory location itself may never have actually held this value.

Where a sequence of instructions writes data to different memory locations, assumptions cannot be made about the order in which those locations are assigned data. For example, it cannot be assumed in the sequence

ldc A; ldl Location; stnl 0;	— store constant A into *Location*	(3)
ldl (Location+1); ldnl 0	— load content of *Location+1*	

that the write to location at *Location* is made before the read from *Location+1*.

The above behavior may cause problems where external devices are memory mapped. The behavior of such devices is often strictly defined by the ordering of read and write cycles to locations mapped onto ports within the device, and it is important that all data written to a memory mapped location is really copied to the device. For this reason, the IMS T9000 includes within its instruction set, special instructions for accesses to memory mapped devices. These are shown in table 7.23. Note however that these instructions may also be used to access non-device locations.

† For a resumé of the cache, refer to chapter 15. For a more detailed presentation of the memory architecture, refer to sections *Programmable memory interface* and *Instruction and data cache* of *The T9000 Hardware Reference Manual and to chapter 15 of this book.*

mnemonic	name
devlw	device load word
devls	device load sixteen
devlb	device load byte
devsw	device store word
devss	device store sixteen
devsb	device store byte
devmove	device move

Table 7.23 Device access instructions

The device instructions *devlw*, *devls*, *devlb*, *devsw*, *devss*, *devsb*, and *devmove* use the same registers as memory instructions *ldnl 0*, *ls*, *lb*, *stnl 0*, *ss*, *sb*, and *move* respectively, and in the context of the memory model described above, they have the same behavior.

It is possible to ensure that certain areas of memory are reserved for access by device instructions only. The programmable memory interface (PMI) can allocate memory banks as 'device memory'. This is explained in section *Programmable memory interface* of *The T9000 Hardware Reference Manual*. Also, when running under protection a region descriptor register can mark a region as 'device access only', so that a P-process trap is caused if a non-device instruction makes an access. This is explained in section 9.5.

The behavior therefore of the device instructions with respect to main memory may be different to their memory instruction counterparts. Firstly, the instructions guarantee that for memory marked by the PMI as 'device memory', writes and reads are made to the main memory. Therefore the sequence

ldc A; ldl Location1; devsw;
ldl Location1; devlw; ldl Location2; devsw;
ldl B; ldl Location1; devsw

(compare with sequence 2 above) is guaranteed to make a write, a read and another write to the actual memory location – in particular it will always write the constant *A* to main memory on the first write. Secondly these instructions guarantee that reads and writes to specified locations occur in the same sequence that they appear in the code sequence – e.g. the sequence

ldl Location; devsw;
ldl (Location+1); devlw

(compare with sequence 3 above) ensures that the write to location at *Location* is made before the read from *Location+1*.

devmove performs the minimum number of reads and writes required to copy the block of data. It guarantees that each successive read is from a location with a higher address (more positive) than the previous read, and similarly ensures that each successive write is to a location with a higher address than the previous write. The processor reads and writes complete words of data (rather than bytes), and so where a transfer is from a source block that is not aligned to word boundaries, it is necessary for the processor to read two words from the source before it can form a complete word to write to the destination. This instruction is interruptible (see section 8.2.5).

Where a combination of normal memory access instructions and device access instructions is used, the following rules apply.

- It is guaranteed that a device load instruction is executed after all normal memory load instructions that appear before the device load in the code sequence, and it is executed before all normal memory load instructions that appear after it.
- It is guaranteed that a device store instruction is executed after all normal memory store instructions that appear before the device store in the code sequence, and it is executed before all normal memory store instructions that appear after it.

This feature can be utilized in the following shared memory system example. Consider two processors sharing a particular area of memory. (See figure 7.4.) Each processor can only write to this area when it has been granted permission to do so. This permission is requested by polling on a specified 'permission' memory location using a device load instruction. If this has the special *Available* value, then the processor can claim write permission by writing its own identity into the location with a device store instruction, and can then proceed to write to the shared area using normal memory store instructions. When it has finished writing to shared memory, it writes *Available* back to the 'permission location' using a device store instruction. The other processor, which may now be polling this location, cannot write to the shared memory area until it finds the value *Available*.

Each processor therefore makes a device read and a device write prior to reading from or writing to shared memory. It also makes a device write when it has finished writing to shared memory. The above guarantees thus ensure that the processor does not make any reads from shared memory before it has successfully polled and cannot make any writes to shared memory before it has written back its own identity, and also ensure that the processor does not make any writes to shared memory after it has released its write privilege.

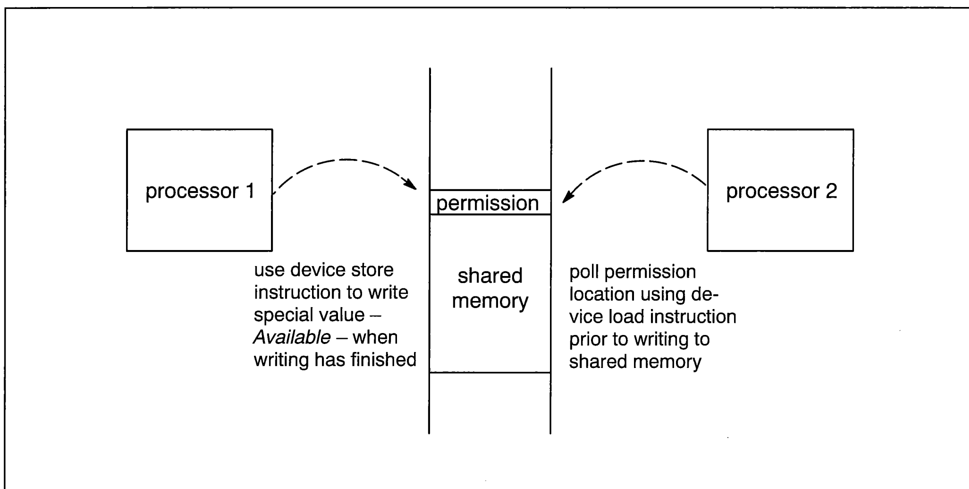


Figure 7.4 A shared memory example

7.15 Specialist instructions

7.15.1 Two dimensional block move

Graphical applications often require the movement of two dimensional blocks of data to perform windowing, overlaying etc. The transputer contains instructions to perform efficient copying, overlaying and clipping of graphics data based on byte sized pixels.

mnemonic	name
move2dinit	initialize data for 2D block move
move2dall	2D block copy
move2dnonzero	2D block copy non-zero bytes
move2dzero	2D block copy zero bytes

Table 7.24 Two dimensional block transfer instructions

A two dimensional array can be implemented by storing rows adjacently in memory. Given any two 2 dimensional arrays implemented in this way, the instructions provided can copy a section (a block) of one array to a specified address in the other.

To perform a two dimensional move, 6 parameters are required – see figure 7.5. These are

- The address of first element of the source block to be copied. This is called the *source address*.
- The address of first element of the destination block. This is called the *destination address*.
- The number of bytes in each row in the block to be copied. This is called the *width* of the block.
- The number of rows in the block to be copied. This is called the *length* of the block.
- The number of bytes in each row in the source array. This is called the *source stride*.
- The number of bytes in each row in the destination array. This is called the *destination stride*.

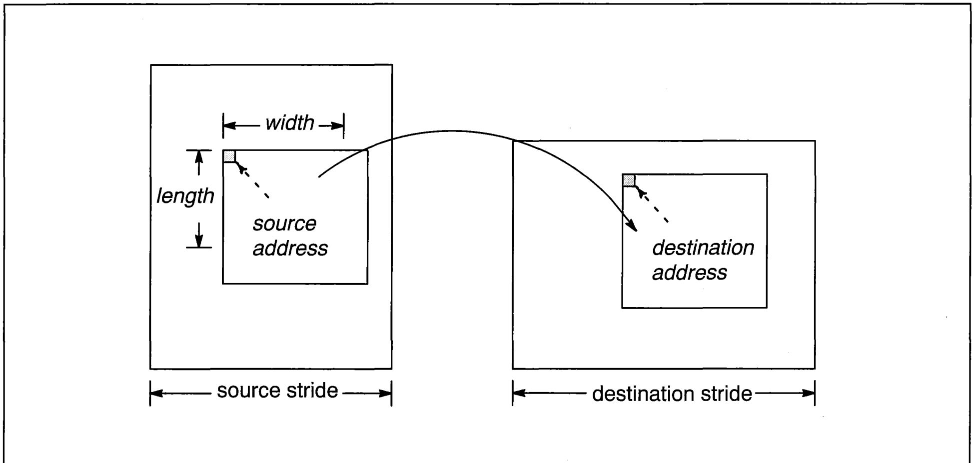


Figure 7.5 Two dimensional block move

The two stride values are needed to allow a block to be copied from part of one array to another array where the arrays can be of differing size.

The *move2dinit* instruction sets up 3 of these parameters. It takes the *source stride* from **Creg** and writes this into **BMReg2**, it takes the *destination stride* from **Breg** and writes this into **BMReg1**, and it takes the *length* from **Areg**, and writes this into **BMReg0**. This must be performed before every two dimensional block move. The integer stack register values are interpreted as unsigned integers, and are undefined after execution.

Each of the 2D block move instructions (*move2dall*, *move2dnonzero*, *move2dzero*) has the *source address* in **Creg**, the *destination address* in **Breg** and the *width* in **Areg** (interpreted as an unsigned integer). These instructions are interruptible (see section 8.2.5), and undefine the integer stack and block move registers after execution.

move2dall copies the whole of the block of *length* rows each of *width* bytes from the source to the destination.

move2dnonzero copies the non zero bytes in the block leaving the bytes in the destination corresponding to the zero bytes in the source unchanged. This can be used to overlay a non rectangular picture onto another picture.

move2dzero copies the zero bytes in the block leaving the bytes in the destination corresponding to the non zero bytes in the source unchanged. This can be used to mask out a non rectangular shape from a picture.

The current process must not be descheduled between the *move2dinit* instruction and the actual 2D move instruction.

None of the two dimensional moves has any effect if either the *width* or *length* of the block to copy is equal to zero. Also a two dimensional block move only makes sense if the *source stride* and *destination stride* are both greater or equal to the *width* of the block being moved. The effect of the two dimensional moves is undefined if the source and destination blocks overlap.

7.15.2 Bit manipulation and CRC evaluation

The instructions listed in table 7.25 allow efficient implementation of some of the low level bit manipulation required in communication protocols etc.

mnemonic	name
bitcnt	count bits set in word
bitrevword	reverse bits in a word
bitrevnbits	reverse bottom n bits in word
crcword	calculate CRC on word
crcbyte	calculate CRC on byte

Table 7.25 Instructions that perform bit manipulation and CRC evaluation

bitcnt counts the number of bits that are set in **Areg**, adds this to the integer in **Breg**, and returns the result in **Areg**. **Creg** is popped up into **Breg**. The use of a register to accumulate the total number of bits set means that this instruction can be used in an inline sequence or a loop to count bits set in an array of words efficiently. Note that a loop using *lend* cannot be used as this has the potential of timeslicing. This instruction has applications in pattern matching and image recognition.

bitrevword reverses the bit pattern of the word held in **Areg**. **Breg** and **Creg** are left unchanged. *bitrevnbits* reverses the bottom **Areg** bits in **Breg**, zeroing all other bits, leaving the result in **Areg** and popping **Creg** up into **Breg**. This result is undefined if **Areg** is greater than the wordlength, or is negative.

These instructions are useful when interfacing the 'little-endian' transputer with other systems that are 'big-endian'.

crcword and *crcbyte* are component instructions in the calculation of the cyclic redundancy check word for a message. This method for checking the correctness of data that has been communicated is based on polynomial division. Both instructions take the data to be processed in **Areg** — though for *crcbyte* it must be in the most significant byte of the word. **Breg** contains the CRC that has already been generated and **Creg** contains the generator. The instruction calculates the CRC by iterating a loop for *BitsPerWord* or *BitsPerByte* iterations. The CRC for one bit is performed by shifting **Breg** and **Areg** left one place as a double word quantity (most significant word in **Breg**) then xor-ing **Creg** into the resulting **Breg** if the bit shifted out of **Breg** was set to 1. At the end the new CRC word generated in **Breg** is left in **Areg** and the generator is left in **Breg**.

Calculating the CRC of a message

The *crcword* and *crcbyte* instructions are designed to be used sequentially in in-line code to enable efficient generation of the CRC of a message.

If a message is word aligned and is a multiple of *BytesPerWord* long, then the CRC can be calculated by loading the generator and the first word of the message into the integer stack. Then each remaining word in turn is loaded and *crcword* applied to it.

The CRC generation can be coded into a loop using *lend*. **Areg** and **Breg** must be preserved over the *lend* instruction in two locals as the process could be timesliced. The following code would evaluate the CRC of a word aligned message *mess* of *len* words.

```
      ldc 0; stl LEDS; ldl len; stl LEDS+1;   — set up 'loop end data structure'
      ldc generator; ldc 0;
      stl temp_crc; stl temp_generator;       — store CRC accumulation (initially zero)
                                              — and generator into temporary variables
L:    ldl temp_generator;                     — load CRC generator
      ldl LEDS; ldl message; wsub; ldnl 0;    — load loop control variable and load
                                              — word from message indexed by
                                              — that variable
      ldl temp_crc; rev;                      — load CRC accumulation
      crcword;
      stl temp_crc; stl temp_generator;       — store CRC accumulation and
                                              — generator into temporary variables
      ldlp LEDS; ldc (END–L); lend;           — load pointer to 'loop end data structure'
                                              — and 'jump offset' prior to testing for loop
                                              — end
END:
```

If the message is not word aligned then more care is needed. *crcbyte* is used to handle any non-word-aligned bytes at either end of the message.

The overhead involved in handling the loop can be reduced by putting more than one *crcword* in-line inside the loop body.

Remember that the transputer is totally 'little-endian' in that more significant data is always to the left of less significant data or at a more positive address. This applies to bits in bytes, bytes in words and words in arrays.

Communications protocols and standard CRCs differ widely in the way they order data so that to calculate the CRC of a message it will often be necessary to use the *bitrevword* and *bitrevnbits* instructions to handle this. Care is needed to ensure that the CRC being calculated is the same as that required and that data is communicated in the correct order. Many protocols make claims of being 'little-endian' or 'big-endian' but this is not always totally correct — for example the CRC is sometimes in the opposite orientation to the data.

1. If *IntegerOverflow* is signalled for these instructions, then a result is returned in **Areg** if and only if trapping is disabled. If a trap is taken, then the contents of the integer stack are undefined when presented to the trap-handler/supervisor. This is discussed in details in section 13.2.2. If no trap is taken, then the result represents the number that would be obtained from repeatedly subtracting 2^{32} from (or adding 2^{32} to) the positive (or negative) number which should be yielded mathematically (if there were no implementation restrictions), until the result is in the range -2^{31} to $(2^{31}-1)$. Hence, for *adc*, *add*, *ladd*, *sub* and *lsub*, the difference between the result and the required result is -2^{32} (or $+2^{32}$), and for *mul* the difference is $n2^{32}$, where n is an integer.

8 Concurrent processes

The transputer is capable of running many processes concurrently. This concept is introduced in section 3.1. The following chapter explains the scheduling mechanism that enables the machine to do this, and discusses how processes communicate.

8.1 Workspace

A process workspace consists of a vector of words in memory. It is used to hold the local variables and temporary values manipulated by the process. The workspace is organized as a falling stack, with 'end of stack' addressing; that is the local variables of a process are addressed as positive offsets from the workspace pointer. Space is allocated and deallocated explicitly using the adjust instructions, and is also allocated implicitly by the procedure call and return instructions.

8.1.1 Process workspace data structure

When a process is descheduled, its workspace pointer is stored as part of the process descriptor (see section 5.3). Certain key data words are stored below the workspace address. This is referred to as the process workspace data structure.

The following slot names are used to name locations relative to a process workspace for an L-process.

word offset	slot name	purpose
0	**pw.Temp**	slot used by some instructions for storing temporary values
–1	**pw.Iptr**	the instruction pointer of a descheduled process
–2	**pw.Link**	the address of the workspace of the next process in scheduling list
	pw.Count	message length in variable length communication
–3	**pw.TrapHandler**	pointer to trap-handler data structure (THDS)
–4	**pw.Pointer**	saved pointer to communication data area
	pw.State	saved alternative state
	pw.Length	length of message received in variable length communication
–5	**pw.TLink**	address of the workspace of the next process on the timer list
–6	**pw.Time**	time that a process on a timer list is waiting for

Table 8.1 Word offsets and names for data slots in a L-process workspace

In the text, a reference such as 'the **pw.Length** slot of the process workspace', would mean the address at an offset of –4 words (–16 bytes) from the workspace address of the process.

Note that in some cases, a word offset is shared by more than one slot name. This is because the location specified by such an offset is used for a number of different purposes at different times. For example when the **pw.Count** slot contains information about the message length, the process is not on a scheduling list and so the location is not required to contain **pw.Link** information.

A small number of instructions — *outword*, *outbyte*, *altwt*, *taltwt*, *disc*, *dist*, *disg*, *diss*, and *altend* — make use of **pw.Temp**. A process must therefore ensure that workspace 0 (**Wptr**+0) is not in use when executing an output instruction, or during the disabling part of an alternative sequence (see section 8.7). Also, care is needed to ensure that the return address of a procedure call, which is stored at (**Wptr**+0) on entry to the procedure, is not lost.

8.1.2 Size of workspace

To ensure that the process workspace data structure is not overwritten, each process must be allocated workspace in addition to that for the local variables. The extra locations are immediately below the workspace pointer address, held in **Wptr**.

process with no i/o	3 words
process with only unconditional i/o using *in* and *out*, or *vin* and *vout*	4 words†
process with only unconditional i/o using *outbyte* or *outword*	4 words
process with alternative input	4 words†
process with timer input	6 words
process with alternative timer input	6 words†

8.2 Scheduling and priority

The concept of scheduling lists and the two priority levels provided by the machine, were introduced in section 3.1. This section provides some more detail on scheduling, descheduling, timeslicing and interrupts.

8.2.1 The current process, the null process, and scheduling lists

The descriptor (see section 5.3) of the process that is currently 'executing' is held in the workspace descriptor register (**WdescReg**). This process is referred to as the 'current process'. If the 'null process' is executing (i.e. no process is executing) then the workspace descriptor register contains the special value *NotProcess.p*. In section 3.1, the term 'active set' was defined. A process is in the active set if:–

- it is the current process,
- it is the interrupted process, or
- it is on one of the scheduling lists.

When a process belongs to this set it may be described as 'running', 'scheduled', or 'active'. A process on one of the scheduling lists may be described as 'queued'.

The processor maintains two lists of processes which are ready to run, one for each priority level. Each list is a linked list of workspace data structures for processes which are ready to be executed, not including the current or interrupted processes. When a process is run, it is added to the end of the appropriate list. When the current process is timesliced, it is placed at the end of the appropriate scheduling list and the new current process is taken from the front of the list.

The front pointer registers, **FptrReg0** and **FptrReg1**, contain the workspace pointers for the next process to be executed at high and low priority respectively. Similarly the back pointer registers, **BptrReg0** and **BptrReg1**, contain the workspace pointers for the last process on each list. Each queued process (except the last in each list) holds, in the **pw.Link** slot of the process workspace data structure (section 8.1.1), a pointer to the workspace of the next process scheduled at the same priority.

The scheduler operates in such a way that non-executing processes do not consume any processor time. Figure 8.1 shows the workspace area and code for processes *P*, *Q*, *R* and *S*, where *S* is the currently executing process, and *P*, *Q* and *R* are active, awaiting execution. Only the low priority process queue registers are shown; the high priority processes are queued and executed in a similar manner.

† In addition to the *below* workspace data structure, these processes also use the **pw.Temp** slot (not included in the above figures). For example, a process cannot use **pw.Temp** as a local variable while performing an alternative input.

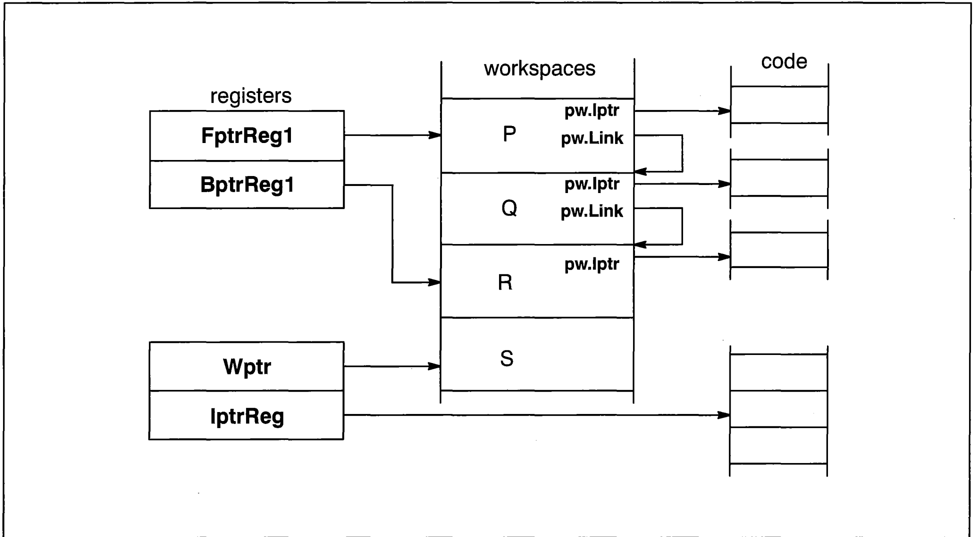


Figure 8.1 Linked scheduling list

Manipulating the scheduling lists

The instructions *swapqueue* and *insertqueue* have been provided for manipulation of the scheduling lists. They are described in section 13.6.

8.2.2 Descheduling

When a process ceases to be the current process, it is said to be 'descheduled'. As described in section 3.1, a process may be descheduled for a number of reasons. Descheduling can only occur after execution of certain instructions. These are called 'descheduling points' and are listed in table 8.2. Some of these instructions have not yet been introduced and are left to later sections.

j	*altwt*	*endp*	*grant*	*in*
lend	*out*	*outbyte*	*outword*	*selth*
stopch	*stopp*	*taltwt*	*timeslice*	*tin*
vin	*vout*	*wait*		

Table 8.2 List of descheduling points

When a process is descheduled, its instruction pointer is stored in the **pw.Iptr** slot of the process workspace data structure and execution of the process is suspended. The integer and floating-point stack registers are not saved when a process is descheduled. This means that an L-process must not attempt to transfer any information in these stacks across any descheduling points. If the process is descheduled, other processes may corrupt the stacks before it is rescheduled. [Note that this is not a concern for P-processes because these do not deschedule. When a P-process takes a trap, the integer state is saved in the P-state data structure.]

8.2.3 Rescheduling after communication

When a communication completes, the waiting process is placed on the end of the relevant scheduling list. The transputer determines the priority of this process by examining the least significant bit of its descriptor (refer to section 5.3). If the waiting process is a high priority process and the transputer is currently running a low priority process, then the waiting process interrupts the current process.

8.2.4 Clocks and timeslicing

mnemonic	name
sttimer	store timer
timeslice	timeslice
settimeslice	set timeslicing status

Table 8.3 Instructions which relate to the timer and timeslicing

The processor contains two clock registers, one for each priority. These registers start incrementing after the processor has been reset only after a store timer — *sttimer* — instruction has been executed. *sttimer* is described in section 13.6.

The high priority clock register ticks (increments) every 1μs and the low priority clock increments every 64μs.

After every 256 ticks of the high priority clock (i.e. 256 μs), a timeslice period is said to have ended. When two timeslice periods have ended while the same low priority process has been continuously executing, the processor will force a timeslice at the next timeslicing point. For an L-process, the only timeslicing points are the instructions *j* or *lend* – i.e. the processor may timeslice after execution of either of these instructions. For a P-process, a timeslicing point is any point at which an interrupt can occur – including during execution of an interruptible instruction (section 8.2.5). See table 8.4.

timeslicing point	note
j†	
lend	
all interrupt points (see section 8.2.5)	protected mode only
† except *j 0* – see section 10.3.1	

Table 8.4 Timeslicing points

For a low priority L-process, a timeslice forcibly deschedules the current process, and immediately reschedules it by placing it at the end of the low priority scheduling list. The next waiting process becomes the current process. Note that since *j* and *lend* may cause a current L-process to deschedule, they are also descheduling points and so have been included in the list given in table 8.2. The same consideration therefore holds for stack corruption across timeslicing points as across descheduling points (see subsection 8.2.2).

For a low priority P-process, a timeslice causes a trap to the supervisor.

The processor does not timeslice high priority processes.

In low priority processes, timeslicing can be disabled with the *settimeslice* instruction. This is discussed in section 13.5.

A timeslice can be forced by execution of the *timeslice* instruction. This will force a timeslice to be taken by a *high* (as well as a low) priority process; and is effective whether or not timeslicing or interrupts are enabled. This is discussed in more detail in section 13.5.

Processes that are not timesliced execute until they are descheduled for another reason (e.g. communication), unless they are interrupted.

Operations for reading the clock values and comparing these values, are described in section 8.5.

8.2.5 Priorities and interruption

The processor can execute processes at one of two priority levels, one level for urgent (high priority) processes, one for less urgent (low priority) processes. A high priority process will always execute in prefer-

ence to a low priority process if both are able to do so. If a high priority process becomes able to run whilst a low priority process is executing, the low priority process is temporarily stopped — 'interrupted' — and the high priority process is executed. When there are no high priority processes able to run, the interrupted process continues executing.

A low priority process may be interrupted after it has completed execution of any transputer instruction†. Furthermore, to minimize the time taken for an interrupting high priority process to start executing, the potentially time consuming instructions shown in table 8.5 are interruptible – i.e. they may be interrupted in the middle of execution. (Some of these instructions have not yet been introduced and are left to later sections.) The points at which a process may be interrupted – i.e. at the end of any instruction or during execution of an interruptible instruction – are referred to as 'interrupt points'.

move	*move2dall*	*move2dzero*	*move2dnonzero*		
in	*vin*	*out*	*outbyte*	*outword*	*vout*
tin	*taltwt*	*dist*			
fprem					
readhdr	*writehdr*				

Table 8.5 List of interruptible instructions

When an interrupt occurs, the process state of the currently executing low priority process, is saved into shadow registers (section 5.1.1), and the high priority process starts to execute. This state is known as the 'shadow state'. Execution of the low priority process is interrupted until no high priority processes are able to run; at which point, the state registers are reloaded from the shadow registers, and execution of the interrupted process continues. There can only ever be at most one interrupted process. Note that the interrupted process is *not* placed onto the low priority scheduling list.

8.2.6 Scheduling/descheduling of L-processes

As mentioned in section 3.1, the processes which are queued on the IMS T9000 scheduling lists are known as L-processes. The status and control bits that relate to an L-process are discussed in section 5.2. The following discusses the scheduling and descheduling mechanisms.

Prior to executing any L-process, the processor loads the trap-handler pointer (stored in the **pw.TrapHandler** slot) into the trap-handler register. This is a pointer to an area of store called a trap-handler data structure (THDS) (detailed in section 10.1.1), which identifies the trap-handler to be run if a trap occurs while running this L-process. Such a trap-handler may be shared by any number of L-processes. Amongst other information, the THDS holds the status and control bits for the L-processes which share it, and upper and lower addresses which specify a watchpoint region (see chapter 14 for further details). When the processor loads the trap-handler register, it also loads the status and control bits for the process from the THDS into the status register, and loads the watchpoint registers (**WlReg** and **WuReg**) if appropriate. When an L-process is descheduled, the processor copies the status and control bits from the status register into the THDS. These status and control bits are therefore *local* to all L-processes which share the trap-handler, and the mechanism described ensures that the status is preserved through execution of other processes which do not share that trap-handler.

8.3 Initiation and termination of processes

All processes must have an area of memory reserved as their workspace — this holds the process' local variables etc. The allocation of space to concurrent processes can often be performed by a compiler, eliminating the overheads of dynamic storage allocation. However, the transputer instructions also allow fully dynamic process initiation and termination.

Initiation and termination of concurrent process can be performed by the instructions shown in table 8.6. With the exception of *ldpri*, all these instructions are privileged.

† Interruption cannot occur after prefix instruction components (see chapter 6). Hence for a secondary instruction, it can only occur during (if interruptible) or after the *opr* instruction component.

mnemonic	name
startp	start process
endp	end process
runp	run process
stopp	stop process
ldpri	load current priority
ldth	load trap-handler

Table 8.6 Instructions for starting and terminating processes

8.3.1 Scheduling parallel processes

The instruction *startp* can be used to run a process in parallel. This can be repeatedly executed to run several processes concurrently. Each parallel process signals completion of execution with the instruction *endp*. The latter instruction interacts with a parallel process data structure to achieve correct synchronization.

Setting up the parallel process data structure

The parallel process data structure is shown in table 8.7. This must be word-aligned. The **pp.Count** slot holds an unsigned count of the number of parallel processes that have not yet terminated. It should hence be initialized to the number of processes that are to be run concurrently. The **pp.IptrSucc** slot should be set to point to the first instruction to be executed by the 'successor process' (i.e. the process that runs when all the parallel processes have terminated).

This data structure also becomes the workspace of the successor process.

word offset	slot name	purpose
1	**pp.Count**	contains unsigned count of parallel processes
0	**pp.IptrSucc**	contains pointer to first instruction of successor process

Table 8.7 Parallel process data structure

Starting a concurrent process

The instruction *startp* schedules a new concurrent process of the same priority as the current process. **Breg** contains the offset from the address of the next instruction to the first instruction of the new process. **Areg** contains the address of the workspace of the new process — this must be a word-aligned address. The processor schedules this new process by placing it at the end of the appropriate scheduling list, and initializing its processes workspace data structure with **pw.Iptr** set to the absolute address of the first instruction, and **pw.TrapHandler** pointing to the data structure of current trap-handler (hence the new process will share the current trap-handler). This instruction does not deschedule the current process, but it does undefine the integer and floating-point stack registers.

Terminating a concurrent process

The instruction *endp* terminates the current process, which is one from a set of processes running concurrently. **Areg** contains a pointer to the parallel process data structure (see table 8.7).

endp has the effect of decrementing the value held in the **pp.Count** slot of the parallel process data structure. If the value of **pp.Count** is1 before execution, indicating that the current process is the last of the concurrent processes to terminate, then it starts the successor process.

Compiling PAR

In a high-level language that supports concurrency, there may be a parallel construct. In occam this is the PAR construct (see chapter 2).

```
PAR
  P1
  P2
  ⋮
  Pn
```

startp and *endp* can be used to implement this. Consider the following high-level code.

```
SEQ
  PredProc
  PAR
    ParProcA
    ParProcB
    ParProcC
  SuccProc
```

An example of how this may be implemented is shown below. An explanation is given in the paragraphs that follow and the workspace layout is illustrated in figure 8.2.

```
      PredProc;          — process PredProc
      ldc 3; stl 1;      — store parallel process count into the pp.Count of the
                         —        parallel process data structure (in this case
                         —        the current process workspace)
      ldc (L5–L6); ldpi; — store the address of instruction to be executed after the
L6:   stl 0;             —        parallel construct, into pp.IptrSucc
      ldc (L1–L2);       — load into the integer stack, the offset address of the first
                         —        instruction of process ParProcA
      ldlp WA;           — load into the integer stack, the workspace address of
                         —        process ParProcA
      startp;            — run process ParProcA
L2:   ldc (L3–L4);       — load into the integer stack, the offset address of the first
                         —        instruction of process ParProcB
      ldlp WB;           — load into the integer stack, the workspace address of
                         —        process ParProcB
      startp;            — run process ParProcB
L4:   ParProcC;          — process ParProcC
      ldlp 0;            — load into the integer stack, the address of the parallel
                         —        process data structure (in this case the
                         —        same as the current process workspace)
      endp;              — terminate ParProcC
L1:   ParProcA;          — process ParProcA
      ldlp –WA;          — load into the integer stack, the address of the parallel
                         —        process data structure (in this case the
                         —        same as the workspace used by PredProc)
      endp;              — terminate ParProcA
L3:   ParProcB;          — process ParProcB
      ldlp –WB;          — load into the integer stack, the address of the parallel
                         —        process data structure (in this case the
                         —        same as the workspace used by PredProc)
      endp;              — terminate ParProcB
L5:   SuccProc;          — process SuccProc
```

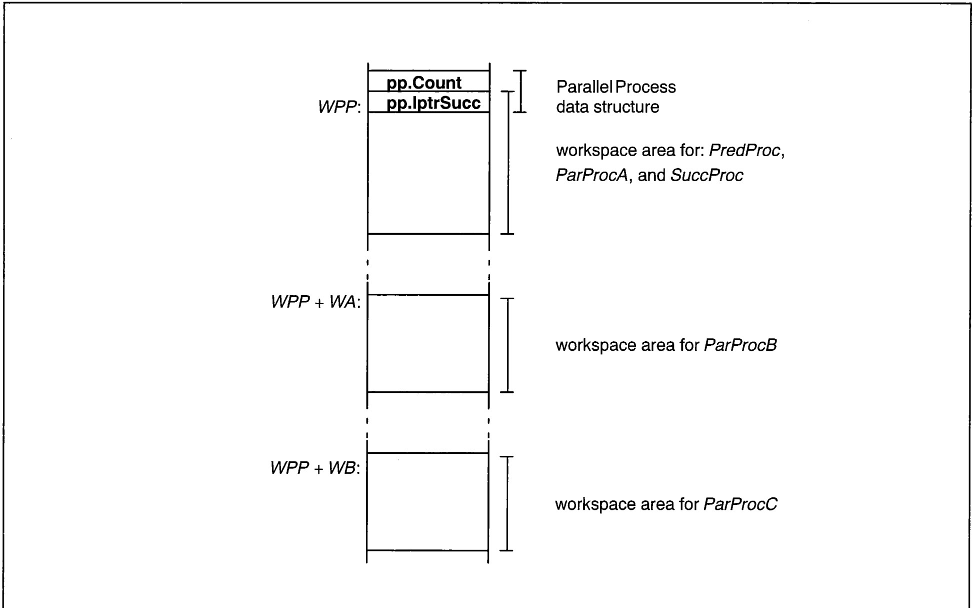


Figure 8.2 Example of workspace layout for compiling parallel processes

where *WPP* is the workspace address of *PredProc*, *WA* is the offset from the workspace of *PredProc* to that of *ParProcA*, and *WB* is the offset from the workspace of *PredProc* to that of *ParProcB*.

The 'predecessor process' (i.e. the process that sets up the parallel construct) must set up the parallel process data structure before it schedules the other concurrent processes. Note, when the predecessor has finished starting all the parallel processes, that it will itself become a component of the parallel construct. It must therefore include itself in the parallel process count when assigning **pp.Count**.

In this example, the processes *PredProc*, *ParProcC* and *SuccProc* share the same workspace. Firstly the code representing the process *PredProc* (the predecessor processor) runs. Then the three parallel processes (*ParProcA*, *ParProcB* and *ParProcC*) run concurrently. Finally when all three of these processes have terminated, the process *SuccProc* runs.

Since the parallel process data structure provides the workspace of the successor process (*SuccProc*), and (in this example) the successor process has the same workspace as the predecessor process (*PredProc*), the latter initializes the data structure by setting (**Wptr**+1) (which is the equivalent to **pp.Count**) and (**Wptr**+0) (which is the equivalent to **pp.IptrSucc**), to the required values.

In general, there is for each parallel process (component of the parallel construct), a start sequence, the code for the process itself, and a terminating sequence. The start sequence is executed by the predecessor process, and comprises loading the integer stack with the appropriate parameters and executing *startp*. Note that *ParProcA* and *ParProcB* are by default, processes with the same priority as *PredProc* (and hence *ParProcC*, which inherits the priority of *PredProc*). The terminating sequence is executed by the parallel process component itself, and comprises loading the integer stack with a pointer to the parallel process data structure (workspace address of the continuation process) and executing *endp*. *SuccProc* will not start until *ParProcA*, *ParProcB* and *ParProcC* have all terminated. Note that *ParProcC* does not have a start sequence because it automatically supercedes the current process (*PredProc*), but it does have a terminating sequence because this is essential for synchronization.

Since the processes *ParProcA*, *ParProcB* and *ParProcC* are run concurrently, it is not known which process will finish executing first. The *endp* instructions may thus be executed in any order. The concurrent scheduling of processes has been described in section 8.2.

Be aware that the above is just one way that a parallel construct could be implemented. The continuation process *SuccProc* could for example use a completely different workspace to *PredProc.*

8.3.2 Other scheduling instructions

The *runp* instruction schedules an L-process, the descriptor of which is held in **Areg**. The workspace pointer field in this process descriptor points to a process workspace data structure, in which the **pw.Iptr** slot contains the value to be loaded into **IptrReg** when the process is scheduled, and the **pw.TrapHandler** slot contains a pointer to a THDS. *runp* can be used to start an L-process at either priority level by setting or clearing the priority bit in the process descriptor in **Areg**. This instruction does not deschedule the current process, but it does undefine all integer and floating-point stack registers.

The instruction *stopp* simply terminates the current process, saving the value of **IptrReg** and **ThReg** in the process workspace data structure. The process is not put onto the scheduling lists so to restart it a *runp* instruction is needed.

The instruction *ldpri* pushes the value of **Priority** (the priority of the current process – see section 5.3) onto the integer stack.

The instruction *ldth* pushes the pointer to the THDS (see section 3.3) of the current process onto the integer stack. More detail on this instruction and other trap-handling instructions are given in chapter 10.

If *J* is a variable holding the address of the first instruction of a new process, and *W* holds the address of a workspace for the process, then the following code sequence will start a process that shares its trap-handler with the current process and has the same priority.

ldth; ldl W; stnl –3;	— store a pointer to the current process's THDS into the
	— **pw.TrapHandler** slot of the new process's
	— workspace data structure (*W*)
ldl J; ldl W; stnl –1;	— store the address of the first instruction that the new process
	— should execute (*J*) in the **pw.Iptr** slot of its
	— workspace data structure (*W*)
ldl W;	— load the workspace address into the integer stack
ldpri; or;	— form process descriptor by combining the workspace
	— address with the priority of the current process
runp;	— schedule the L-process specified by the descriptor formed
	— above

If it is known that the current priority is high, then

ldth; ldl W; stnl –3; ldl J; ldl W; stnl –1; ldl W; runp;

will start an L-process at the same priority, sharing the same trap-handler. Here, because *W* is word-aligned, the bottom bit of the process descriptor, which represents **Priority**, is 0 (hence specifying high priority).

8.4 Channel communication, synchronization and data-transfer

Section 3.2 introduces the concepts of channel communication, synchronization and data-transfer. This section introduces the various channel implementations, discusses synchronization, details the different types of channel communication, and finally explains how internal channels are implemented and initialized.

8.4.1 Channels

A channel is used for synchronization and data-transfer between two processes. It may be implemented either by a word in memory – for communication between processes on the same transputer (internal channel), or by an external link – for communication between transputers or between a transputer and

an external device (external channel). A channel is uniquely identified by a 'channel address'. For an internal channel, the channel address corresponds to a memory location which may be allocated by the compiler. For an external channel, the channel address belongs to a range of addresses reserved for external channels (this is discussed further in chapter 12).

A process can be written and compiled without knowledge of whether its channels are connected to other processes on the same transputer, or other transputers. The same instruction sequence is used in both cases. That is, a process that uses instructions to communicate on a channel, does not need to know how the channel is implemented. However some instructions can only be used on specified implementations. The various implementations for channels are: internal channels, virtual channels, byte-stream channels, and event channels. Virtual, byte-stream and event channels are referred to generically as external channels. The instructions determine the channel implementation at run-time.

- An internal channel (sometimes referred to as a soft channel) connects processes that are on the same processor.
- A virtual channel allows communication between processes on different IMS T9000 transputers. These transputers do not have to be directly connected, provided there is a connecting path via a communications network.
- A byte-stream channel allows communication between a process on an IMS T9000 transputer and a process on a neighboring T2/T4/T8-series transputer. (N.B. Links between these transputers must be connected via a system protocol converter. See the *IMS C100 data sheet – document number 42 1475 01* – for details.)
- An event channel allows a communication between a process and an external device. This type of channel can carry no message: it is purely for synchronization. For example one use of the event channel is as an external interrupt input.

8.4.2 Synchronization

Before data-transfer can occur, there must be synchronization between the inputting and outputting process. The general philosophy is that there are always two sides to any communication, i.e. an input and an output. There are three ways of achieving synchronization, which are specific to channels. (More generally the semaphore mechanism provided on the IMS T9000 – section 8.6 – can also be considered as a synchronization mechanism.)

Simple synchronization

Simple synchronization occurs when one process is executing an input instruction, and another process executing an output instruction, on the same channel.

When a communication instruction is executed (either input or output) on a channel, the action taken depends on the type of channel and on whether or not a complementary communication instruction (output or input respectively) has already been executed on that channel.

If a process executes a communication instruction on

(i) an internal channel on which there has been *no* previous complementary execution of a communication instruction

(ii) an external channel

then the process is descheduled. That is, the current process is descheduled until the another process executes the other communication instruction. By the time the process is rescheduled, the data-transfer will have occurred.

If, when a process executes a communication instruction on an internal channel, there *has* been a complementary execution of a communication instruction on the same channel, then the data-transfer can take place immediately.

For simple synchronization, the only instructions that are required are the input and output instructions which also handle the data-transfer. No extra instructions are needed to establish the communication. This is not the case with the other forms of synchronization which are discussed below.

Alternative synchronization

The transputer provides a synchronization mechanism that enables one channel to be selected for input, from a number of 'alternatives'. That is, several inputs are considered for synchronization, but only one is selected. If no inputs are ready immediately, then the inputting process is descheduled until one is ready. Each input has an associated piece of code that is run if selected. When an input is selected, the inputting process synchronizes with the process that has executed an output instruction on the same channel, and the data-transfer can be completed by the selected code segment. The instructions that are used to implement this synchronization are known as 'the alternative sequence'. The mechanism and related instructions are described in greater detail in section 8.7.

The alternative sequence thereby gives the user the capability of implementing a construct where one communication is selected from several possibilities (e.g. the occam ALT construct).

Resource synchronization

Resource synchronization provides a means of implementing many-to-one communication.

One commonly occurring concurrent process paradigm is known as the client-server model. In this model, there may be several processes acting as servers and several processes acting as clients to those servers. Any one of the clients may at any time want to communicate with any one of the servers. Since any server can be accessed by a number of clients, it may receive requests at a faster rate than it can deal with them.

The resource mechanism provides a direct way of implementing the client-server model. It provides a resource queue (data structure), which can be associated with a server, so that channel connections can be allocated in a fair way. It also provides a resource mode for channels that are to be used in this way. A channel in resource mode is associated with a particular resource. When an output is made on such a channel, the behavior depends on whether the resource is waiting. If it is, then the communication can be immediately synchronized ready for data-transfer. Otherwise, the outputting process (client) is descheduled, and the channel is attached to the resource queue until it is selected. If the inputting process (server) is ready but has no clients to service, then this is descheduled until a client becomes ready (makes a 'claim'). The name given to a channel which can be set to resource mode, is a 'resource channel'. This is described in greater detail in section 8.8.

8.4.3 Communication

The three synchronization mechanisms discussed in section 8.4.2, can all be used in conjunction with zero, fixed or variable length communication.

- In zero-length communication, there is synchronization but no data-transfer (no message is passed). It is just used to synchronize the processes which are connected by the channel on which the communication occurs. Zero-length communication can occur on internal channels, virtual channels or event channels, but cannot occur on byte-stream channels.

- In fixed-length channel communication, there is synchronization and data-transfer. The data-transfer requires both communicating processes to have knowledge of the length of the message that is to be transferred. Fixed-length communication can occur on internal channels, virtual channels or byte-stream channels, but cannot occur on event channels.

- In variable-length communication, there is synchronization and data-transfer. The data-transfer requires only the outputting process to have knowledge of the length prior to transfer, while the inputting process specifies a maximum allowable length. The actual message length is communicated to the inputting process at the time of transfer, along with the message itself. Variable-length communication can occur on internal channels or virtual channels, but cannot occur on event channels or byte-stream channels.

It is guaranteed that both processes are in the active set immediately after completion of any of these communications. If a process had been descheduled while waiting for the process at the other end to communicate, then this process is rescheduled by its host processor.

Occam has operations which directly implement input from and output to a channel to enable synchronized transfer of messages between concurrent processes. The syntax for each operation is as follows, where <channel> specifies a channel which has previously been declared using the keyword CHAN, and <message> is the data which is to be transferred.

input	<channel> ? <message>
output	<channel> ! <message>

These operations are used below to demonstrate the IMS T9000 communication instructions: *in*, *out*, *vin*, *vout*, *outbyte*, and *outword*.

If the receiving process knows the length of the message that is to be received, fixed-length communication can be used. Otherwise, either the length of the communication must be communicated first, or variable-length communication must be used.

N.B. It is not recommended that a message length larger than half of the address space (2 Gbytes) is used to transfer data in the same processor memory space – hence message transfer on internal channels should adhere to this.

Fixed-length communications

mnemonic	name
in	input message
out	output message

Table 8.8 Fixed length i/o operation instructions

A process inputs a message of known size by executing the instruction *in*. The process loads:–

- the address of the destination buffer (where the message is to be stored) into **Creg**,
- the address of the channel into **Breg**,
- and the length of message in bytes (as an unsigned integer) into **Areg**;

and then executes *in*. For example

c? *v* = *address(v); address(c); length(v); in*

where the *address* is defined on page 29 and *length* is defined in section 7.5.4.

If the channel is an internal channel that has already had an output performed on it, then there is a message waiting and this is transferred to the buffer specified by **Areg** and **Creg**. If the channel is an external channel or an empty internal channel then the process stores the pointer to the destination buffer in the **pw.Pointer** slot of the process workspace data structure, and deschedules.

A process outputs a message of known size by executing instruction *out*. The process loads:–

- the address of the source buffer (where the message is currently stored and from where it is to be copied) into **Creg**,
- the address of the channel into **Breg**,
- and the length of message in bytes (as an unsigned integer) into **Areg**;

and then executes *out*. For example

c! *v* = *address(v); address(c); length(v); out*

If the channel is an internal channel that has already had an input performed on it, then there is a destination buffer ready and the message specified by **Areg** and **Creg** is transferred to this buffer. If the channel is an external channel or an empty internal channel then the process stores the pointer to the message in the **pw.Pointer** slot of the process workspace data structure, and deschedules. (There are further considerations for the output instruction if the channel is a resource channel or has been enabled by an *enbc* instruction, but this detail is left until those mechanisms are explained.)

The contents of the integer and floating-point stack registers are undefined after execution of *in* or *out*. These instructions are privileged and interruptible, and are descheduling points. They signal *IntegerError* if **Breg** does not contain a valid channel address (see section 12.4), and signal *Unalign* if the channel address is not word-aligned. Both the input and output ends of a communication should have the same value in **Areg** when executing *in* and *out*. These instructions may be applied to any channel (internal or external).

If different lengths are specified by the *in* and *out* instructions, then the behavior is undefined.

Variable-length communications

mnemonic	name
vin	variable-length input message
vout	variable-length output message
ldcnt	load message byte count

Table 8.9 Variable length i/o operation instructions

The previous type of communication is called fixed-length because both the sending and receiving process need to know the length of the message prior to transfer. In variable-length communication, it is only necessary for the sending process to know the exact length of the message prior to transfer (although the receiving process must know the maximum length). If the *in* and *out* instructions are to be used for transferring variable-length messages, then some protocol is needed by which the length is communicated before the actual message. The IMS T9000 provides three more instructions to obviate the need for such protocol.

To allow the secure and efficient communication of variable length data, the *vin* and *vout* instructions may be used instead of *in* and *out*. *vout* may send a message of any length. *vin* provides an input buffer to receive the message. When the transaction is complete, the inputting process can find out the length of the message using a special instruction for that purpose (*ldcnt*). These instructions may be applied to internal channels, virtual channels or event channels.

When both a *vin* and a *vout* instruction have been executed by processes referring to the same channel, data is transferred from the outputting process to the inputting process in a similar way to a communication which uses *in* and *out*. However, in the case where the length specified by *vout* exceeds that specified by *vin*, the message cannot be transferred. Whether or not the data-transfer has been successful, it is guaranteed that both processes are in the active set immediately after the communication. An additional instruction, *ldcnt*, is provided to enable the inputting process to determine how much data (if any) has been transferred or whether an error has occurred. Note that it is not possible to mix the use of fixed length and variable length transfer in a single communication. For example if *out* is used to transmit a message, it is not possible to receive that message with *vin*.

The mechanism for a variable length communication is similar to that described for a fixed-length communication. The following descriptions detail the differences.

vin

A process inputs a message of variable length by executing the instruction *vin*. The process loads:–

- the address of the destination buffer (where the message is to be stored) into **Creg**,
- the address of the channel into **Breg**,

- and an unsigned integer representing the maximum message length in bytes into **Areg**;

prior to executing *vin*. The maximum message length must be less than *MostPosUnsigned*. The values in the integer and floating-point stack registers are destroyed by the instruction.

This instruction is privileged, interruptible and a descheduling point. It signals *IntegerError* if **Areg** contains *MostPosUnsigned*. It also signals *IntegerError* if **Breg** does not contain a valid channel address, and signals *Unalign* if the channel address is not word-aligned.

When *vin* is executed, the behavior depends on whether or not the message being received is longer than the maximum length **Areg**. The data is only transferred successfully if message length is less than or equal to the maximum. There is a special **pw.Length** slot in the workspace to give information about the length or error status of the message.

When the communication is successful, the complete message is transferred into the destination buffer and the length of the message is stored in the **pw.Length** slot.

If the message is too long, the complete message cannot be received and a special value *LengthError.p* is written into the **pw.Length** slot to indicate that there has been an error. Whether or not part of the message is received, is undefined. Data is never written into any part of memory not specified by the input process, hence ensuring that buffer overflow does not occur.

vout

A process outputs a message in a variable-length communication by executing the instruction *vout*. The process loads:–

- the address of the source buffer (where the message is currently stored and from where it is to be copied) into **Creg**,
- the address of the channel into **Breg**,
- and an unsigned integer representing the message length in bytes into **Areg**;

prior to executing *vout*. The values in the integer and floating-point stack registers are destroyed by the instruction.

This instruction is privileged, interruptible and a descheduling point. It signals *IntegerError* if **Breg** does not contain a valid channel address, and signals *Unalign* if the channel address is not word-aligned.

Where the value in **Areg** is longer than the maximum message length specified by the receiving process, the complete message cannot be sent. Whether or not part of the message is sent, is undefined.

ldcnt

ldcnt pushes the length of a successfully received message onto the integer stack. Otherwise, it signals *IntegerError* to indicate that the previous *vin* instruction did not successfully receive a message because the message available on the channel was too long.

The processor obtains the message length from the **pw.Length** slot of the current process workspace data structure (pointed to by **Wptr**), and this is copied into **Areg**. Hence if the communication is unsuccessful, it copies the special value *LengthError.p* onto the stack.

Note that for this instruction to yield correct information, it should be used after *vin* and before any subsequent descheduling point, timeslicing point, adjust workspace operation or procedure call. It is a privileged instruction.

Zero-length communications

Where synchronization is required between processes, but no message needs to be transferred, zero-length communication can be used. This comprises the synchronization stage of a channel communica-

tion but not the data-transfer. Either of the above pairs of communication instructions can be used for zero-length communication – i.e. *in*/*out* or *vin*/*vout*. In both cases the message length must be set to zero when passed to the appropriate instructions.

Zero-length is the only form of communication which is allowed on an event channel, because messages cannot be transferred on these channels. (If a non-zero message length is specified on an event channel, then a zero length communication will still occur.)

Zero-length communication cannot occur on byte-stream channels because synchronization cannot occur unless at least one byte of data is transferred.

Single word and byte transfer

The common cases of single word and byte transfer can be optimized.

mnemonic	name
outbyte	output byte
outword	output word

Table 8.10 Single byte transfer instructions

Byte transfer

outbyte outputs the single byte in the least significant byte of **Areg** down the channel specified in **Breg**. It uses the slot **pw.Temp** as a temporary variable. The three most significant bytes of **Areg** are ignored.

If *a* and *b* are both single byte elements and *e* is a byte valued expression then compiled code for the transfers are

```
c? b    =    address(b); address(c); ldc 1; in
c! e    =    address(c); e; outbyte
```

Word transfer

outword outputs the single word in **Areg** down the channel specified in **Breg**. It uses the slot **pw.Temp** as a temporary variable.

If *x* and *y* are both single word elements and *e* is a word valued expression then compiled code for the transfers are

```
c? x    =    address(x); address(c); ldc BytesPerWord; in
c! e    =    address(c); e; outword
```

[When the target wordlength is unknown (e.g. for code that must operate on transputers of different wordlengths), single word channel input can be generalized to

address(x); address(c); ldc 1; bcnt; in

outbyte and *outword* are privileged instructions, and are descheduling points. They signal *IntegerError* if **Breg** does not contain a valid channel address, and signal *Unalign* if the channel address is not word-aligned.

8.4.4 Implementation of channels

Initializing internal channels

In a high-level language that supports concurrency, there may be a data type that supports message passing. In occam there is a channel type, which is declared with the keyword CHAN. The INMOS 'C' compiler also provides a channel type – `Channel`. A channel provides unbuffered, unidirectional point-to-point communication of values between two concurrent processes.

An internal channel is represented by a word in memory. The address of this word is the channel address, which is passed via **Breg** in the communication instructions. Before a memory location can be used as a channel, it must be initialized to *NotProcess.p* to indicate that no process is waiting for communication on that channel. This value can be obtained by the *mint* instruction. It is convenient to do this when a channel declaration is executed. For example

```
CHAN OF PROTOCOL c :

    mint; stl c;                    — initialize channel by setting memory address
                                    —       corresponding to channel to
                                    —       NotProcess.p
```

or

```
[n]CHAN OF PROTOCOL c :

    ldc 0; stl (LEDS+le.Index);     — set up the loop end data structure with initial
    ldc n; stl (LEDS+le.Count);     —       Index=0 and Count=n
L:  mint; ldl c; ldl (LEDS+le.Index);  — initialize channel at memory word address
    wsub; stnl 0;                   —       (c+Index)
    ldlp LEDS; ldc (END–L); lend;   — load the address of the loop data structure
                                    —       and the offset for the next instruction
                                    —       into the integer stack prior to
                                    —       executing lend
END:
```

The input and output instructions use the memory location (channel word) to provide synchronized communication between two concurrent processes. After each communication, the store location is returned to its initial value, *NotProcess.p*. This general mechanism of message transfer across internal channels is discussed below in more detail.

Implementation of internal channel communication

An internal communication occurs when two processes on the same processor communicate. Because a transputer is only executing one process at any time, one of these two processes will become ready to communicate first. For example, in figure 8.3, process *P* is about to execute an *output* instruction, specifying a message to be transferred.

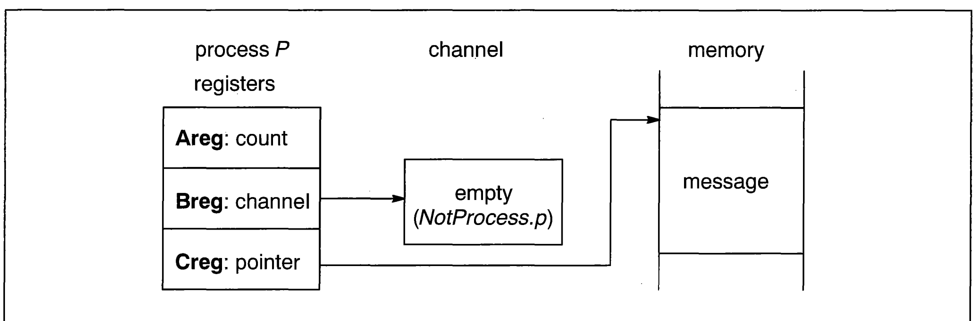


Figure 8.3 Output to empty channel

When the first process to become ready to communicate executes its communication instruction, the transputer will find the value *NotProcess.p* in the channel word — this signifies that the other process is not ready to communicate.

The process then

- copies the current process descriptor from the workspace register (**WdescReg**) into the channel word
- writes the address of its communication area to the **pw.Pointer** slot of the current process workspace data structure – if the first communication instruction is an input, this will be a pointer to the buffer area where the message should be placed when received – if the first communication instruction is an output, this will be a pointer to the message that needs to be transmitted
- (for variable-length only) writes length / maximum length of message into the **pw.Count** slot of the current workspace data structure
- deschedules itself

Figure 8.4 shows the state after *output* has executed.

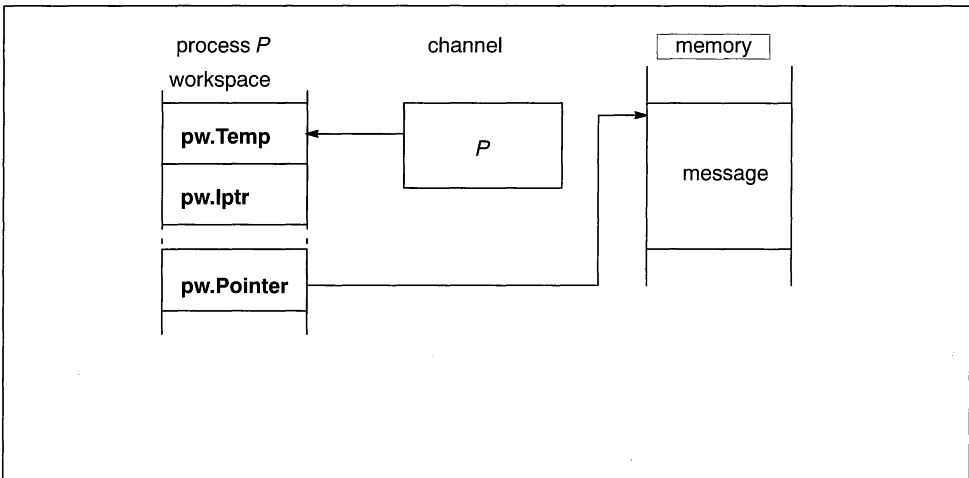


Figure 8.4 Process descriptor of first process left in channel

When the second process to become ready to communicate executes its communication instruction, the transputer reads the value in the channel word and finds the value is a process descriptor (i.e. not *NotProcess.p)*, thus identifying the process that is waiting to communicate. Figure 8.5 shows process *Q* about to *input* from the channel that is being *output* to by process *Q*.

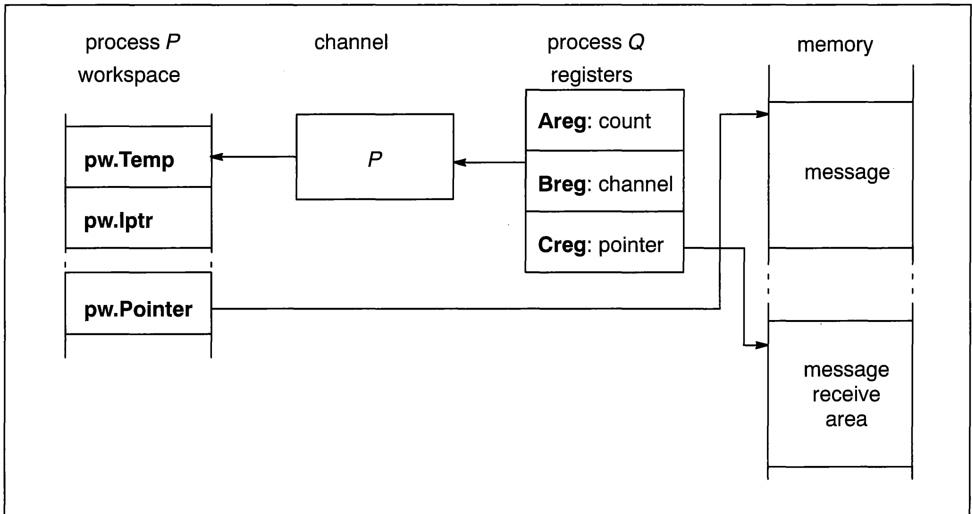


Figure 8.5 Input from waiting channel

The second process then

- (for variable-length only) writes the message length (or *LengthError.p*) into **pw.Length** of the current process workspace
- resets the value of the channel word to *NotProcess.p*
- (provided message length does not exceed maximum) performs a block move using its source or destination address, its length, and the destination or source address in the workspace pointed to by the channel word
- reschedules the first process

Figure 8.6 shows process *P* and the channel and memory state when process *Q* has completed its *input* instruction.

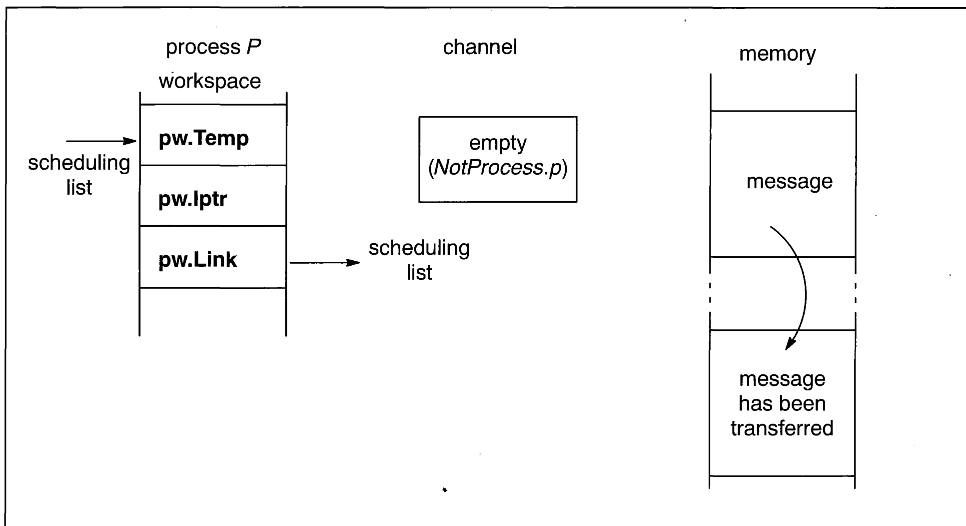


Figure 8.6 Message transfer complete

External channels

For communication between processes on different transputers, or communication between a process and a non-transputer device, external channels must be used. There are three types of external channel: virtual channels, event channels and byte-stream channels. The implementation details of these are discussed in chapter 12.

In summary, a channel may be implemented as: internal, virtual, byte-stream or event; where virtual, byte-stream and event channels are forms of external channel. Synchronization is achieved via: the simple mechanism, the alternative mechanism or the resource mechanism. Communication may be: zero-length, fixed-length or variable-length. The channel can be considered as the medium of message transfer while the type of synchronization and communication refers to the usage of that medium.

8.5 Time

On a transputer, time is cyclic. There are two clock registers, one for each priority level, $\mathbf{ClockReg}_0$ and $\mathbf{ClockReg}_1$. The high priority clock $\mathbf{ClockReg}_0$ increments every 1µs. The low priority clock $\mathbf{ClockReg}_1$ increments every 64µs. Whenever **ClockReg** = *MostPos*, it is 'incremented' to *MostNeg*.

mnemonic	name
ldtimer	load timer
tin	timer input

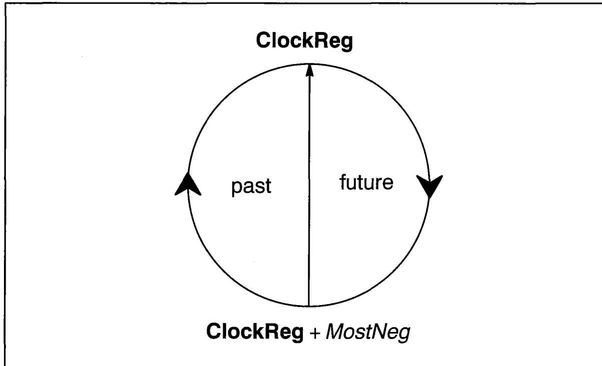
Table 8.11 Instructions which use the on-chip clocks

8.5.1 Past and future

For each priority level, 'future' and 'past' are defined as follows.

$$(\mathbf{ClockReg}\ \mathit{PLUS}\ \mathit{MostNeg}) \leq \text{past} < \mathbf{ClockReg}$$

$$\mathbf{ClockReg} \leq \text{future} \leq (\mathbf{ClockReg}\ \mathit{PLUS}\ \mathit{MostPos})$$



That is, all times which are between (**ClockReg** *PLUS MostNeg)* and **ClockReg** are considered to be in the past, and those which are between *(* **ClockReg** *PLUS 1)* and *(* **ClockReg** *PLUS MostPos)* are considered to be in the future.

The AFTER relation

Care is needed when operating on cyclic quantities such as time. The usual 'greater than' relation is replaced by the relation AFTER which is defined by

(x AFTER *y)* ≡ *((x MINUS y) > 0)*

and can be translated into

x; y; diff; ldc 0; gt

The usual transitive property does not hold for the after relation, that is:

(x AFTER *y)* ∧ *(y* AFTER *z)* does *not* imply *(x* AFTER *z)*

A consequence of this property of cyclic time is that a group of times are only unambiguous if they are all contained within a half cycle of timer ticks – i.e. within a range of size (*MostPos* – *MostNeg*) ÷ 2.

8.5.2 Reading the clock

The current value of the processor clock can be read by executing a 'load timer' instruction *ldtimer*. This reads the value of the high priority clock when executed in a high priority process and the low priority clock when executed in a low priority process, and pushes this onto the integer stack.

8.5.3 Timer input

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, *tin*, requires a time to be supplied in **Areg**. This time is referred to as the 'alarm-time' and it specifies that the process should not be in the active set up to and including that time. If this time is in the 'past' — i.e. **ClockReg** AFTER **Areg** — then the instruction has no effect. If the time is the current time or it is in the 'future' — i.e. **Areg** = **ClockReg** or **Areg** AFTER **ClockReg** — then the process is descheduled. The process is rescheduled when the specified time is reached (i.e. the value in the appropriate clock register is AFTER the alarm-time). The process will not necessarily start to execute immediately it is scheduled, as other processes may already be waiting on the scheduling list. Consequently when the process starts to execute, the value in the clock may be some time after the time specified in the timer input.

tin is therefore a descheduling point, and is privileged and interruptible. The integer and floating-point stacks are left undefined by this instruction, even if the time specified is in the past.

For example the following code sequence executed in a low priority process would cause the process to be descheduled for (at least) one second by waiting for *1000000* $\div$ *64* (= *15625*) ticks of the clock.

ldtimer; ldc 15625; sum; tin

Note that when dealing with time the unsigned modulo arithmetic operations *sum*, *diff*, must be used rather than *add* and *sub* which would cause an arithmetic overflow when the value representing the time wrapped round from *MostPos* to *MostNeg*.

8.5.4 Timer lists

The following provides some background information on the mechanism of the timer lists which are used to ensure that processes are rescheduled at the correct time, having been descheduled by a *tin* or *taltwt* (see section 8.7) instruction.

Each priority level has a timer list. Each timer list contains information about processes that are waiting, and is represented as a linked list of process workspace data structures. The address of the workspace of the head of each timer list is stored in one of the timer list pointer registers: **TptrReg0** for high priority, and **TptrReg1** for low priority. The linked list is implemented by storing the address of the next workspace in the **pw.TLink** slot of a process workspace. The alarm-time for each process is stored in the in **pw.Time** slot. This is the time at which the process will become ready — so if a process executes

ldc X; tin

the time *(X+1)* will be entered into **pw.Time** slot. The end of a timer list is signified by a link address of *NotProcess.p*.

Each list is ordered so that each process in the list is waiting for a time no earlier than that of the process before it and no later than that of the process after it. For each list, the time of the first process — the next time that is required — is stored in a alarm register: **TnextReg0** for high-priority, and **TnextReg1** for low-priority.

Timer input

When a timer input is performed — either because of a *tin* instruction or by a timer guard in an alternative sequence (see section 8.7) — then the process is inserted into the relevant timer list. This involves searching down the list until the time requested lies between the times of two adjacent entries so that when the process is inserted there, the ordering of the list is maintained. This means that instructions that manipulate the timer lists take (on average) time proportional to the length of the timer lists. Because of this, these instructions are interruptible.

Manipulating the timer lists

The instruction *swaptimer* has been provided for manipulation of the timer lists. It is described in section 13.6.

8.6 Semaphores

Semaphores were introduced in 1965 by E.W. Dijkstra as a means of controlling the execution of concurrent processes. An *n*-valued semaphore ensures that from the set of processes that are *waiting* on that semaphore, at most *n* can run concurrently. If a process has a critical piece of code which must only execute when the semaphore allows it to, then the code *waits* on the semaphore before commencing, and *signals* to the semaphore when it has finished.

The IMS T9000 transputer provides an efficient implementation of an *n*-valued semaphore for processes on the **same processor**. *signal* and *wait* instructions (table 8.13) are provided which operate on a data structure which may be located at any word aligned address in memory. A semaphore is implemented by a three word data structure. The word locations in the data structure are shown in table 8.12. The data structure must be initialized with the **s.Count** slot set to *n* for an *n*-valued semaphore and with the **s.Front** slot set to *NotProcess.p*.

word offset	slot name	purpose
2	**s.Back**	back of waiting queue
1	**s.Front**	front of waiting queue
0	**s.Count**	number of extra processes that the semaphore will allow to continue running on a *wait* request

Table 8.12 Word offsets and names for data slots in a semaphore data structure

When a process executes a *wait* instruction, it will only be able to proceed without delay provided there are currently less than *n* processes that have claimed the semaphore – i.e. the number of *wait* instructions executed on the semaphore by all processes, does not exceed by more than *n,* the number of *signal* instructions executed by all processes. If this is not the case, the process executing the current *wait* instruction is temporarily descheduled. If further processes execute *wait* instructions prior to a *signal* instruction, they will also be descheduled and all such processes are placed in a queue awaiting restart. For each *signal* instruction that is received, a queued process is restarted. In summary, a process can claim a 'run slot' from the *n* available, using *wait*, and it can free that slot for another process using *signal*.

mnemonic	name
wait	wait
signal	signal

Table 8.13 Semaphore operation instructions

Both instructions are privileged and require a semaphore data structure address in **Areg**. All integer and floating-point stack registers are left undefined after these operations.

The *wait* instruction examines the value of the counter in **s.Count**. If this is greater than zero, then it decrements this count and takes no further action. In this case the executing process is free to continue. If the counter is equal to zero, then the current process is descheduled and appended to the end of the process queue pointed to by **s.Front** and **s.Back**. Hence by implication, this instruction is a descheduling point.

The effect of *signal* is to allow one extra process to execute a *wait* without being suspended. Unless **s.Front** contains the value *NotProcess.p*, it contains a process descriptor and the process that it describes can be run. Hence this process is removed from the front of the semaphore queue and is rescheduled. If **s.Front** contains the special value *NotProcess.p*, then there are no processes waiting. The action taken in this case, is to increment the value in **s.Count**. If the count overflows, then *IntegerOverflow* is signalled.

A semaphore may be shared between processes of different priorities, but processes are removed from the semaphore in the order that they are queued, rather than in priority order.

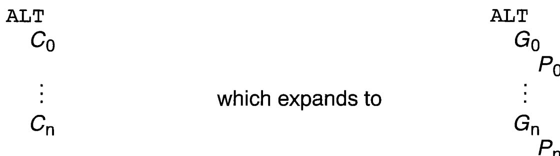
8.7 Alternative input

The ALT construct in occam allows a process to make a choice over its future behavior dependent on the readiness of other concurrent processes to communicate with it.

It can be implemented by the instruction set in one of two ways. The first method is the more direct one and uses the instruction set 'alternative sequence', described in this section. The second method is by using resource channels and this is described in section 8.8.

8.7.1 The occam ALT construct

In occam the construct which describes an alternative is called the ALT construct, the syntax of which is



Each G_i, P_i (i = 0 to *n)* pair is referred to as 'component alternative' *i* – or C_i. G_i is a 'component guard' and P_i is the 'component process' which is executed if component *i* is selected. Just one of the component alternatives is selected, namely, any component whose guard happens to be ready. If no guard is ready then the process waits until one is.

Each component may have one of the following 'guards' where *e* is a boolean expression.

skip component		*e* & SKIP *P*

channel component	*c*? *v*
	  P
		e & *c*? *v*
		  P

timer component	*timer*? AFTER *t*
	  P
		e & *timer*? AFTER *t*
		  P

The skip guard is ready whenever *e* is *true*. The channel guard is ready when both *e* is *true* and the input channel is ready to receive. The timer guard is ready when both *e* is *true* and the alarm-time has expired.

Guards which do not have a boolean conjunct to them have TRUE & implicitly added.

```
c? v              =  TRUE & c? v
  P                    P

timer? AFTER t    =  TRUE & timer? AFTER t
  P                    P
```

8.7.2 The 'alternative sequence'

The instruction set 'alternative sequence' is a sequence of instructions that can be used to select one of the component alternatives, and initiate execution of the instruction sequence associated with that alternative.

The list of instructions which may be used in an alternative sequence is shown in table 8.14. These instructions are privileged.

mnemonic	**name**
alt	alt start
altwt	alt wait
altend	alt end
talt	timer alt start
taltwt	timer alt wait
enbs	enable skip
diss	disable skip
enbc	enable channel
disc	disable channel
enbt	enable timer
dist	disable timer
enbg	enable grant
disg	disable grant

Table 8.14 Instructions required to implement an alternative sequence

The occam ALT (or similar high-level construct) can be implemented by: an *alt* (or *talt*) instruction, a sequence of enable instructions (one for each component guard), an *altwt* (or *taltwt*) instruction, a sequence

of disable instructions (one for each component guard) and an *altend* instruction. Hence the chronological order of events is:

- The *alt* (or *talt*) instruction signifies the start of the sequence.
- For each guard of the ALT, there is an enabling sequence, which uses one of the instructions: *enbs*, *enbc*, *enbt*, *enbg*.
- An *altwt* (or *taltwt*) instruction forces the current process to deschedule until one of the guards is ready.
- There is for each guard a disabling sequence, which uses one of the instructions: *diss*, *disc*, *dist*, *disg*.
- The *altend* instruction signifies the end of the alternative sequence, and the code for the selected component is run.

The order in which the alternatives are enabled is unimportant, but the order in which they are disabled determines the priority of the alternatives. The first ready alternative to be disabled is selected. If none of the component alternatives is a timer component, then the instructions *alt* and *altwt* must be used; otherwise the instructions *talt* and *taltwt* must be used instead.

Workspace pointer during selection

The workspace pointer must have the same value at each of the *alt*, *altwt*, *altend* instructions and the enable and disable instructions. It may however be changed, for example, while evaluating an argument to an enable instruction. However, the user must not change the **pw.State** slot of the process workspace data structure between execution of *alt* and *altend*, and when the alternative sequence has a timer component, the user must not change either **pw.State** or **pw.TLink** between *talt* and *altend*. The slot **pw.Temp** has a special use, and so must not be written to, and is not preserved over the disabling sequence (from the *altwt* instruction to the *altend* instruction).

Instructions

The effect of *alt*, is to put the special value *Enabling.p* into the **pw.State** slot of the process workspace. The *talt* instruction is similar but it also puts the special value *TimeNotSet.p* into the **pw.TLink** slot. This indicates that there is a timer component in the alternative sequence, which has not yet been enabled. The stack registers are not affected by these instructions.

For all four enabling instructions (*enbs*, *enbc*, *enbt*, *enbg*), the boolean expression value of the guard is passed in **Areg**. If the boolean in **Areg** is *true* then that guard is 'enabled'.

> *enbs* enables the guard by placing *Ready.p* into the **pw.State** slot of the process workspace data structure.
>
> For *enbc*, the channel address is passed in **Breg**. If the specified channel is ready to communicate, then the instruction sets the **pw.State** slot to the special value *Ready.p* to indicate that this is the case.
>
> For *enbt*, the time at which the component process should be run (if selected) is passed in **Breg**. The instruction sets the **pw.TLink** slot of the process workspace data structure to *TimeSet.p* to indicate a timer component has been enabled in the alternative sequence. It also stores the alarm-time value, passed in **Breg**, in the **pw.Time** slot of the process workspace data structure, unless there is already an earlier time value stored in that slot.
>
> *enbg* will be described in section 12.7.

enbs has no effect on any of the values in the integer stack. *enbc*, *enbt* and *enbg* leave the boolean value in **Areg**, and pop the value held in **Creg** into **Breg** leaving **Creg** undefined.

The *altwt* instruction examines the **pw.State** slot of the process workspace. If it contains the special value *Ready.p*, then at least one of the guards is ready and so the instruction takes no further action. Otherwise

the instruction sets the **pw.State** slot to the special value *Waiting.p* and deschedules the current process until one of the guards becomes ready. The integer and floating-point stacks are left undefined by this instruction, even if one of the guards is ready.

The *taltwt* instruction is similar to *altwt* but if no communication channels are ready, then it also considers the case where timer guards have been enabled. If the alarm-time recorded in the **pw.Time** slot is in the past, then the **pw.State** slot is set to *Ready.p* the current process continues. If the alarm-time is in the future, then the process is put onto the timer list and descheduled. As for *altwt*, the integer and floating-point stacks are left undefined, even if one of the guards is ready. *taltwt* is interruptible.

For the disabling instructions (*diss*, *disc*, *dist*, *disg*), **Areg** contains an offset from the instruction following the *altend* to the start of the code for that branch of the alternative, and **Breg** contains the boolean expression value of the guard.

> For *disc*, the channel address is passed in **Creg**.
>
> For *dist*, the alarm-time is passed in **Creg** (this should be the same as the time passed for the *enbt*). If the process is still on the timer list, it is removed. *dist* is interruptible.
>
> *disg* will be described is section 12.7.

The first ready component alternative to be disabled is selected. The code offset for the selected component is loaded from **Areg** into **pw.Temp**. These instructions: return a boolean in **Areg** which is *true* only if that branch of the alternative is the one to have been selected, and leave **Breg** and **Creg** undefined.

The instruction *altend* marks the end of the alternative sequence. When this is executed one of the guards has been selected by the disabling sequence. This instruction forces a jump to the code associated with the selected guard. *altend* achieves this by adding the offset for this code, held in **pw.Temp,** to the address of the next instruction, and loading the result into **IptrReg**. Note that if the selected guard is a channel, then the alternative sequence does *not* perform the data-transfer. This should be done by an input instruction in the component process.

enbc and *disc* signal *IntegerError* if the channel parameter is not a legal channel address, and signal *Unalign* if the channel address is not word-aligned.

8.7.3 Execution of the alternative sequence

Section 8.7.2 describes the code necessary to implement an alternative sequence. This section overviews the changes that are made to the process workspace data structure and to the channel words during the execution of this sequence.

The three phases of enabling, waiting and disabling are considered separately.

The value held in the **pw.State** slot of the process workspace data structure, is known as the *ALT* state of the alternative, and it has one of the following values

> *Enabling.p* = *MostNeg*+1
> *Waiting.p* = *MostNeg*+2
> *Ready.p* = *MostNeg*+3

pw.State has the same offset in the process workspace data structure, as **pw.Pointer**, but because none of the above values is a valid pointer to an input message buffer, an outputting process is able to distinguish between an unconditional input and an alternative input on the channel.

In addition for a timer component, the timer list link, held in the **pw.TLink** slot, has one of the following values

> *TimeSet.p* = *MostNeg*+1
> *TimeNotSet.p* = *MostNeg*+2

Enabling

An alternative is enabling between the execution of the *alt* or *talt* instruction and the start of the execution of the *altwt* or *taltwt* instruction.

The processor sets the *ALT* state (**pw.State** slot) to *Enabling.p* to indicate that the guards of an alternative construct are being enabled. If any guard is immediately ready — e.g. is a `SKIP` guard or a channel guard on a ready channel — then this location is set to *Ready.p* to indicate that a guard is ready.

Timer alternatives

A record of the earliest timer guard yet encountered is kept during the enabling sequence of a timer alternative. The **pw.TLink** slot contains *TimeNotSet.p* until the first timer guard is enabled and then it contains *TimeSet.p* with **pw.Time** containing the earliest time encountered.

Waiting

An alternative is waiting between the start of execution of the *altwt* or *taltwt* instruction and the start of the next instruction.

The processor initializes the **pw.Temp** slot to *NoneSelected.o* to indicate that no branch has yet been selected. If the *ALT* state (**pw.State** slot), is not *Ready.p* then the processor sets this to *Waiting.p* and deschedules the process.

Any communication to a waiting alternative causes the *ALT* state to be set to *Ready.p*. When one of the alternative guards becomes ready the process executing the alternative is rescheduled. The waiting period ends when this process comes to the front of its scheduling list and starts to execute.

Timer alternatives

If no other guards are ready, an additional check is made to see if the earliest enabled time is earlier than the current time. If so, the process is not descheduled and the *ALT* state is set to *Ready.p* as a timer guard is ready. If a guard is ready then the current time is recorded in the **pw.Time** slot to indicate when the timer finished waiting. If no guard is ready then the process is descheduled and inserted into the appropriate timer list.

Disabling

An alternative is disabling between the execution of the instruction after the *altwt* or *taltwt* instruction and the execution of the *altend* instruction.

When a guard becomes ready the disabling sequence is executed. When the first ready guard is disabled, the offset to the component process code is stored in **pw.Temp** to indicate that it has been selected.

Timer alternatives

If necessary, the current process is removed from the appropriate timer list. The current process will not be on the timer list if one of the timer guards is ready.

Communication on a guarded internal channel

For an internal channel, the action of an output instruction when outputting to an alternative input, is slightly different from its action when outputting to an unconditional input. (Note however that this is transparent to the outputting process.) The following describes the various assignments that are made to the channel.

If there is no process descriptor in the channel when it is enabled, then *enbc* leaves the descriptor of the current (inputting) process in the channel.

When an output instruction is executed on the channel, if there is no process descriptor held in the channel, then the descriptor of the outputting process is left there and the process deschedules as for simple synchronization. However, if there is a process descriptor held in the channel, then the **pw.Pointer/pw.State** slot of the process workspace data structure is examined. If the value is a valid pointer to a message buffer, then this is a simple synchronization, and the data-transfer occurs as described previously. Otherwise the value in **pw.State** is the *ALT* state, and the inputting process is operating an alternative sequence. Hence, the communication does not occur as the guard has not yet been selected.

- If the *ALT* state is *Enabling.p*, then the inputting process is enabling, but has not yet found a guard that is ready. Since there is now a channel ready to communicate, the *ALT* state is changed to *Ready.p*.
- If the *ALT* state is *Waiting.p*, then the inputting process is descheduled and waiting for a guard to become ready. Since there is now a channel ready to communicate, the *ALT* state is changed to *Ready.p* and the inputting process is rescheduled.
- If the *ALT* state is *Ready.p*, then there is another guard ready in the alternative sequence, and the inputting process is already aware of this, so the *ALT* state does not need to be modified.

In each case, the outputting process leaves its own descriptor in the channel and deschedules itself.

When the inputting process subsequently executes *disc*, the presence in the channel, of a descriptor that is not of the current (inputting) process, will indicate that this guard is ready. If the inputting process selects this guard (in the disabling sequence), then a subsequent input instruction will transfer the data and reschedule the outputting process.

8.7.4 Compiling an ALT statement

Section 8.7.1 introduces the occam ALT construct. To translate this, each component alternative requires an enabling sequence, *enable(C_i)*, a disabling sequence *disable(C_i,offset)*, and a process sequence, *process(C_i)*. These sequences are defined later. The ALT statement presented in subsection 8.7.1, can then be translated as follows.

```
        alt;
        enable( C0 ); enable( C1 ); . . . ; enable( Cn );
        altwt;
        disable( C0, P0–A ); disable( C1, P1–A ); . . . ; disable( Cn, Pn–A );
        altend
A:      ...
P0:     process( C0 ); j END;
P1:     process( C1 ); j END;
        . . .
Pn:     process( Cn ); j END;
END:
```

Note that *talt* and *taltwt* would need to be used if any of the guards G_i is a timer guard.

Enabling and disabling component alternatives

Component alternatives are enabled and disabled by the following sequences of instructions

component – *C*			enable/disable sequence
e & SKIP	: *enable(C)*	=	*e; enbs*
P	: *disable(C, L)*	=	*e; L; diss*
e & *c? v*	: *enable(C)*	=	*c, e, enbc*
P	: *disable(C, L)*	=	*c; e; L; disc*
e & *timer*? AFTER *t*	: *enable(C)*	=	*t, e, enbt*
P	: *disable(C, L)*	=	*t; e; L; dist*

where *L* is the offset from the instruction which follows *altend* to the start of the instruction sequence corresponding to process *P*.

Component alternative process sequences

For the component alternative which has a channel communication guard, the process *P* should be preceded by the input instruction. The following shows this.

component – *C*			process sequence
e & SKIP *P*	: *process(C)*	=	*P;*
e & *c*? *v* *P*	: *process(C)*	=	*c*? *v; P;*
e & *timer*? AFTER *t* *P*	: *process(C)*	=	*P;*

where *c*? *v* is translated by fixed or variable input as described in section 8.4.3.

8.7.5 Trapping degenerate alternatives

It is possible for all the guards of an alternative to fail due to all the boolean components being *false*. In some circumstances this might need to be reported as an error, because the alternative can never proceed. Each enable instruction terminates with the value of its boolean expression in **Areg**. This can be used during the enabling sequence to detect whether the boolean expressions in all the alternatives are *false*. For example

alt;
enable(C_0)*; stl F;*
enable(C_1)*; ldl F; or; stl F;*
⋮
enable(C_n)*; ldl F; or; ldc 1; ccnt1;*
altwt

Thus the *ccnt1* instruction will signal *IntegerError* if the disjunct of all the boolean components is *false*. This uses the temporary local variable *F* to evaluate the disjunct. This is so that its value is preserved in the event of the process descheduling, or integer stack pushing. If it is known that none of the enabling sequences can cause the process to be descheduled, and the evaluation of the two operands to the enable requires no more than two registers, then the following sequence could be used .

alt;
enable(C_0)*;*
enable(C_1)*; or;*
⋮
enable(C_n)*; or; ldc 1; ccnt1;*
altwt

8.7.6 Replicated ALT

An ALT construct in occam can use a replication sequence.

ALT *i* = *b* FOR *c*
  C_i
	which expands to	ALT *i* = *b* FOR *c*
		  G_i
		    P(i)

The enabling sequence for this involves using a loop round the enable guard instructions. This is achieved using *lend* as described in section 7.10.

The disabling sequence for a replicated ALT is more complex as the value of the control variable *i* for the branch selected must be passed into the execution of *P*. Each disable instruction terminates with **Areg**

holding *true* if the alternative is selected, and *false* otherwise. This allows the disabling sequence for a replicated alternative to record the selected value of the control variable. The disabling sequence for C_i, is

disable(C_i); cj M; ldl i; stl selected_i; M:

where the selected process *P(i)* will use *selected_i* as its constant *i*.

8.7.7 PRI ALT

In occam, there is also a PRI ALT construct.

```
PRI ALT
  G0
    P0
  ⋮
  Gn
    Pn
```

This is similar to an ALT construct, but gives priority to its components in the order that they are listed. Hence if two guards become ready at the same time, then the first component listed (of the two whose guards are ready) in the PRI ALT construct is selected. For an ordinary ALT construct, the selected component is indeterminate when more than one guard is ready (although this may be determinate for particular compiler implementations).

This is easy to implement with the alternative sequence. Each component should be disabled in the same order as they are listed in the occam construct. The first guard found to be ready in the disabling sequence, is the one whose component is selected. This provides the required priority ordering.

8.8 Resource channels

The synchronization mechanism provided by resource channels has been introduced in section 8.4.2 as a means of allowing many-to-one communication. This section presents a particular programming problem for which resource channels can be used to solve, further explains the mechanism, and gives the required data structures.

8.8.1 The client-server model

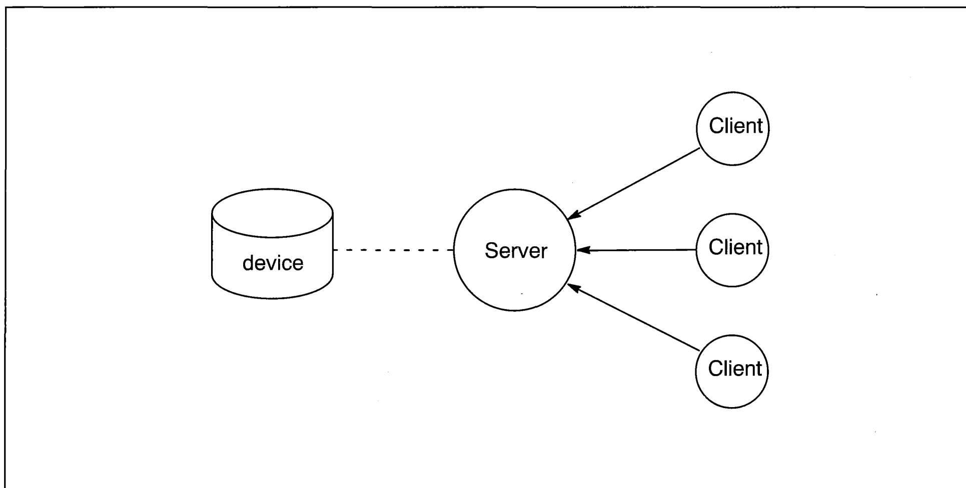
The problem

In concurrent programming, it is often the case that several processes require a communication path to the same process. It may be for example, that there is a central resource such as a printer which is being controlled by a process, and there are several other processes which are potential users of this resource. This is referred to as a client-server model, where the user processes are the clients, and the process controlling the resource is the server. At a high-level this is most easily demonstrated in occam.

```
PROC Server( []CHAN OF ANY in )
  SEQ
    ...
    WHILE Serving
      ... declarations
      ALT i = 0 FOR SIZE in
        in[i]? Message
          Serve( Message, i )
    ...
:

PROC Client( CHAN OF ANY out )
  ... declarations
  SEQ
    ...
    out! Message
    ...
:

[n]CHAN OF ANY client.to.server:
PAR
  ... declarations
  Server( client.to.server )
  PAR i = 0 FOR n
    Client( client.to.server[i] )
```



In this example, a single `Server` process is running concurrently with *n* `Client` processes. Each client is connected to the server via an element of the array of channels, `client.to.server`, and the only action by the client in which we are interested, is an output on that channel which sends a message to the server. The message may be anything from a single byte token, to a large message with complex protocols. The server executes a continuous loop, and in each branch of the loop, it waits for a message to be received from one of its clients. When it receives a message, it executes a procedure (`Serve`) dedicated to handling that message on behalf of the client.

In practice there may be many servers, each of which has a set of clients, where these sets may not be mutually exclusive.

The solution

One way to implement this model in the transputer instruction set is with an 'alternative sequence' (described elsewhere – see section 8.7). The disadvantage of this is that for a large *n*, each loop of the alternative may take a long time to run, because every channel must be explicitly enabled and disabled.

A better approach (for large *n*) is to implement every channel as a resource channel; and to associate each of these with the server process. The following describes how this can be achieved with the IMS T9000 instruction set, and the next section gives an overview of the data structures and state assignments. Further implementation details and full descriptions of the associated instructions are given in section 12.7.

The coding of an outputting process is independent of the synchronization mechanism. The only requirement is that an extra two words are reserved for each channel that is to be used as a resource channel. The client process therefore tries to communicate with a server using a normal output instruction (*out*, *outbyte*, *outword*, *vout*). There is nothing then to prevent the channel from being used for either normal channel communication or resource channel communication. Such a channel is thus considered to have two modes of operation: 'normal mode' and 'resource mode'. It is in fact possible to change the mode of a channel after the client has executed the *out* instruction (while the client process is descheduled).

The server process, which has several clients inputting information via channels, can set any number of these to resource mode, using the instruction *mkrc*. When the server is ready to receive an input from one of these channels, it executes a *grant* instruction. If none of the resource channels are ready to communicate then the server process is descheduled until one of the clients does try to communicate. If one or more have already tried to output, then one of them is selected. In either case, when the server is ready to proceed, it should then execute an input instruction (*in*, *vin*) on the selected channel. The *grant* instruction also sets the channel back to normal mode, so it must be explicitly set back to resource mode before attempting another resource communication.

Note that although the resource mechanism has been proposed as an alternative for implementing an ALT construct, it is not as general as the 'alternative sequence'. In particular: (i) it cannot easily implement boolean guards (ii) the `Serve` procedure (in the above example) cannot use the other components of channel array `client.to.server`.

In this example the server knows all it needs to know about its clients. This is the 'omniscient server' that is described in section 12.7.5, which also presents some other uses of resource channels.

8.8.2 Resource mechanism and data structures

A resource is represented in memory by a 'resource data structure' (RDS). The format of the RDS is shown in table 8.15, which also shows how this data structure should be initialized. The RDS is resident on the memory of the transputer that hosts the server process. When a client process becomes ready to output, it executes an output instruction specifying the resource channel address.

claim

If the channel specified by an output instruction is in resource mode, then the processor makes a 'claim' to the resource on behalf of the client. That is:–

- If there is a waiting server, then it is granted an immediate communication.
- Otherwise it is attached to the end of a queue of waiting clients (or the front of that queue if there are no others waiting). The first and last channels on this queue are pointed to by the **rds.Front** and **rds.Back** slots of the RDS respectively.

When the server process becomes ready to communicate, it executes a *grant* instruction (for full details of this instruction, see section 12.7.4), supplying a pointer to the RDS. The first channel on the queue is removed and that channel is made ready to communicate. If there is no client process ready when a server executes a *grant*, then the server's descriptor is left in the **rds.Proc** slot of the RDS and the server is descheduled until the next client output.

word offset	slot name	purpose	initial value
2	**rds.Back**	pointer to back of resource channel queue	any
1	**rds.Front**	pointer to front of resource channel queue	*NotProcess.p*
0	**rds.Proc**	process descriptor of server	*NotProcess.p*

Table 8.15 Resource data structure (RDS)

To implement a channel that can be set to resource mode, it is necessary to allocate a resource channel data structure. This is shown in table 8.16, which also shows how this data structure should be initialized.

The **rc.Id** slot is used to indicate the mode of the channel. If it has the special value *NotProcess.p*, then the channel is in normal mode, but if it has any other value then it is in resource mode. Because a channel is in normal mode when its memory space is first allocated, the **rc.Id** slot should always be initialized to *NotProcess.p*. The instruction *mkrc* (see section 12.7.4) will set this to a unique 'resource channel identifier' when setting the channel to resource mode. *mkrc* must never assign the value *NotProcess.p* to the resource channel identifier.

When the channel is in resource mode, the slots are used as follows. Prior to making an output on a resource channel, the **rc.Ptr** slot points to the RDS. Where a channel is linked into a resource channel queue on an RDS as described above, **rc.Ptr** points to the next channel in the queue. The 'resource channel identifier', which is held in the **rc.Id** slot, is written to the location specified as a parameter to the *grant* instruction, when the latter selects this channel. It can then be used by the server process, to determine the channel address for the subsequent input instruction.

word offset	slot name	purpose	initial value
1	**rc.Id**	resource channel identifier / mode indicator	*NotProcess.p*
0	**rc.Ptr**	pointer to RDS or next resource channel	any

Table 8.16 Resource channel data structure

Very often the identifier stored in the resource channel data structure will be the channel address itself. If this is returned by the grant instruction, it is easy for the subsequent input instruction, to input from the correct channel. In some circumstances however, it may be more convenient for this to be an integer that represents an offset in a channel array. Such an index can have a dual use. Firstly it may be used to calculate the actual channel address from a base address, or as an index into a table of channel addresses. Secondly it may be used as a parameter to the subsequent procedure, perhaps to enable the procedure to identify the requesting client. An example of this is the parameter *i* in procedure `Serve` above.

The positioning of the resource channel data structure, the implementation of channels in resource mode, and the instructions applicable to the use of resource channels, are fully described in chapter 12.

9 Protection and memory management

The protection and memory management mechanism on the IMS T9000 provides:

- instruction protection – preventing interference between executing processes
- memory protection – allowing regions of memory to be write and/or execute protected
- memory management – provides mapping from logical to physical address space

These features are available when running code 'under protection' (we sometimes say running in 'protected mode', or running a P-process). They are designed to support the development and debugging of programs, to allow the safe execution of insecure languages, and to support address translation. The mechanism does not provide virtual memory, or page-based memory protection, but does provide for stack extension when the stack overflows the region allocated for it – hence allowing dynamic allocation of a calling stack.

N.B. A P-process should not be considered as 'protected'; it is the supervisor and the other transputer processes that are protected *from* the P-process. Furthermore, a P-process should not be considered as a distinct process, but as a mode of a transputer process (where the L-process is the other mode).

9.1 The mechanism

Any L-process can cause the IMS T9000 to run code under protection, acting as the 'supervisor' of that code. A supervisor is responsible for initializing and maintaining two data structures. The first of these is the PDS (P-state data structure) which is primarily used for storing the state of the P-process. The second is the region descriptor data structure which contains information controlling memory protection and memory mapping. (More detail on these data structures is given in section 9.7.) Provided that these data structures are initialized correctly, execution by the supervisor of the instruction *goprot*, puts the machine into protected mode. The priority of the process executing under protection is unchanged from that of the supervisor.

The process then continues to run under protection (as a P-process) until it takes a trap. A number of different conditions can cause a 'P-process trap'. These are:–

- the occurrence of an error, which includes accessing an illegal address (*AccessViolation*) or executing a privileged instruction (*PrivInstruction*),
- the expiry of a timeslice period (see section 8.2.4),
- the execution of the *syscall* instruction (system call), or
- a debugging event (breakpoint, watchpoint, single-step, *causeerror*).

(More detail on trap causes is provided in section 10.3.) When a P-process traps, it restarts its supervisor at the instruction following *goprot*. The supervisor then has access (via the PDS) to the state of the process prior to the trap. If the code running under protection causes an error, the supervisor is able to handle the error much as a trap-handler is able to handle an error caused by an L-process. (More detail on state storage and the trap mechanism is given in section 10.2 and section 13.2.)

A trapped P-process can be *re*started by the supervisor, using *goprot*.

9.2 Instruction protection – privileged instructions

Code run under protection is prevented from interfering with the execution of other processes in the transputer. This is achieved because only a subset of the transputer instruction set may be executed under protection. The set of instructions that may **not** be executed under protection are known as 'privileged instructions'. These include all scheduling and communication instructions and all instructions that configure the transputer.

If a P-process attempts to execute a privileged instruction, then it signals *PrivInstruction* and traps to its supervisor. Note that in this case, the supervisor has enough information to execute the instruction on behalf of the P-process if required.

The complete list of privileged instructions is:–

alt	*altend*	*altwt*	*chantype*	*disc*
disg	*diss*	*dist*	*enbc*	*enbg*
enbs	*enbt*	*endp*	*erdsq*	*fdcl*
goprot	*grant*	*icl*	*in*	*initvlcb*
insertqueue	*insphdr*	*intdis*	*intenb*	*irdsq*
ldchstatus	*ldcnt*	*ldconf*	*ldmemstartval*	*ldresptr*
ldshadow	*ldth*	*mkrc*	*out*	*outbyte*
outword	*readbfr*	*readhdr*	*resetch*	*restart*
runp	*selth*	*setchmode*	*sethdr*	*settimeslice*
signal	*startp*	*stconf*	*stopch*	*stopp*
stresptr	*stshadow*	*sttimer*	*swapbfr*	*swapqueue*
swaptimer	*talt*	*taltwt*	*testpranal*	*tin*
tret	*unmkrc*	*vin*	*vout*	*wait*
writehdr				

9.3 Address translation, memory protection, and stack extension

When code is running under protection all addresses are treated as logical addresses and are translated into physical addresses before access is made to memory. Each such address should be within one of four logical address regions, the range of which is specified by four words called 'region descriptors' which are contained in special-purpose registers called the region descriptor registers. If an instruction specifies an address that is not in one of these logical regions, then it signals *AccessViolation*. Each descriptor specifies the size of its associated region and the mapping that is used by the hardware to translate logical addresses to a physical addresses. The region of memory which is thus mapped to by the descriptor is referred to as a physical address region.

A region descriptor also contains protection information, stating whether the locations within that region are read-only or read-write, whether or not instructions may be fetched from that region, and whether or not device access instructions must be used to access these locations. Each address access (whether read, write or instruction fetch) is therefore checked by the hardware to ensure that there is not a protection violation. If an error is detected, then *AccessViolation* is signalled and a P-process trap is taken.

In addition to checking the validity of memory accesses, whenever a non-privileged instruction adjusts the workspace pointer (**Wptr**), the hardware checks that the new value in **Wptr** is the address of a *writable* location (if this were not the case then the stack would be read-only) and is not a location reserved for a device. If in protected mode, a *call*, *ajw* or *gajw* instruction causes the workspace pointer to address an invalid location then it traps to the supervisor. When a trap is taken as a result of such a workspace adjustment, the supervisor can restart execution of the error causing instruction in protected mode after extending the region. In this way it is possible to execute stack extension on demand. This is the only case where it is possible to restart an instruction which has trapped due to a non-floating-point error. (See section 13.2.2 for further details.)

The IMS T9000 allows independent relocation of each region. A region may be of size 2^n bytes, with a minimum size of 256 bytes (64 words) and a maximum size of 2^{30} bytes. A region of size 2^n bytes may be translated onto any 2^n byte boundary in the physical address space. It is the programmer's responsibility to ensure that the physical address regions do not overlap.

9.4 Regions

In protected mode, the logical address space is divided into four address quarters. Associated with each quarter is a logical address region, which is specified by a region descriptor. Each region is sized, positioned, assigned access permissions; and addresses within each region are independently translated. The two most significant bits of a logical address determine which region is being referenced. The terms *region 0*, *region 1*, *region 2*, and *region 3* are used to refer to the regions having addresses with the most significant bits set to 00, 01, 10 and 11 respectively. It is not possible to access addresses outside these regions.

The region associated with each quarter of the address space occupies either the top 2^n addresses or the bottom 2^n addresses within that quarter. The size and position of the region is fully specified by the region descriptor. Table 9.1 and figure 9.1 show the range of accessible (legal) addresses for each region.

region	size	positioned from top of address quarter		positioned from bottom of address quarter	
		most positive address	most negative address	most positive address	most negative address
0	2^l	$2^{30}-1$	$2^{30}-2^l$	2^l-1	0
1	2^k	$2^{31}-1$	$2^{31}-2^k$	$2^{30}+2^k-1$	2^{30}
2	2^n	$-2^{30}-1$	$-2^{30}-2^n$	$-2^{31}+2^n-1$	2^{31}
3	2^m	-1	-2^m	$-2^{30}+2^m-1$	-2^{30}

Table 9.1 Region addresses

A consequence of this is that, except for when the maximal sized region (2^{30} bytes) is in use, it is possible to ensure that the addresses 0 and #80000000, which are commonly used as null pointers, do not correspond to legal addresses and so access to such an address is immediately detected as a violation.

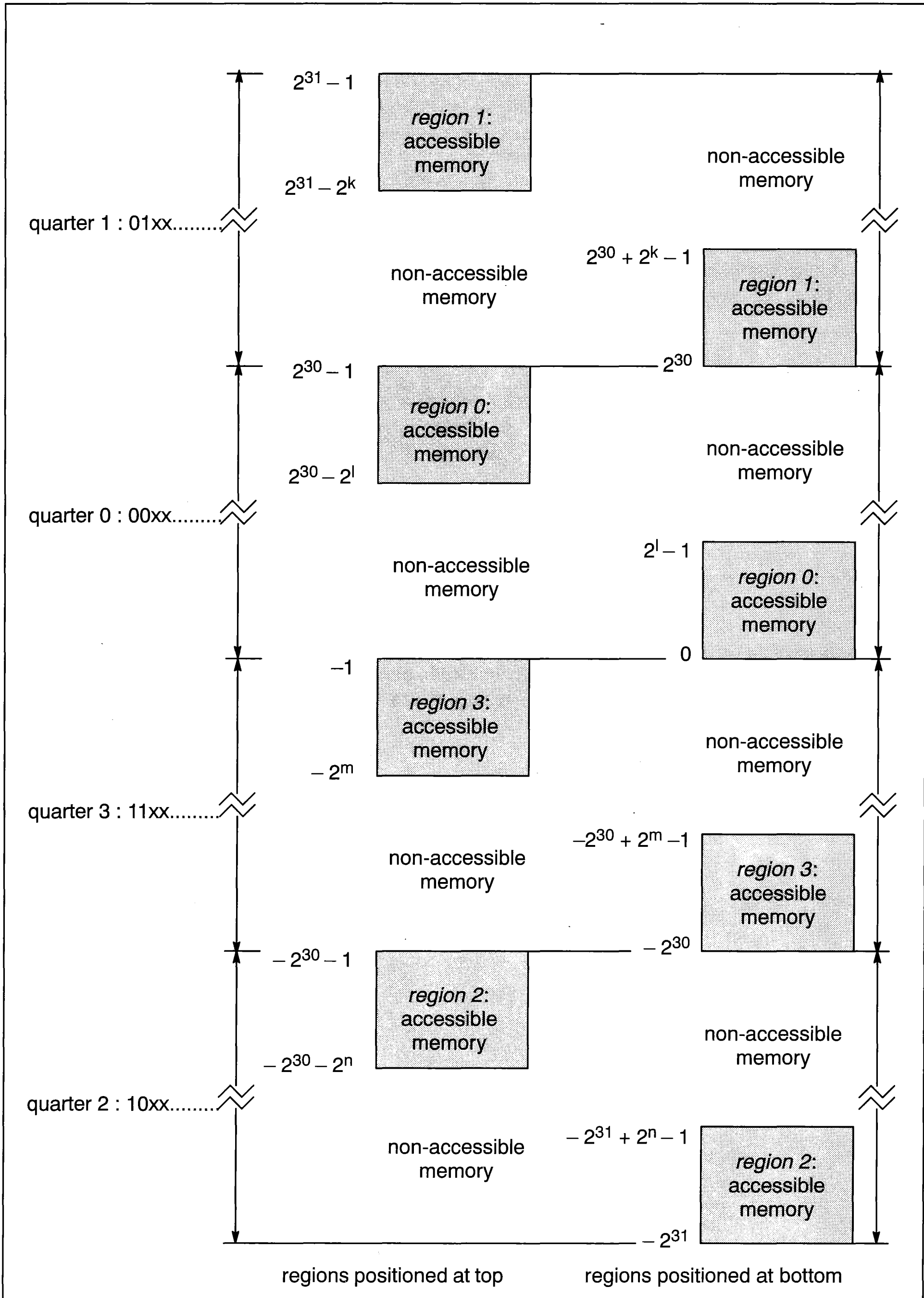


Figure 9.1 Position of region addresses in logical memory space

9.5 Region descriptors

A region descriptor defines the size of a region, the position of the logical region, the address translation, and the write, execute and device permissions associated with that region.

A region descriptor is a single word which specifies a region of size 2^n, where the minimum allowed value of n is 8. It contains the following fields.

- bit 0 indicates whether writes may be made to the region (1 = write-permit)
- bit 1 indicates whether instructions may be fetched from the region (1 = execute-permit)
- bit 2 indicates the position of the logical region (1 = top, 0 = bottom)
- bit 3 indicates whether non-device instructions are allowed to access locations within the region (1 = device-access-only)
- bits 4 to n–2 **must** be set to 0
- bit n–1 specifies the size of the region – it is set to 1 to indicate that the region size is 2^n
- bits n to 31 specifies the address of the physical region to which the logical region should be relocated – these bits replace the corresponding bits in the logical address which is being translated

A region may be disabled by setting the region descriptor to the special value #8000000, the 'null descriptor'. This sets the region to zero size. That is, there are no accessible addresses in the quarter of the address space associated with this descriptor. The region is 'disabled'.

A descriptor with a form different to that described above is invalid and must not be used. Also the behavior is undefined if the two physical address regions overlap. No two distinct logical addresses may translate to the same physical address.

The above mechanism is illustrated in the following diagram.

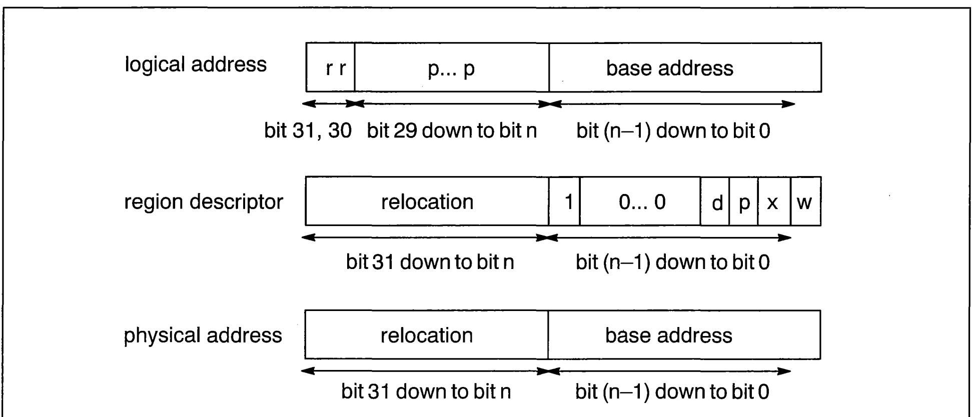


Figure 9.2 Logical to physical address translation

The next diagram gives a specific example where the logical address #BFFFF1C5 is translated to the physical address #002831C5. Region 2 is being used here, its size is 4096 (2^{12}), it is positioned at the top of the address quarter, and is read-only executable, and is not memory mapped to a device area.

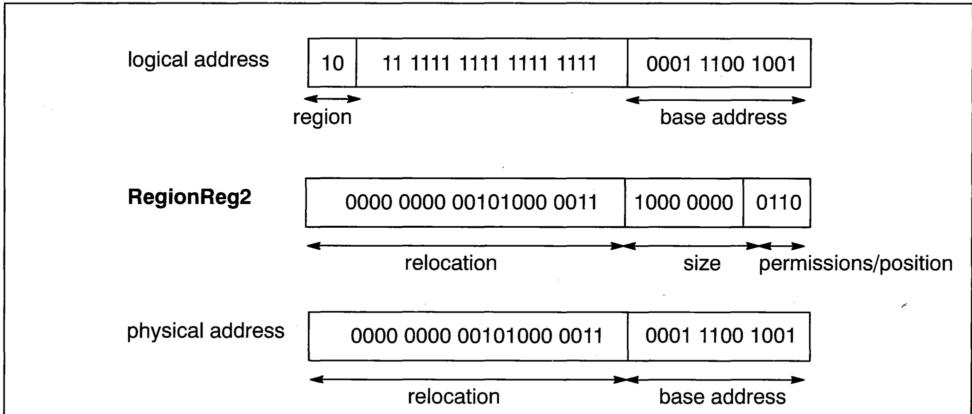


Figure 9.3 An example of logical to physical address translation

Note that for the logical address to be valid, bits n to 29 must be 1s if the position bit (bit 2) in the region descriptor is set to 1, and must be 0s if the position bit is 0. Otherwise *AccessViolation* is signalled.

9.6 Registers

The IMS T9000 has several registers which are used only by the protection mechanism. The following text describes the contents of these registers whilst the code is running under protection.

register	name / description
RegionReg0	region descriptor register 0
RegionReg1	region descriptor register 1
RegionReg2	region descriptor register 2
RegionReg3	region descriptor register 3
PstateReg	protected state register – pointer to the P-state data structure
WdescStubReg	process descriptor of the supervisor

Table 9.2 Registers used by protection mechanism

The **RegionReg*X*** register contains the region descriptor for region *X*. This information is loaded from the region descriptor data structure on execution of *goprot*. As described above, the region descriptor defines the size, position, physical address and permissions of a region as a single word.

The **PstateReg** register contains a pointer to the P-state data structure (detailed in section 9.7).

The **WdescStubReg** register contains the process descriptor of the supervisor while the code is running under protection.

9.7 Data structures

This section describes the two data structures that need to be initialized prior to starting a P-process for the first time with *goprot*:

- The P-state data structure
- The region descriptor data structure

9.7.1 P-state data structure (PDS)

The data structure referred to as the P-state data structure (or PDS) is shown in table 9.3.

word offset	slot name	purpose
10	**ps.sXreg**	internal state – loaded into / stored from **Xreg** when protected mode entered/exited in the middle of executing an interruptible instruction
9	**ps.sEreg**	internal state – loaded into / stored from **Ereg** when protected mode entered/exited in the middle of executing an interruptible instruction
8	**ps.sCreg**	P-process C register – loaded into / stored from integer stack C register when protected mode entered/exited
7	**ps.sBreg**	P-process B register – loaded into / stored from integer stack B register when protected mode entered/exited
6	**ps.sAreg**	P-process A register – loaded into / stored from integer stack A register when protected mode entered/exited
5	**ps.sIptr**	P-process instruction pointer – loaded into / stored from instruction pointer register when protected mode entered/exited
4	**ps.sWptr**	P-process workspace pointer – loaded into / stored from **Wptr** when protected mode entered/exited
3	**ps.eWu**	upper bound of P-process watchpoint region – may be loaded into upper watchpoint register when protected mode entered
2	**ps.eWl**	lower bound of P-process watchpoint region – may be loaded into lower watchpoint register when protected mode entered
1	**ps.Eptr**	pointer to instruction causing trap – loaded into / stored from error pointer register when protected mode entered/exited
0	**ps.Cntl**	control word

Table 9.3 P-state data structure (or PDS)

This section describes very generally the source/destination of data loaded into / copied from this data structure. There are several anomalies, and for a thorough understanding of state storage and retrieval, refer to section 13.2.2. For discussion of the trapping mechanism, also refer to chapter 10.

The control word: ps.Cntl

When a process starts to execute under protection, the processor loads the process status and control bits from the P-state control word (the **ps.Cntl** slot of the PDS) into the status register (**StatusReg**). The control word thus determines the conditions under which the P-process traps or sets flags (see section 10.3). When the process takes a P-process trap, these status and control bits are written back into the control word in the PDS.

[The order of bits within the control word is the same as the status and control bits in the status register, as set out in table 5.5.]

P-process state: ps.sWptr, ps.sIptr, ps.sAreg, ps.sBreg, ps.sCreg, ps.sEreg, ps.sXreg

The state of the P-process is loaded from the PDS when *goprot* is executed. The processor loads the contents of the **ps.sWptr** and **ps.sIptr** slots into **Wptr** and **IptrReg** respectively, and loads the contents of **ps.sAreg**, **ps.sBreg** and **ps.sCreg** into the integer stack. When a P-process takes a trap back to its su-

pervisor, the processor saves the contents (at the time of the trap) of all these registers, in the appropriate slots of the PDS. When a supervisor starts a P-process for the first time, it is usually the case that only **ps.sWptr** and **ps.sIptr** contain useful information (see below: *Creation of a PDS*). Note also if a trap is taken in the middle of an interruptible instruction, that the internal registers are saved in **ps.sEreg** and **ps.sXreg**, and are reloaded from those slots if the trapped P-process is restarted.

Trap causing instruction: ps.Eptr

When a P-process takes a trap, the address of the trap causing instruction (which is held in **EptrReg**), is written into the **ps.Eptr** slot. [This is different to the content of the **ps.sIptr** slot (see above), into which is stored the address of the instruction which was due to be executed *next*.] This error pointer is not loaded back into **EptrReg** unless *goprot* is being used to restart an interrupted process (see section 13.4).

Watchpoints: ps.eWl and ps.eWu

If watchpoints are enabled in the P-state control word (**sb.WtchPntEnbl** is set to 1), then when a process starts to execute under protection, the contents of the **ps.eWl** and **ps.eWu** slots are loaded into the watchpoint registers (**WlReg** and **WuReg** respectively). These addresses specify the lower bound and upper bound of the watchpoint region. While enabled, any instruction writing to a location in this region causes the process to take a trap. Note, that since the processor is in protected mode, the watchpoint region specifies logical addresses.

More details on watchpointing are provided in chapter 14.

The supervisor workspace

While the process is executing under protection, the process descriptor of the supervisor is held in **WdescStubReg** (see section 9.6). The processor reloads this descriptor into the workspace descriptor register when the P-process takes a trap.

The supervisor's instruction pointer

While the process is executing under protection, the instruction pointer of the supervisor is stored in the **pw.Iptr** slot of the supervisor's workspace data structure. The processor reloads the instruction pointer register (**IptrReg**) with the content of **pw.Iptr** when the P-process takes a trap.

Creation of a PDS

Any suitably sized word-aligned block of store may be used as a P-state data structure.

When *goprot* is used to start a P-process for the *first* time, it is always necessary to ensure that the **ps.Cntl**, **ps.sWptr** and **ps.sIptr** slots are initialized correctly with:–

- **ps.Cntl** specifying the conditions under which a trap should be taken,
- **ps.sWptr** pointing to the workspace of the P-process, and
- **ps.sIptr** pointing to the next instruction which will be executed under protection.

N.B. When the machine switches to protected mode, the values that have been loaded from **ps.sWptr** and **ps.sIptr** are treated as logical addresses.

9.7.2 Region descriptor data structure (RDDS)

The data structure referred to as the region descriptor data structure is shown in table 9.4.

word offset	slot name	purpose
3	**pc.RegionReg3**	loaded into region descriptor register 3 when protected mode entered
2	**pc.RegionReg2**	loaded into region descriptor register 2 when protected mode entered
1	**pc.RegionReg1**	loaded into region descriptor register 1 when protected mode entered
0	**pc.RegionReg0**	loaded into region descriptor register 0 when protected mode entered

Table 9.4 Region descriptor data structure

Any suitably sized word-aligned block of store may be used as a region descriptor data structure. It is necessary to initialize the data structure with region descriptor information for each quarter of the address space. Its contents are then loaded into the region descriptor registers as indicated in the table.

9.8 Instructions

The following instructions are used to switch to and from protected mode. (Execution of *syscall* is just one of the causes of a trap to the supervisor. The others are listed in section 10.3.1.)

mnemonic	name
goprot	go protected
syscall	system call

Table 9.5 Instructions used for switching to and from protected mode

goprot

goprot starts execution of code in protected mode. It is a privileged instruction. Before it is executed, **Areg** contains a word-aligned address that points to a PDS (P-state data structure). **Breg** contains a word-aligned address that points to a region descriptor data structure. *goprot* can thus be used to

- start a P-process for the first time
- restart a P-process that has been trapped
- restart a P-process that has been interrupted†

The following paragraphs describe how *goprot* saves and loads state.

The current state of the L-process is saved. If the process has a trap-handler, then the trap, watchpoint, timeslice and single-step enable bits, and all flags in the status register (**StatusReg**) are written into the **th.Cntl** slot of the THDS (see section 10.1.1). The content of the instruction pointer register (**IptrReg**) is copied into the **pw.Iptr** slot of the local workspace. The content of the workspace descriptor register is copied into the workspace descriptor stub register (**WdescStubReg**). Note that the content of the trap-handler register (**ThReg**) does not need to be saved because this register is not affected when the machine is running under protection.

The processor then loads the state of the P-process. The address of the PDS (P-state data structure) contained in **Areg** is loaded into the protected state register (**PstateReg**). The four words stored in the region descriptor data structure (pointed to by **Breg** – table 9.4) are loaded into the corresponding region descriptor registers. The contents of the **ps.sWptr** and **ps.sIptr** slots of the PDS are copied into **Wptr**

† N.B. Unless the shadow state is altered, the state of an interrupted P-process is automatically reloaded from the shadow registers when there are no high priority processes running – i.e. *goprot* is not required. However a program such as an operating system kernel might be required to store the shadow state and restart the interrupted process later. *goprot* can be used for this. See section 13.4 for details on this application.

and the instruction pointer register (**IptrReg**); and the **ps.sAreg**, **ps.sBreg** and **ps.sCreg** slots are copied into the integer stack registers. The trap, watchpoint, timeslice and single-step enable bits and all flags are loaded into the status register, from the **ps.Cntl** slot of the PDS. Also if the watchpoint enable bit (**sb.WtchPntEnbl**) is set, the contents of the **ps.eWl** and **ps.eWu** slots are loaded into the watchpoint registers (**WlReg** and **WuReg**). The new workspace area specified by the address held in **ps.sWptr** should be writable, otherwise *AccessViolation* will be signalled as soon as a local variable is stored. Furthermore, this address must be word-aligned otherwise undefined behavior will result.

If *goprot* is restarting the code at an instruction that has been interrupted or timesliced in the middle of execution, then it loads the contents of the **ps.sEreg** and **ps.sXreg** slots into the internal registers (**Ereg** and **Xreg** respectively).

The protection bit in the status register (**sb.IsPprocessBit**) is set and the processor begins to execute, under protection, the code specified by the PDS.

Section 10.4 describes what happens when this instruction triggers a single-step or watchpoint trap.

syscall

When *syscall* is executed in protected mode, the effect is to force a trap to the supervisor. More detail on this is provided in section 10.4.

10 The trap mechanism

When a process takes a trap, control is transferred in one of the following ways.

machine executing an L-process with a trap-handler specified	⇒	control is transferred to trap-handler of L-process
machine executing an L-process with a null trap-handler	⇒	machine is halted
machine executing a P-process	⇒	control is transferred to supervisor of P-process

The first two are referred to as L-process traps, and the third is referred to as a P-process trap.

The following provides a complete description of traps including:

- an overview of the L-process traps including the trap-handler data structure (a similar overview of the P-process trap is given in chapter 9);
- details on the state storage and retrieval when a trap is taken;
- a comprehensive list of trap causes and errors;
- a list of the instructions that may be used when dealing with traps, together with a description for each.

10.1 The trap-handler

All L-processes have associated with them, a 'trap-handler pointer'. For the currently executing process, this is held in the trap-handler register (**ThReg**). For a non-executing process, it is stored in the **pw.TrapHandler** slot of the process workspace. Normally an L-process has an associated trap-handler, in which case the trap-handler pointer is the address of a trap-handler data structure (THDS), which itself provides a pointer to the trap-handler code to be run when a trap is taken. More than one L-process may share a trap-handler. Alternatively, an L-process may have a null trap-handler, in which case its trap-handler pointer is a null-pointer and has the special value *NotProcess.p.*

10.1.1 The THDS (trap-handler data structure)

For an L-process to be associated with a trap-handler, its trap-handler pointer must be the word aligned address of a THDS.

A THDS records the state of the trapped process, and provides a pointer to the trap-handler code which runs when a trap is taken. Its complete construction is shown in table 10.1.

word offset	slot name	purpose
11	**th.sCreg**	L-process C register – stored from / loaded into integer stack C register when trap-handler entered/exited
10	**th.sBreg**	L-process B register – stored from / loaded into integer stack B register when trap-handler entered/exited
9	**th.sAreg**	L-process A register – stored from / loaded into integer stack A register when trap-handler entered/exited
8	**th.sIptr**	L-process instruction pointer – stored from / loaded into instruction pointer register when trap-handler entered/ exited
7	**th.sWptr**	L-process workspace pointer – stored from / loaded into **Wptr** when trap-handler entered/exited
6	**th.eWu**	upper bound of L-process watchpoint region – may be loaded into upper watchpoint register when an L-process is executed[1]
5	**th.eWl**	lower bound of L-process watchpoint region – may be loaded into lower watchpoint register when an L-process is executed[1]
4	**th.Eptr**	pointer to instruction causing trap – stored from error pointer register when trap-handler entered
3	**th.Bptr**	back of trap sharing process queue
2	**th.Fptr**	front of trap sharing process queue
1	**th.Iptr**	trap-handler instruction pointer – loaded into instruction pointer register when trap-handler entered
0	**th.Cntl**	control word

Table 10.1 Trap-handler data structure (or THDS)

This section describes very generally the source/destination of data loaded into / copied from this data structure. There are several anomalies, and for a thorough understanding of state storage and retrieval, refer to section 13.2.2. For discussion of the trapping mechanism, also refer to later sections in this chapter.

The control word: th.Cntl

When the processor starts execution[1] of an L-process, it loads the process status and control bits from the trap-handler control word (the **th.Cntl** slot of the THDS) into the status register (**StatusReg**). The control word thus determines the conditions under which the L-process traps or sets flags (see section 10.3). When the process takes an L-process trap, these status and control bits are written back into the control word in the THDS.

[The order of bits within the control word is the same as the status and control bits in the status register, as set out in table 5.5.]

There is a extra bit in the control word known as the trap-handler in use bit (**sb.ThInUse** – bit 31). This bit is *not* loaded into the status register. It provides an interlock which prevents two or more processes that share a trap-handler from simultaneously requiring use of that trap-handler. The processor achieves this by setting the bit to 1 when a trap-handler is entered and clearing it when the trap-handler is exited. (See below.)

L-process state: th.sWptr, th.sIptr, th.sAreg, th.sBreg, th.sCreg

When an L-process takes a trap, its current state is saved in the THDS. The workspace and instruction pointer are saved in **th.sWptr** and **th.sIptr** respectively. The integer stack is saved in **th.sAreg, th.sBreg** and **th.sCreg**. The processor restores this state to the processor if it is required to restart the trapped process on return from the trap.

The trap-handler's instruction pointer: th.Iptr

The instruction pointer (i.e. the entry point) of the trap-handler code is stored in the **th.Iptr** slot. When an L-process takes a trap, the content of **th.Iptr** is copied into the instruction pointer register (**IptrReg**).

Trap causing instruction: th.Eptr

When an L-process takes a trap, the address of the instruction that caused the trap (which is held in **EptrReg**), is written into the **th.Eptr** slot. [This is different to the content of the **th.sIptr** slot (see above), into which is stored the address of the instruction that was due to be executed *next*.]

Watchpoints: th.eWl and th.eWu

If watchpoints are enabled in the trap-handler control word (**sb.WtchPntEnbl** is set to 1), then when an L-process starts execution[1], the contents of the **th.eWl** and **th.eWu** slots are loaded into the watchpoint registers (**WlReg** and **WuReg** respectively). These addresses specify the lower bound and upper bound of the watchpoint region. While enabled, any instruction writing to a location in this region causes the process to take a trap.

More details on watchpointing are provided in chapter 14.

The trap-handler queue: th.Fptr and th.Bptr

Before the processor executes an L-process it checks the trap-handler in use bit to determine whether or not the trap-handler is in use. If the trap-handler is in use then the process is appended to the trap-handler queue. (If an L-process were executed while its trap-handler was in use by another process, then the trap-handler could not be invoked if a trap occurred.) This queue is specified by a front pointer (**th.Fptr**) and a back pointer (**th.Bptr**) and is linked in the same manner as the high and low priority scheduling lists (see section 8.2.1). When the trap-handler is exited, the entire trap-handler queue is inserted at the front of the appropriate priority scheduling list (i.e. the same priority at which the trap-handler is running).

The trap-handler workspace

When an L-process takes a trap, the processor loads the workspace pointer register field (**Wptr**)† with the content of the trap-handler register (**ThReg**). Hence the trap-handler can immediately use the area below the THDS as its workspace.

Since a trap-handler's workspace pointer initially points to the THDS, the trap-handler code must avoid corrupting this information. In particular, if any of the trap-handler's instructions use workspace location 0 (**pw.Temp** – see section 8.1.1), then the code should firstly adjust the workspace pointer (e.g. execute *ajw –1*) so that it points to a free memory location. Likewise, the workspace pointer must be adjusted if local variables are allocated.

The trap-handler's 'trap-handler'

The trap-handler itself initially executes with a null trap-handler, and so the trap-handler register is loaded with *NotProcess.p*.

Creation of a THDS

Any suitably sized word-aligned block of store may be used as a trap-handler data structure. The area below this block will also initially be the trap-handler workspace when an L-process takes a trap (although the trap-handler code may change this on entry). The block must be initialized: with the **th.Fptr** slot set to *NotProcess.p* to indicate that the trap-handler queue is empty, with the **th.Iptr** slot set to the appropriate address for the trap-handler code, and with the trap-handler in use bit (**sb.ThInUse**) of the control word set to 0, the flags set appropriately, and the other bits selecting traps as desired. If watchpointing is selected in the control word, then the **th.eWl** and **th.eWu** slots should be set to define the watchpoint region.

Changing a trap-handler

There are two instructions that an L-process may use to manipulate its trap-handler. The first is *load trap handler - ldth* and the second is *select trap handler - selth*. These are detailed in section 10.4.

† The process priority is not changed when the context changes to a trap-handler.

10.1.2 Sharing a trap-handler data structure

It is possible to a share a THDS (trap-handler data structure) between any number of processes. However, there are certain restrictions

- The processes sharing a THDS must be of the same priority.
- All processes sharing a THDS also share the control word in the THDS, which implies that they all have the same traps enabled (floating-point, watchpoint etc.).

  Note that it is possible to change the status and control bits in the status register while a process is executing; but be aware that if/when a timeslice or trap subsequently occurs, some of these bits are written back into the **th.Cntl** slot of the THDS and hence are shared by all processes which use this trap-handler.

The IMS T9000 will not execute† a process if its trap-handler is currently in use‡ by another process. This ensures that when a process takes a trap, a second process cannot take a trap to the same trap-handler whilst the first trap is being handled. This restriction is enforced automatically by the IMS T9000 scheduler as described in section 10.1.1.

10.1.3 The null trap-handler

If the trap-handler pointer of an L-process is the special value *NotProcess.p*, then it has a null trap-handler. Before the process is executed:

- The null trap-handler value *(NotProcess.p)* is loaded into the trap-handler register.
- The status register (**StatusReg**) is loaded with the default control word (see section 5.2). This specifies that the only enable option that is selected is the integer overflow trap (**sb.IntOvTeBit** is set to 1). Hence an L-process executing with the null trap-handler, does not trap on floating-point exceptional conditions and misalignment errors, and does not have watchpointing enabled.

If an L-process takes a trap while a null trap-handler is installed (a null trap), the processor is halted.

The main use of the null trap-handler is to provide some way of executing L-processes without a trap-handler (often just for short periods). This can be essential. For example, trap-handlers (which are themselves run as L-processes) must always be ready to execute, and cannot therefore themselves have a trap-handler when first invoked. If this were allowed, it might be that the secondary trap-handler (i.e. the trap-handler's trap-handler) would be in use by another process at the time that the primary trap-handler was invoked. The primary trap-handler would then not be ready to execute. However although a trap-handler must have a null trap-handler at the time of invocation, it is perfectly acceptable to provide it with its own trap-handler while it is executing (making use of the *select trap handler* instruction – see section 10.4).

10.2 State storage and retrieval when a trap is taken

When an L-process takes a trap, it relinquishes control to its trap-handler (or halts the machine if there is a null trap-handler specified). This is referred to as an L-process trap.

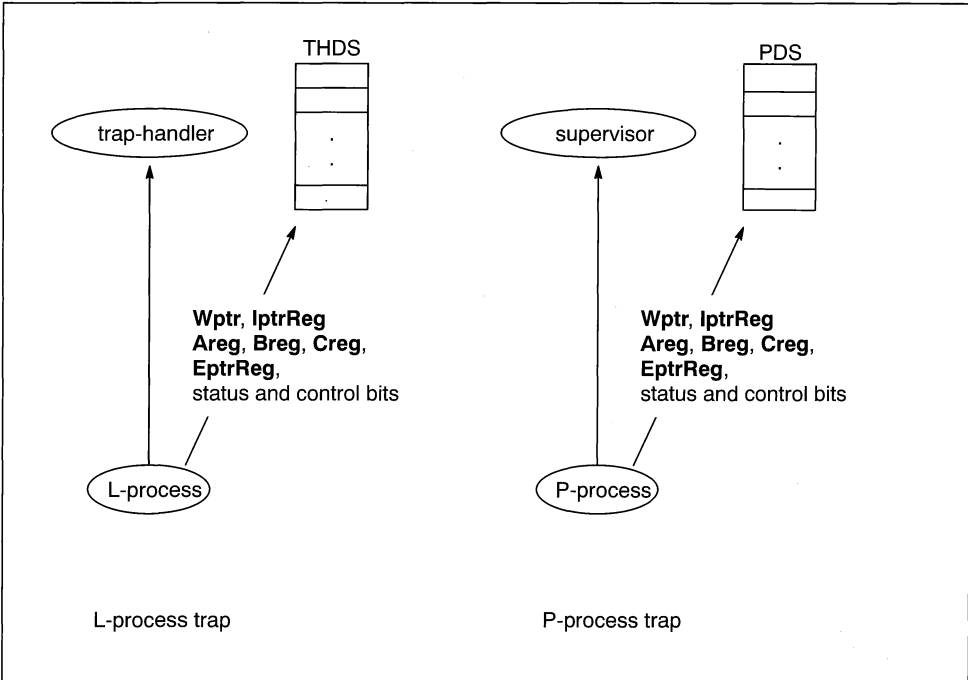
When a P-process takes a trap, it relinquishes control to its supervisor. This is referred to as a P-process trap.

† Note carefully the word 'execute' here. The transputer will *schedule* a process (i.e. place its process on the scheduling list) without checking whether its trap-handler is in use. It is only when it starts *execution* of the process (i.e. install it as the current process) that it checks. It will at this point prevent the process from being executed if a process sharing the same THDS has trapped.

‡ The term 'trap-handler in use' means that a trap has already been taken to the trap-handler specified by that particular trap-handler data structure, but the corresponding 'trap return' instructions (*tret*) has not yet been executed. However, it is possible for two L-processes with different trap-handler data structures, to use the same trap-handler code. That is, the **th.Iptr** slot can be a pointer to the same code in two or more trap-handler data structures. L-processes can thus share the same trap-handler code, without sharing workspace. There is **no restriction** on execution of L-processes which share trap-handler code but have separate trap-handler data structures.

Except when a null trap is taken (see section 10.3.3), the reason for the trap is encoded in **Areg**, and if the trap has been caused by an error, the type of error is encoded into **Breg**. **Creg** is left undefined. This section provides some detail on state storage[2] and retrieval, and a comparison between the two types of trap.

If a trap occurs as a result of a floating-point exception, then prior to taking the trap, the state of the floating-point unit is restored to the state that it had before the floating-point operation (refer to section 11.13 for more details).



An L-process trap

When an L-process takes a trap, the action depends on the trap-handler pointer (stored in the trap-handler register – **ThReg**). If it points to a THDS, then the current state is stored and the trap-handler state is set up as described below. If the trap-handler pointer is the special constant *NotProcess.p*, which signifies that there is a null trap-handler, then the machine halts (takes a null trap).

The THDS (trap-handler data structure) which is pointed to by the trap-handler register is used both to save the state of the L-process when it takes a trap, and to contain the necessary details required to invoke the trap-handler.

The contents of the instruction pointer register (**IptrReg**), **Wptr**, and integer stack registers are all written into the THDS. This action *cannot* signal a watchpoint or *AccessViolation*. The address of the instruction that has caused the trap (held in **EptrReg**) is written into the **th.Eptr** slot. The error, watchpoint, timeslice and single-step trap enable bits and all error flags in the status register (**StatusReg**) are written into the **th.Cntl** slot. Also the trap-handler in use bit (**sb.ThInUse**) is set in this slot (to prevent execution of the processes which share this trap-handler).

The workspace pointer and instruction pointer of the trap-handler are (re)installed: the content of the trap-handler register is copied into the workspace pointer register field (**Wptr**); and the content of the **th.Iptr** slot is copied into the instruction pointer register. Observe that the process priority bit of the workspace descriptor register is left unaltered. The trap-handler continues to run as an L-process at the same priority.

The trap-handler register is given the special constant value (*NotProcess.p*) which signifies that the trap-handler itself initially has a null trap-handler. The status register is loaded with the default control word (all zeroes except for **sb.IntOvTeBit**).

A P-process trap

When a P-process takes a trap, the current state is stored and the supervisor's state is retrieved as described below.

The PDS (P-state data structure) which is pointed to by the protected state register (**PstateReg**), is used both to save the state of the P-process when it takes a trap, and to contain the necessary details required to restart its supervisor.

The contents of the instruction pointer register (**IptrReg**), **Wptr**, and the integer stack registers are all written into the PDS. This action *cannot* signal a watchpoint or *AccessViolation*. The address of the instruction that has caused the trap (held in **EptrReg**) is written into the **ps.Eptr** slot. The error, watchpoint, timeslice and single-step trap enable bits, and all error flags in the status register (**StatusReg**) are written into the **ps.Cntl** slot. If the trap is taken in the middle of an interruptible instruction (i.e. due to a timeslice), then the contents of internal registers **Ereg** and **Xreg** are saved in the **ps.sEreg** and **ps.sXreg** slots of the PDS respectively.

The workspace pointer and instruction pointer of the supervisor are reinstalled: the content of the workspace descriptor stub register (**WdescStubReg**) is copied into workspace descriptor register; and the content of the **pw.Iptr** slot of the supervisor's workspace data structure, is loaded into the instruction pointer register.

If the trap-handler register points to a THDS, then the supervisor has a trap-handler installed and its state is restored from the THDS as follows. The error, watchpoint, timeslice and single-step trap enable bits and all error flags are loaded into the status register, from the **th.Cntl** slot of the THDS, and the **sb.IsPprocessBit** is reset. Also if the watchpoint trap enable bit (**sb.WtchPntEnbl**) is set, the contents of the **th.eWl** and **th.eWu** slots in the THDS are loaded into the watchpoint registers (**WlReg** and **WuReg**).

If the supervisor's trap-handler pointer has the special constant value *NotProcess.p*, then it has a null trap-handler and so the status register is loaded with the default control word (all zeroes except for **sb.IntOvTeBit**).

A comparison

It is useful to observe the similarities and differences in behavior between the above two mechanisms.

- For each there is a data structure which is used to store the state of the process when trapped. This state is required if it is necessary to examine the process state and/or it is required to restart the process at the point where the trap was taken.
- In both data structures, the control word, the workspace pointer, the instruction pointer, the integer stack registers and the address of the trap causing instruction for the trapped process are stored automatically.
- The THDS address is held in the trap-handler register, whereas the PDS address is held in the protected state register.
- The workspace pointer of the trap-handler is held in the trap-handler register, whereas the workspace pointer of the supervisor is held in the workspace descriptor stub register.
- The instruction pointer for the trap-handler code is held in a slot in the THDS, but the instruction pointer for the supervisor is held in the supervisor's workspace data structure.
- When a trap-handler is started, this must itself have a null trap-handler, and so in order to load the correct state when an L-process traps to its trap-handler, the processor only needs to install the workspace pointer and instruction pointer, and to set the status register to a default control word. However a supervisor may have its own THDS, and so when a P-process traps to its super-

visor, it is also necessary to load the process bits from the control word of its THDS and, if appropriate, load the watchpoint registers.

Exiting from a trap-handler or a supervisor

When a trap-handler has performed its function, it must execute *tret* (the 'trap return' instruction – see section 10.4), which may either:–

- terminate the execution of the process which trapped, or
- continue its execution.

If **Areg** is 0 when *tret* is executed, then the process is continued from the point where it trapped†, otherwise it is terminated. The workspace pointer register field (**Wptr**) should hold the address of the THDS when *tret* is executed. *tret* must be executed (as opposed to stopp) even when the trapped process is not to be restarted. This ensures that the 'trap-handler in-use' bit is reset, and any accumulated process queue is transferred to the scheduling list.

A supervisor restarts a trapped P-process with the *goprot* instruction.

More details on these instructions are given in section 10.4.

Extra state storage and retrieval

Observe that the floating-point and block move register state is not automatically saved when a trap is taken. There is no need for the mechanism to do so, because this state is not overwritten by the context change. However subsequent execution may overwrite the contents of the floating-point and block move registers at any time, so it is left to the trap-handler or supervisor to save and restore this state using instructions included for the purpose. Refer to section 13.2.3.

10.3 Trap causes and signalling of errors

This section is concerned with the events that can cause a process to take a trap, or set flags. The description applies to both P-process traps and L-process traps, unless explicitly stated.

10.3.1 Trap causes

As summarized in the introduction to this chapter, there are two types of trap. When an L-process is executing, a trap transfers control to its trap-handler (L-process trap), and when a P-process is executing, a trap returns control to its supervisor. This subsection considers the various trap causes, discusses which of these can occur simultaneously, and explains how these causes are presented to the trap-handler or supervisor.

The events that can cause traps are given in table 10.2 and explained below.

name	**notes**
single-step	
error	
syscall (system call)	
causeerror	
breakpoint	
watchpoint	
timeslice	P-process trap only

Table 10.2 General trap causes and their abbreviations

† unless the trap-handler has modified the **th.slptr** slot

If a process has single-stepping enabled, as indicated by the single-stepping trap enable bit (see section 14.2) in the status register, then it takes a trap when it has finished executing any instruction.

If an error (or a floating-point exceptional condition) is detected, then this is signalled and may cause a trap. There are a number of such 'signals' and these are considered separately in section 10.3.2.

When a process executes *syscall*, *causeerror* (see section 10.4) or the breakpoint instruction (*j0*), then it traps to its trap-handler or supervisor.

If a process has watchpointing enabled, as indicated by the watchpoint trap enable bit (see section 14.3) in the status register, then it takes a trap if an instruction attempts to write to an address in the watchpoint region.

When a timeslice becomes due while a low priority P-process is executing, then the P-process traps to its supervisor at the next interrupt point.

Traps for multiple reasons

The trap causes listed above may not be unique (i.e. there may be more than one cause of a trap).

Table 10.3 shows which trap causes can occur together. An entry of 'Y' indicates that the two trap causes can be coincident, an entry of 'N' that they cannot.

	single-step	error	*syscall*	*causeerror*	breakpoint	watchpoint	timeslice‡
single-step	–	Y	Y	Y	Y	Y	Y‡
error	Y	–	N	N	N	Y†	N
syscall	Y	N	–	N	N	N	N
causeerror	Y	N	N	–	N	N	N
breakpoint	Y	N	N	N	–	N	N
watchpoint	Y	Y†	N	N	N	–	Y‡
timeslice‡	Y‡	N	N	N	N	Y‡	–

† a watchpoint can only be coincident with an error (viz. *AccessViolation*) in a P-process

‡ a timeslice only causes a trap from a P-process

Table 10.3 Trap causes that can occur simultaneously

Indication of cause of traps

The trap-handler or supervisor must be able to determine the trap cause (or causes). The reporting of the reason for a trap is essentially the same for both an L-process trap and a P-process trap. Firstly, the address of the instruction that caused the trap and/or the address of the next instruction to be executed are written into the handler's data structure. Secondly, two values are used to convey why the instruction trapped; the first, delivered in **Areg** (see table 10.5), indicates the trap reason, the second, delivered in **Breg** (see section 10.3.2) indicates which type of error (if any) has occurred.

The address of the trap causing instruction is written into **th.Eptr** of the THDS for an L-process trap, or to **ps.Eptr** of the PDS for a P-process trap. The only times where this information is not valid is when a P-process trap has been taken between instructions due to a timeslice, or when a memory semantics error has occurred (see page 120).

It is possible that an L-process is due to be descheduled when it is trapped. More specifically, descheduling can be coincident with a single-step or a watchpoint (if the process workspace data structure is watchpointed). Although the descheduling action is not a trap cause, this information is encoded into the the word delivered in **Areg**. The trap-handler is then aware that the trapped process cannot be immediately restarted on trap return.

Similarly, it is possible for an L-process trap to occur from an instruction that would normally change the context of the process. More specifically, *goprot*, *selth* (which changes the trap-handler – see section 10.4)

and *restart* (which restarts an interrupted L-process – see section 13.4) can cause a single-step or watchpoint trap. In this situation, these instructions simply take a trap after they have written the current state into the control word of the trap-handler data structure. It is then left to the trap-handler to change the context. Although the context change is not a trap cause, this information is encoded into the the word delivered in **Areg**. The trap-handler is then aware that the trapped process cannot be restarted using *tret*.

The trap reason value, delivered in **Areg**, is encoded as shown in table 10.4.

field	bit 4	bits 3 .. 1	bit 0
meaning	1: single-step 0: not single-step	000: error 001: breakpoint 010: none of this set 011: timeslice 100: *syscall* 101: deschedule 110: *causeerror* 111: context	1: watchpoint 0: not watchpoint

Table 10.4 Bit mapping of trap reason delivered in **Areg**

This coding uses 5 bits which allows 32 possible values. Of these only 23 can ever occur, according to the combinations of trap causes that can occur simultaneously (as given in table 10.3). (Note that the trap causes encoded by bits 3 .. 1 cannot be coincident.) Table 10.5 summarizes.

trap reason value (hex)	**trap reason symbol**	**meaning**	**note**
0 (#00)	**t.Error**	error	
1 (#01)	**t.WatchError**	watchpoint and error	P-process trap only
2 (#02)	**t.Break**	breakpoint	
3-4 (#03-#04)		*invalid*	
5 (#05)	**t.Watch**	watchpoint	
6 (#06)	**t.Time**	timeslice	P-process trap only
7 (#07)	**t.WatchTime**	watchpoint and timeslice	P-process trap only
8 (#08)	**t.Scall**	*syscall*	
9-10 (#09-#10)		*invalid*	
11 (#0B)	**t.WatchDesch**	watchpoint and deschedule	L-process trap only
12 (#0C)	**t.Cerror**	*causeerror*	
13-14 (#0D-#0E)		*invalid*	
15 (#0F)	**t.WatchContext**	watchpoint and context	L-process trap only
16 (#10)	**t.StepError**	single-step and error	
17 (#11)	**t.StepWatchError**	single-step, watchpoint and error	P-process trap only
18 (#12)	**t.StepBreak**	single-step and breakpoint	
19 (#13)		*invalid*	
20 (#14)	**t.Step**	single-step	
21 (#15)	**t.StepWatch**	single-step and watchpoint	
22 (#16)	**t.StepTime**	single-step and timeslice	P-process trap only
23 (#17)	**t.StepWatchTime**	single-step, watchpoint and time-slice	P-process trap only
24 (#18)	**t.StepScall**	single-step, *syscall*	
25 (#19)		*invalid*	
26 (#1A)	**t.StepDesch**	single-step and deschedule	L-process trap only
27 (#1B)	**t.StepWatchDesch**	single-step, watchpoint and des-chedule	L-process trap only
28 (#1C)	**t.StepCerror**	single-step and *causeerror*	
29-31 (#1D)		*invalid*	
30 (#1E)	**t.StepContext**	single-step and context	L-process trap only
31 (#1F)	**t.StepWatchContext**	single-step, watchpoint and con-text	L-process trap only

Table 10.5 Possible trap reasons as loaded into **Areg** when a is taken

10.3.2 Signalling of errors

A process executing on an IMS T9000 can detect a variety of errors. The precise errors that a particular process detects depend on the mode of the process (i.e. L-process or P-process). The signalling of certain errors may cause a trap to be taken, or flags to be set within the status register. This is controlled by the setting of bits known as 'error trap enable bits'. This subsection presents these flags and trap enable bits, and details the different types of errors.

The errors, that a process executing on the IMS T9000 can detect, are:–

- errors that are explicitly set or checked – These are caused by the execution of an error setting instruction (e.g. *causeerror* – see section 10.4) or by error detection instructions. An example of the latter is an array subscript error detected by a *csub0* instruction.
- integer overflow – This is signalled when there is an integer overflow or integer division by zero.
- misalignment – This is signalled when a instruction attempts to load or store a 32-bit or 16-bit object from a byte address that is not 32-bit or 16-bit aligned.
- attempt to execute illegal instruction – This is signalled when *opr* is executed with an invalid operand – i.e. not an instruction.
- attempt to execute privileged instruction in protected mode
- access violation in protected mode
- floating-point exceptional conditions – The IMS T9000 can detect the conditions specified as exceptional conditions by the IEEE floating-point standard. These are 'invalid operation', 'divide by zero', 'overflow', 'underflow' and 'inexact result'. In addition to these it also detects a condition called, 'floating-point error'. The latter is detected whenever an arithmetic operation has as an operand, an infinity or a not-a-number, and when the conditions 'invalid operation', 'divide by zero', or 'overflow' are true. (Detection of 'floating-point error' ensures compatibility with the treatment of floating-point errors by the IMS T805.)
- memory semantics violation – see page 120

The detection of any of these distinct operating conditions is signalled, either by the setting of one or more flags, or by a trap being taken. The complete set of 'error signals' are defined in table 10.6.

error signal	brief description	note
IntegerOverflow	integer overflow or integer divide-by-zero	
IntegerError	integer error other than *IntegerOverflow* – e.g. explicitly checked or explicitly set error, misuse of channel	
Unalign	address of instruction operand is not aligned to the correct boundary	
IllegalInstruction	attempt to execute an illegal instruction	
PrivInstruction	attempt to execute a privileged instruction in protected mode	P-process trap only
AccessViolation	attempt to access a memory protected or non-existent address	P-process trap only
FPError	floating-point 'error'	
FPInvalidOp	IEEE floating-point 'invalid operation'	
FPDivideByZero	IEEE floating-point 'divide by zero'	
FPOverflow	IEEE floating-point 'overflow'	
FPUnderflow	IEEE floating-point 'underflow'	
FPInexact	IEEE floating-point 'inexact result'	

Table 10.6 Definition of errors signalled by the IMS T9000

IntegerError and *IntegerOverflow* are sometimes referred to generically as integer errors, and *FPError*, *FPInvalidOp*, *FPDivideByZero*, *FPOverflow*, *FPUnderflow* and *FPInexact* are referred to as floating-point exceptional conditions.

Flags

Various bits within the status register (**StatusReg**) are reserved as flags. (See section 5.2.) These are set when certain errors are signalled as outlined in table 10.7. More than one error may be signalled by an instruction, and more than one flag may be set; however no flags are set if a trap is taken[2].

name	error signals	notes
sb.IntOvFlag	*IntegerOverflow*	
sb.FPErrorFlag	*FPError*	corresponds to IMS T805 floating-point error flag
sb.FPInOpFlag	*FPInvalidOp*	
sb.FPDivByZeroFlag	*FPDivideByZero*	
sb.FPOvFlag	*FPOverflow*	
sb.FPUndFlag	*FPUnderflow*	
sb.FPInexFlag	*FPInexact*	

Table 10.7 Flags and the error signals that cause them to be set

Error trap enable bits

Various bits within the status register (**StatusReg**) are reserved as error trap enable bits. (See section 5.2.) Table 10.8 shows which error signals cause a trap to be taken when each of these bits is set.

name	error signals
sb.IntOvTeBit	*IntegerOverflow*
sb.UnalignTeBit	*Unalign*
sb.FPErrorTeBit	*FPError*
sb.FPInOpTeBit	*FPInvalidOp*
sb.FPDivByZeroTeBit	*FPDivideByZero*
sb.FPOvTeBit	*FPOverflow*
sb.FPUndTeBit	*FPUnderflow*
sb.FPInexTeBit	*FPInexact*

Table 10.8 Error trap enable bits and error signals that cause a trap when bits are set

If a trap for a certain error signal is explicitly disabled, and a trap does not occur for any other reason, then the instruction completes execution, and a flag is set according to table 10.7 (although there is no flag for *Unalign*). If the trap is enabled, a value is presented to the trap-handler or supervisor (in **Breg**) to indicate the error that has occurred (see later – table 10.10).

Observe that although two types of integer error have been classified (table 10.6), it is only possible to selectively enable trapping on the *IntegerOverflow* error. Usually an integer error is either very serious (e.g. a range check fails) or is deliberate (e.g. if forced by the *causeerror* instruction). In these cases the error cannot be ignored and the option to disable traps is not provided. For the potentially less serious integer errors such as arithmetic overflow and division by zero, this option is provided with **sb.IntOvTeBit**.

Effect of error signals

Table 10.9 summarizes the effect of each of the error signals.

error signal	extra condition for trap to be taken	flags set if trap not taken
IntegerOverflow	**sb.IntOvTeBit**	**sb.IntOvFlag**
IntegerError	*true*‡	not applicable
Unalign	**sb.UnalignTeBit**	none
IllegalInstruction	*true*‡	not applicable
PrivInstruction	in protected mode	none
AccessViolation	in protected mode	none
FPError	**sb.FPErrorTeBit**	**sb.FPErrorFlag**
FPInvalidOp	**sb.FPInOpTeBit**	**sb.FPInOpFlag**
FPDivideByZero	**sb.FPDivByZeroTeBit**	**sb.FPDivByZeroFlag**
FPOverflow	**sb.FPOvTeBit**	**sb.FPOvFlag**
FPUnderflow†	**sb.FPUndTeBit**	**sb.FPUndFlag**
FPInexact	**sb.FPInexTeBit**	**sb.FPInexFlag**

† To comply with the IEEE standard, *FPUnderflow* is signalled on detection of 'tininess' if trapping is enabled, but otherwise on detection of 'tininess' *and* 'inexact'. See section 11.2.5.

‡ i.e. a trap is always taken when these errors are signalled.

Table 10.9 Effect of signalling errors

When an error is signalled, a trap is taken if the appropriate trap enable bit is set according to table 10.9. For some signals (viz. *IntegerError*, *IllegalInstruction*, *PrivInstruction*, and *AccessViolation*), there is no trap enable bit. These signals always cause a trap to occur. *PrivInstruction* and *AccessViolation* can only be generated when running under protection.

If an L-process has the null trap-handler, the signalling of *IntegerError*, *IntegerOverflow* or *IllegalInstruction* causes the processor to stop†. (Note that there is no equivalent to a null trap-handler when running a P-process. A trap *always* returns control to its supervisor.)

Indication of which errors (if any) have caused the trap

When an error causes a process to take a trap, the processor loads a unique value into **Breg** to indicate which particular error was the cause. A complete list of the values that may be loaded into **Breg**, is given in the first column of table 10.10. The second column of this table gives the symbol for each error. The third column states the condition that causes each error to occur. The fourth column shows the meaning of each error.

† In this way, the execution of an L-process without a trap-handler is compatible with the execution of process on the IMS T805 in 'halt-on-error' mode.

error type value (hex)	error type symbol	condition for trap to occur	error meaning
0 (#00)	**et.NoError**	any non-error trap reason	trap has not been caused by an error
1 (#01)	**et.PrivInstruction**	*PrivInstruction* (when running under protection)	attempt to execute a privileged instruction in protected mode
2 (#02)	**et.IllegalInstruction**	*IllegalInstruction*	attempt to execute illegal instruction
3 (#03)	**et.Unalign**	*Unalign* AND **sb.UnalignTeBit**	address of instruction operand is not aligned to the correct boundary
4 (#04)	**et.AccessViolation**	*AccessViolation* (when running under protection)	attempt to access a memory protected or non-existent address
5 (#05)	**et.IntegerError**	*IntegerError*	integer error, not indicated by integer overflow
6 (#06)	**et.IntegerOverflow**	*IntegerOverflow* AND **sb.IntOvTeBit**	integer overflow or integer divide-by-zero
7 (#07)	**et.FPError**	*FPError* AND **sb.FPErrorTeBit**	floating-point error†
8 (#08)	**et.FPInvalidOp**	*FPInvalidOp* AND **sb.FPInOpTeBit** AND (NOT *'conditions for **et.FPError**'*)	IEEE floating-point 'invalid operation'
9 (#09)	**et.FPDivideByZero**	*FPDivideByZero* AND **sb.FPDivByZeroTeBit** AND (NOT *'conditions for **et.FPError**'*)	IEEE floating-point 'divide by zero'
10 (#0A)	**et.FPOverflow**	*FPOverflow* AND **sb.FPOvTeBit** AND (NOT *'conditions for **et.FPError**'*)	IEEE floating-point 'overflow'
11 (#0B)	**et.FPUnderflow**	*FPUnderflow* AND **sb.FPUndTeBit** AND (NOT *'conditions for **et.FPError**'*)	IEEE floating-point 'underflow'
12 (#0C)	**et.FPInexact**	*FPInexact* AND **sb.FPInexTeBit** AND NOT (*'conditions for **et.FPError**' OR 'conditions for **et.FPOverflow**' OR 'conditions for **et.FPUnderflow**'*)	IEEE floating-point 'inexact result'
13 (#0D)	**et.MemSemError**	see below	memory semantics error
† This is not an IEEE exception. See section 11.13 for more details.			

Table 10.10 Possible error types as loaded into **Breg** when a trap is taken

Because of the concurrent nature of instruction execution in the pipeline, it is not always possible for the processor to instantly determine the exact instruction that has caused an error. The processor may need to re-execute some of the most recently executed instructions, one at a time. It should be able to reproduce the same error and thereby take a trap with the correct error type value. This behavior is transparent to

the user. However it is possible that on re-execution, the error does not occur because the same values have not been read from memory. This is known as a memory semantics error and can only happen if

(i) there is a memory fault, or

(ii) a memory location has changed.

The following might cause the latter, but these practices *should be avoided*.

- *locations that are used by external channels* – The data in locations below *MemStart* (see section 12.4) may change asynchronously. In particular, the programmer should not attempt to directly access a virtual link control block or a packet buffer associated with a virtual link while that link is communicating. If this is attempted then there is no guarantee that two consecutive reads from such locations will yield the same data, and so a memory semantics error might occur.
- *memory mapped device in uncached memory* – A memory mapped device should be marked as 'device memory'. As explained in section 7.14, this ensures that only device instructions are used to access a device. A memory semantics error cannot occur when a device instruction is executed. However, if a normal load instruction is used to read from a device, this might result in a memory semantics error.
- *sharing memory between processors* – Care must be taken if using transputers in a shared memory system. One scheme to ensure that a memory semantics error does not occur, is to follow the procedure described in section 7.14. However if a program allows a piece of shared memory to be written by another processor while it is reading from that memory, a memory semantics error might occur.

If the processor detects such a memory semantics violation, it presents the special error type **et.MemSemError** to the trap-handler in place of the original error. But note that a memory semantics error will only occur as a result of the processor attempting and failing to determine the initial error that caused the trap.

T8-series compatible error handling

For an L-process with a trap-handler data structure, it is possible to detect the same errors that would be detected on a T8-series transputer. This is ensured by enabling integer overflow trapping, disabling misalignment trapping and all the floating-point traps.

For an L-process with a null trap-handler the error handling is compatible with an IMS T805 in halt-on-error mode.

Summary

Any error flag can only be set by a unique error signal (see table 10.7), but note that more than one flag may be set if more than one signal is raised.

Similarly, when a trap is taken, all error types that may be presented in **Breg**, are due to a unique error signal. In this case however, when more than one signal is raised concurrently, only one 'error' is presented to the trap-handler or supervisor. If an error is signalled and the corresponding trap enable bit(s) is set, then a trap is taken. If this is the only signal with its trap enable bit set, then this is the error presented to the trap-handler or supervisor in **Breg**. If there is more than one error signal with its trap enable bit set, then the error type presented in **Breg** is determined according to the conditions listed in table 10.10. For a full explanation of the precedence rules, refer to section 11.13.

10.3.3 Null trap causes

Only a subset of the trap causes listed in table 10.2 can cause a trap to occur while a process has a null trap-handler. More explicitly, it is possible for '*syscall*', '*causeerror*', 'breakpoint' and 'error' traps to be taken, but it is not possible for a 'watchpoint' or 'single-step' trap to be taken.

When a process is first installed with a null trap-handler, the status register is set to the 'default control word' (see section 5.2). Hence the only error signals that are initially trapped are *IntegerError*, *IntegerOverflow* and *IllegalInstruction*. A process can however explicitly set the the *error* trap enable flags using the *stflags* instruction. So for example a null trap can be forced when *Unalign* is signalled. Note that *stflags* does not change the values of the single-step and watchpoint trap enable bits, so single-step and watchpoint traps can never occur when a null trap-handler is being used.

Since a null trap causes the processor to halt, it is not possible to determine the trap reason and error type until the processor is rebooted. This is discussed in the *Control system* chapter of *The T9000 Hardware Reference Manual*.

10.4 Instructions

The instructions listed in this section are relevant to the use of traps.

mnemonic	name
syscall	system call
causeerror	causeerror
tret	trap return
goprot	go protected
ldth	load trap-handler
selth	select trap-handler
ldflags	load error flags
stflags	store error flags

Table 10.11 Instructions used in conjunction with trap-handling

syscall

The user can force control to be transferred to a trap-handler or supervisor by executing *syscall*.

syscall forces the current process to take a trap. Hence in an L-process it transfers control to the trap-handler (or takes a null trap), and in a P-process it returns control to the supervisor. Unless a null trap is taken (see section 10.3.3), the reason for the trap is delivered in **Areg** (see table 10.5) to inform the trap-handler or supervisor why a trap has been taken. The only trap cause that can occur simultaneously with *syscall* is a 'single-step' because, if enabled, this occurs after all instructions. The trap reason assigned to **Areg** is therefore either **t.Scall** or **t.StepScall**. The state is saved as for all traps, and this can be inspected or manipulated by the trap-handler or supervisor. The error type **et.NoError** is loaded into **Breg**. **Creg** is left undefined.

Note that when a trap is taken, the integer stack is stored into the THDS or PDS. This enables the executing process to pass up to three parameters if required.

For more detail on state storage and retrieval, refer to section 10.2.

causeerror

A program can simulate the occurrence of an error with the *causeerror* instruction. The instruction obtains the type of error to simulate from the value held in **Areg**. This value should correspond to one of the error types given in table 10.10. If it does not, then *causeerror* signals *IntegerError*.

causeerror forces the current process to take a trap. Hence in an L-process it returns control to the trap-handler (or takes a null trap), and in a P-process it returns control to the supervisor. The reason for the trap is delivered in **Areg** (see table 10.5) to inform the trap-handler or supervisor why a trap has been taken. The only trap cause that can occur simultaneously with *causeerror* is a 'single-step' because, if en-

abled, this occurs after all instructions. The trap reason assigned to **Areg** is therefore either **t.Cerror** or **t.StepCerror**. The state is saved as for all traps, and this can be inspected or manipulated by the trap-handler or supervisor. The type of the error being simulated (i.e. the value initially in **Areg**) is loaded into **Breg**. **Creg** is left undefined.

Note that this instruction forces a trap to be taken regardless of whether trapping has been enabled for the particular error that is being simulated.

For more detail on state storage and retrieval, refer to section 10.2.

tret

tret is used to return from a trap-handler. If the value stored in **Areg** is zero then it returns control to the L-process that has its state stored in the THDS. Otherwise it allows the next process on the scheduling list to be executed. Before *tret* is executed, the workspace pointer register field (**Wptr**) should hold the address of the THDS (trap-handler data structure) for the current trap-handler. The following paragraphs describe how *tret* saves and loads state.

The current state of the trap-handler, including the workspace pointer and the instruction pointer, are not saved. The trap-handler in use bit (**sb.ThInUse**) is reset in the **th.Cntl** slot of the THDS to signify that the trap-handler is no longer in use. If the THDS has accumulated a local process queue pointed to by the **th.Fptr** and **th.Bptr** slots, then this entire queue is placed on the front of the scheduling list of the appropriate priority (i.e. the same priority as the trap-handler).

If the value held in **Areg** is non-zero, then the processor terminates the trapped L-process, and starts to execute the next process on the scheduling list.

If the value in **Areg** is zero, then the processor reloads the registers with the state saved in the THDS. The address of the THDS contained in **Wptr** is copied into the trap-handler register (**ThReg**). The contents of the **th.sWptr** and **th.sIptr** slots of the THDS are copied into **Wptr** and the instruction pointer register (**IptrReg**); and the **th.sAreg**, **th.sBreg** and **th.sCreg** slots are copied into the integer stack registers. The error, watchpoint, timeslice and single-step trap enable bits, and all error flags are loaded into the status register, from the **th.Cntl** slot of the THDS. If the watchpoint trap enable bit (**sb.WtchPntEnbl**) is set, the contents of the **th.eWl** and **th.eWu** slots are copied into the watchpoint registers (**WlReg** and **WuReg**).

tret is a privileged instruction. Prior to executing *tret*, the trap-handler itself must have a null trap-handler. If this is not the case (i.e. **ThReg** holds a value other than *NotProcess.p*) then the instruction signals *IntegerError*. If the trap-handler queue is non-empty when *tret* is executed, then the contents **th.Fptr** and **th.Bptr** slots must be word aligned otherwise the behavior of the instruction is undefined. Note that provided these slots are initialized correctly, they will always be word aligned unless explicitly manipulated by the user.

goprot

When *goprot* is executed, the IMS T9000 begins to execute in protected mode. More detail on this is provided in section 9.8.

If this instruction triggers a single-step or watchpoint trap, then it does not start the P-process. The trap delivers in **Areg**, one of the trap reasons **t.WatchContext**, **t.StepContext** or **t.StepWatchContext**.

ldth

ldth pushes a copy of the trap-handler pointer (held in the trap-handler register – **ThReg**) on to the integer stack. This instruction must be used if the current process needs the address of its THDS. It may, for example, be necessary to store this for later use, if a new trap-handler is to be installed (see *selth*). It is a privileged instruction.

selth

The *selth* instruction can be used to change the trap-handler of the current process. Its action is to load the trap-handler register (**ThReg**) with the value held in **Areg**. The current process status and control bits

are saved in the old trap-handler (THDS) and new values for these bits are loaded from the new trap-handler. This instruction undefines the integer and floating-point stacks. It is privileged, and is a descheduling point.

If the value currently held in the trap-handler register is the special constant *NotProcess.p*, which represents the null trap-handler, then it is not necessary to store any status information. Otherwise the error, watchpoint, timeslice and single-step trap enable bits, and all error flags in the status register (**StatusReg**) are written into the **th.Cntl** slot of the current trap-handler.†

If this instruction triggers a single-step or watchpoint trap, then it does not install the new trap-handler. The trap delivers in **Areg**, one of the trap reasons **t.WatchContext**, **t.StepContext** or **t.StepWatchContext**.

Assuming a trap is not taken, the value in **Areg** is loaded into the trap-handler register. This value is either the word-aligned address of the THDS for the new trap-handler, or is the special value indicating a null trap-handler. (If address held in **Areg** is not word-aligned, then undefined behavior will result.) If the new trap-handler specified is the null trap-handler then the default control word is loaded into the status register. Otherwise, unless the new trap-handler is in use, the trap, watchpoint, timeslice and single-step trap enable bits, and all flags are loaded into the status register, from the **th.Cntl** slot of the new THDS. If the watchpoint trap enable bit (**sb.WtchPntEnbl**) is set, the contents of the **th.eWl** and **th.eWu** slots are loaded into the watchpoint registers (**WlReg** and **WuReg**). If the new trap-handler is currently in use, then the current instruction pointer (content of **IptrReg**) and the new value in the trap-handler register are saved in the **pw.Iptr** and **pw.TrapHandler** slots below the local workspace; and the current process is descheduled and appended to the queue of that trap-handler.

Note that the state of the old trap-handler is saved into the old control word before the state is loaded from the new control word. This implies that if **Areg** specifies the existing trap-handler, the instruction does not affect the status register. *selth* can therefore *not* be used to change the contents of the status register without changing the trap-handler.

ldflags

The instruction *ldflags* copies all the error flags and error trap enable bits from the status register (**StatusReg**) into the corresponding bit positions in **Areg**. All other bits in **Areg** are set to 0. The values previously held in **Areg** and **Breg** are pushed into **Breg** and **Creg** respectively. N.B. This instruction only writes into **Areg**, the bits listed in tables 10.7 and 10.8 which do *not* include **sb.StepBit** and **sb.WtchPntEnbl**.

stflags

The action of the *stflags* instruction is to overwrite the error flags and error trap enable bits in the status register with the values specified at the corresponding bit positions in **Areg**. The setting of the other bits in **Areg** is ignored. The integer stack is popped one level by this operation, leaving the value **Creg** undefined. N.B. This instruction only writes into **Areg**, the bits listed in tables 10.7 and 10.8 which do *not* include **sb.StepBit** and **sb.WtchPntEnbl**. Single-stepping and watchpointing can only be enabled by writing to the trap-handler (or supervisor) control word. (See also chapter 14.)

† When *selth* is executed, there will never be any processes on the queue of the trap-handler which is being replaced, because the trap-handler cannot be in use. By definition, it is not being used by the current process, and if any other process were using it, the current process wouldn't have been allowed to execute.

1. 'starts execution' in this context means installs the L-process as the current process. This may occur either when an L-process comes to the front of the scheduling list, or when a trapped L-process is restarted.

2. Although this chapter generally discusses process state storage when a trap occurs, the details of state storage are different for various types of trap – in particular traps caused by errors or floating-point exceptions. These anomalies are thoroughly presented in section 13.2.2.

11 Floating-point instructions

11.1 IEEE floating-point arithmetic

Parts of this chapter assume a thorough understanding of the *ANSI/IEEE standard 754-1985 – An American national standard for binary floating-point arithmetic*. Henceforth, this will be referred to as 'the IEEE standard' or simply 'the standard'.

In summary, the standard specifies:–

- single and double precision floating-point number formats
- arithmetic operations: add, subtract, multiply, divide, square-root, remainder; and compare operations
- conversions between integer and floating-point formats
- conversions between different floating-point formats
- floating-point exceptions and the behavior required of an implementation when certain exceptional conditions occur

It also states that there should be

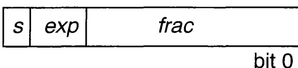
- two representations of zero (± 0),
- two infinities ($\pm\infty$) and
- two types of special symbol – quiet NaNs (not-a-numbers) and signalling NaNs.

11.2 The implementation of IEEE floating-point arithmetic on the IMS T9000

11.2.1 Formats

The IMS T9000 implements single precision and double precision formats of floating-point numbers as described in the IEEE standard. It does not implement the extended formats.

Single precision (`REAL32`) and double precision (`REAL64`) values are stored in the following formats



where *s* is the sign bit, *exp* is the exponent and *frac* is the fraction. For single precision, *s* is 1 bit wide, *exp* is 8 bits wide and *frac* is 23 bits wide. For the double precision, *s* is 1 bit wide, *exp* is 11 bits wide and *frac* is 52 bits wide. Whenever the *exp* field is not 0 the actual fraction of the number represented has an 'implied' 1 placed on the left of the *frac* value.

A floating-point value is given by

$$val \; \boxed{s \mid exp \mid frac} = \begin{cases} (-1)^s \times 1.\text{frac} \times 2^{\text{exp}-\text{bias}}, & \text{if exp} \neq 0; \\ (-1)^s \times 0.\text{frac} \times 2^{1-\text{bias}}, & \text{if exp} = 0; \end{cases}$$

where *bias* is 127 for single precision and 1023 for double precision

11.2.2 Floating-point operations

A *floating-point operation* is one of the operations described in section 5 of the IEEE standard. All of these operations are implemented by floating-point instructions described in this chapter.

The following sets of floating-point instructions implement floating-point operations: the arithmetic instructions (including *fprem* and *fprange*), the comparison instructions, and real to real type conversion instructions (including *fpint*, *fprtoi32* and *fpadddbsn*[1]). The full list of instructions that implement floating-point operations is therefore:–

fpadd	*fpldnladdsn*	*fpmulby2*	*fpeq*	*fpr32tor64*
fpsub	*fpldnladddb*	*fpdivby2*	*fpgt*	*fpr64tor32*
fpmul	*fpldnlmulsn*	*fpexpinc32*	*fpge*	*fpint*[1]
fpdiv	*fpldnlmuldb*	*fpexpdec32*	*fplg*	*fprtoi32*[1]
fprem		*fpabs*	*fpordered*	*fpadddbsn*[1]
fprange		*fpsqrt*		

11.2.3 Exceptions

IEEE exceptions

In accordance with the IEEE standard, there are a number of conditions – referred to as '(floating-point) exceptional conditions' – that are detected. Detection of each of these conditions causes one of the signals described in section 10.3.2. These are reproduced in table 11.1.

signal	exception for which condition has been detected
FPInvalidOp	IEEE 'floating-point invalid operation'
FPDivideByZero	IEEE 'floating-point divide by zero'
FPOverflow	IEEE 'floating-point overflow'
FPUnderflow	IEEE 'floating-point underflow'
FPInexact	IEEE 'floating-point inexact result'

Table 11.1 Signals raised when an IEEE exceptional condition is detected

One of two actions is taken as a result of such a detection.

- a trap is taken – Control is transferred to a trap-handler or supervisor which deals with the exception.
- a flag is set – If the trap is not selected, the appropriate flag is set, and for floating-point instructions that deliver a floating-point result, that result is defined as follows:for '(floating-point)† invalid operation' the result is a quiet not-a-number, for a 'divide by zero' or a 'floating-point overflow' the result is an infinity, and for an 'underflow' or an 'inexact result' the result is the correctly rounded value.

Special exceptional condition – *FPError*

The IMS T9000 provides an additional exceptional condition – *FPError.* This is signalled when either a not-a-number or an infinity is used as an operand to a floating-point operation, or when one or more of the conditions *FPInvalidOp*, *FPOverflow*, or *FPDivideByZero* is detected. This allows the user to make use of a simpler means of indicating 'errors', as discussed in section 11.13.

This chapter does not explicitly list exceptional conditions for each instruction. They are however comprehensively listed in appendix A.

11.2.4 Not-a-Number representations (NaNs)

The IEEE standard provides a special category of floating-point 'value' – namely, a 'not-a-number' (abbreviated to 'NaN'). There are two types of NaNs: signalling NaNs and quiet NaNs. They are represented by floating-point number with its exponent field set to to all '1's and with a non-zero fraction field.

† 'floating-point' will be excluded from now on when referring to the IEEE exceptions, except for 'floating-point overflow' which could be confused with its integer equivalent.

- *signalling NaN*

  The occurrence of a signalling NaN as the operand of any floating-point operation signals *FPInvalidOp*.

  The IMS T9000 implements signalling NaNs as having the most significant bit of their fraction part set to 0.

- *quiet NaN*

  The occurrence of a quiet NaN as an operand to any floating-point operation is not necessarily an exceptional condition.

  The IMS T9000 implements quiet NaNs as having the most significant bit of their fraction part set to 1.

Where a floating-point dyadic operation is required to return a floating-point value, it has the following behavior with regard to Not-a-Number operands. If only one operand is a NaN, then if it is a *quiet* NaN then it is returned as the result, whereas if it is a *signalling* NaN, a quiet version of that NaN is returned. This ensures that NaNs propagates through expressions. If both operands are NaNs then to meet the IEEE standard, one must be returned — the IMS T9000 produces results according to the following rules. If both operands are quiet NaNs, then the result is the NaN in **FPAreg**. If the operand in **FPAreg** is a signalling NaN, then the result is a quiet version of that NaN. Otherwise the result is a quiet version of the NaN in **FPBreg**. (See section 11.14 for more detail.)

For every other case where a floating-point operation signals *FPInvalidOp* and requires a floating-point result, a unique quiet NaN results if trapping is disabled. In this way if the result of an operation is a Not-a-Number it is possible to detect what type of error has occurred. (See also section 11.14.)

11.2.5 Implementation of underflow

The IEEE standard defines two criteria for detecting 'underflow': 'tininess' and 'loss of accuracy'. For the 'underflow' trap to be taken, it is sufficient that 'tininess' has been detected. For the 'underflow' flag to be set, both 'tininess' and 'loss of accuracy' must have been detected.

Section 7.4 of the IEEE standard gives the implementor the choice between detecting tininess before or after rounding. The IMS T9000 detects tininess after rounding. This applies whether or not the underflow trap is enabled.

The standard also gives the implementor the choice of detecting loss of accuracy via denormalization loss or inexact result. The IMS T9000 detects loss of accuracy via inexact result.

11.3 Floating-point stack

In addition to the three deep stack of integer registers — **Areg**, **Breg** and **Creg** (see section 7.1) — the processor contains a three deep stack of floating-point registers:

FPAreg	floating-point stack register A
FPBreg	floating-point stack register B
FPCreg	floating-point stack register C

Each floating-point register can hold either a single precision or a double precision value† and has a tag associated with it (stored in the floating-point status register) which signifies the precision of the value it contains. The floating-point stack behaves in a similar manner to the integer stack. When a value is loaded in **FPAreg** the values in **FPAreg** and **FPBreg** are pushed down into **FPBreg** and **FPCreg** respectively. When a value is stored from **FPAreg**, **FPBreg** is popped into **FPAreg** and **FPCreg** into **FPBreg**.

Manipulation of floating-point stack

The instructions listed in table 11.2 provide direct manipulation of the floating-point stack. They correspond to the integer instructions *rev* and *dup* (see section 7.2), but operate on the floating-point stack as opposed to the integer stack.

† The single and double precision formats supported are as specified in the IEEE 754 standard.

mnemonic	name
fprev	floating-point reverse
fpdup	floating-point duplicate

Table 11.2 Instructions which are used to directly manipulate the floating-point stack

fprev swaps the contents of **FPAreg** and **FPBreg**, not affecting **FPCreg**.

fpdup takes a copy of the content of **FPAreg** and pushes this into the floating-point stack, leaving two identical values in **FPAreg** and **FPBreg**, and the old value of **FPBreg** in **FPCreg**.

11.4 Loading and storing floating-point values

This section introduces the instructions that are used to store floating-point values in memory, or to load values into the floating-point stack. Both single precision and double precision floating-point values are considered. Single precision format floating-point values are represented in memory within a single machine word (32-bit). Double precision format floating-point values are represented in memory by two contiguous machine words. The word that contains the sign bit is held at the memory location with the higher address of the two.

Addresses for loading and storing floating point values are formed on the integer stack, and floating-point values are transferred between the addressed memory locations and the floating point stack. A 'floating-point pointer' is the address of the location(s) that holds a floating-point number. For a double precision floating-point number, this is a pointer to the memory location with the lower address of the two.

11.4.1 Loading

The instructions used to load floating-point numbers into the floating-point stack, are shown in table 11.3.

mnemonic	name
fpldnlsn	floating-point load non-local single
fpldnldb	floating-point load non-local double
fpldnlsni	floating-point load non-local indexed single
fpldnldbi	floating-point load non-local indexed double

Table 11.3 Floating-point load instructions

fpldnlsn or *fpldnldb* respectively load single or double precision floating-point values from memory into the floating-point stack. These instructions read the floating-point value from the location(s) specified by the floating-point pointer in **Areg**. The new floating-point value is pushed onto the floating-point register stack (see section 11.3). The integer stack is popped to remove the pointer in **Areg**.

These instructions signal *Unalign* if the address in **Areg** is not word-aligned. The double precision load instructions do *not* insist that the address is two-word-aligned.

For example to load the content of (**Wptr**+5) as a single precision value onto the floating-point stack the instruction sequence below is used.

ldlp 5; fpldnlsn

The state of the two stacks during this sequence is shown in figure 11.1.

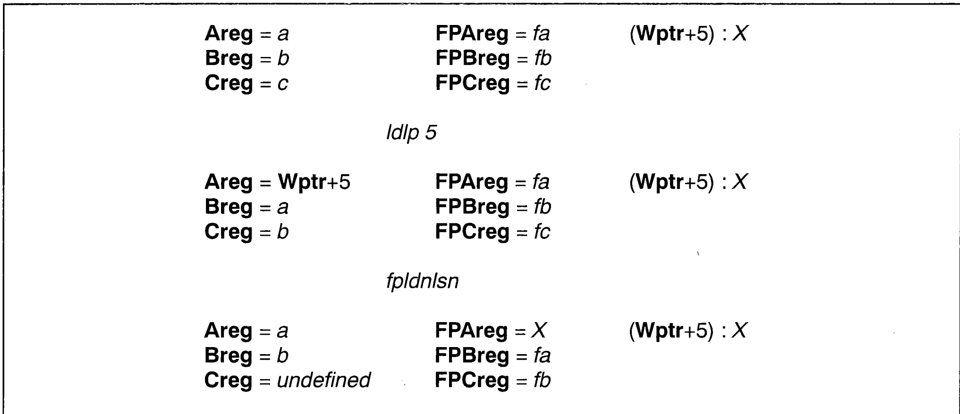


Figure 11.1 Stack use in floating-point load

To aid code compactness in loading from arrays, two indexed floating-point loads are provided.

fpldnlsni = *wsub; fpldnlsn*
fpldnldbi = *wsubdb; fpldnldb*

The following sequences can be used for loading from component *X[e]* of a floating-point array *X*, where *e* is an integer expression for the array index.

e; ldlp X; fpldnlsni — where *X* is a `REAL32` array

e; ldlp X; fpldnldbi — where *X* is a `REAL64` array

Loading constants

There is no special instruction for loading an arbitrary constant into the floating-point stack. A constant must be in memory, prior to loading into the stack. This is most conveniently achieved by forming a pointer into a table of constants.

Since 0.0 is a common constant, two instructions *fpldzerosn* and *fpldzerodb* (shown in table 11.4) are provided to load single and double precision 0.0.

mnemonic	name
fpldzerosn	load zero single
fpldzerodb	load zero double

Table 11.4 Instructions for loading floating-point zero

11.4.2 Storing

The instructions used to store floating-point numbers in memory, are shown in table 11.5.

mnemonic	name
fpstnlsn	floating-point store non-local single
fpstnldb	floating-point store non-local double

Table 11.5 Floating-point store instructions

fpstnlsn or *fpstnldb* respectively save single or double precision floating-point values into memory from the floating-point stack. These instructions store the floating-point value held in **FPAreg** into the location(s)

specified by the floating-point pointer in **Areg**. Both integer and floating-point stacks are popped to remove the data that has just been used.

These instructions signal *Unalign* if the address in **Areg** is not word-aligned. The double precision store instructions store onto any word boundary.

The compiler is expected to ensure that single precision data is stored with a *fpstnlsn* and double precision with a *fpstnldb*. The processor makes no check on the correctness of the precision, and the behavior of mismatched stores is undefined — the compiler should prevent this from happening.

The following instruction sequence stores the double precision value in **FPAreg** to the word address (**Wptr**+7) — N.B. the double word value is stored in (**Wptr**+7) and (**Wptr**+8).

ldlp 7; fpstnldb

The state of the two stacks during this sequence is shown in figure 11.2.

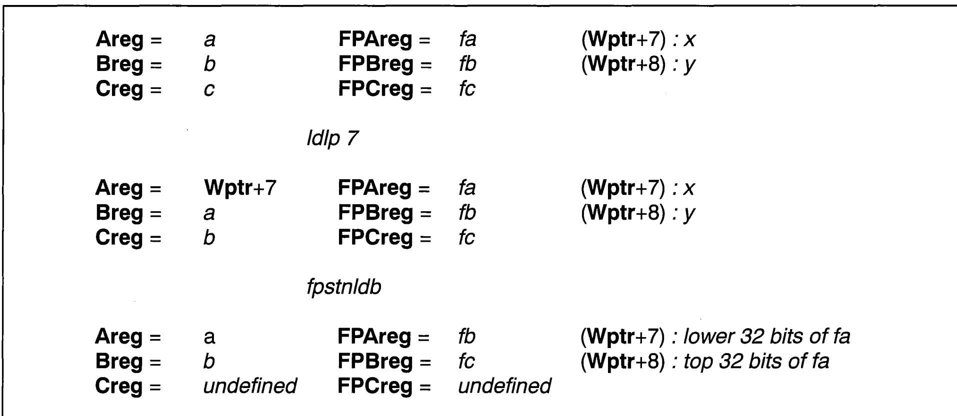


Figure 11.2 State of stacks in floating-point store

Storing to arrays is similar to loading from arrays except that there are no store indexed instructions. The following sequences can be used for storing to component *X[e]* of a floating-point array *X*.

e; ldlp X; wsub; fpstnlsn — where *X* is a `REAL32` array

e; ldlp X; wsubdb; fpstnldb — where *X* is a `REAL64` array

11.5 Compiling floating-point expressions

Compilation of expressions to be evaluated on the floating-point stack can be done in much the same way as for integer expressions.

A compiler loads a variable, *X*, by loading into the floating-point stack, the memory address allocated to that variable

address[2]*(X); fpldnlsn* — for single precision *X*
address(X); fpldnldb — for double precision *X*

A compiler best loads floating-point constants by arranging all constants in a table. The following code loads a constant *C* that is positioned at offset $Constant_C$ from the a table with base address *Constants*.

ldlp Constants; ldnlp $Constant_C$*; fpldnlsn — for single precision*
ldlp Constants; ldnlp $Constant_C$*; fpldnldb — for double precision*

N.B. As for the integer table of constants (described in section 7.3.2), this table and the floating-point constants within it must be word-aligned.

An expression *e1 op e2* is evaluated as shown in the algorithm in section 7.3, but the following sequences are replaced.

rev	becomes	*fprev*
stl temp	becomes	*ldlp temp; fpstnl*
ldl temp	becomes	*ldlp temp; fpldnl*

where *fpstnl* and *fpldnl* here stand for either the single or double precision load or store instruction depending on the type of the value being placed in a temporary variable.

When **FPAreg** and **FPBreg** need to be loaded with specific values — e.g. for a comparison — then code sequences similar to those given in the section on loading integer operands can be used (section 7.3.1).

11.6 Floating-point rounding mode

When an operation yields a floating-point result, this result is by default *rounded* to the nearest representable value to the exact result (*Round-to-nearest*). The IEEE standard does however provide three other *rounding modes*. The IMS T9000 provides instructions to set the mode for execution of the subsequent instruction.

mnemonic	name
fprn	set rounding mode to round nearest
fprz	set rounding mode to round zero
fprp	set rounding mode to round plus
fprm	set rounding mode to round minus

Table 11.6 Rounding mode setting instructions

The floating-point rounding mode is reset to *Round-to-nearest* at the end of all other floating-point instructions (except *fpldall*). To use any other rounding mode, one of the set rounding instructions listed in table 11.6 should be executed before the floating-point operation. These explicitly set the rounding mode. If there is no explicit selection of a rounding mode then the mode is *Round-to-nearest*.

Round-to-zero mode provides truncation, while the *Round-to-plus-infinity* and *Round-to-minus-infinity* modes have their uses in interval arithmetic and elsewhere.

11.7 Floating-point arithmetic instructions

Floating-point expression evaluation is performed using the floating-point stack. This section introduces instructions that are used for

- dyadic floating-point arithmetic – arithmetic instructions with two floating-point operands
- monadic floating-point arithmetic – arithmetic instructions with one floating-point operand

11.7.1 Dyadic operations

mnemonic	name
fpadd	floating-point add
fpsub	floating-point subtract
fpmul	floating-point multiply
fpdiv	floating-point divide
fprem	floating-point remainder

Table 11.7 Arithmetic instructions with two floating-point operands

The dyadic floating-point arithmetic instructions are listed in table 11.7. These instructions evaluate **FPBreg** *op* **FPAreg** leaving the result in **FPAreg**, and popping **FPCreg** into **FPBreg** like the integer arithmetic instructions. (Refer also to section 11.7.2 which describes some multiply and divide instructions that have one user specified operand and one implicit operand.)

The same instructions are used for single and double precision arithmetic instructions. The arithmetic instructions return the result as defined by the IEEE 754 standard. They assume both operands are of the same format — if not the result is undefined. For each instruction the destination (result) of the operation is the same format as its operands.

For a full discussion of *fprem* and the associated instruction *fprange*, refer to section 11.8.

Load and operate instructions

mnemonic	**name**
fpldnladdsn	floating-point load non-local and add single
fpldnladddb	floating-point load non-local and add double
fpldnlmulsn	floating-point load non-local and multiply single
fpldnlmuldb	floating-point load non-local and multiply double

Table 11.8 Floating-point load and operate instructions

To make the floating-point code more compact some common pairs of instructions can be replaced with a single instruction. The load and operate instructions shown in table 11.8 are equivalent to the instruction pairings shown below. These are the four instructions with the greatest effect on the size of code.

(fprx); *fpldnladdsn* = *fpldnlsn;* (fprx); *fpadd*
(fprx); *fpldnladddb* = *fpldnldb;* (fprx); *fpadd*
(fprx); *fpldnlmulsn* = *fpldnlsn;* (fprx); *fpmul*
(fprx); *fpldnlmuldb* = *fpldnldb;* (fprx); *fpmul*

where the optional instruction 'fprx' is one of the rounding mode instructions explained in section 11.6.

Prior to executing any of these instructions, the first operand has already been loaded onto the floating-point stack. Therefore the floating-point data being loaded must have the same precision as the data in **FPAreg**. Otherwise the operation is undefined. That is **FPAreg** must contain single precision data prior to execution of *fpldnladdsn* and *fpldnlmulsn*, and must contain double precision data prior to execution of *fpldnladddb* and *fpldnlmuldb*.

11.7.2 Monadic operations

mnemonic	**name**
fpmulby2	floating-point multiply by 2
fpdivby2	floating-point divide by 2
fpexpinc32	floating-point multiply by 2^{32}
fpexpdec32	floating-point divide by 2^{32}
fpabs	floating-point absolute value
fpsqrt	floating-point square root

Table 11.9 Arithmetic instructions with one floating-point operand

The monadic floating-point arithmetic instructions are listed in table 11.9. These instructions take the value of the operand from **FPAreg** and load the result into **FPAreg**, overwriting the operand value. The other floating-point stack registers are unaffected.

Multiplying and dividing by special values

Multiplication and division by 2.0 are common. Two instructions *fpmulby2* and *fpdivby2* perform these operations. These are considerably faster than loading 2.0 and doing an *fpmul* or *fpdiv* as they operate directly on the data in **FPAreg**.

Similarly multiplication and division by 2^{32} are provided by *fpexpinc32* and *fpexpdec32* mainly for use in the conversion routines.

Sign bit manipulation

fpabs replaces **FPAreg** with its absolute value. i.e. it makes the sign bit positive, except when the operand is a NaN. For a NaN, the sign bit is left unaltered.

Square root instruction

This section and section 11.8 use tables to document the effect of instructions in terms of signals and results for various operand values. The symbol 'NaN' in the 'operand conditions' column means either a quiet NaN *or* a signalling NaN. The meaning of 'finite_num' is any floating-point value that is not zero, ± infinity or NaN, while the meaning of *any* means literally any possible bit pattern. Where 'finite_num' or 0 are specified without a sign in the 'operand conditions' column, this means that the condition includes both positive and negative values. Where no sign is included in the 'result' column, the result is the same sign as the first (or only) operand in the 'operand conditions' column. Where there can be no meaningful result for the operation, the processor generates a quiet NaN. A full list of NaNs that the processor may produce is shown in table 11.29.

fpsqrt takes the a single floating-point operand and calculates the square root of this value. The result and signals raised are shown in table 11.10.

operand conditions SQRT **FPAreg**	**signals**	**result (square root)**
SQRT(NaN)	as detailed in section 11.14	as detailed in section 11.14
SQRT(−∞)	*FPInvalidOp, FPError*	*NegSqrtNaN*
SQRT(+∞)	*FPError*	+∞
SQRT(−0)	none	−0
SQRT(+0)	none	+0
SQRT(−finite_num)	*FPInvalidOp, FPError*	*NegSqrtNaN*
SQRT(+finite_num)	none†	$\text{SQRT}_{\text{IEEE}}$(+finite_num)
† *FPInexact* may be signalled		

Table 11.10 Signals raised and result of *fpsqrt* instruction

11.8 Remainder and range instructions

This section gives a detailed treatment of the dyadic operation *fprem* (already introduced in section 11.7.1) and the associated instruction *fprange* (table 11.11).

mnemonic	**name**
fprem	floating-point remainder
fprange	floating-point range reduce

Table 11.11 Floating-point remainder and range reduction instructions

fprem

The operation REM as as defined in the IEEE 754-1985†.

The instruction *fprem* calculates the remainder when evaluating the integer quotient of **FPBreg** divided by **FPAreg**. The result is loaded into **FPAreg**. The value of **FPCreg** is popped into **FPBreg** and **FPCreg** is left undefined by this instruction. The operands must either both be single precision or both be double precision floating-point numbers, otherwise the result is undefined. The integer stack is unaffected.

The instruction is interruptible. Floating-point exceptional conditions are signalled according to table 11.12. The result of REM is always exact.

operand conditions **FPBreg** REM **FPAreg**	**signals**	**result (remainder)**
NaN REM any any REM NaN	as detailed in section 11.14	as detailed in section 11.14
0 REM 0 finite_num REM 0	*FPInvalidOp, FPError*	*RemainderByZeroNaN*
$\pm\infty$ REM 0 $\pm\infty$ REM finite_num $\pm\infty$ REM $\pm\infty$	*FPInvalidOp, FPError*	*RemainderFromInfNaN*
± 0 REM $\pm\infty$	*FPError*	0
finite_num REM $\pm\infty$	*FPError*	finite_num
$finite_num_1$ REM $finite_num_2$ (if underflow occurs)	*FPUnderflow*†	not applicable†
$finite_num_1$ REM $finite_num_2$ (no underflow)	none	$finite_num_1$ REM_{IEEE} $finite_num_2$
† As the result cannot be 'inexact', REM cannot underflow unless a trap is taken. The condition is therefore only applicable if the 'underflow' trap is enabled. See also section 11.13.2.		

Table 11.12 Signals raised and result of *fprem* instruction

fprange

The instruction *fprange* is similar to *fprem*, but has two essential differences:–

- In addition to calculating the remainder, it produces the integer quotient result (in floating-point format).
- It has a limited range of operation. The integer quotient must be less than or equal to 2^{24}–1 for single precision arithmetic, or less or equal to than 2^{53}–1 for double precision.

fprange takes the top two floating-point operands on the floating-point stack and calculates the integer quotient and remainder when dividing the value in **FPBreg** by the value in **FPAreg**. The quotient is loaded into **FPBreg** and the remainder is loaded into **FPAreg**. Because the quotient is given by this instruction, it is suitable for use in range reduction (for example in trigonometric functions). The operands must be of the same precision, otherwise the result is undefined. **FPCreg** and the integer stack are unaffected by this instruction.

Note that whereas *fprem* is interruptible, this instruction is not. This is because the maximum execution time for *fprange* is much shorter than *fprem*. Floating-point exceptional conditions are signalled as shown in table 11.13.

† In summary, when $y \neq 0$, the remainder $r = x$ REM y is defined by

$r = x - y \times n$
where n is the nearest integer to the exact value x/y.

operand conditions **FPBreg** RANGE **FPAreg**	signals	result (remainder)	result (integer quotient)
NaN RANGE any any RANGE NaN	as detailed in section 11.14	as detailed in section 11.14	undefined
0 RANGE 0 finite_num RANGE 0	*FPInvalidOp, FPError*	*RemainderByZeroNaN*	undefined
$\pm\infty$ RANGE 0 $\pm\infty$ RANGE finite_num $\pm\infty$ RANGE $\pm\infty$	*FPInvalidOp, FPError*	*RemainderFromInfNaN*	undefined
–0 RANGE $-\infty$	*FPError*	–0	+0
–0 RANGE $+\infty$	*FPError*	–0	–0
+0 RANGE $-\infty$	*FPError*	+0	–0
+0 RANGE $+\infty$	*FPError*	+0	+0
–finite_num RANGE $-\infty$	*FPError*	–finite_num	+0
–finite_num RANGE $+\infty$	*FPError*	–finite_num	–0
+finite_num RANGE $-\infty$	*FPError*	+finite_num	–0
+finite_num RANGE $+\infty$	*FPError*	+finite_num	+0
$finite_num_1$ RANGE $finite_num_2$ (if underflow occurs)	*FPUnderflow*†	not applicable†	not applicable†
$finite_num_1$ RANGE $finite_num_2$ (integer quotient not in range of destination)	*FPInvalidOp, FPError*	*RangeQuotErrorNaN*	undefined
$finite_num_1$ RANGE $finite_num_2$ (no underflow and integer quotient within range of destination)	none	$finite_num_1$ REM_{IEEE} $finite_num_2$	$finite_num_1$ INT_DIV‡ $finite_num_2$

† As the results cannot be 'inexact', RANGE cannot underflow unless a trap is taken. The condition is therefore only applicable if the 'underflow' trap is enabled.

‡ floating-point representation of the nearest integer to the exact value of $finite_num_1$ /$finite_num_2$ – if the exact value is equidistant from two integers, then the even integer is chosen

Table 11.13 Signals raised and result of *fprange* instruction

As an example of the use of the *fprange* instruction, consider the following.

example – use of fprange *for range reduction*

An occam implementation of sine where *SINEPRIM* is a function that evaluates sines over [$-\pi$, π] could be

```
REAL32 FUNCTION SINE( VAL REAL32 X )
  VAL REAL32 Two.Pi IS 6.283185307 (REAL32):
  REAL32 Reduced.X:
  VALOF
    Reduced.X := X REM Two.Pi
    RESULT SINEPRIM( Reduced.X )
:
```

However in practice the value of 2π that would be used (*Two.Pi*) would not be exact. As the quotient (*X* REM *Two.Pi*) increased this error in 2π would be reflected in an increasingly large error in *Reduced.X* — i.e. the value used in the primary range calculation would become inaccurate. Suppose in this example that the value π_{REAL32} is used to derive *Two.Pi* where

$\pi_{REAL32} = \pi + \varepsilon$

then the reduced range of *X* evaluates to

$$
\begin{aligned}
Reduced.X &= X - m \times Two.Pi \\
&= X - m \times 2 \times \pi_{REAL32} \\
&= X - 2m \times (\pi + \varepsilon) \\
&= (X - 2m \times \pi) - 2m \times \varepsilon
\end{aligned}
$$

So the *Reduced.X* calculated consists of the true reduced argument plus the error term $-2m \times \varepsilon$. As *m* is $INT\left(\frac{X}{2\pi}\right)$ this error grows unacceptably large as *X* grows — for example at $X = 100\pi$ the error is 6 bits. To get around this problem an approximation to this error can be added back to the remainder by multiplying an approximation of ε by $INT\left(\frac{X}{2\pi}\right)\pi$. In effect this is using a value of π with twice as many significant bits as the format provides.

This error correction is needed in all the standard functions so support for it is useful. When calculating a remainder the quotient is also being developed so the *fprange* instruction returns the quotient in **FPBreg**. If *X* is very much larger than *Y* then $INT\left(\frac{X}{Y}\right)$ cannot be exactly represented in the floating-point format. Sufficient conditions for **FPBreg** to contain the quotient after the remainder are that (*X.exp* – *Y.exp*) is less than 23 for single precision and 52 for double precision values. If this is the case then a fast and accurate range reduction of *X* into $\left[-\frac{1}{2}Y,\ \frac{1}{2}Y\right]$ can be implemented by

```
address(X); fpldnlsn;             -- load X into floating-point stack
address(Y); fpldnlsn;             -- load Y into floating-point stack
fprange; fprev;                   -- execute 'range' operation and reverse
                                  -- order of 'quotient' and 'remainder' in stack
address(Y.error); fpldnlsn;       -- load Y.error (the known error in Y due to loss
                                  -- of precision) into floating-point stack
fpmul; fpadd;                     -- calculated (X REM Y) + Y.error × (X/Y)
address(Y); fpldnlsn; fprem       -- perform an extra REM Y (see below)
```

The final *fprem* is required because after adding the error term the result may possibly lie just outside the range $\left[-\frac{1}{2}Y,\ \frac{1}{2}Y\right]$. If *Y* has last bit accuracy then this can be corrected by taking the remainder by *Y*. In this sequence, since the proximity of the first operand can be guaranteed, *fprem* executes very quickly.

Note that both *fprem* and *fprange* can only signal *FPUnderflow* if the 'underflow' trap is enabled. This is because although they can produce a tiny non-zero result, it is not possible for either of these operations to lose precision. For the 'underflow' flag to be set, precision must have been lost. Whereas for an 'underflow' trap to be taken, it is sufficient for the result be tiny and non-zero.

11.9 Comparisons

11.9.1 Comparison instructions

The IMS T9000 provides the comparison instructions shown in table 11.14.

fpgt, *fpge*, *fpeq* and *fplg* perform (**FPBreg** *comp* **FPAreg**). *fpordered* tests if **FPAreg** and **FPBreg** can be 'ordered' in the IEEE sense.

mnemonic	name
fpeq	floating-point equality
fpgt	floating-point greater than
fpge	floating-point greater than or equals
fplg	floating-point less than or greater than
fpordered	floating-point orderability

Table 11.14 Floating-point comparison instructions

Each instruction implements a comparison of two operands. The left hand operand is taken from **FPBreg** and the right hand operand is taken from **FPAreg**. The two operands are popped from the floating-point stack in all the comparison instructions except *fpordered* for which the floating-point stack is unaffected by execution. The operands must be of the same precision, otherwise the result is undefined. Every comparison instruction returns a boolean result, which is pushed onto the integer stack.

The instructions: *fpeq*, *fpgt*, *fpge* and *fplg* implement the IEEE comparison operations '=', '>', '>=' and '<>' respectively. If one or both of the operands is a NaN, then the result is *false*. When neither operand is a NaN, the result of these instructions is as would be expected from the following predicate.

$$-\infty < -\text{finite_num} < -0 = +0 < +\text{finite_num} < +\infty$$

fpordered implements the IEEE 'NOT(?)' operation, where '?' is the IEEE 'unordered' relation. A relation is ordered provided that both operands are numeric floating-point values. *fpordered* therefore always evaluates to *true* provided neither operand is a NaN. This instruction can be used to filter out the case where the result of another comparison instruction is *false* because of a NaN operand. It does not pop the floating-point stack.

Table 11.15 summarizes the results of these instructions. The relationship between any two floating-point numbers is one and only one of the four relations shown in the table: 'greater than', 'less than', 'equal' or 'unordered'. That is, these relations are mutually exclusive. The table shows the result returned by each comparison instruction for each of these relations. For example if the number in **FPBreg** is 'less than' the number in **FPAreg**, then *fpeq*, *fpgt* and *fpge* return *false* in **Areg**, whereas *fplg* and *fpordered* return *true*.

instruction	IEEE 'ad hoc'	relations			
		greater than	less than	equal	unordered
fpeq	=	*false*	*false*	*true*	*false*
fpgt	>	*true*	*false*	*false*	*false*
fpge	>=	*true*	*false*	*true*	*false*
fplg	<>	*true*	*true*	*false*	*false*
fpordered	NOT(?)	*true*	*true*	*true*	*false*

Table 11.15 Results of comparison instructions for all possible relations

The conditions for which these instructions raise signals are shown in table 11.16.

instruction	signal raised if either operand is a signalling NaN	signal raised if either operand is a quiet NaN but neither operand is a signalling NaN	signal raised if either operand is an infinity but neither operand is a NaN
fpeq	*FPInvalidOp, FPError*	*FPError*	*FPError*
fpgt	*FPInvalidOp, FPError*	*FPInvalidOp, FPError*	*FPError*
fpge	*FPInvalidOp, FPError*	*FPInvalidOp, FPError*	*FPError*
fplg	*FPInvalidOp, FPError*	*FPInvalidOp, FPError*	*FPError*
fpordered	*FPInvalidOp, FPError*	*FPError*	*FPError*

Table 11.16 Signals raised by comparison instructions, for various operand conditions

11.9.2 Implementation of IEEE comparisons

The five instructions described in section 11.9.1, are provided as primitives for building comparison operations. The comparisons given in the IEEE standard can be constructed from these instructions. This is

shown in table 11.17. For the code sequences in this table, the following shorthand is used for loading a floating-point number into the floating-point register.

fld(X, Y)   <u>is defined as</u>   'load *X* into **FPBreg** and load *Y* into **FPAreg** using suitable instructions as described elsewhere'

IEEE *ad hoc* comparison	>	<	=	?	***FPInvalidOp***	**code sequence**
A > B	T	F	F	F	yes	*fld(A, B); fpgt*
NOT(A > B)	F	T	T	T	yes	*fld(A, B); fpgt; eqc 0*
A >= B	T	F	T	F	yes	*fld(A, B); fpge*
NOT(A >= B)	F	T	F	T	yes	*fld(A, B); fpge; eqc 0*
A < B	F	T	F	F	yes	*fld(B, A); fpgt*
NOT(A < B)	T	F	T	T	yes	*fld(B, A); fpgt; eqc 0*
A <= B	F	T	T	F	yes	*fld(B, A); fpge*
NOT(A <= B)	T	F	F	T	yes	*fld(B, A); fpge; eqc 0*
A <> B	T	T	F	F	yes	*fld(A, B); fplg*
NOT(A <> B)	F	F	T	T	yes	*fld(A, B); fplg; eqc 0*
A <=> B	T	T	T	F	yes	*fld(A, B); fpordered; fpgt; or*‡
NOT(A <=> B)	F	F	F	T	yes	*fld(A, B); fpordered; fpgt; or; eqc 0*‡
A ? B	F	F	F	T	no	*fld(A, B); fpordered; eqc 0*†
NOT(A ? B)	T	T	T	F	no	*fld(A, B); fpordered;*†
A ?<> B	T	T	F	T	no	*fld(A, B); fpeq; eqc 0*
A = B	F	F	T	F	no	*fld(A, B); fpeq*
A ?> B	T	F	F	T	no	*fld(B, A); fpordered; cj end; fpge;* *end:* *eqc 0*†
NOT(A ?> B)	F	T	T	F	no	*fld(B, A); fpordered; cj end; fpge;* *end:* *<next_instr>*†
A ?>= B	T	F	T	T	no	*fld(B, A); fpordered; cj end; fpgt;* *end:* *eqc 0*†
NOT(A ?>= B)	F	T	F	F	no	*fld(B, A); fpordered; cj end; fpgt;* *end:* *<next_instr>*†
A ?< B	F	T	F	T	no	*fld(A, B); fpordered; cj end; fpge;* *end:* *eqc 0*†
NOT(A ?< B)	T	F	T	F	no	*fld(A, B); fpordered; cj end; fpge;* *end:* *<next_instr>*†
A ?<= B	F	T	T	T	no	*fld(A, B); fpordered; cj end; fpgt;* *end:* *eqc 0*†
NOT(A ?<= B)	T	F	F	F	no	*fld(A, B); fpordered; cj end; fpgt;* *end:* *<next_instr>*†
A ?= B	F	F	T	T	no	*fld(A, B); fpordered; eqc 0; fpeq; or*
NOT(A ?= B)	T	T	F	F	no	*fld(A, B); fpordered; fpeq; eqc 0; and*

† leaves two values on the floating-point stack

‡ *fpgt* forces *FPInvalidOp* to be signalled for unordered

† leaves two values on the floating-point stack only if the predicate is 'unordered'

Table 11.17 Code sequences that should be used to implement IEEE comparisons

N.B. Remember (section 11.9.1) that either *X* and *Y* must both be single precision, or they must both be double precision. Otherwise the behavior of the comparison instructions is undefined.

11.9.3 Some anomalies

It is important when using the comparison instructions, that the user fully understands which instructions to use for the correct treatment of NaNs.

If all NaNs are to be treated as errors, then it does not matter what result is returned from an 'unordered' relation – e.g.

(A > B)	is equivalent to	NOT(B >= A)	*only* if not considering
(A >= B)	is equivalent to	NOT(B > A)	result for a NaN operand
(A = B)	is equivalent to	NOT(A <> B)	

In this case the code can use either *fpgt* or *fpge* (with or without a logical inversion as appropriate) for any of the comparisons: '>', '<=', '<' and '>='; and can use either *fpeq* or *fplg* (with or without a logical inversion as appropriate) for the comparisons: '=' and '<>'.

However, often the above rules do not apply, since if either operand is a NaN, the result should be *false* for the comparisons: '=', '<>', '>', '<=', '<' and '>=' – e.g.

(A > B)	is **not** equivalent to	NOT(B >= A)	in the general case where
(A >= B)	is **not** equivalent to	NOT(B > A)	NaN operands are considered
(A = B)	is **not** equivalent to	NOT(A <> B)	

For this reason it is essential to make the correct choice of instructions for these comparisons – i.e. as shown in table 11.17.

Furthermore when implementing the IEEE comparisons that test for 'unordered' (those that have '?' in table 11.17), *FPInvalidOp* should *not* be signalled (unless one of the operands is a signalling NaN). The instructions *fpgt*, *fpge* and *fplg* signal *FPInvalidOp* when one of the operands is a quiet NaN, and so execution of these instructions should be avoided for these comparisons if the relation is unordered – e.g.

(A > B)	is **not** equivalent to	NOT(B ?>= A)
(A >= B)	is **not** equivalent to	NOT(B ?> A)

If in any doubt, use the sequences recommended in table 11.17.

11.10 Class analysis

mnemonic	name
fpnan	floating-point NaN
fpnotfinite	floating-point not finite

Table 11.18 Class analysis instructions

The instructions shown in table 11.18 are provided to allow a rudimentary check to be made on the class of the value held in **FPAreg**. They push a boolean value into **Areg** and do not affect the floating-point stack.

fpnan tests to see if **FPAreg** is a Not-a-Number and *fpnotfinite* tests to see if **FPAreg** is not finite — i.e. is a Not-a-Number or an infinity.

11.11 Type conversion

The transputer has facilities to enable conversions between the integer types (represented by single word and double word formats) and the floating-point types (represented by single precision and double preci-

sion formats). Several instructions are provided to perform the component parts of the various conversions. Each conversion can be constructed by using a suitable sequence of these components.

11.11.1 REAL to REAL conversions

mnemonic	name
fpr32tor64	floating-point REAL32 to REAL64
fpr64tor32	floating-point REAL64 to REAL32

Table 11.19 Real to real conversion instructions

The two instructions *fpr32tor64* and *fpr64tor32* (table 11.19) convert the floating-point value in **FPAreg** from one floating-point format to the other. *fpr32tor64* is an exact conversion involving no rounding. *fpr64tor32* rounds during the conversion so a set rounding mode instruction must precede it if a rounding mode other than *Round-to-nearest* is required.

The behavior of these instructions is undefined if **FPAreg** does not initially hold the correct precision floating-point data – i.e. for *fpr32tor64* it must hold a single precision value, and for *fpr64tor32* it must hold a double precision value.

An infinity is represented in the IEEE standard by a floating-point number with maximum exponent but zero fraction. A Not-a-Number has a maximum exponent but a non-zero fraction. When converting from one format to the other, infinities are preserved. When a Not-a-Number is converted from single precision (REAL32) to double precision (REAL64), it is converted to a quiet NaN (if it isn't already quiet). When a double precision NaN is converted to single precision, it always becomes *R64ToR32NaN*, which is itself a quiet NaN. (See section 11.14 for full details).

11.11.2 REAL to INT conversions

mnemonic	name
fpint	round to floating integer
fpchki32	floating-point check in range of INT32
fpchki64	floating-point check INT64
fprtoi32	REAL to INT32
fpstnli32	floating-point store non-local INT32

Table 11.20 Real to integer conversion instructions

fpint converts a floating-point number to an integer value in the same floating-point format. This is the *'Round Floating-Point Number to Integer Value'* function specified by the IEEE standard. It takes the value in **FPAreg** and rounds it, according to the current rounding mode, to an integer value. If a rounding mode other than *Round-to-nearest* is required for the conversion then this instruction should be preceded immediately by the mode selection instruction. For example if **FPAreg** contained *345.678* then after

fprz; fpint

FPAreg would contain *345.0*.

fpchki32 and *fpchki64* check that the floating-point value in **FPAreg** (regardless of precision) lies in the range of the relevant integer type. If the value lies outside the range then *FPInvalidOp* and *FPError* are signalled.

To aid code compactness the most common floating-point to single word integer case

fpint; fpchki32

can be replaced with the single instruction

fprtoi32

fpstnli32 firstly converts the floating-point number in **FPAreg** to an integer value, rounding towards $-\infty$. It then converts the number to a 64-bit twos-complement integer, and stores the least significant 32-bits of this integer in the location pointed to by **Areg**.

The behavior of this instruction is only defined if **FPAreg** contains a floating-point number, the integer part of which lies within the range of a 64-bit integer. Note that this excludes infinities and NaNs.

Code sequences for the conversions to integer from floating-point are shown below. Note that rounding to an integer must be performed by *fpint* (or *fprtoi32*) as described above prior to applying the rest of the conversion sequence.

- floating-point variable (*fp*) to single word integer, *Round-to-nearest* mode, error checked, storing the result in *X*:

  address²(fp); fpldnl†; fprtoi32; address(X); fpstnli32

- floating-point variable (*fp*) to double word integer, truncated (*Round-to-zero* mode), unchecked, storing the result in *Y*:

  address(fp); fpldnl; fprz; fpint; fpdup; address(Y); dup; fpstnli32; ldnlp 1; fpexpdec32; fpstnli32

  In this sequence, the floating-point number is duplicated using *fpdup*. The first copy of the number is used to convert the less significant part of the number. The second copy of the number divided by 2^{32} using *fpexpdec32* prior to conversion in order to yield the more significant part of the number.

11.11.3 INT to REAL conversions

mnemonic	name
fpi32tor64	INT32 to REAL64
fpi32tor32	INT32 to REAL32
fpb32tor64	BIT32 to REAL64
fpadddbsn	floating-point add double producing single

Table 11.21 Integer to real conversion instructions

The various integer to real type conversions can be provided by code sequences using the instructions listed in table 11.21. The following describes each instruction and provides suggested sequences for these conversions. Each sequence should be followed by code to store the converted value into its destination if necessary. Where required the rounding mode for the conversion can be set.

Single word integer to double precision floating-point

fpi32tor64 takes the single word integer value from the address contained in **Areg** and converts this to a double precision floating-point number in **FPAreg**. This is an exact conversion (i.e. no rounding is required).

For example a single word integer (*int32*) is converted to a double precision floating-point number by

address²(int32); fpi32tor64

Single word integer to single precision floating-point

The sequence

† *fpldnl* is defined on page 131

*fpi32tor64; (*fprx*); fpr64tor32*

where the optional instruction 'fprx' is one of the rounding mode instructions explained in section 11.6, converts a single word integer to a single precision floating-point number. *But* this can be replaced by the shorter equivalent sequence

*(*fprx*); fpi32tor32*

fpi32tor32 takes the single word integer value from the address contained in **Areg** and converts this (rounding if necessary) to a single precision floating-point number in **FPAreg**. This is preceded by a round mode selection instruction if a rounding mode other than *Round-to-nearest* is required.

For example a single word integer (*int32*) is converted to a single precision floating-point number in *Round-to-zero* mode by the sequence

address[2](int32); fprz; fpi32tor32

Double word integer to double precision floating-point

A conversion from double word integer to double precision floating-point is performed by converting each word of the double word integer to double precision and adding them together. The more significant word is converted to floating-point by fpi32tor64 and the less significant word by fpb32tor64.

fpb32tor64 takes the unsigned 32 bit value from the address contained in **Areg** and converts this to a double precision floating-point number of the same value in **FPAreg**. This is an exact conversion.

Hence the following instruction sequence converts from double word integer (*int64*) to double precision floating-point using *Round-to-nearest* mode.

address[2](int64); dup; fpb32tor64; ldnlp 1; fpi32tor64; fpexpinc32; fpadd

Here, firstly the less significant word of the double word integer is converted by *fpb32tor64* as an *unsigned* integer to double precision floating-point format. Secondly the more significant word is converted by *fpi32tor64* as a *signed* integer, and then multiplied by 2^{32}. The two results are summed to give the double precision representation. This last action may cause the result to be rounded.

Double word integer to single precision floating-point

fpadddbsn adds two double precision floating-point numbers in **FPAreg** and **FPBreg** to produce a correctly rounded single precision value in **FPAreg**. **FPCreg** is popped into **FPBreg** leaving **FPCreg** undefined. This should be preceded by a round mode selection instruction if a rounding mode other than *Round-to-nearest* is required.

Both operands must be double precision, otherwise the result is undefined. *FPInvalidOp* is caused by signalling NaNs and adding infinities of opposite sign. If either operand is a NaN then the result is *R64ToR32NaN*.

For example, using the following sequence, a double word integer (*int64*) is converted to a single precision floating-point number in *Round-to-minus-infinity* mode.

address[2](int64); dup; fpb32tor64; ldnlp 1; fpi32tor64; fpexpinc32; fprm; fpadddbsn

The user may instead be tempted here to use the same conversion sequence as that described for conversion from double integer to double floating-point, followed by an *fpr64tor32* instruction. This would not be accurate since rounding would occur after both *fpadd* and *fpr64tor32*; the *fpadddbsn* removes this double rounding.

In the conversions from double word integer, the round mode selection takes place immediately before the results from converting the two halves of the double word integer are added together as the sub-conversions from integer to floating-point are exact.

11.12 Floating-point state

The floating-point state comprises the content of the floating-point stack, and the floating-point status word. The latter, which contains rounding mode and precision information, is discussed below, as are the effects of descheduling, interrupting, and trapping on the floating-point state, and the instructions that can be used to store and load this state.

11.12.1 Floating-point status word

The currently executing process has a floating-point status word associated with it. This is stored within the floating-point status register (**FPstatusReg**). Its format is shown in table 11.22. This shows that four pieces of information are stored in this word: the floating-point rounding mode, and the type of floating-point value stored in each floating-point stack register.

field (bit numbers)	**meaning**
0–1	rounding mode
2–3	type of floating-point value in **FPAreg**
4–5	type of floating-point value in **FPBreg**
6–7	type of floating-point value in **FPCreg**
8 to 31	reserved – read and write as zeros

Table 11.22 The format of the floating-point status word

The rounding mode may be one of the four modes specified by IEEE 754. This is encoded into a two bit field and the binary value for each mode is shown in table 11.23. Although the rounding mode is reset after execution of every instruction (section 11.6), the mode needs to be stored as part of the shadow state when an instruction is interrupted.

value	meaning
0	IEEE round zero
1	IEEE nearest
2	IEEE round + infinity
3	IEEE round – infinity

Table 11.23 The floating-point rounding mode field values

The format of the floating-point value stored in any of the floating-point stack registers, can be single precision or double precision. This information is represented as shown in table 11.24.

value	meaning
0	single precision – IEEE format 32-bit floating-point number
1	double precision – IEEE format 64-bit floating-point number
2	reserved
3	reserved

Table 11.24 The floating-point type field values

11.12.2 Saving the floating-point state

Timeslicing and descheduling

In the same manner that information must not be left on the integer stack when a process may be descheduled, care must be taken with the floating-point stack.

The floating-point registers are not saved when a process is descheduled. This is the same as for the integer registers. To take account of this a compiler must ensure that at all descheduling points, there is no information being stored on the floating-point stack. Any data that is needed later must be stored in temporary variables.

When a process is scheduled it can make no assumptions about the contents of the floating-point registers. If floating-point arithmetic is to be used then data needs to be loaded into the floating-point registers thus setting the round mode to *Round-to-nearest*. Hence the value of the rounding mode when a process is scheduled, and by implication when a process is descheduled, is irrelevant.

Interrupts

When a high priority process interrupts a low priority process the floating-point state is copied into shadow registers and retrieved when control is returned to the low priority process. The conditions required to ensure correct behavior of low priority processes are sufficient to ensure correct behavior of high priority processes.

Traps

When a process traps, the trap-handler or supervisor can store the floating-point state by executing *fpstall*. Conversely when returning from the trap, the state can be reloaded with *fpldall*. These instructions are explained in section 11.12.3, and an explanation of how they would normally be used is given later in section 13.2.3.

11.12.3 Instructions for saving and loading floating-point state

mnemonic	name
fpstall	floating-point store all
fpldall	floating-point load all

Table 11.25 Floating-point state instructions

fpstall

The instruction *fpstall* stores the floating-point register values into the data structure addressed by the pointer in **Areg**. The floating-point stack is undefined, and the integer stack is popped leaving **Creg** undefined.

The data structure pointed to by **Areg** contains seven words of data. The instruction writes the floating-point status word (see section 11.12.1) into the single word **fp.FPstatusReg** slot of the data structure. It then writes the current value of each floating-point stack register into the other three slots. Each of these slots is two words wide so that double precision floating-point values can be stored. Where a floating-point stack register holds single precision data, this is loaded into its two word slot at the location with the lower address. This is summarized in table 11.26.

fpldall

The instruction *fpldall* loads the floating-point register values from the data structure addressed by the pointer in **Areg**. The integer stack is popped leaving **Creg** undefined.

The data structure pointed by **Areg** contains seven words of data. The instruction loads the floating-point status word (see section 11.12.1) from the single word **fp.FPstatusReg** slot of the data structure. It then restores each floating-point stack register from the other three slots. This is summarized in table 11.26. Note that an effect of this instruction is that the rounding mode may change.

word offset	slot name	purpose
5	**fp.FPCreg**	loaded into / stored from floating-point stack register C by *fpldall* / *fpstall*
3	**fp.FPBreg**	loaded into / stored from floating-point stack register B by *fpldall* / *fpstall*
1	**fp.FPAreg**	loaded into / stored from floating-point stack register A by *fpldall* / *fpstall*
0	**fp.FPstatusReg**	loaded into / stored from floating-point status register by *fpldall* / *fpstall*

Table 11.26 Floating-point state data structure

11.13 Exception handling mechanism

This section discusses the IMS T9000 exception handling mechanism and explains how to implement an IEEE trap handler. The term 'T9 exception handler' is used here to mean trap-handler or supervisor depending on whether a trap has been taken from an L-process or P-process. The term 'IEEE (trap) handler' refers the the 'trap-handler' specified in the standard (section 8).

The IMS T9000 has a flag and a trap enable bit associated with each of the floating-point exceptions. When the IMS T9000 detects an exceptional condition, it raises one of the signals shown in table 11.27. These exceptional conditions represent a subset of the error conditions that are detected by the IMS T9000. (A more general treatment of signalling and handling of errors is given in section 10.3.) Table 11.27 states for each signal, which flag is set, or under what circumstances a trap is taken.

signal	extra condition for trap to be taken	flags set if trap not taken
FPError	**sb.FPErrorTeBit**	**sb.FPErrorFlag**
FPInvalidOp	**sb.FPInOpTeBit**	**sb.FPInOpFlag**
FPDivideByZero	**sb.FPDivByZeroTeBit**	**sb.FPDivByZeroFlag**
FPOverflow	**sb.FPOvTeBit**	**sb.FPOvFlag**
FPUnderflow†	**sb.FPUndTeBit**	**sb.FPUndFlag**
FPInexact	**sb.FPInexTeBit**	**sb.FPInexFlag**
† To comply with the IEEE standard, *FPUnderflow* is signalled on detection of 'tininess' if trapping is enabled, but otherwise on detection of 'tininess' *and* 'inexact'.		

Table 11.27 Effect of signals raised due to detection of exceptional conditions

The IMS T9000 provides a general floating-point exception ('floating-point error') as an alternative (simpler) method of handling exceptions. If enabled, this exception occurs when *FPError* is signalled. The conditions that cause this signal are a superset of the conditions which cause *FPInvalidOp*, *FPDivideByZero* and *FPOverflow* signals. *FPError* is also signalled when a NaN (either signalling or quiet) or an infinity is used as an operand of a floating-point operation. This is of interest to the user who does not want to analyze the type of exception, but is just interested in detecting any extraordinary floating-point conditions. For example many high-level languages treat these conditions as errors. However note that *FPUnderflow* and *FPInexact* do not cause 'floating-point error', so if these conditions need to be detected, then this must be done explicitly (by enabling the respective traps or examining the flags).

[The 'floating-point error' exception is extra to the IEEE specification, and provides compatibility with the IMS T805 floating-point error flag. The latter treats *FPInvalidOp*, *FPDivideByZero*, *FPOverflow* and the use of any NaN or infinity as an operand to a floating-point operation as a floating-point error. The IMS T9000 retains compatibility with the IMS T805 by providing a flag and a trap enable bit associated with the T805 error conditions. These are the **sb.FPErrorFlag** and the **sb.FPErrorTeBit** which are used to implement 'floating-point error'.]

Exceptions are either represented by the setting of a flag or by the occurrence of a trap. Unless a trap is taken, the flag reserved for that exception is set (table 11.27). If the trap is enabled, a value is presented to the T9 exception handler (in **Breg**) to indicate the exception that has occurred (table 11.28), but no flags are set.

error type value (hex)	**error type symbol**	**condition for trap to occur**	**error meaning**
7 (#07)	**et.FPError**	*FPError* AND **sb.FPErrorTeBit**	'floating-point error'†
8 (#08)	**et.FPInvalidOp**	*FPInvalidOp* AND **sb.FPInOpTeBit** AND (NOT *'conditions for **et.FPError**'*)	IEEE 'floating-point invalid operation'
9 (#09)	**et.FPDivideByZero**	*FPDivideByZero* AND **sb.FPDivByZeroTeBit** AND (NOT *'conditions for **et.FPError**'*)	IEEE 'floating-point divide by zero'
10 (#0A)	**et.FPOverflow**	*FPOverflow* AND **sb.FPOvTeBit** AND (NOT *'conditions for **et.FPError**'*)	IEEE 'floating-point overflow'
11 (#0B)	**et.FPUnderflow**	*FPUnderflow* AND **sb.FPUndTeBit** AND (NOT *'conditions for **et.FPError**'*)	IEEE 'floating-point underflow'
12 (#0C)	**et.FPInexact**	*FPInexact* AND **sb.FPInexTeBit** AND NOT *('conditions for **et.FPError**' OR 'conditions for **et.FPOverflow**' OR 'conditions for **et.FPUnderflow**')*	IEEE 'floating-point inexact result'
† This is not an IEEE exception. See above.			

Table 11.28 Exceptional conditions and error types

When a floating-point instruction is executed, it is possible for more than one signal to be raised. More precisely: an *FPError* can coincide with *FPInvalidOp*, *FPDivideByZero*, *FPOverflow*, or *FPInexact*; an *FPOverflow* can coincide with *FPInexact*; and an *FPUnderflow* can coincide with an *FPInexact*.

However, only one error type value (first column of table 11.28) is provided to the T9 exception handler and this value is supplied according to these rules: where there is any conflict, the 'floating-point error' takes precedence, followed by 'floating-point overflow' or 'underflow' (these are mutually exclusive), followed by 'inexact'. The precedence of the IEEE exceptions is that set out in the IEEE 754 standard. (N.B These precedence rules are incorporated into the column labelled 'conditions for trap to occur' in table 11.28.) In contrast, when traps have not been enabled, the signalling of multiple floating-point exceptional conditions gives rise to the setting of all the corresponding flags.

For example, consider what happens if say both *FPOverflow* and *FPInexact* are signalled. While **sb.FPErrorTeBit** is set, it is **et.FPError** which is presented to the T9 exception handler (loaded into **Breg**). If on the other hand, this bit is not set, but both **sb.FPOvTeBit** and **sb.FPInexTeBit** are set, then **et.FPOverflow** is presented since this has the higher priority. If only **sb.FPInexTeBit** is set, **et.FPInexact** is presented. In all these cases, no flags are set as a result of these error signals because a trap is taken. Finally if no trap enable bits are set, then a flag is set for each error according to table 11.27.

Thus if any of the trap enable bits are set such that a single exceptional condition causes a trap to be taken, then the number for that exception is presented to the T9 exception handler. If more than one trap enabled exceptional condition is signalled, then only one exception is presented to the T9 exception handler according to the above precedence rules. If a trap is not taken, a flag is set for each exception.

11.13.1 State delivered by floating-point exception – Implementing an IEEE (trap) handler

Where a trap occurs as a result of a floating-point exceptional condition, the floating-point state is restored to the state that was present *before* the operation was performed† (but see section 11.13.2). This state can be stored in the appropriate data structure using *fpstall* (section 11.12.3).

It is therefore straightforward for the T9 exception handler to invoke an IEEE handler, passing via parameters the information that the standard requires to be delivered to an IEEE handler. The floating-point state can be used to calculate this information as described below. Where it is necessary to re-execute a trapped instruction, the instruction opcode can be obtained from the location pointed to by the address in the **th.Eptr** (or **ps.Eptr**) slot, and the initial state can be reloaded with *fpldall*.

- All exceptional conditions signalled by the instruction can be ascertained by examining the flags (using *stflags*) having re-executed the instruction with traps disabled.
- The kind of operation performed can be ascertained by examining the instruction's opcode.
- The destination's format can be determined by examining the floating-point status word and the instruction's opcode.
- The operand values are part of the restored floating-point state.
- For 'inexact', the result to be delivered to the IEEE handler (correctly rounded) can be obtained by re-executing the instruction with traps disabled.
- For 'overflow' and 'underflow', the result to be delivered to the IEEE handler can be computed from the original values.

Finally when the IEEE handler has fully analyzed the trap cause, the T9 exception handler might for example be required to present its own result for the instruction. It may do this as follows:–

- by loading the floating-point stack and floating-point status register with the appropriate data (using *fpldall*)
- by setting the appropriate flags in the status register (using *ldflags* and *stflags*) to indicate which exceptional conditions have been detected
- by restarting the code from the correct point by executing *tret* (or *goprot*).

11.13.2 Some anomalies

Compound instructions

Care must be taken with the compound instructions – namely: *fpldnladdsn*, *fpldnladddb*, *fpldnlmulsn*, *fpldnlmuldb*, *fpi32tor32* and *fprtoi32*. Either stage of these two-part instructions can cause a trap. The processor behaves in the same way that it would if both instructions (that the compound instruction replaces) were executed separately.

For the load and operate instructions (*fpldnladdsn*, *fpldnladddb*, *fpldnlmulsn* and *fpldnlmuldb*), floating-point exceptional conditions can only be signalled by the operate part of the instruction (i.e. add or multiply). When a trap is taken due to such a signal, the state delivered is the state prior to the *operate* part. This is similar for *fpi32tor32* (which is equivalent to *fpi32tor64* followed by *fpr64tor32*) since *fpi32tor64* cannot cause a floating-point trap.

fprtoi32 (which comprises *fpint* and *fpchki32*) can signal floating-point exceptional conditions in either operation. The state delivered is the floating-point state before the trap causing part of the instruction was executed. Similarly, the trap behavior of *fpadddbsn* is the same as the behavior of the sequential execution of *fpadd* and *fpr64tor32*, even though the function is slightly different.

fprem

The possible exceptions for *fprem* are 'underflow' and 'invalid operation'. The only time that the initial operands are not restored is when *fprem* has caused an 'underflow' trap. In this case, the initial value is restored to **FPAreg** (the divisor), but the value restored to **FPBreg** may be less than the initial value (the dividend) by a multiple of the value in **FPAreg**. For 'underflow' (as well as 'overflow', and 'inexact') traps, the IEEE

† This implies that the result of a floating-point operation is only delivered if a trap is *not* taken.

standard requires that the correctly rounded result is presented to the IEEE handler. This can still be achieved for *fprem* by re-executing the instruction with traps disabled. For 'invalid operation' (as well as 'divide by zero'), the standard requires that the initial operands are presented to the IEEE handler. This can be achieved for *fprem* (as for all instructions) because the initial values are restored when an 'invalid operation' trap is taken.

11.14 Implementation of NaNs

Section 11.2.3 explains that if a trap is not taken when a 'invalid operation' occurs, then any floating-point result should be a NaN. Since this NaN is the result of an operation, it must also be a quiet NaN (in accordance with the IEEE standard). Under these circumstances, the IMS T9000 therefore generates one of the values listed in table 11.29. These are the only NaNs generated by the IMS T9000 unless one of the operands is a NaN (discussed later).

error	constant name	single precision value (quiet NaN)	double precision value (quiet NaN)
divide zero by zero	*DivZeroByZeroNaN*	#7FC00000	#7FF80000 00000000
divide infinity by infinity	*DivInfByInfNaN*	#7FE00000	#7FFC0000 00000000
multiply zero by infinity	*ZeroMulInfNaN*	#7FD0000	#7FFA0000 00000000
addition of opposite infinities or subtraction of like infinities	*AddOpInfsNaN*	#7FC80000	#7FF90000 00000000
negative square root (non-zero)	*NegSqrtNaN*	#7FC40000	#7FF88000 00000000
REAL64 (double precision) to REAL32 (single precision) NaN conversion	*R64ToR32NaN*	#7FC20000	(not applicable)
remainder from infinity	*RemainderFromInfNaN*	#7FC04000	#7FF80800 00000000
remainder by zero	*RemainderByZeroNaN*	#7FC02000	#7FF80400 00000000
quotient out of range in result of *fprange*	*RangeQuotErrorNaN*	#7FC01000	#7FF80200 00000000

Table 11.29 Quiet NaNs generated when *FPInvalidOp* is signalled but no trap is taken

When a NaN is changed from single precision to double precision (*fpr32tor64*), the sign bit is set to 0, the exponent field is extended from eight '1's to eleven '1's (to maintain the representation as a NaN), and the 'value' of the fraction field is maintained by setting the 29 least significant bits to 0. The most significant bit of the fraction field is set to 1 to ensure that it is a *quiet* NaN. The same meaning (e.g. divide zero by zero) is thus preserved.

When a NaN (either a signalling or quiet) is used as an operand to a floating-point operation, then *FPError* is signalled. (This causes 'floating-point error' – if enabled – which provides compatibility with the IMS T805 transputer. It is not an IEEE exceptional condition.) Also, if a signalling NaN is used as an operand to a floating-point operation, then *FPInvalidOp* is signalled.

Tables 11.30 and 11.31 show the signals raised and the values returned (unless a trap is taken) when NaNs are present as the inputs to floating-point operations†. Q(*value*) means that the signalling NaN *value* is converted to a quiet NaN (by setting the most significant bit of its fraction). The word 'number' means a floating-point representation that is not a NaN.

† This does not apply to *fpadddbsn*, which returns the quiet NaN *R64ToR32NaN* if either of its operands is a NaN.

input	signals	result (unless trap taken)
signalling NaN	*FPInvalidOp, FPError*	Q(input)
quiet NaN	*FPError*	input

Table 11.30 Behavior of monadic floating-point operations for NaN input

inputs		signals	result
A	B		(unless trap taken)
signalling NaN	signalling NaN	*FPInvalidOp, FPError*	Q(A)
signalling NaN	quiet NaN	*FPInvalidOp, FPError*	Q(A)
quiet NaN	signalling NaN	*FPInvalidOp, FPError*	Q(B)
quiet NaN	quiet NaN	*FPError*	A
signalling NaN	number	*FPInvalidOp, FPError*	Q(A)
quiet NaN	number	*FPError*	A
number	signalling NaN	*FPInvalidOp, FPError*	Q(B)
number	quiet NaN	*FPError*	B

Table 11.31 Behavior of dyadic floating-point operations for NaN inputs

The stack manipulation instructions, the class analysis instructions, the load and store instructions, *fpldall*, *fpstall*, the integer to real conversion instructions (except *fpadddbsn*[1]), and the real to integer conversion instructions (except *fpint* and *fprtoi32*[1]), are not classified as floating-point operations and so do **not** cause *FPInvalidOp* or *FPError* when there is a NaN (of any kind) present as an input, nor do they convert a signalling NaN to a quiet NaN.

Note that since a signalling NaN is never generated by the machine, when one is detected it must have been either created explicitly or created accidentally (e.g. uninitialized or retyped data). The programmer can hence take advantage of the detection of signalling NaNs. For example it might be useful to 'initialize' floating-point data to signalling NaNs, so that if this data is used prior to being assigned a proper floating-point number, then this is signalled.

1. *fpint* is grouped with the 'real to integer' type conversion instructions because it is used in those conversion sequences. However it does itself produce a result in floating-point format, and so is a floating-point operation. *fprtoi32* is a compound instruction which incorporates *fpint* and so this part of the instruction is a floating-point operation. Similarly *fpadddbsn* is grouped with the 'integer to real' type conversion instructions because it is used in those conversion sequences, but this has floating-point operands and floating-point results and so is also classified as a floating-point operation.

2. *address(X)* loads into the integer stack, the address of the slot that holds variable *X* – see section 29.

12 Channels

The concepts of channels and communication between transputer processes were introduced in section 3.2, and more detail was provided in section 8.4, which also considered the implementation issues for internal channels. This chapter gives more detail on the three types of external channel and considers some of the implementation details of various communications on these channels. It also considers how to reset a channel in the event of failure; it provides further details on implementation of resource channels, including the instructions which are required; and finally it describes some other instructions which relate to channel communication.

12.1 Compilation and configuration of channels – an overview

Before considering the details of communication between processes, it is worthwhile having a general picture of the sort of scheme that could be used to produce machine code for a network of IMS T9000 transputers. The example given is not the only approach that can be taken but it demonstrates some of the principles which should be understood when designing tools for such a network. The description is based on a process/channel model which can be described directly in occam, but can also be implemented in other high-level languages.

In this system, there are two tools provided to produce machine code and data structures from a high-level description. Firstly there is a compiler, which compiles segments of code and assumes that this code is to be placed on a single transputer. Secondly there is a configurer which maps these segments onto a network of transputers.

The code generated by a compiler is always targeted onto a single transputer. This is referred to as an 'SC (separately compiled) unit'. It may consist of any number of parallel (or 'concurrent') processes, but execution of these processes is only concurrent in concept, since they are all sharing the same CPU. The channels that connect the processes within an SC unit, can all be implemented as internal channels as it is known that each end of the channel is connected to a process that is on the same transputer. (See figure 12.1.) It is therefore possible to allocate these channels at compile time to channel words in memory. The SC unit may also have channels as external variables or parameters. The compiler does not know if these are internal or external channels. They will connect to processes in other SC units, but it does not know whether or not the code for that unit will be loaded onto the same transputer. It therefore leaves the allocation of these to the configurer.

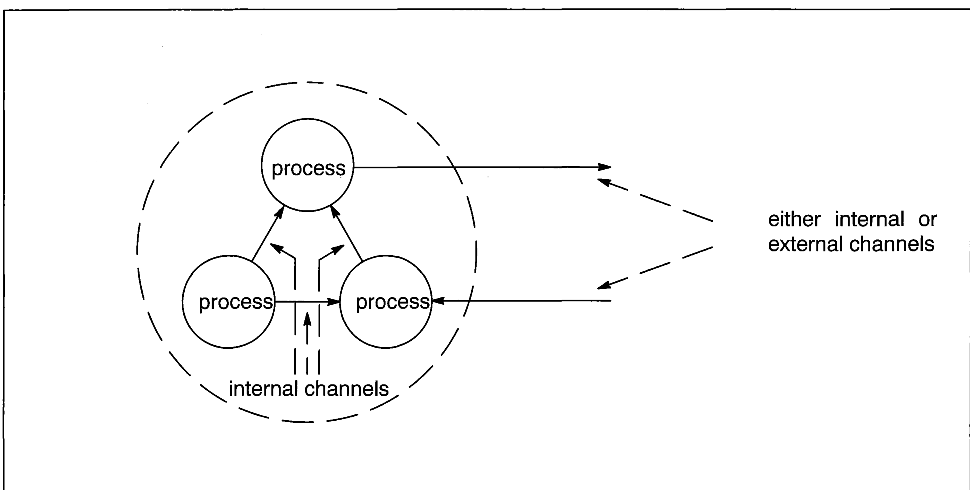


Figure 12.1 An SC unit – a separately compiled unit of code

Network mapping of processes onto transputers can be described by:–

- a high-level description of how the SC units are connected,
- a network description that describes the hardware, and
- a mapping from one to the other.

This is illustrated by the small example shown in figure 12.2. The configurer takes this information and deduces where to place each SC unit and how to implement the channels which connect the SC units. Note that the channels between SC units on different transputers, are implemented as external channels, while the channels between the SC units on the same processes are implemented as internal channels.

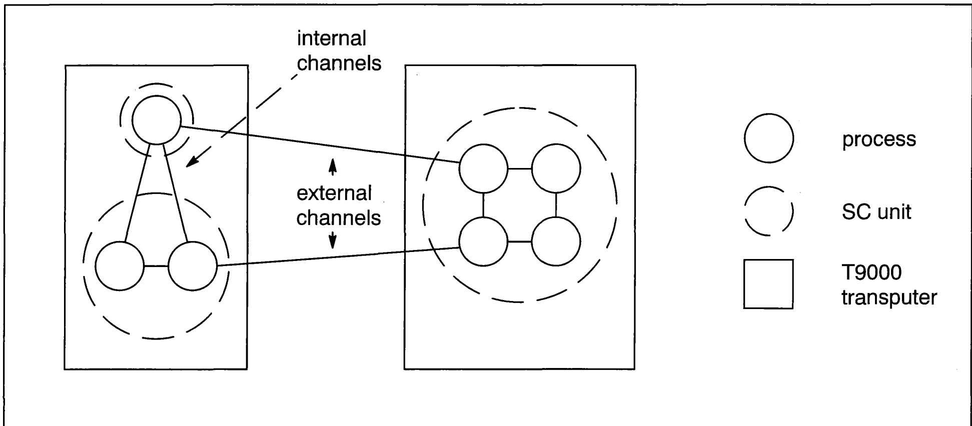


Figure 12.2 A small network

The instructions which are used to communicate between processes (*in*, *out*, *outbyte*, *outword*, *vin*, *vout*, *disc*, *enbc*), do not need to know the type of the channel. One of the parameters that is passed to them in the integer stack, is the channel address. The value of this address enables the processor to determine the type of the channel, and act accordingly. Therefore when writing the process code, or designing the code generator of a compiler, the programmer does not need to know the channel type.

The scheme described above assumes that the configuration is static – i.e. it does not change as the program is run on the network. A more complicated system might require processes to be dynamically loaded and channel connections to be made and broken under the control of a distributed operating system. This is equally achievable with a network of IMS T9000 transputers.

12.2 External channels

As introduced in section 3.2, a channel can either be internal (a communication path between processes on the same processor) or external (a communication path between processes on different processors). The implementation of internal channels is discussed in section 8.4. This section describes the three types of external channel: virtual channels, byte-stream channels, and event channels.

The processor determines the type of the channel from the instruction's channel address parameter. The channel address mapping is given in subsection 12.4.

A transputer has 4 physical (data) links. Each of these can either be used for virtual channels or for byte-stream channels, but not both.

12.2.1 Virtual channels

A virtual channel can connect any two processes in a network of IMS T9000 transputers and IMS C104 dynamic routing devices. The processors that host the processes do not have to be adjacent, provided

there is a connecting path through the network. This path must be specified by a 'packet header' which is associated with the virtual channel. The inputting end of each channel must have a buffer specified which is capable of storing the maximum sized packet expected. This section provides the background needed to set up and use virtual channels.

A virtual channel is referenced by using one of the virtual channel addresses (see section 12.4), and the physical link associated with that channel should be set to 'virtual mode' by configuring the appropriate VCP link mode register (see section 12.5). A communication must never be attempted on a virtual channel for which the physical link hasn't been set to virtual mode.

The IMS T9000 incorporates a hardware communications processor, called the *Virtual Channel Processor* (VCP), which is able to multiplex any number of *virtual channels* over each physical link. Each message is split into a sequence of packets, and packets from different messages may be interleaved over each physical link. Interleaving packets from different messages allows any number of processes to communicate simultaneously via each physical link. IMS T9000 transputers may be connected directly or via a network of IMS C104 dynamic routing devices. Communication channels can be established between any two processes regardless of where they are physically located, or whether the channels are routed through a network. Thus, programs can be independent of network topology.

In order that the VCP of the receiving transputer can distinguish packets that are part of different messages, each received packet contains one or two bytes that identify a virtual input channel of the receiving transputer. When a packet is transmitted it may also contain information to route the packet through a packet switching network of IMS C104s. The combination of any routing information and the identification of the virtual input channel of the receiving transputer is called the packet header. Every packet of a message ends with an end–of–packet (EOP) token, except the last packet which ends with an end–of–message (EOM) token.

The maximum length of data in each packet is 32 bytes (excluding the header and the EOP/EOM token). All but the last packet of a message contain the maximum amount of data; the last contains the maximum amount of data or less. Each packet has the structure illustrated in figure 12.3. The header bytes (containing routing and channel information) are transmitted first, followed by the data bytes of the packet (if any), followed by the encoded end of packet marker. The VCP can thus encode short messages (not longer than 32 bytes) in a single packet.

The message protocol details are not required by the programmer but the interested reader is referred to the *Communications* chapter of *The T9000 Hardware Reference Manual*.

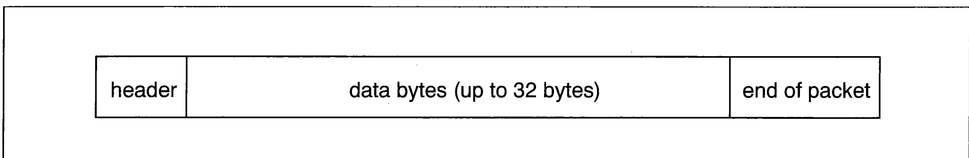


Figure 12.3 Structure of a packet

The VCP enforces a high-level protocol on each virtual channel. Each packet of data sent along a virtual channel must be acknowledged before the next is sent to ensure that no data is lost. An acknowledge packet is sent automatically by the VCP of the receiving IMS T9000. The transmitting IMS T9000 waits for the last packet to be acknowledged prior to rescheduling the outputting process, hence ensuring synchronized communication. Data packets on a virtual channel are acknowledged by the VCP by sending acknowledge packets on another virtual channel back to the VCP which sent them. This acknowledgement is process-to-process and is transparent to intermediate network components.

Virtual channels always occur in pairs between pairs of communicating processors, with one virtual channel in each direction. If a message is being communicated in one direction, the virtual channel in the opposite direction is used to return acknowledge packets to the sender. The pair of associated virtual channels is referred to as a *virtual link*. A virtual link can transfer messages in both directions at the same time with data packets and acknowledge packets being interleaved on both of the virtual channels. Because virtual

channels are always paired in this way it is not necessary to include source information in the packets. Thus packet headers need only represent their destinations.

When the CPU of one transputer performs an output instruction (*output, outbyte, outword* or *variable output*), its VCP sends the first packet of the message to another transputer on a virtual channel. When the VCP of the second transputer receives the packet, it uses the packet header to identify the virtual channel on which the packet was received. If a process on the second transputer has performed an input instruction on the channel, the data contained in the packet is stored in the data space of the inputting process. If there is no process ready to receive the first packet, then it is placed in an in-store packet buffer associated with the virtual link. When the inputting process becomes ready, the first packet is copied from the buffer into the data space of the process and an acknowledge packet is sent. This buffer is transparent to the communicating processes because it is empty when the input process is active.

In order that the data contained in a buffer is not overwritten, the VCP of a transputer that has sent one packet of a message on a virtual channel to another transputer, does not send another packet on that channel until it receives an acknowledgment that a process on the second transputer has become ready to receive the message.

Virtual link control blocks

For each virtual link, a data structure, called a *virtual link control block* (VLCB), is stored in the memory of both transputers connected by that link. A VLCB stores information to control the operation of the virtual link; it is 8 words long and aligned on an 8-word boundary. The details of the information stored in this block are private to the VCP and of no interest to the programmer, but the following provides a brief description of its usage and type of information that it holds.

A number of instructions (described in section 12.6.1) are provided on the IMS T9000 for manipulating VLCBs. These instructions may be used to establish the virtual links, dynamically alter the connections, activate, deactivate and reset the channels, place channels into resource mode, and assist in the debugging parallel programs.

The physical links are shared by a number of virtual links by threading on linked lists, the control blocks of the virtual links waiting to use the physical links.

Each VLCB has slots to store the process descriptors of the processes (if any) sending and/or receiving messages on the virtual link. These slots may be referred to as the channel words for the virtual output and virtual input channels respectively.

To enable the first packet of any message to be received at any time on a virtual input channel, it is necessary to have a buffer area to hold that packet until the receiving process is ready to transfer the message into the data area specified by the input instruction. The address of this buffer, which must be word-aligned, is copied into the VLCB from **Breg**, when *initvlcb* is executed (see section 12.6.1).

The virtual link's header contains routing information for its output channel's forward path and its input channel's acknowledge path. The VLCB specifies this header which is included with each packet sent.

- If a header is up to 3† bytes long it is held in the VLCB itself. This is known as a 'short header'.
- If a header is longer than 3† bytes, it is held in a special region of memory. This is known as a 'long header'. For a long header, the VLCB holds the length of the header and an offset to the location in memory where it may be found.

The VLCB marks the header as a null header when initialized by *initvlcb*. The encoding of short headers, where possible, within the VLCB saves a memory access on every packet sent.

In addition to the 8 word VLCB, there may be another two words for the resource channel data structure if the receiving channel is a resource channel, but these are held in a separate area of memory (see section 12.4).

Errors

The VCP can detect a length overrun on an *input*. This is dealt with by recording an invalid message length (*LengthError.p*) in the **pw.Length** slot of the inputting process workspace data structure. The process

† This value could change in future revisions of the IMS T9000.

must recognize and handle the error after it has been rescheduled, which it can do with the *ldcnt* instruction.

Other VCP and link errors are fully discussed in the *Communications* and *Data/Strobe links* chapters of *The T9000 Hardware Reference Manual.*

12.2.2 Byte-stream channels

A communication on a byte-stream channel is between processes on adjacent processors. That is, the processors' links must be physically connected (via a system protocol convertor or a link adaptor). The variable-length communication instructions cannot be used on byte-stream channels.

A byte-stream channel is referenced by using one of the byte-stream channel addresses (see section 12.4), and the physical link associated with that channel should be set to 'byte-stream mode' by configuring the appropriate VCP link mode register (see section 12.5). This mode of operation enables the IMS T9000 to communicate with a T2/T4/T8-series transputer, via the IMS C100 system protocol convertor. It does *not* provide a communication channel between adjacent IMS T9000 transputers. A communication must never be attempted on a byte-stream channel for which the physical link hasn't been set to byte-stream mode.

Output to a byte-stream channel

When an output communication instruction is executed on a byte-stream channel, the current process descriptor (content of the workspace descriptor register), the pointer to the message source, and the message length are saved, and the process is descheduled. The current process descriptor is copied into the relevant link channel word (see section 12.4).

When all the message has been output and the final acknowledge received, the process is rescheduled and the link channel word reset to *NotProcess.p*.

Input from a byte-stream channel

When an input communication instruction is executed on a byte-stream channel, the current process descriptor, the pointer to the message destination and the message length are saved, and the process is descheduled. The current process descriptor is copied into the relevant link channel word.

When all the message has been input and the final acknowledge received the process is rescheduled and the link channel word reset to *NotProcess.p*.

12.2.3 Event channels

The event-in pins (**EventIn0**, **EventIn1**, **EventIn2**, **EventIn3**) and event-out pins (**EventOut0**, **EventOut1**, **EventOut2**, **EventOut3**) provide an asynchronous handshake interface between external events and internal processes. Event channels provide process synchronization but cannot transfer any data. Each pair of event-in and event-out pins (0 to 3) can act independently as either an input or an output event channel, but not both.

Input event channel

When an external event takes an event-in pin high, the associated external event channel becomes ready. A process may have already performed an input on this channel. In this case, the process is rescheduled and the associated event-out pin is set high. If there has not yet been an input on that channel, then the processor does not set the event-out pin high until one occurs.

The processor then resets the event-out pin to low when it sees the event-in pin go low. See figure12.4.

Output event channel

When a process performs an output on an event channel, the IMS T9000 asserts the event-out pin and deschedules that process. This action instructs the external hardware, which connects

to the event pins, to perform an action. When the processor sees a low-to-high transition on the associated event-in pin, it reschedules the process and sets the event-out pin to low. The external device should then lower the signal on the event-in pin to complete the handshake. See figure 12.4.

If no process performs an output on a particular event channel, then the event-out pin for that event is never be taken high. The event-out pins are taken low when reset occurs or when a *resetch* instruction (see section 12.8.1) is executed on that channel.

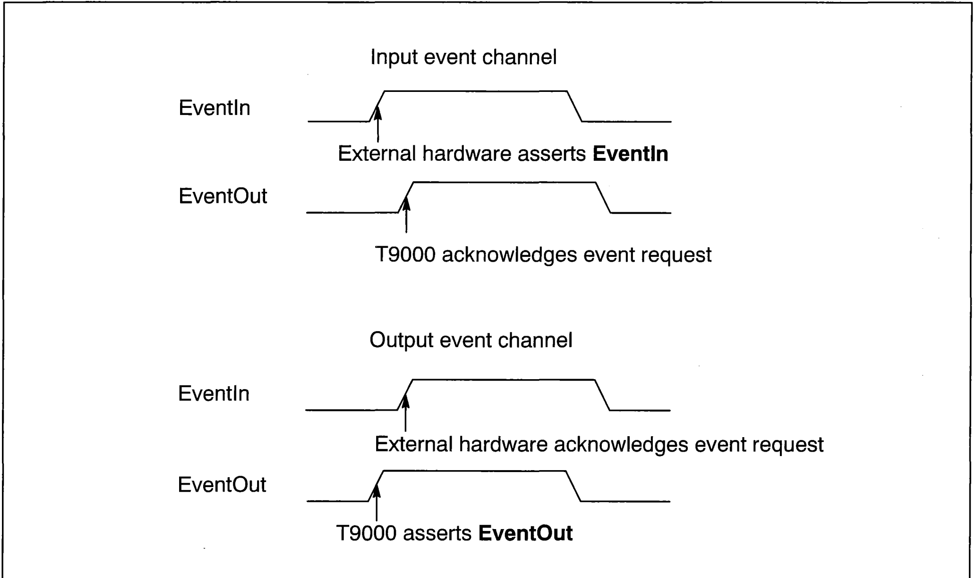


Figure 12.4 Event channel full handshake

The addresses of these event channels are shown in figure 12.8. For each pair, the input channel is one channel address word lower than its corresponding output channel. This is consistent with the channel address space for virtual channels and byte-stream channels. Note also that channel address #†80000020 is an event input channel, and this is consistent with the T2/T4/T8-series transputer.

When the state of an event channel is *empty* (this and other channel states/modes are defined in section 12.3), the channel word has the value *NotProcess.p*. If the state is *waiting* (inputting, outputting or enabled) then it contains the process descriptor of the communicating process. If the event channel is in the resource mode *idle* state, then it contains the value *ResChan.p*. The reverse channel word – that is, the other channel word of the input/output pair – holds the value *NotProcess.p* when the event channel is activated. When the channel is deactivated, it holds the special value *Deactivated.p*. Hence for an event channel to be initialized ready for communication, both the forward and reverse channel words should be assigned the value *NotProcess.p*. This is achieved by applying to the channel address, the instructions *resetch* and *setchmode* which are detailed elsewhere in this chapter.

Use of event channels as interrupts

These channels can be used to handle external interrupts. In real-time applications, it is important for a processor to be able to respond quickly to a signal from an external device. Input event channels can thus be used as interrupt pins with a full handshake facility. The following procedure explains how to do this.

An interrupt handler, implemented as a high priority process, should execute an input instruction on an input event channel. If the associated event-in pin is low, then this process is descheduled. It takes no

† #n represents a hexadecimal value.

further action unless that pin goes high. If it does, then the high priority process is rescheduled, and provided that the processor is currently executing a low priority process and interruption is enabled, an interrupt will occur at the next interrupt point. The high priority interrupt handler then resumes execution.

12.3 Channel states and modes of operation

12.3.1 Normal channel states

A 'normal' channel is defined as a channel which cannot be in resource mode. Such a channel can be essentially in one of two states: *empty* or *waiting*. Prior to receiving its first communication, the state of the channel is described as *empty*. When a single communication instruction (input or output) has been applied to that channel, its state is described as *waiting*. A channel is also *waiting* if it has been enabled in an alternative sequence. It remains in the *waiting* state until the communication is synchronized with the second communication instruction.

A virtual channel may also be in a third state: *stopping*. This is a 'winding down' state which is necessary when a virtual channel is stopped. When an input channel is *stopping*, any received packets are acknowledged. When an output channel is *stopping*, any packet queued will not be sent, but the channel must receive an acknowledge for any packet which has already been sent.

12.3.2 Resource channel states

A resource channel (as explained in section 8.8) has two modes: normal mode and resource mode. Within each of these modes the channel may be in one of two states.

In normal mode the resource channel behaves identically to a 'normal' channel.

In resource mode, the synchronization mechanism of the resource channel is as described in section 8.8. Prior to receiving an output, the state of the channel is described as *idle*. When an output instruction is applied to a resource channel, a 'claim' is made for the resource. If the server process is waiting on the resource, then the 'claim' is granted, and the channel is returned to normal mode. Otherwise, the channel is put on the resource queue and the state is described as *queued*.

12.3.3 Virtual and event channel activation modes

Each end of a virtual or event channel can be either 'activated' or 'deactivated'.

Both ends of the channel must be *activated* for communication to occur.

No communication can occur while either end of the channel is *deactivated*, but no state or data is lost. This puts the channel in temporary suspension. For example, a virtual channel remains deactivated until it is known that the virtual link headers have been set up correctly at both ends. When deactivated, an output channel cannot send any data packets, and cannot cause the VCP to schedule a process. If a packet is received on a deactivated input channel, then it is stored as normal, but no acknowledge packet is sent, and no scheduling actions can occur.

12.4 Channel configuration and mapping

The VCP and CPU (in common with a number of other sub-systems of the IMS T9000) are controlled via registers in a configuration space. The registers are accessed via the *ldconf* and *stconf* instructions, or via *CPeek* and *CPoke* command messages (described in the *Control system* chapter of *The T9000 Hardware Reference Manual*) received along control link **CLink0**. This subsection describes some of these configuration registers and also defines the relationship between the addressing of channels and the addressing of memory.

A channel address is the value used to access a channel in a communication instruction. The IMS T9000 channel address space is shown in figure 12.5. The special channel address names, *MemStart* and

MinInvalidChannel, are introduced later in this section. *MinVirtualChannel* and *MinEventChannel* are constants defined in section A.2.7 of appendix A.

Event channels are always accessed via event channel addresses. The physical data links are normally accessed via virtual channel addresses. However, each IMS T9000 physical link may be set to operate in 'byte-stream mode' for use in mixed transputer systems. They are then accessed via the byte-stream channel addresses.

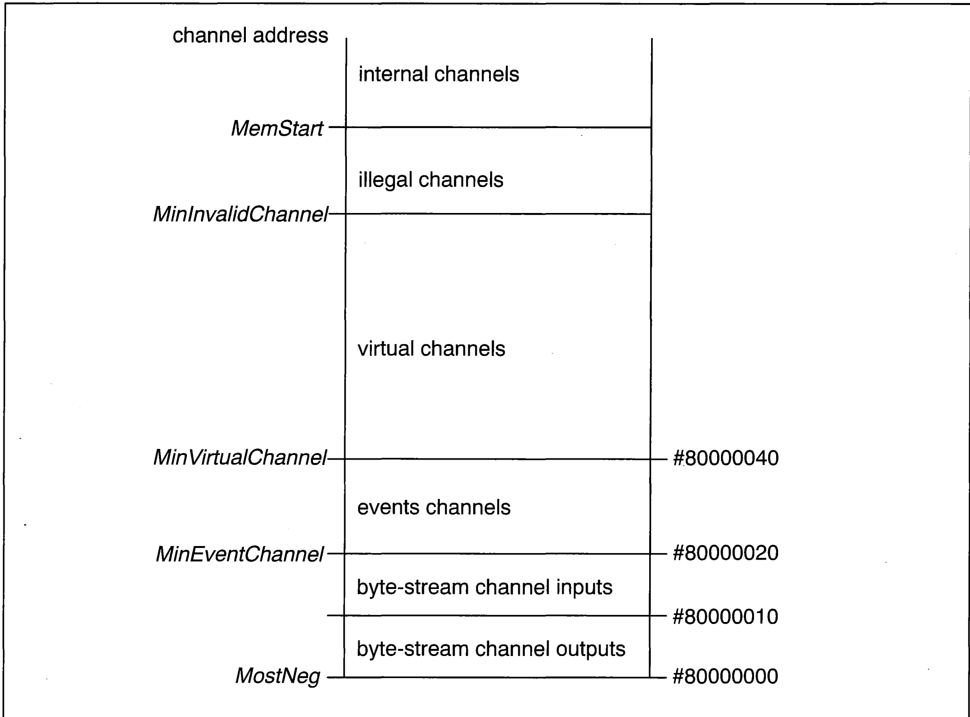


Figure 12.5 IMS T9000 channel address space

The IMS T9000 memory space is shown in figure 12.6. The special memory address names, *MemStart*, *HdrAreaBase* and *ExternalRCbase*, are introduced later in this section.

The user must not directly access the memory space below *MemStart*. To examine and assign the channel control information, the user should execute the instructions described in section 12.6.1 which access this area via the channel address space.

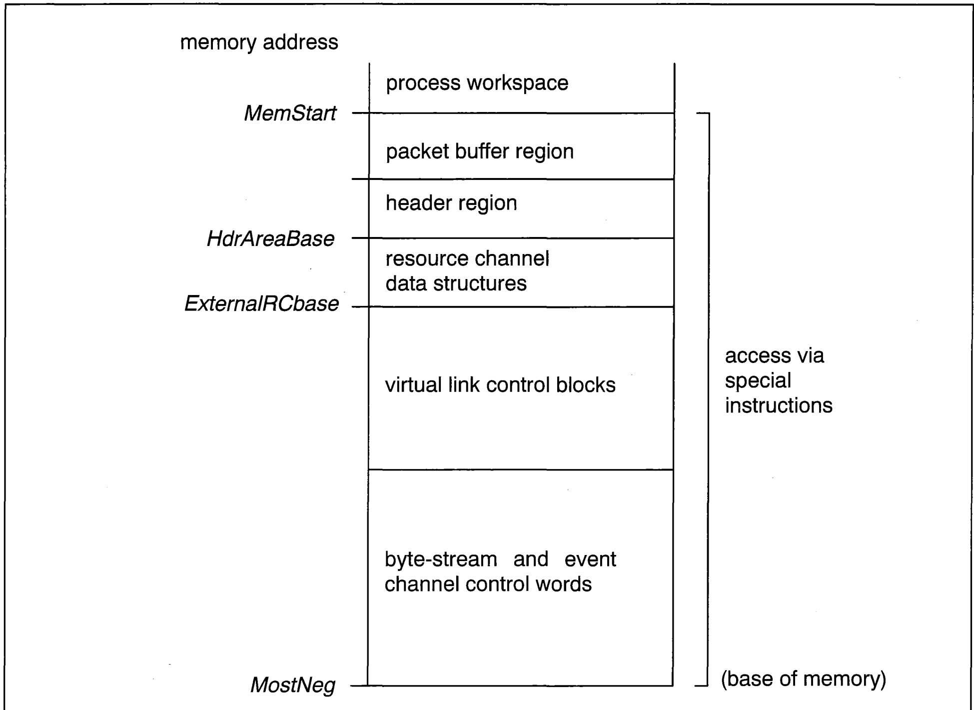


Figure 12.6 IMS T9000 memory map

Note that a virtual channel address does *not* equate to the memory address of the VLCB which implements that virtual channel.

12.4.1 Configuration register instructions

mnemonic	name
stconf	store to configuration register
ldconf	load to configuration register

Table 12.1 Configuration instructions

A complete list of the configuration registers and their addresses is given in the *Configuration register reference guide* chapter of *The T9000 Hardware Reference Manual*. The instructions in table 12.1 are used to write data to and read data from these registers.

stconf loads the data held in **Breg** into the configuration register specified by the configuration write-address in **Areg**. **Creg** is popped into **Areg** leaving **Breg** and **Creg** undefined.

ldconf reads the data held in the configuration register specified by the configuration read-address in **Areg**, and loads this into **Areg**. **Breg** and **Creg** are unaffected.

Both instructions are privileged. They both signal *IntegerError* if an invalid configuration address is passed in **Areg**.

12.4.2 Configuration registers used for memory mapping

Free memory start value register

The register **MemStart** holds a pointer to the start of free memory – *MemStart*. This identifies the first word of memory that is not used by the VCP for implementation of external channels. An example of setting this register up is given on page 169.

The communications instructions operate by treating all channel addresses at or above *MemStart* as internal channels – that is channels between processes executing on the same processor. All channel communications below this address are transferred to the VCP, after checking for illegal addresses.

mnemonic	name
ldmemstartval	load value of *MemStart* address

The *ldmemstartval* instruction can be used to obtain the free memory start value. This pushes *MemStart* onto the integer stack. It is a privileged instruction.

Packet buffers

Each virtual link must have an allocated packet buffer, which may be placed in any area of memory space that is not reserved for other use. An efficient strategy is to use maximum size (8 word) packets and align these to cache lines (4 words) in a region below *MemStart* but above the region allocated for headers.

Minimum invalid virtual channel address register

There is a range of channel addresses below *MemStart* that do not correspond to valid external channels. The memory below the equivalent memory address *MemStart* normally contains virtual link control blocks and headers. The lowest channel address that corresponds to an invalid channel address – *MinInvalid-Channel* – is held in the minimum invalid channel address register (**MinInvalidChannel**). An example of setting this register up is given on page 169.

External resource channel base address register

A resource channel is a channel that may be in *normal mode* or *resource channel mode*. It is implemented in the same way as a normal channel, but has an additional two word resource channel data structure (RCDS – see sections 8.8 and 12.7 on resources). For resource channels connecting processes on the same processor (i.e. internal resource channels), the RCDS is contiguous with the word used as the channel. For resource channels connecting processes on different processors (i.e. external channels) an RCDS is associated with each input virtual resource channel and event input resource channel. These extra words are allocated together in a block, and the base of the block (*ExternalRCbase*), which must be word-aligned, is defined by the external resource channel base address register (**ExternalRCbase**).

Header region word offset base register

If the header associated with a virtual output channel is longer than three bytes, it is not held in the VLCB associated with that channel, but resides in a separate region of store – the 'header region'. The base of this region – *HdrAreaBase* – is defined by the header base address register (**HdrAreaBase**) and must be word aligned.

12.4.3 Virtual link mapping functions

Allocation of virtual link numbers

Any two virtual channels that have source and destination processes on opposing transputers can be paired together and assigned to a unique virtual link on each transputer. Figure 12.7 shows a small network of processes statically connected via virtual links across a communications network. Each virtual link is given a 'virtual link number'.

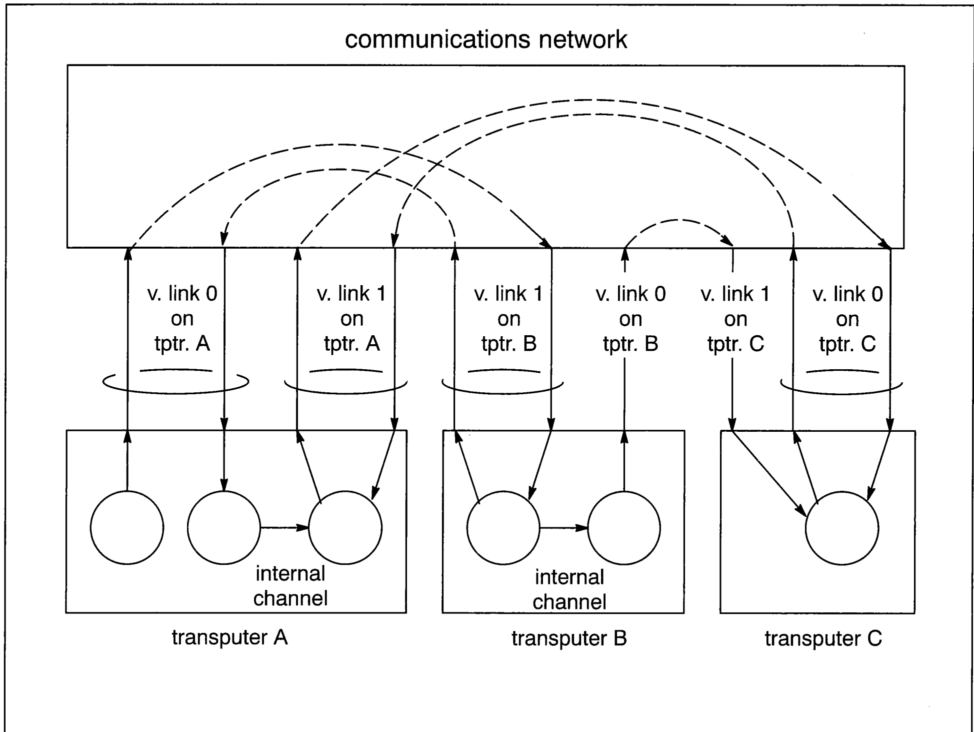


Figure 12.7 Network of processes connected by virtual channels

NB1: It does not matter if the source end of one channel and the destination end of another channel in a virtual link pair are connected to different processes, provided that these processes are on the same transputer. For example virtual link 0 on transputer *A* pairs two channels that connect to different processes.

NB2: If it is not possible to pair a virtual channel with a return channel, then that channel must be given a virtual link to itself on each transputer. For example there is only one channel between virtual link 0 on tranputer *B* and virtual link 1 on transputer *C*. But be aware that the return path is still used to transmit acknowledges so it is essential that both links are assigned the correct headers.

Mapping packet headers to virtual link numbers

When a packet is received via a physical link, the VCP converts the header number to a virtual link number, by subtracting the value stored in the appropriate header offset register.

The header offset registers (**VCPLink0HdrOffset**, **VCPLink1HdrOffset**, **VCPLink2HdrOffset**, **VCPLink3HdrOffset**) are each programmed with an offset. The number of the virtual link to which a packet is directed is calculated by the VCP hardware using the following mapping function: –

virtual link number = (Header – *HeaderOffsetN*)

where *HeaderOffsetN* is the content of the header offset register for physical link *N* via which the packet was received.

Section 12.4.4 briefly illustrates how this mapping is put into effect for message routing.

N.B. The header offset value would normally be expected to be no greater than the header value. However where this is not the case the VCP applies the following mapping function when calculating the virtual link number:–

virtual link number = (Header + 2^{16} – *HeaderOffsetN*) **rem** 2^{16}

where **rem** is the integer remainder operator – i.e. it performs a modulo 2^{16} subtraction.

Mapping virtual channel addresses to virtual link numbers

When an instruction references a virtual channel address, the processor automatically converts this address to the number of the virtual link associated with that channel. This mapping is given arithmetically by:–

virtual link number = ('virtual channel address' – *MinVirtualChannel*) ÷ 8 (4)

because there are 2 channels mapped to each link, and each channel address is separated by 4.

Hence channel addresses #80000040 and #80000044 are implemented by virtual link 0 and #80000048 and #8000004C are implemented by virtual link 1.

Mapping virtual link number to VLCB memory address

For each IMS T9000 in a network, the VLCB for its virtual link 0 is positioned in memory immediately above its byte-stream and event channel words at *MinVirtualChannel*, and is aligned to an eight word boundary (see figure 12.8). The VLCB for virtual link 1 is positioned immediately above virtual link 0 and so is also eight word aligned. All other VLCBs are stacked thus in ascending order of virtual link numbers.

The VCP maps each virtual link to its VLCB address by the formula:–

VLCB memory address = *MinVirtualChannel* + ('virtual link number' × *BytesPerVLCB*)

where *MinVirtualChannel* is #80000040 and *BytesPerVLCB* is 32 for the IMS T9000.

N.B. The user does not need to know this mapping function, but may need to know that 8 words are allocated for each virtual link in order to determine the required value of *ExternalRCbase* (see figure 12.6).

Figure 12.8 shows the mapping of channel addresses and header numbers to virtual link numbers, and also shows how each virtual link is allocated a unique VLCB. The example given shows 3 virtual links (6 virtual channels) using 2 words for long headers.

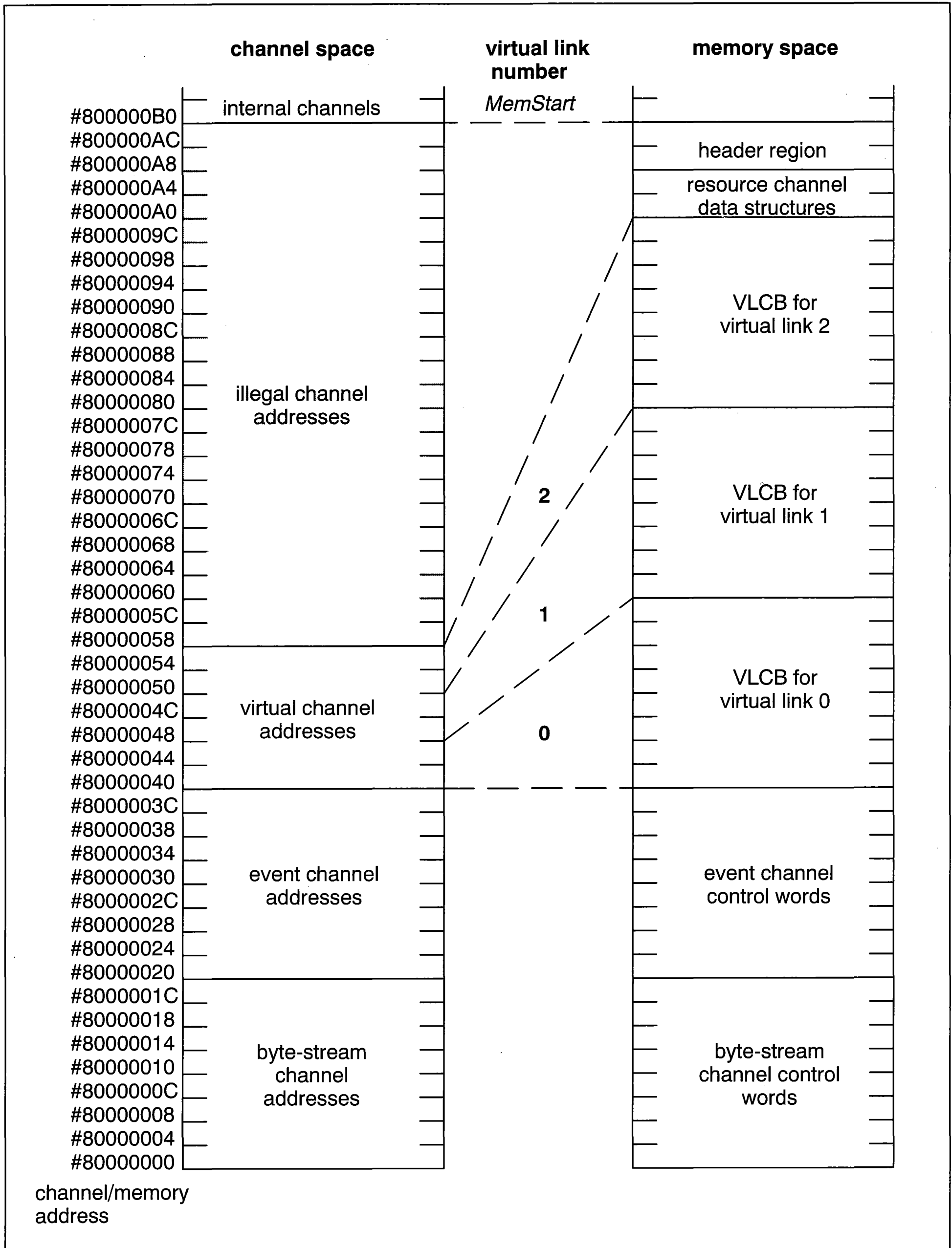


Figure 12.8 Mapping of channel addresses and header numbers to the memory address of the VLCB

12.4.4 Packet header labelling

There are two strategies that the programmer might adopt for routing messages in a network of IMS T9000 transputers and IMS C104 packet routing switches:–

- header deletion with local link numbering
- global link numbering.

In header deletion, the header comprises two parts. The first part of the header specifies the destination transputer. The IMS C104 packet routing switches are programmed to remove this part of the header prior to routing the rest of the packet to the destination processor. Hence the receiving processor just sees the second part of the header. This 'local header' is used to calculate the virtual link number and hence the memory address of the VLCB using the above mapping functions. In this system, the header offset values are usually set to zero. The local headers correspond to the virtual link numbers and there need be no correlation between the local header value and the number of the physical link on which the packet was received. Note that a header is transmitted in ascending order of byte significance, and so the routing header should be placed below the local header in memory when the header is written (see *writehdr* in section 12.6.1).

In global link numbering, each end of every virtual link in the network is given a unique destination address. This address is then used as the header. The IMS C104 packet routing switches are *not* programmed to delete headers in this system and so each packet is routed with its header according to the interval labelling mechanism described in *The IMS C104 Datasheet*. Provided that the receiving header addresses of the virtual links are such that the range of header addresses received on each physical link is contiguous, it is always possible for the IMS C104 packet routing switches to correctly route all messages. By setting the correct header offset value for each physical link, the IMS T9000 can calculate the virtual link number and hence the memory address of the VLCB using the above mapping functions.

For example consider the IMS T9000 shown in figure 12.9. The network is configured such that only messages with headers within a fixed range are received on each physical link.

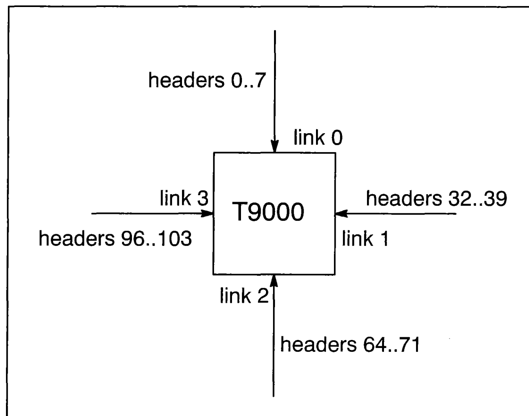


Figure 12.9 An IMS T9000 receiving packets with headers in specified ranges

If the contents of the header offset registers are set as

VCPLink0HdrOffset	=	0
VCPLink1HdrOffset	=	24
VCPLink2HdrOffset	=	48
VCPLink3HdrOffset	=	72

then the header to link mapping would be

headers 0..7	→	virtual links 0..7
headers 32..39	→	virtual links 8..15
headers 64..71	→	virtual links 16..23
headers 96..103	→	virtual links 24..31

12.5 Other configuration registers for setting up links and VCP

Packet header limit registers

The base and limit of packet headers which are acceptable on each physical link are stored in the header lower limit registers (**VCPLink0MinHeader**, **VCPLink1MinHeader**, **VCPLink2MinHeader**, **VCPLink3MinHeader**) and the header upper limit registers (**VCPLink0MaxHeader**, **VCPLink1Max-Header**, **VCPLink2MaxHeader**, **VCPLink3MaxHeader**). Out of range headers cause the associated packets to be discarded and, unless the **LocalizeError** flag (see under 'VCP link mode register' in this section) is set, generate link errors. These registers can be used for enhanced system security.

VCP link mode register

The VCP link mode registers (**VCPLink0Mode**, **VCPLink1Mode**, **VCPLink2Mode**, **VCPLink3Mode**) contain information about the physical links 0 to 3 respectively.

Bit	Bit field	Function
0	**ByteMode**	when set to 1, the associated physical link is set to operate in byte-stream mode – otherwise it is set to operate in virtual mode
1	**LocalizeError**	when set to 1, link errors detected by the VCP are no longer reported to the control unit – see also the *Communications* chapter of *The T9000 Hardware Reference Manual*
2	**HeaderLength**	programs the expected length of the incoming packet header (1 or 2 bytes) for each physical link 0 1 byte header 1 2 byte header

Table 12.2 Bit fields in registers **VCPlink0–3Mode**

VCP command register

The VCP command register (**VCPcommand**) enables commands to be issued to the VCP. Each bit of the register corresponds to a command, see table 12.3 below. The command is executed when the bit is set. Each write to the register can set only one bit.

It is important that the the VCP is not started until the VLCBs have been set up. An example is shown on page 171. For further details on starting, stopping and resetting the VCP, refer to the *Communications* chapter of *The T9000 Hardware Reference Manual.*

Bit	Bit field	Function
0	**Reset**	Reset the VCP – stops the VCP and resets the registers to their undefined level 2 state.
1	**Start**	Start the VCP
2	**Stop**	Stop the VCP 'cleanly' so that channel states are preserved. The VCP accepts messages currently in transit but no new messages can be sent.

Table 12.3 Bit fields in the **VCPcommand** register

12.6 Setting up the virtual link control blocks

12.6.1 Instructions for setting up a VLCB

To set up virtual channel communication, it is necessary to provide each VLCB with: a physical link number, header information, and buffer information for input channels. Although it is useful to understand the contents of the VLCB as described in section 12.2.1, it is not necessary for the programmer to know its precise structure and memory map. The IMS T9000 provides a set of privileged instructions that allow the user to set up the VLCB. These are listed in table 12.4 and explained below.

mnemonic	name
initvlcb	initialize vlcb
sethdr	set virtual channel header
writehdr	write virtual channel header
readhdr	read virtual channel header
insphdr	inspect virtual channel header
readbfr	read buffer pointer from VLCB
swapbfr	swap buffer pointer in VLCB
setchmode	set channel mode

Table 12.4 Instructions used for setting up and manipulating the VLCB

The instructions *initvlcb*, *sethdr*, *writehdr*, *swapbfr* and *setchmode* are used for setting up the VLCB. The instructions *readhdr*, *insphdr*, *swapbfr* and *readbfr* are used for analyzing the current state of the VLCB.

initvlcb

initvlcb initializes the VLCB associated with the channel specified in **Areg**. It also associates the word-aligned packet buffer specified in **Breg** with the VLCB, and ensures that the VLCB marks the header as a null header (see section 12.6.2). Both the virtual input and the virtual output channels associated with this VLCB are deactivated by the instruction. **Areg** inherits the value of **Creg** leaving **Breg** and **Creg** undefined. The example shown on page 171 illustrates the use of this instruction.

If the packet buffer specified in **Breg** is not word-aligned, then the instruction signals *Unalign*.

sethdr

sethdr establishes the physical link number and header type for the VLCB associated with the channel specified in **Areg**. **Breg** holds the physical link number to be associated with the VLCB, or a special value – *NullHeader* – to indicate that a null header is required (see section 12.6.2). **Creg** holds the word offset into the header region to specify the location of the header, or it contains a special value – *NullOffset* – to indicate that the header is a short header. The integer stack is left undefined after execution. The example shown on page 171 illustrates the use of this instruction.

If the unsigned value passed in **Breg** is greater than or equal to the number of physical links (4) and is not equal to *NullHeader*, then *sethdr* signals *IntegerError*. If the unsigned value passed in **Creg** is greater than the maximum header region word offset – *MaxHeaderOffset* – and is not equal to *NullOffset*, then the instruction signals *IntegerError*. It only sets to a null header if both input and output channels for the VLCB are deactivated. If this is not the case, it signals *IntegerError*.

If **Breg** contains the value *NullHeader*, then the VLCB marks the header as a null header.

If **Breg** contains a valid physical link number (0 ... 3), then *sethdr* records this in the VLCB. If **Creg** contains a valid header region word offset (0 ... *MaxHeaderOffset*), then it also records this in the VLCB. Note carefully that this offset is a *word* offset and not a byte offset. The absolute address of the header in the header

region is (**HdrAreaBase** + 'header region word offset' × 4). If **Breg** contains a valid physical link number and **Creg** contains *NullOffset*, then the link number is recorded in the VLCB but there is no offset to be recorded.

writehdr

writehdr assigns a packet header to the VLCB associated with the channel specified in **Areg**. **Breg** holds the length in bytes of the header. **Creg** holds the address of the data area where the header is stored. The integer stack is left undefined after execution. The example shown on page 171 illustrates the use of this instruction.

If the value passed in **Breg** is zero, or is greater than the maximum header length (#FF), then *writehdr* signals *IntegerError*. If the value passed in **Breg** is greater than the maximum short header length (3) but the VLCB has not been set up (using *sethdr*) with a header region word offset, then the instruction signals *IntegerError*. The instruction is interruptible.

Prior to execution of this instruction, the header is stored in a contiguous block that is **Breg** bytes long and begins at the address held in **Creg**.

If the header length in **Breg** is in the range 1 to 3, then the header is a short header, and so is stored in the VLCB. The length of the header and the header itself are copied into the VLCB, from **Breg** and data area pointed to by **Creg**.

If the header length in **Breg** is greater than 3, then the header is a long header. This length is written into the VLCB. The VLCB should have already been initialized with a header region word offset (using *sethdr*), which identifies the place where the header should be stored. The header is copied to that header region, from the data area pointed to by **Creg**.

A packer header is transmitted in ascending order of byte significance, and so the byte pointed to by **Creg** will be the first byte transmitted.

readhdr

readhdr copies the packet header of the VLCB associated with the channel specified in **Areg**, into an area of store beginning at the address held in **Creg**. **Breg** holds the length in bytes of that header. The integer stack is left undefined after execution.

If the value passed in **Breg** is not equal to the header length, then the instruction signals *IntegerError*. If the VLCB has been set to a null header, then the instruction signals *IntegerError*. The instruction is interruptible.

If the header length in **Breg** is in the range 1 to 3, then the header is a short header. The header is therefore copied from the VLCB to the block pointed to by **Creg**.

If the header length in **Breg** is greater than 3, then the header is a long header. The header is therefore copied from the special header region, to the block pointed to by **Creg**.

insphdr

insphdr loads the integer stack with information stored in the VLCB associated with the channel specified in **Areg**. It loads **Areg** with the number of the physical link (0 ... 3) on which the specified channel communicates, loads **Breg** with the length in bytes of the header, and loads **Creg** with the header region word offset (if applicable).

If the header is a null header, then *NullHeader* is loaded into **Breg** and *NullOffset* is loaded into **Creg**. If the header is a short header then the its length in bytes is loaded into **Breg**, and *NullOffset* is loaded into **Creg**. If the header is a long header, then the length is loaded to **Breg**, and the header region word offset is loaded into **Creg**.

swapbfr

swapbfr exchanges the buffer pointer in **Breg** with the buffer pointer of the VLCB associated with the input channel specified in **Areg**.

The instruction signals *IntegerError* if the channel address is not a virtual *input* channel address, and signals *Unalign* if the packet buffer specified in **Breg** is not word-aligned.

The buffer pointer held in **Breg** is written into the word in the VLCB reserved for the buffer pointer. This establishes a (new) buffer pointer for the VLCB. Any pointer which may have been previously associated with the VLCB is loaded into **Areg**.

Breg inherits the value previously in **Creg**, and **Creg** is left undefined after execution.

readbfr

readbfr loads into the integer stack, the buffer pointer, buffer length and received packet status of the VLCB associated with the input channel specified in **Areg**. **Areg** is loaded with the buffer pointer. **Breg** is loaded with length in bytes of the packet currently stored in the buffer. **Creg** is loaded with an integer representing the receive state.

The instruction signals *IntegerError* if the channel address is not a virtual *input* channel address.

The receive state in **Creg** is as coded in table 12.5.

receive state code	meaning
0	no packet – there is no packet currently stored in the buffer
1	last packet – the packet stored in the buffer is the last packet in the message
2	other packet – there is a packet in the buffer but it is not the last packet of the message

Table 12.5 Meaning of value loaded into **Creg** by *readbfr*

setchmode

setchmode activates the virtual channel specified by the channel address in **Areg** if **Breg** holds the value *true*, or deactivates the channel if it hold the value *false*. **Areg** inherits the value previously in **Creg**, and **Breg** and **Creg** are left undefined after execution. The example shown on page 172 illustrates the use of this instruction.

If the channel address in **Areg** associates with a VLCB that has a null header when a channel is being activated, then the instruction signals *IntegerError.* Also if the value in **Breg** is not *true* or *false*, then it signals *IntegerError*. *setchmode* can also be used on an event or a byte-stream channel (see section 12.9.3), but signals *IntegerError* if the channel address is not an external channel address.

If the channel address is that of a virtual channel (with a non-null header), then the VCP records the activation mode of the channel in the VLCB.

Programming for independence of short/long header knowledge

There is no guarantee that future versions of the IMS T9000 will have the same maximum short header length – namely 3 for the current version. It may be prudent therefore for programs not to assume this value.

It is possible to set up a header using *sethdr* and *writehdr* without knowledge of the maximum length of a short header. To do this, it should firstly be assumed that the header is a long header and *sethdr* should be used to reserve this space in the header region. *writehdr* then writes the header either into this space if it is a long header or into the VLCB if it is a short header. Finally, *insphdr* can be used to determine whether or not the header is a long or short header, allowing the unused memory in the header region to be freed if it is a short header.

12.6.2 Null header

When a VLCB is first created or is cleared using *initvlcb*, the channels associated with it are deactivated, and there is no header information – i.e. it has a 'null header'.

It is guaranteed whenever a virtual link has a null header, that the virtual channels using the link are deactivated. This prevents communication on these channels and, before communication instructions can use them, it is necessary to install the virtual link with a header using *sethdr* and *writehdr*. The channels can then be activated using *setchmode*.

Note if *setchmode* attempts to activate a virtual link that has a null header, then it signals *IntegerError*. Also, *sethdr* signals *IntegerError* if either of the channels associated with the specified virtual link are activated. These instructions thus ensure that a null header cannot be installed with activated channels.

12.6.3 An example

This subsection presents a small example for setting up the virtual links control blocks as well as their associated buffers. Some assumptions are made as a starting point:–

- no virtual input channels are to be used as resource channels
- all headers are short headers
- the number of virtual links is known and is stored in a local variable – *NumVirtualLinks*

An outline of this procedure is:

```
SEQ
  ... set up configuration registers
  ... stop the VCP
  ... set up virtual link control blocks
  ... start the VCP
  ... activate channels
```

Each stage is described in more detail below.

Set up configuration registers and pointer to packet buffer region

In this example the channel and memory maps are to be set up as shown in figure 12.10. There is no requirement for a resource channel data structure region or a header region; none of the virtual channels are resource channels and all headers are short headers to be saved in the virtual link control blocks.

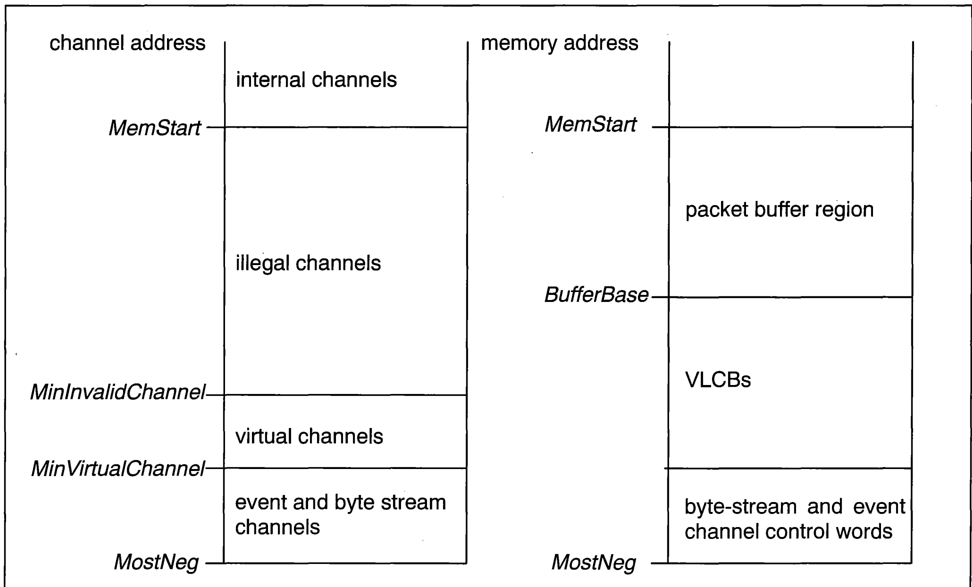


Figure 12.10 Example channel and memory maps

The configuration registers that need to be assigned, are: **MemStart**, and **MinInvalidChannel**. It is also necessary to assign a variable – *BufferBase* – that points to the start of the packet buffer region.

The value *MinInvalidChannel* can be calculated from knowledge of the number of virtual links – *NumVirtualLinks*. Equation (4) above shows that the mapping from virtual channel addresses to virtual link numbers is obtained by dividing by 8. This mapping can be used reciprocally to determine the minimum invalid channel address from the number of virtual links.

```
ldl NumVirtualLinks;                — calculate the offset address from the minimum virtual
ldc 8; prod;                        — channel to the first invalid channel address;
ldc MinVirtualChannel; sum;         — add offset to minimum virtual channel address
                                    — to obtain the first invalid channel address;
ldc MinInvalidChannelConfigAddr;    — store MinInvalidChannel into configuration register;
stconf;
```

The buffer region is placed immediately above the VLCB region. The address of the start of this region – *BufferBase* – can be calculated from knowledge of the number of virtual links and the number of words that should be allocated for each VLCB.

```
ldl NumVirtualLinks;
ldc BytesPerVLCB;
prod;                               — calculate total number of bytes required for VLCB region;
ldl MinVirtualChannel;              — calculate memory address above VLCB region
sum;                                — (N.B. would be ExternalRCbase if resource channels
                                    — were being used);
stl BufferBase
```

MemStart can be calculated from knowledge of *BufferBase*, and the size of each packet buffer; in this example each packet buffer is allocated enough memory for a maximum sized packet – *MaxPacketLength*.

```
ldl NumVirtualLinks;
ldc MaxPacketLength;
prod;                        — calculate total number of bytes required for buffer region;
ldl BufferBase; sum          — calculate memory address above buffer region;
ldc MemStartConfigAddr;      — store MemStart into configuration register;
stconf;
```

Stop the VCP

It is advisable to ensure that the VCP has been reset and then put into the waiting state before configuring the processor's virtual channels.

```
ldc 1;
ldc VCPcommandConfigAddr;    — set bit '0' in VCPcommand to reset
stconf;                      — the VCP – see table 12.3;

ldtimer;
ldc WaitTimeForVCPreset;
sum;
tin;                         — wait for VCP to be reset

ldc 0;
ldc VCPcommandConfigAddr;    — clear bit '0' in VCPcommand to return
stconf;                      — the VCP to waiting state;
```

The value of *WaitTimeForVCPreset* depends on the priority of the current process. The VCP needs at most 20μs for the reset to be effective. At low priority therefore, set *WaitTimeForVCPreset* to 2, and at high priority, set it to 21.

Set up virtual link control blocks and packet buffers

In this example, when this code is loaded into memory space, it is followed by a block of encoded data that comprises a 3 block data structure for each virtual link to be set up. Each data structure holds the following information that is needed for loading into each VLCB.

word offset	slot name	purpose
2	HeaderOffset	header to be associated with virtual link
1	HeaderLenOffset	length of the header in bytes (must be $\leq$ 3) – N.B. must be stored from least significant end of word
0	PhyLinkOffset	number specifying physical link on which virtual link will transmit packets

These data structures are stored contiguously from memory label *VLCBdata_label*. Note that this information could be stored in a more compact form. For example, the length of the header (which in this example will never be more than 3) and the physical link number could be coded in 4 bits rather than 2 words. However this would require masking to extract the information, and this example has been intentionally simplified.

```
ldc 0; stl (LEDS+le.Index);           — set up 'loop end data structure';
ldl NumVirtualLinks; stl (LEDS+le.Count);

ldc MinVirtualChannel;                — initialize CurrentVirtualChannel,
stl CurrentVirtualChannel;            — a variable that is going to be used to
                                      — point to each virtual output channel
                                      — in turn;
ldl BufferBase;                       — initialize CurrentBuffer_ptr, a variable
stl CurrentBuffer_ptr;                — that is going to be used to point to each
                                      — buffer area in turn;
```

```
        ldc (VLCBdata_label – NextInst_label)   — initialize CurrentVLCBdata_ptr, a variable
        ldpi                                    — that is going to be used to point to the
NextInst_label:                                 — VLCB information data structure for
        stl CurrentVLCBdata_ptr;                — each virtual link;

LOOP:
        ldl CurrentBuffer_ptr;                  — load pointer to buffer area
        ldl CurrentVirtualChannel;              — load the virtual channel address;
        initvlcb;                               — initialize VLCB for the 'current channel'
                                                — and set up its packet buffer;
        ldc NullOffset;                         — all headers are short headers in this
                                                — example;
        ldl CurrentVLCBdata_ptr;                — load the physical link number for this
        ldnl PhyLinkOffset;                     — virtual link;
        ldl CurrentVirtualChannel;              — load the virtual channel address;
        sethdr;                                 — set up the virtual channel header;

        ldl CurrentVLCBdata_ptr;
        ldnlp HeaderOffset;                     — load pointer to the short header;
        ldl CurrentVLCBdata_ptr;
        ldnl HeaderLenOffset;                   — load pointer to the header length;
        ldl CurrentVirtualChannel;              — load the virtual channel address;
        writehdr;                               — write header into VLCB;

        ldl CurrentVirtualChannel;              — adjust the current virtual channel
        ldnlp 2;                                — address to that of the next virtual
        stl CurrentVirtualChannel;              — input channel;

        ldl CurrentVLCBdata_ptr;                — adjust CurrentVLCB_ptr to point to
        ldnlp 3;                                — the next VLCB information
        stl CurrentVLCBdata_ptr;                — data structure;

        ldl CurrentBuffer_ptr;
        ldc MaxPacketLength; sum;               — adjust CurrentBuffer_ptr to point to
        stl CurrentBuffer_ptr;                  — the next buffer area;

        ldlp LEDS; ldc (END – LOOP)
        lend                                    — test for last iteration;
END:

NextInst_label:                                 — load encoded VLCB data here;
```

Start the VCP

Now that the virtual link control blocks and their associated buffers have been assigned, the VCP can be started.

```
ldc 2;
ldc VCPcommandConfigAddr;            — set bit '1' in VCPcommand to start
stconf;                              — the VCP – see table 12.3;
```

Activate channels

Finally the virtual channels can be activated.

```
        ldc 0; stl (LEDS+le.Index);           — set up 'loop end data structure' –
        ldl NumVirtualLinks;                  — N.B. the number of virtual links is multiplied
        ldl 2; prod; stl (LEDS+le.Count);     — by 2 to obtain the number of virtual channels;
        ldc MinVirtualChannel;                — initialize CurrentVirtualChannel, a
        stl CurrentVirtualChannel;            — a variable that is going to be used to
                                              — point to each virtual output channel
                                              — in turn;
LOOP:                         ldc true;
        ldl CurrentVirtualChannel;
        setchmode;                            — activate channel;

        ldl CurrentVirtualChannel;
        ldnlp 1;
        stl CurrentVirtualChannel;            — increment to next virtual link;

        ldlp LEDS; ldc (END – LOOP)
        lend
END:
```

12.7 Resource channels

Section 8.8 introduced the concept of a resource and discussed its purpose and application. Resource channels can be implemented on internal, virtual and event channels. This section provides details. It also explains how a reverse channel from server to client can be implemented. It describes all the instructions which are used in conjunction with resource channels. Finally it describes how resource channels should be used to implement various client-server models.

12.7.1 Implementation of internal resource channels

To implement an internal resource channel, a resource channel data structure (RCDS – presented in section 8.8.2) should be positioned immediately above the word in memory used to implement an internal channel (refer to section 8.4.4). The latter can still be used as a normal channel word, but holds the special value *ResChan.p*, when the channel is in *idle* state. If the channel word has this value when an output instruction is executed, then the processor recognizes that it is in resource mode, and a 'claim' (this operation is defined in section 8.8.2) is made on behalf of the client process executing the output instruction.

12.7.2 Implementation of external resource channels

A virtual input resource channel

A communication on a virtual channel can be queued on a resource data structure in exactly the same way as a communication on an internal channel.

The resource (and server process) is on the the same transputer as the receiving end of the virtual channel, and so it is on this processor that the communication is queued. It is therefore only the *input* channel which needs to be in resource mode.

To implement a virtual input resource channel, an RCDS should be associated with the VLCB for that channel. The positioning of this virtual RCDS is discussed below under 'Placement of resource channel data structures for external channels'. When an input is received while this channel is in resource mode, the VCP makes a 'claim' on behalf of that channel to the resource specified by the **rc.Ptr** slot. The action taken then is as described in section 8.8.2.

An event resource channel

A communication on an event channel can be queued on a resource in exactly the same way as a communication on an internal channel or a virtual channel.

To implement an event resource channel, an RCDS should be associated with the that event channel. The positioning of this event RCDS is discussed below under 'Placement of resource channel data structures for external channels'. When an input is received while this channel is in resource mode, the processor makes a 'claim' on behalf of that channel to the resource specified by the **rc.Ptr** slot. The action taken then is as described in section 8.8.2.

Placement of resource channel data structures for external channels

The resource channel data structures for external channels are located in a separate area to their associated channel data structures – i.e. the RCDS for a resource input channel is not contiguous with the VLCB for that channel, and the RCDS each event channel is not contiguous with the channel word for that channel. The area which stores these data structures is the 'external RCDS block' and begins at the memory address stored in the external resource channel base address register (**ExternalRCbase**). This address should be set up by the program responsible for allocating virtual channels and resource channels. Although this area can be placed anywhere, it is best placed below *MemStart.* The programmer must also be aware of the other structures which are mapped below *MemStart* (see figure 12.8).

There is a one to one sequential correspondence between each external RCDS and its associated external channel (refer again to figure 12.8). That is, the first two words in the external RCDS block are the RCDS for event input channel 0 (channel address #80000020), the next two words for event input channel 1 (channel address #80000028) etc. The first two words after the four event resource channel data structures are the RCDS for the first virtual input channel (channel address #80000040), the next two for the second virtual input channel (channel #80000048) and so forth.

Observe that if say the tenth virtual input channel (channel address #80000088) is used as a resource channel, then space must be reserved in the external RCDS block for all the event resource channel data structures and the first nine virtual resource channel data structures. This is the case whether or not any these other channels are being used as resource channels. It therefore makes good sense in terms of memory space economy, to map all the virtual input channels used as resource channels onto to the lowest address channels possible (from channel address #80000040 upwards), and similarly for event channels.

12.7.3 Reverse channel

Where a server process is inputting a message on a resource channel, it is often convenient to use the channel address immediately above the input channel, as an output channel to the client process. This channel is referred to as the 'reverse channel'. For a virtual channel, the reverse channel is the other channel (i.e. the output channel) in the pair of channels implemented by the virtual link. For an internal channel, the reverse channel word is the **rc.Ptr** slot of the resource channel data structure, and so it is only under certain circumstances that this can be used as a reverse channel.

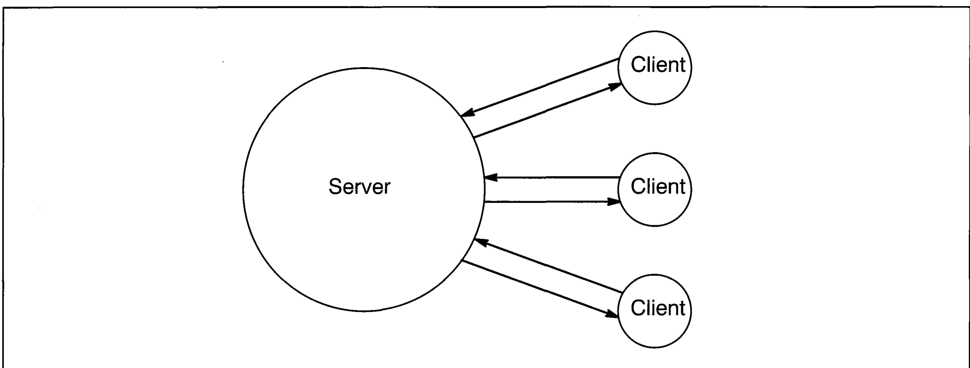
When the resource channel is in normal mode, it is not associated with an RDS and cannot be on a resource queue, and so the **rc.Ptr** slot of the RCDS is not used to hold a pointer. Hence, since this slot is positioned immediately above the channel word for an internal resource channel, it may be used as the reverse channel provided that the channel is in normal mode.

For example consider the following piece of occam code for a client and server model.

```
PROC Server( []CHAN OF ANY in, out )
  WHILE TRUE
    ... declarations
    ALT i = 0 FOR SIZE in
      in[i]? message
        SEQ
          Task( message, i )
          out[i]! task_complete
:
```

```
PROC Client( CHAN OF ANY out, in )
  ... declarations
  SEQ
    ...
    out! message
    in? task_complete
    ...
:

[n]CHAN OF ANY client.to.server, server.to.client:
PAR
  ... declarations
  Server( client.to.server, server.to.client )
  PAR i = 0 FOR n
    Client( client.to.server[i], server.to.client[i] )
```



In this example, the client sends a message to the server and waits for a response (`task_complete`), indicating that the server has performed its task, prior to continuing. The server receives the message, performs the task and acknowledges completion to the client. If this is implemented with the resource mechanism, then the reverse channel can be used for each component of `server.to.client`. This is possible because the *grant* instruction sets the channel to normal mode. Conversely observe that it isn't possible to use the **rc.Ptr** slot as a reverse channel until the *grant* instruction has been executed. In practice, the reverse channel should not be used before an input instruction has been executed on the 'forward' channel, because this would result in deadlock.

Note that as far as the server is concerned, it does not need to know whether the reverse channel is an internal channel or an external (virtual) channel. In both cases, the channel address of the reverse channel is immediately above the input channel in channel address space.

12.7.4 Instructions for setting and using the resource mechanism

This section contains a description of the instructions which are specifically used in conjunction with the resource mechanism. This is followed by some rules on how some of these should be used. The instructions covered here are shown in table 12.6. These instructions assume that the resource channel and resource data structures have been allocated as described earlier.

mnemonic	name
mkrc	mark resource channel
grant	grant resource
unmkrc	unmark resource channel
erdsq	empty resource data structure queue
irdsq	insert at front of RDS queue
ldresptr	load resource queue pointer
stresptr	store resource queue pointer
enbg	enable grant
disg	disable grant

Table 12.6 Instructions used for resource mechanism

mkrc

mkrc takes as its parameters: a channel address in **Areg**, a pointer to an RDS in **Breg** and a resource channel identifier in **Creg**. The instruction sets the channel to resource mode by associating it with the RDS, and giving it an identifier. If there has already been an output to this channel, then the resource channel is linked into the RDS. The stack is left undefined after this operation.

mkrc is a privileged instruction. If the value in **Areg** is not word aligned, then the instruction signals *Unalign*; and if the value is not a valid internal, virtual or event channel address, then it signals *IntegerError*.

If the **rc.Id** slot already contains a resource channel identifier (i.e. it does not contain *NotProcess.p*), then the channel is already in resource mode and so the instruction has no effect. Otherwise, the following occurs. The resource channel identifier (**Creg**) is assigned to the **rc.Id** slot. The state of the channel is examined to determine whether or not the channel is *empty* or *waiting*. If it is *empty*, then the **rc.Ptr** slot is assigned the RDS pointer (**Breg**), and the the state becomes *idle*. If it is *waiting*, the resource channel is attached to the end of the RDS queue and the state becomes *queued*.

For an internal channel that is *empty* or an external channel (virtual or event input), the channel word is set to *ResChan.p*. This is not the case for an internal channel which is *waiting*, because the memory word is used to store the process descriptor of the outputting process.

mkrc should not be applied to a channel that has been put into the *waiting* state by an input instruction. It may be applied to a channel which is *waiting* due to an output instruction, but it may not be applied when both the channel is *waiting* and the resource has a waiting server. This is because the instruction does not perform a complete 'claim'; it just attaches the channel to the resource queue. This only needs to be considered in practice, when a controller process (see section 12.7.5) is executing a *mkrc* instruction; because when a server executes *mkrc*, it cannot be waiting, and similarly when a client executes *mkrc*, there cannot be an output pending.

grant

grant takes as its parameters: the address of the RDS in **Areg**, and the 'identifier store address' in **Breg**, where the identifier store address is the address of a memory location where the result of the instruction is to be stored. If there is no client waiting for the resource, then the current process deschedules, otherwise the resource channel identifier associated with the client is written into the identifier store address, and the resource channel of that client is set back to normal mode. The the integer and floating-point stacks are left undefined after this operation.

grant is a privileged instruction. If the value in **Areg** or **Breg** is not word aligned, then the instruction signals *Unalign*.

The instruction inspects the RDS queue. If it is empty, then the current process is descheduled, leaving a copy of its process descriptor in the **rds.Proc** slot of the RDS, and the identifier store address in the

pw.Pointer slot of the process workspace. If there is one or more resource channels in the RDS queue, then: the processor: removes the channel at the front of the queue, writes the resource channel identifier (value held in the **rc.Id** slot) into the identifier store address, sets the channel back to normal mode by setting the **rc.Id** slot to *NotProcess.p*, and initializes the reverse channel by setting the **rc.Ptr** slot to *NotProcess.p*.

In order to complete the communication, it is necessary for an input instruction to be subsequently executed.

unmkrc

unmkrc takes as its parameters a channel address in **Areg**. The instruction sets the specified resource channel to normal mode. The value in **Areg** is popped from the stack leaving **Creg** undefined.

unmkrc is a privileged instruction. If the value in **Areg** is not word aligned, then the instruction signals *Unalign*; and if the value is not a valid internal, virtual or event channel address, then it signals *IntegerError*.

If the value held in the slot **rc.Id** is *NotProcess.p*, then the channel is already in normal mode, and no further action is taken. Otherwise, the channel is set to normal mode by assigning *NotProcess.p* to the **rc.Id** slot. If the channel is *idle*, it is set to the *empty* state, and if the channel is *queued* it is set to the *waiting* state.

Note that the resource channel identifier is removed whether or not the channel is on the queue. It is assumed that under normal circumstances, when one resource channel is unmarked, all other resource channels that might be on the queue are also unmarked. This includes any channels that may happen to have made 'claim's while the queue is being unmarked. The communications on unmarked channels must be completed as normal channel communications.

erdsq

erdsq takes as its parameter, a pointer to an RDS in **Areg**. The instruction empties the queue associated with the RDS specified, and pushes the front and back pointers of the queue that is removed on to the integer stack. The value previously in **Breg** is pushed into **Creg**.

erdsq is a privileged instruction. If the value in **Areg** is not word aligned, then the instruction signals *Unalign*.

The instruction loads the value held in the **rds.Front** slot into **Areg** and stores the special value *NotProcess.p* into the slot; and also loads the value held in **rds.Back** into **Breg**.

irdsq

irdsq takes as its parameter, a pointer to an RDS in **Areg**, and pointers to a linked list of resource channels in **Breg** and **Creg**. The instruction concatenates the specified linked list onto the front of the list (if there is one) already associated with the specified RDS. The stack is left undefined after this operation.

irdsq is a privileged instruction. If the value in **Breg** or **Creg** is not word aligned, then the behavior is undefined, and if the value in **Areg** is not word aligned, then the instruction signals *Unalign*.

The front pointer of the new queue is passed to this instruction in **Breg** and is written into the **rds.Front** slot of the RDS specified by **Areg**. If the previous front pointer was *NotProcess.p*, then the resource queue was empty, at the time of execution, and so the back pointer of the new queue, which is passed in **Creg**, is written into the **rds.Back** slot of the RDS. If the resource queue is non-empty at the time of execution, then it is concatenated onto the end of the new queue by setting the **rc.Ptr** slot of the last channel of the new queue (pointed to by **Creg**) to the previous value of the **rds.Front** slot.

ldresptr

ldresptr takes as its parameter in **Areg** the address of a *queued* resource channel. The instruction loads into **Areg**, the address of the next channel in the resource queue. The values in **Breg** and **Creg** are unaffected.

ldresptr is a privileged instruction. If the value in **Areg** is not word aligned, then the instruction signals *Unalign*; and if the value is not a valid internal, virtual or event channel address, then it signals *IntegerError*.

The value held in the **rc.Ptr** slot of the channel specified, is loaded into **Areg**, overwriting that channel address currently stored in that register. If the channel is *queued*, then the address is either that of the next channel on the queue, or is undefined if the channel is the last in the queue. The result is also undefined if the channel is *idle* or if it is in normal mode.

stresptr

stresptr takes as its parameters: the address of a *queued* resource channel in **Areg**, and a new channel address **Breg**. The instruction places the resource channel pointed to by **Breg** behind the resource channel pointed to be **Areg**, in the resource queue. The value in **Creg** is popped into **Areg**, leaving **Breg** and **Creg** undefined.

stresptr is a privileged instruction. If the value in **Areg** is not word aligned, then the instruction signals *Unalign*; and if the value is not a valid internal, virtual or event channel address, then it signals *IntegerError*.

The value contained in **Breg** is written into the **rc.Ptr** slot of the channel specified by **Areg**, overwriting the value currently stored in that location.

enbg

enbg takes as its parameters: in **Areg** – the boolean expression value of a component guard, and in **Breg** – address of an RDS. If the value held in **Areg** is *true*, then the instruction enables the specified resource as part of the alternative sequence described in section 8.7.2. **Areg** is unaffected by this instruction, and **Creg** is popped into **Breg**, leaving **Creg** undefined.

enbg is a privileged instruction. If the value in **Breg** is not word aligned, then the instruction signals *Unalign*.

If **Areg** is *true*, then the resource is enabled as follows. The current process descriptor is written into the **rds.Proc** slot of the RDS. If there is a client already on the queue, then the value *Ready.p* is written into the **pw.State** slot of the current process workspace.

disg

disg takes as its parameters: in **Areg** – an offset from the start of the instruction following the *altend* to the start of the code for that branch of the alternative, in **Breg** – the boolean expression value of a component guard, and in **Creg** – address of an RDS channel address. If the value held in **Areg** is *true*, then the instruction disables the specified resource as part of the alternative sequence described in section 8.7.2. If this component alternative is selected, then **Areg** is loaded with *true*, otherwise it is loaded with *false*. **Creg** is popped into **Breg**, leaving the former undefined.

disg is a privileged instruction. If the value in **Breg** is not word aligned, then the instruction signals *Unalign*.

If the value held in **Breg** is initially *false*, then this is not changed and the component alternative cannot be selected. If **Breg** is *true*, then the resource is disabled as follows. The resource server's process descriptor is read from the **rds.Proc** slot of the RDS. If this isn't a valid process descriptor, then the resource must have already been disabled, and so this component cannot be selected. Otherwise the **rds.Proc** slot is assigned the value *NotProcess.p*. If there are no clients on the queue then this component alternative is not ready and cannot be selected. If there is at least one client in the queue, then this component is ready and available for selection. It is only selected if it is the first ready component to be disabled.

In order to complete the communication if the component is selected, it is necessary for *grant* and an input instruction to be executed in the component branch code.

12.7.5 Usage of resource channels

This section describes the ways in which the resource channel instructions of the T9000 are intended to be used.

Overview of Resource Channels

In summary, resource channels are a means of synchronizing many 'client' processes with a single 'server' process in an efficient way. The processes may be on the same or different transputers. The synchronization is achieved by means of instructions which manipulate data structures in the memory of the IMS T9000 where the server process resides, together with the mechanism of virtual channels.

When a resource channel has both been put into resource mode, i.e. associated with a specific RDS and been output to by a client process, this combination of events is called a 'claim'. The RDS may have a *queue* of resource channels which have performed 'claim's upon it. When a server executes *grant* on the RDS, it obtains the next resource channel to have made a 'claim'. It can then synchronize with (and receive data from) the client which is outputting to that resource channel and interact with it until it wishes to service the next client.†

To increase the flexibility of servers, the whole RDS can be treated as one branch of an alternative. One server can thus service more than one RDS.

General comments about usage of resource instructions

Listed here are some general comments about how instructions must be used with resources.

The client-server model which is described in section 8.8, may or may not have an extra process which controls the connections between the clients and servers. For the purposes of this text, this process is called a 'controller'. (It would typically be part an operating system.)

The instructions *out*, *vout*, *outbyte* and *outword*, which perform a 'claim' operation for the resource on behalf of the channel, are executed by a client. They may also be performed by a server on the reverse channel, but only after the server has received the resource channel identifier of the forward channel via execution of a *grant* instruction. In the latter case, there is no 'claim' operation because the reverse channel is a normal channel (not a resource channel).

The instructions *grant*, *in*, *vin*, *enbg*, *disg*, *mkrc* and *unmkrc* are executed by a server. *in* and *vin* may also be performed by a client, on the reverse channel, but only after the client has performed an output instruction. *mkrc* and *unmkrc* may also be performed by a controller.

This instructions *erdsq*, *irdsq*, *ldresptr* and *stresptr* are expected to be used by a controller‡, for queue inspection and manipulation. The resource queue can be extracted with *erdsq*, examined and/or manipulated with *ldresptr* and *stresptr*, and restored to the resource with *irdsq*. When the queue is taken away from the resource, no *grant*s by the server should be permitted, because the server is assumed not to be waiting for a 'claim' when the queue is returned. This can be readily enforced with a binary semaphore (see section 8.6). *ldresptr* and *stresptr* should only be applied to channels that are in a queue which has been removed from a resource by the *erdsq* instruction. Since *ldresptr* does not return a useful result when the channel is last in the queue, then it should not be applied to such a channel. This can be avoided by comparing the channel address with that of the queue's back pointer prior to execution.

The controller may also want control of which resource channels should be in resource/normal mode and when. For this purpose, the controller would make use of the instructions *mkrc* and *unmkrc.*

The instructions *enbg* and *disg* should be used a part of the alternative sequence described in section 8.7.2. This allows the alternative and resource synchronization mechanisms to be mixed (see section 8.4.2). This would be needed for example where implementing an occam ALT which has lots of channel guards (the resource mechanism can be used for these) together with some timer guards or SKIP guards. The code associated with the guard needs to execute a *grant* instruction and an input instruction in exactly the same way as any other server.

Omniscient Servers

The most straightforward usage of the resource channel mechanism is as a more efficient replacement for a replicated ALT. This is the example illustrated in section 8.8.1. It is repeated below.

† Note that the same resource channel might be associated with different 'RDS's at different times, different client processes might use the same resource channel at different times, and different server processes might use the same RDS at different times..
‡ But is also possible for a server to manipulate the queue on behalf of a controller.

```
PROC Server( []CHAN OF ANY in )
  SEQ
    ...
    WHILE Serving
      ... declarations
      ALT i = 0 FOR SIZE in
        in[i]? Message
          Serve( Message, i )
    ...
:

PROC Client( CHAN OF ANY out )
  ... declarations
  SEQ
    ...
    out! Message
    ...
:

[n]CHAN OF ANY client.to.server:
PAR
  ... declarations
  Server( client.to.server )
  PAR i = 0 FOR n
    Client( client.to.server[i] )
```

The channels `client.to.server` must be resource channels – that is a resource channel data structure must be allocated and initialized. When the process `Server` executes the 'WHILE `Serving`' loop, an RDS must be allocated and initialized, and the resource channels must be set into resource mode by *mkrc*. The server process is able to do this because it knows all the resource channels (it is 'omniscient'). This means that the client processes can be compiled without knowledge of whether they will be served using the alternative or the resource mechanism; this knowledge is restricted to the server. In order to support the full generality of this situation, the server gives each resource channel an identifier from which it can reconstruct the channel (in simple cases it can simply *be* the channel address). It is this resource channel identifier that is delivered to server when it does a *grant* on the RDS. Note, however, that this mechanism will not implement all varieties of replicated ALT; for example it is difficult to use it if there are boolean guards, and it is an absolute restriction that the process `Server` does *not* use any of the other channels `client.to.server[j]`, $j \neq i$.

Inside the loop, instead of enabling and disabling the whole array of channels (as required for the alternative sequence), the server merely has to execute the sequence

grant;
Address(RCidentifier, ChannelAddress);
Input(ChannelAddress, Message);
Serve(Message, RCidentifier);
mkrc

grant returns the channel *RCidentifier*. This is used by *Address* to determine to determine *ChannelAddress*. *Input* uses an input instruction to receive a *Message* from the channel. This is passed to *Serve* which serves the client. Finally *mkrc* returns the channel to resource mode so that the client can be served again in the future.

When `Server` exits the 'WHILE `Serving`' loop, it must put each channel in the the array back into normal mode by applying the *unmkrc* to each one; unless the `client.to.server` channels go out of scope, or it is known that another process is going to use the same RDS with the same set of resource channels (for example if there is an enclosing loop).

Ignorant Servers

An alternative usage of the resource mechanism is to implement servers that do not know who their clients are, nor even how many there may be ('ignorant servers'). In this case the client process must know that

it is going to use a resource channel, and what the address of the RDS is. Provided it is on the same processor as the server, the client can allocate a resource channel for itself, associate it with the RDS by using *mkrc*, and put itself on the queue by outputting. However, the resource channel data structure must be on the same processor as the server and *mkrc* must be executed there, clients on remote processors must rely on an auxiliary process to assist them in establishing their connection to the server. Since we may also wish to allocate the virtual channels dynamically, the most general sequence of events is as follows:

- The client process makes a request to a 'controlling process' on its processor. Note that it can do this by dynamically allocating an internal resource channel and queueing itself on an RDS used by its controller.
- This controller communicates (along a pre-existing virtual channel) with a controlling process on the same processor as the server; the controlling processes each allocate a VLCB and set up the headers thereof so as to form a virtual link.
- The controller on the server's processor executes *mkrc* to link the newly-established input virtual channel with the server's RDS.
- The controller on the client's processor now replies to the client's request by giving it the addresses of the channels (both halves of the virtual link) along which it can communicate with the server.

If the client happens to be on the same processor as the server, the two controlling processes are one and the same, and this process can allocate an internal resource channel for the client. In this situation neither the client nor the server need know where the other is located. In environments which are not completely dynamic, some or all of this sequence of operations can be performed by a compiler so as to reduce or eliminate the need for 'controlling processes'.

Note that it is *illegal* for a controlling process to apply *mkrc* to a *waiting* resource channel (because it cannot be sure that server is not also waiting for a client).

It might also be unwise to apply *unmkrc* to a *queued* channel if the server is ignorant. The problem here is that the although *unmkrc* changes the mode of *queued* channel from resource to normal, it doesn't actually remove it from the queue. It does however remove the client's resource channel identifier from the resource channel data structure, so when the channel reaches the from of the queue, an execution of *grant* by the server will yield a null identifier (*NotProcess.p*). An omniscient server would be able to handle this, but an ignorant server might not.

Robust Servers

Whether a server is omniscient or ignorant, it may be required to make it tolerate the failure of a client. This would certainly be the case if the server were a component of an operating system which communicates with user programs. There are two ways to achieve this:

1. Restrict the way in which clients are allowed to interact with the server so that once a client has made a 'claim' on the server it is guaranteed to complete its interaction correctly. This could be achieved by means of library procedures *etc*.
2. Make the server able to recover from a failure. This would require that the main body of the server always records enough information about its status so that it can be re-started, and any inconsistencies tidied up (close any open files or whatever), and that it has another component (which may be part of an operating system) that notices when the main body has failed (e.g. by timeout) and re-starts it.

Unwanted Clients

In an operating system (o/s) environment it is necessary to be able to remove processes that are part of a failed (or aborted) program. In the case of resources, one solution is for the o/s to inform the server of the identity of the unwanted process(es), and then wait until it comes to the front of the queue. This implies an unbounded wait, however, so instructions (*erdsq*, *ldresptr* etc.) are provided to allow the RDS queue to be scanned for unwanted resource channels, and for them to be unlinked from the queue. This operation

can either be performed by the server itself, or by another process (e.g. part of the o/s). However if it is performed by another process, it is essential that the server is prevented from performing a *grant* on the RDS until the operation is complete. This can be achieved by guarding the RDS with a semaphore which the server must obtain before performing a *grant*. This imposes an overhead on the operation of the server, but permits the processing of the queue to proceed whilst the server is still interacting with a client.

Note that since it is unknown whether the unwanted resource channel is *idle* or *queued*, *unmkrc* must be applied to it before the queue is scanned, otherwise the resource channel can make a 'claim' which can 'slip by' the scanning operation. Having applied *unmkrc*, it might be sensible to determine whether or not it is *idle* or *queued* before it was unmarked. This is achieved either by inspecting the resource channel data structure if the channel is an internal channel, or by executing *ldchstatus* (see section 12.9.3) if the channel is external. Then the resource queue need only be inspected if the channel is *queued*.

If the removal of unwanted clients is to be combined with the second strategy for constructing robust servers, it will be necessary for the server to record the identity of the client which it is serving. It may also be necessary to protect this value with a semaphore in order to ensure atomic access.

12.8 Resetting and stopping a channel

T9000 links have been engineered so that when used within a system of devices connected on a printed circuit board or via a backplane, they are extremely reliable. When used within their specification, the links implement arbitrary numbers of virtual channels as dependably as the memory implements an arbitrary number of variables.

In other circumstances, such as communication between a development system and a target system, or communication via an unreliable interconnect, it is still possible to use transputer links. However, this requires careful programming and the use of special instructions.

Even in an ideal environment it may be necessary to recover the use of correctly functioning virtual channels which are in operation, for example when a distributed operating system kills a collection of communicating processes. Again, this requires careful programming and the use of special instructions.

12.8.1 Dealing with a communication failure

Detecting communication failure

Communication failures will, of necessity, vary from system to system. In a development system connected to a target system, an error in the target's software may prevent a communication with the development system from occurring; in a system where links may be unplugged, communications may start but fail to complete. For many systems, use of timeouts, and checking lengths of messages received is sufficient to detect failure; this normally requires the use of an additional process operating concurrently with the communicating process. The following example uses a timeout to detect the failure of an output:–

```
CHAN OF BOOL completed:
PAR
  SEQ
    c! message
    completed! TRUE
  ALT
    completed? ok
      ... completed communication
    TIME? AFTER timeout
      ... failure detected
```

Having determined that a communication failure has occurred, there are two actions that may be required. The first is to allow the process whose communication has failed to resume execution; the second is to recover use of the channel.

Restarting processes

The example above cannot terminate when a communication failure occurs because the outputting process never completes. The *resetch* instruction permits the process that detected the failure to recover the

descriptor of the outputting process. The outputting process can then be run, enabling it to complete. At the same time, *resetch* resets the channel so that should the communication subsequently complete the outputting process will not be scheduled a second time.

resetch

mnemonic	name
resetch	reset channel

resetch takes as its parameter a channel address in **Areg**. It returns a value in **Areg**, leaving **Breg** and **Creg** unaffected.

This is a privileged instruction. It signals *Unalign* if the channel address is not word aligned. It signals *IntegerError* if the address is not a channel address.

For an internal channel the channel word is set to *NotProcess.p* and the previous value of the channel word is returned in **Areg**.

For a byte-stream channel in byte-stream mode, the corresponding link is reset and restarted. If a communication was in progress the descriptor of the process communicating is returned in **Areg**; otherwise *NotProcess.p* is returned in **Areg**.

For a byte-stream channel not in byte-stream mode, the corresponding link is reset and restarted, and **Areg** is undefined.

A virtual or event channel is reset into the *empty* state. If a communication was in progress the descriptor of the process communicating is returned in **Areg**; otherwise *NotProcess.p* is returned in **Areg**.

The following example shows the two processes that must be run in parallel to allow continuation after the communication failure.

The first process attempts to output the message and then signals to the other process.

Process 1 – outputting process (this must start to execute before the controlling process)

```
SEQ
  c! message
  completed! TRUE
```

The second process either receives the signal or times out. If it times out it resets the channel on which the communication is being attempted. If the communication happens to complete after the timeout but before the the channel is reset, the outputting process will be rescheduled and *resetch* will deliver the value *NotProcess.p*. The process descriptor returned by *resetch* is run unless it is *NotProcess.p*. In either case a signal will then be received from the outputting process. The following calls two procedures which are described below.

Process 2 – controlling process

```
ALT
  completed? ok
    SKIP
  TIME? AFTER timeout
    INT pid:
    SEQ
      resetch( c, pid )
      IF
        pid = NotProcess.p
          SKIP
        TRUE
          run( pid )
      completed? done
        ok := FALSE
```

The procedure `resetch` has two parameters. The first parameter is a value parameter, and is the channel address which is loaded into **Areg** before application of the *resetch* instruction. The second parameter is a variable parameter, and is the value loaded into the integer stack as a result of the *resetch* instruction.

The procedure `run` has one parameter. This is a value parameter, and is the descriptor of the communicating process which is loaded into **Areg** before application of the *runp* instruction.

Note that it is essential that the outputting process executes the output instruction before the second process starts to execute. This ensures that *resetch* is not applied before the communication has been attempted.

Restarting communication

If the processes communicating over an external channel wish to reuse the channel after they have each recovered from a communication failure, they must both apply *resetch* again. Each process must allow sufficient time between its applications of *resetch* to ensure that

(i) the other process has already performed its first reset, and

(ii) the channel has become completely quiescent.

Virtual channels cannot successfully be reused unless the underlying hardware is functioning correctly. For example, if a link has been unplugged, it must be reconnected and restarted.

12.8.2 Recovering the use of a virtual channel which is in operation

In a complex system the use of timeouts, as described above, is complicated as detailed knowledge about the whole system is needed to compute the timeout periods. However, in the absence of communication failure, a distributed operating system can recover a virtual channel which may be in use, without timeouts.

To recover a virtual channel it is not sufficient to put each end of the channel into the *empty* state because there may still be data or acknowledge packets in transit. The *stopch* instruction can be used to ensure that there are no further packets in transit. The channel can then be made ready for reuse.

To know that there are no packets in transit it is only necessary to wait for the output end of the channel to arrive at the state where every packet sent has been acknowledged. Thus it is necessary to ensure that all data packets sent are acknowledged. Consequently, recovering use of the virtual channel involves action by a process at each end. The process at the output end has to wait until every data packet has been acknowledged; the process at the input end has to ensure that they are acknowledged. In each case this is achieved by the process executing a *stopch* instruction.

stopch

mnemonic	name
stopch	stop virtual channel

stopch takes as its parameter a virtual channel address in **Areg**. It leaves the integer and floating–point stacks undefined.

The instruction signals *Unalign* if the channel address is not word aligned, and signals *IntegerError* if the channel address is not a virtual channel address. It is a privileged instruction. It may also deschedule the current process if the channel is an output channel, and so it is a descheduling point.

If the channel is a virtual input channel then any further packets received on the channel are acknowledged, but otherwise ignored, until *resetch* is applied. The process is not descheduled.

If the channel is a virtual output channel, the behavior depends upon whether or not there is an output in progress. If there is no output in progress the instruction completes without taking any further action. Otherwise the process is descheduled. When every packet sent has been acknowledged the process is rescheduled and the channel is left in the *empty* state.

To put the input end of the channel in an *empty* state, the process at the input end has to execute *resetch* after the process at the output end has completed its execution of *stopch*. In the following example this is achieved by means of a communication on another channel.

Process 1 – controlling process used to stop a virtual output channel on processor 1

```
SEQ
  ... establish that output channel is to be stopped
    stopch( channel )              — described below
    completed! 0
```

Process 2 – controlling process used to stop a virtual input channel on processor 2

```
INT ANY:
SEQ
  ... establish that input channel is to be stopped
  stopch( channel )                — described below
  completed? ANY
  resetch( channel, ANY )          — described in section 12.8.1
```

These processes call two procedures that are not defined in the code. `resetch` is described in section 12.8.1. `stopch` is described below.

The procedure `stopch` loads its single value parameter into **Areg** and applies the *stopch* instruction. The `resetch` procedure is the same as before but observe that the variable parameter returned (`ANY`) is not used.

12.9 Channel instructions according to usage

This section groups all instructions that reference channel addresses according to their usage. It includes some instructions that have been described in chapter 8, some instructions that have been described in this chapter, and a few that have not yet been described. Full descriptions are included for instructions that have not been previously described.

12.9.1 Instructions that can be applied to all channels

The following instructions may be applied to all channels except *vin* and *vout* which cannot be applied to byte-stream channels, and *resetch* which cannot be applied to event channels.

mnemonic	name
in	input message
out	output message
outbyte	output byte
outword	output word
vin	variable-length input message
vout	variable-length output message
enbc	enable channel
disc	disable channel
resetch	reset channel
chantype	channel type

These instructions have the following in common.

- They are privileged instructions.

- They have a channel address parameter passed to them via one of the integer stack registers.
- They signal *Unalign* if the channel address is not word aligned.
- They signal *IntegerError* if the channel address does not specify a channel.

in, *out*, *outbyte*, *outword*, *vin*, *vout*, *enbc* and *disc* are described in chapter 8. *resetch* is described in section 12.8.1.

chantype

chantype (*channel type*) takes the channel address in **Areg** and assigns *true* to **Areg** if the channel is internal, or *false* if it is an external channel. **Breg** and **Creg** are unaffected by this operation.

This is a privileged instruction.

12.9.2 Instructions that can be applied to resource channels

All the instructions in this category are applied to resource channels, and so can be applied to internal channels, virtual input channels and event channels.

mnemonic	name
mkrc	mark resource channel
unmkrc	unmark resource channel
ldresptr	load resource queue pointer
stresptr	store resource queue pointer

They have the following in common.

- They are privileged instructions.
- They have a channel address parameter passed to them in the integer stack register A.
- They signal *Unalign* if the channel address is not word aligned.
- They signal *IntegerError* if the channel address does not specify an internal channel, a virtual input channel, or an event channel.

They are described in section 12.7.4.

12.9.3 Instructions that can be applied to external channels

The following instructions may only be applied to external channels.

mnemonic	name
setchmode	set channel mode
ldchstatus	load channel status

These instructions have the following in common.

- They are privileged instructions.
- They have a channel address parameter passed to them in the integer stack register A.
- They signal *Unalign* if the channel address is not word aligned.
- They signal *IntegerError* if the channel address does not specify an external channel.

The channels modes and states referenced below are defined in section 12.3.

setchmode

setchmode has been described in section 12.6.1 for use on a virtual channel. It can also be applied to event and byte-stream channels.

setchmode either activates the event channel specified by the channel address in **Areg** if **Breg** holds the value *true*, or deactivates the channel if it holds the value *false*. **Areg** inherits the value previously in **Creg**, and **Breg** and **Creg** are left undefined after execution. The reverse event channel word (i.e. the word used to store the process descriptor etc. when the event is used to communicate in the opposite direction) is used to store the activation state. If activation has been requested, then *NotProcess.p* is loaded into the reverse channel word. If deactivation has been requested, then *Deactivated.p* is loaded into the reverse channel word.

If the channel address is that of a byte-stream channel, then the processor either starts the physical link associated with that channel address if **Breg** holds the value *true*, or resets the physical link if **Breg** holds the value *false*. The behavior of this instruction does not depend on the mode of the physical link – i.e. it starts/resets channel when it is in either virtual or byte-stream mode.

ldchstatus

ldchstatus takes as its parameter an external channel address in **Areg**. The instruction loads status information into the integer stack. *ldchstatus* is a privileged instruction. The format of the information returned depends on whether the channel is a virtual channel, an event channel or a byte-stream channel.

For a virtual channel and an event channel, the status word returned in **Areg** is as shown in table 12.7. The state is determined by examining the VLCB for a virtual channel, or the channel words for an event channel.

bit number	description
0	activation mode: '1' = activated, '0' = deactivated
1	set if channel stopping
3	channel in resource mode
4	set if packet or acknowledge pending
5	set if schedule pending
31	channel type: '1' = virtual, '0' = event

Table 12.7 Bit settings for virtual/event channel status in **Areg**

The instruction has the following effect when examining a virtual input channel or an event input channel. If the channel is *empty* or *stopping*, then it loads *NotProcess.p* into **Breg** and undefines **Creg**. If the channel is *waiting*, then it loads the process descriptor of the inputting process into **Breg** and undefines **Creg**. If the channel is in resource mode, then it loads the content of **rc.Ptr** slot into **Breg** and the content of **rc.Id** into **Creg**.

The instruction has the following effect when examining a virtual output channel or an event output channel. If the channel is *empty*, then it loads *NotProcess.p* into **Breg** and undefines **Creg**. If the channel is *waiting*, then it loads the process descriptor of the outputting process into **Breg** and undefines **Creg**.

For a byte-stream channel, the status word returned in **Areg** is as shown in table 12.8. The state is established by the VCP.

bit number	description
0	set if error detected (i.e. parity or disconnect)
3	set if parity error detected
4	set if disconnect error detected
31	always 0 (indicates byte-stream channel)

Table 12.8 Bit settings for byte-stream channel status in **Areg**

The instruction has the following effect when examining a byte-stream channel. If the channel is *empty*, then it loads *NotProcess.p* into **Breg** and undefines **Creg**. If the channel is *waiting*, then it loads the process descriptor of the communicating process into **Breg** and undefines **Creg**.

12.9.4 Instructions that can be applied to virtual channels

The following instructions may only be applied to virtual channels.

mnemonic	name
initvlcb	initialize vlcb
sethdr	set virtual channel header
writehdr	write virtual channel header
readhdr	read virtual channel header
insphdr	inspect virtual channel header
readbfr	read buffer pointer from VLCB
swapbfr	swap buffer pointer in VLCB
stopch	stop virtual channel

All these instructions have the following in common.

- They are privileged instructions.
- They have a channel address parameter passed to them in the integer stack register A.

  A VLCB controls the communication of an input and output channel pair. Each of these virtual channels have a different channel address (the input channel has an even address and the output has an odd address). In these instructions, the address of either channel can be used to identify the channel's controlling VLCB.
- They signal *Unalign* if the channel address is not word aligned.
- They signal *IntegerError* if the channel address does not specify a virtual channel.

These instructions are described in section 12.6.1 except *stopch* which is described in section 12.8.2.

13 Process state

This chapter provides detail on state changes that can be caused by various mechanisms of the processor or explicitly by execution of certain instructions. It reviews the various forms of context switch that can occur, including how to access and manipulate process state under certain circumstances; it explains how timeslicing and interruption can be disabled; and it presents the instructions that can be used to manipulate the scheduling and timer lists.

Prior to reading this chapter, the reader should be familiar with chapter 5, and sections 8.2, 8.3 and 8.5 of chapter 8, as well as sections 9.7 and 10.1.

13.1 Context switching

Chapter 5 introduced the registers that completely define the processor's state. Because the transputer is a machine that can run many processes concurrently by time-sharing execution, it is required to save and/or reload the contents of these registers, as the current process is changed. This change of state is known as a context switch.

There are broadly speaking two types of context switch on the IMS T9000. There is a partial context switch, which stores some part of the state. This is used for scheduling/descheduling including timeslicing between processes, and for the trap mechanism. There is also a full context switch, which stores the entire state of the current process. This is used when a low-priority process is interrupted by a high-priority process. The next two sections consider these separately.

13.2 Partial context switch – descheduling and trapping

When a context switch occurs, it is not always necessary or efficient to store the entire process state. A partial context switch occurs when

- The current process is descheduled. This may be because
  - it is waiting to communicate with another process
  - it is waiting on a semaphore
  - it is waiting on a timer
  - it is being timesliced
  - it has executed an instruction that explicitly forces the current process to be stopped – e.g. *stopp*, *endp*
- A trap is being taken or returned from.

In each of these cases, part of the current state is saved, and is often replaced with the contents of a data structure defining the state of another context.

The data saved/reloaded in a context switch due to descheduling are values contained in: the workspace descriptor register, the instruction pointer register, the status register, and the trap-handler register.

The data saved/reloaded in a partial context switch due to the trap/protection mechanism are as for descheduling/scheduling, but in addition include the values contained in the integer stack registers.

A trap-handler or supervisor can optionally save/reload the floating-point and 2D block move state of its subordinate process(es). Special instructions are provided for this.

13.2.1 Descheduling and execution of the next process

When an L-process is descheduled, the instruction pointer and the trap-handler pointer are both saved in data slots immediately below the workspace area of the process being descheduled. Also, the process

status and control bits (see section 5.2) are saved in the **th.Cntl** slot of the THDS. These bits are not saved if a null trap-handler has been specified. Table 13.1 summarizes. The process descriptor (content of **WdescReg**), which uniquely identifies that process, is usually saved somewhere but its exact storage location depends on why the process is being descheduled. For example if the process has descheduled due to communication, the descriptor is stored in the channel word. The descriptor is not saved if the process has been explicitly descheduled with with a *stopp*, *endp*, or *restart* instruction.

Observe that all other state information is lost when a process is descheduled. The points at which a process may be descheduled are called descheduling points and these are listed in section 8.2.2. It is left to the programmer to ensure that no important information is in any of the other state registers when a process is descheduled. In particular, it is important that if an expression is being evaluated across a descheduling point, that any intermediate results are stored in local variables rather than in the registers.

register	old register value
StatusReg	the process status and control bits are saved in the **th.Cntl** slot of the THDS, unless the process has a null trap-handler, in which case, these bits are unsaved
WdescReg	saved somewhere – see above
IptrReg	saved in **pw.Iptr** slot† of process workspace data structure
ThReg (for an L-process)	saved in **pw.TrapHandler** slot† of process workspace data structure
all other state registers	unsaved
† values are not saved when descheduled by *endp* or *restart* instruction	

Table 13.1 State register values when a process is descheduled

To execute an L-process, the workspace pointer is removed from the front of the scheduling list and loaded into the workspace descriptor register. Given the workspace address, the process then determines the instruction pointer and trap-handler of the new process, from the process workspace data structure. The processor then loads the process status and control bits into the status register, and may load the watchpoint registers with the values stored in the THDS. If a null trap-handler has been specified, then the status register is loaded with the default control word. Table 13.2 summarizes.

register	new register value
StatusReg	the process status and control bits are loaded from the **th.Cntl** slot of the THDS – otherwise, unless the process has a null trap-handler, in which case, the status register is loaded with the default control word
WdescReg	the **Wptr** register field is loaded with a new value from the front of the current scheduling list
IptrReg	loaded with content of the **pw.Iptr** slot of the process workspace data structure of new process
ThReg	loaded with **pw.TrapHandler** slot of process workspace data structure
WlReg	if watchpointing is specified in the trap control word, loaded with content of the **th.eWl**
WuReg	if watchpointing is specified in the trap control word, loaded with content of the **th.eWu**

Table 13.2 Register values after loading state for execution of process

13.2.2 Trapping

State storage and retrieval for the trap mechanisms is described in chapter 10. This information is repeated here with more detail in tabular form. Tables 13.3 and 13.4 show the state storage and retrieval for an L-process trap and execution of *tret*. Tables 13.5 and 13.6 show the state storage and retrieval for a P-process trap and execution of *goprot*.

Most traps are taken at the end of the instruction that causes the trap. The state delivered to the trap-handler or supervisor is therefore the state after execution of this instruction. There are some traps for which this is not true, and this is discussed later with other anomalous issues.

L-process trap

register	old register value	new register value
StatusReg	process status and control bits† are saved in the **th.Cntl** slot of the THDS†	default control word
WdescReg	the workspace pointer field of this register is saved in the **th.sWptr** slot of the THDS	the workspace pointer field of this register is loaded with the address of THDS
IptrReg	saved in the **th.sIptr** slot of the THDS‡	loaded with the content of the **th.Iptr** slot of the THDS
Areg	saved in the **th.sAreg** slot of the THDS☆†	trap reason
Breg	saved in the **th.sBreg** slot of the THDS☆†	error type
Creg	saved in the **th.sCreg** slot of the THDS☆†	undefined
ThReg	copied into **Wptr**	*NotProcess.p*
FPstatusReg	not saved	unchanged☆† or restored‡
FPAreg	not saved	unchanged☆† or restored‡
FPBreg	not saved	unchanged☆† or restored‡
FPCreg	not saved	unchanged☆† or restored‡
BMreg0	unchanged☆†	unchanged☆†
BMreg1	unchanged☆†	unchanged☆†
BMreg2	unchanged☆†	unchanged☆†
WIReg	not saved	undefined
WuReg	not saved	undefined
EptrReg	saved in the **th.Eptr** slot of the THDS★	undefined
all other state registers	not saved	undefined

† all bits listed in table 5.5 except the **sb.WtchPntPend** bit – see 'Single-step and watchpoint traps', page 196

‡ if trap is due to floating-point exception, then the content is restored to value that register had prior the floating-point operation that trapped – see section 11.13 and 'Trapping a floating-point exception', page 195

☆ undefined if trap is due to a non-recoverable error – see 'Trapping due to non-floating-point errors – non-restartable', page 195

† undefined when a memory semantics error causes a trap – see 'Memory semantics error', page 196

‡ not saved if trap is due to a recoverable error – see 'Trapping due to non-floating-point errors – restartable', page 195

★ may not point to exactly the right instruction when a memory semantics error causes a trap – see 'Memory semantics error', page 196

Table 13.3 State register values before and after an L-process trap

For the 'trap return' mechanism, the value in **Areg** at the time the instruction is executed, determines whether or not the trapped process is to be restarted. If **Areg** is zero, then it is not restarted, and the next process on the scheduling list is executed. The new register values in this case are those shown in table 13.2. If **Areg** is non-zero, then the trapped state is reloaded as shown in table 13.4.

register	old register value	new register value
StatusReg	any bits overwritten are not saved	process status and control bits† are loaded from the **th.Cntl** slot of the THDS‡
WdescReg	the workspace pointer register field of this register (**Wptr**) is saved in **ThReg**	the workspace pointer field of this register is loaded with the content of the **th.sWptr** slot of the THDS‡
IptrReg	not saved	loaded with the content of the **th.sIptr** slot of the THDS‡
AReg	not saved	loaded with the content of the **th.sAreg** slot of the THDS‡
BReg	not saved	loaded with the content of the **th.sBreg** slot of the THDS‡
CReg	not saved	loaded with the content of the **th.sCreg** slot of the THDS‡
ThReg	not saved	loaded with the content of **Wptr**‡
WIReg	not saved	if watchpointing specified, then loaded with content of the **th.eWl** slot of the THDS‡
WuReg	not saved	if watchpointing specified, then loaded with content of the **th.eWu** slot of the THDS‡
all other state registers	not saved	undefined

† all bits listed in table 5.5 except the **sb.WtchPntPend** bit – see 'Single-step and watchpoint traps', page 196

‡ new register values here are only applicable if **Areg** is 0 when *tret* is executed – otherwise see table 13.2

Table 13.4 State register values before and after a *tret* instruction

P-process trap

register	old register value	new register value
StatusReg	process status and control bits[†] are saved in the **ps.Cntl** slot of the PDS[†]	**sb.IsPprocessBit** is cleared – otherwise as for loading and executing an L-process – see table 13.2
WdescReg	the workspace pointer field of this register is saved in the **ps.sWptr** slot of the PDS	loaded with the content of **WdescStubReg**
IptrReg	saved in the **ps.sIptr** slot of the PDS[‡]	loaded with the content of the **pw.Iptr** slot of the process workspace data structure of the supervisor
Areg	saved in the **ps.sAreg** slot of the PDS[☆†]	trap reason
Breg	saved in the **ps.sBreg** slot of the PDS[☆†]	error type
Creg	saved in the **ps.sCreg** slot of the PDS[☆†]	undefined
ThReg	unaffected	same as old value
FPstatusReg	not saved	unchanged[☆†] or restored[‡]
FPAreg	not saved	unchanged[☆†] or restored[‡]
FPBreg	not saved	unchanged[☆†] or restored[‡]
FPCreg	not saved	unchanged[☆†] or restored[‡]
BMreg0	unchanged[☆†]	unchanged[☆†]
BMreg1	unchanged[☆†]	unchanged[☆†]
BMreg2	unchanged[☆†]	unchanged[☆†]
WlReg	not saved	as for loading and executing an L-process – see table 13.2
WuReg	not saved	as for loading and executing an L-process – see table 13.2
Ereg	copied into **ps.sEreg** slot of PDS[◇]	undefined
Xreg	copied into **ps.sXreg** slot of PDS[◇]	undefined
EptrReg	saved in the **ps.Eptr** slot of the PDS[★]	undefined
PstateReg		
WdescStubReg	copied into **WdescReg**	undefined
all other state registers	not saved	undefined

† all bits listed in table 5.5

‡ if trap is due to floating-point exception, then the content is restored to value that register had prior the floating-point operation that trapped – see section 11.13 and 'Trapping a floating-point exception', page 195

☆ undefined if trap is due to a non-recoverable error – see 'Trapping due to non-floating-point errors – non-restartable', page 195

† undefined when a memory semantics error causes a trap – see 'Memory semantics error', page 196

‡ not saved if trap is due to a recoverable error – see 'Trapping due to non-floating-point errors – restartable', page 195

★ may not point to exactly the right instruction when a memory semantics error causes a trap – see 'Memory semantics error', page 196; and is undefined when a timeslice trap is taken

◇ only saved when trap occurs in the middle of an interruptible instruction

Table 13.5 State register values before and after a supervisor trap

register	old register value	new register value
StatusReg	as for descheduling an L-process – see table 13.1	process status and control bits† are loaded from the **ps.Cntl** slot of the PDS and **sb.IsPprocessBit** is set
WdescReg	saved in **WdescStubReg**	the workspace pointer field of this register is loaded with the content of the **ps.sWptr** slot of the PDS
IptrReg	as for descheduling an L-process – see table 13.1	loaded with the content of the **ps.sIptr** slot of the PDS
AReg	saved in **PstateReg**	loaded with the content of the **ps.sAreg** slot of the PDS
BReg	not saved	loaded with the content of the **ps.sBreg** slot of the PDS
CReg	not saved	loaded with the content of the **ps.sCreg** slot of the PDS
ThReg	unaffected	same as old value
WlReg	not saved	if watchpointing specified, then loaded with content of the **ps.eWl** slot of PDS
WuReg	not saved	if watchpointing specified, then loaded with content of the **ps.eWu** slot of PDS
Ereg	not saved	loaded with the content of the **ps.sEreg** slot of PDS‡
Xreg	not saved	loaded with the content of the **ps.sXreg** slot of PDS‡
EptrReg	not saved	loaded with the content of the **ps.Eptr** slot of the PDS‡
RegionReg0	not saved	loaded with the content of the **pc.RegionReg0** slot of the RDDS
RegionReg1	not saved	loaded with the content of the **pc.RegionReg1** slot of the RDDS
RegionReg2	not saved	loaded with the content of the **pc.RegionReg2** slot of the RDDS
RegionReg3	not saved	loaded with the content of the **pc.RegionReg3** slot of the RDDS
PstateReg	not saved	loaded with the content of **Areg** – address of PDS
WdescStubReg	not saved	loaded with the content of **WdescReg**
all other state registers	not saved	undefined

† all bits listed in table 5.5

‡ only reloaded when restarting an interrupted instruction – see section 13.4

Table 13.6 State register values before and after a *goprot* instruction

Timeslicing from protected mode

Whereas for a low priority L-process, timeslicing can only occur at a timeslicing point, for a low priority P-process, timeslicing can occur at any interrupt point. It is thus possible to ensure that a user task (implemented as a P-process) cannot hog machine time. This is important for untrusted code. A timeslice in protected mode causes a P-process trap. A trap may thus occur after any instruction or indeed in the middle of any interruptible instruction. The state delivered to the supervisor is the current process state at the time the trap is taken.

aside: The integer stack does not have to be preserved when an L-process is descheduled for a timeslice. This is because the points at which a timeslice can occur are restricted. For a P-process, timeslicing is not restricted in this way and so the integer state is stored in the PDS by the trap mechanism.

Trapping a floating-point exception

When a floating-point operation causes a trap, the machine restores the floating-point state to the state as it was *before* the operation. That is, the floating-point stack, the floating-point status register, and the floating-point flags are restored. The remaining (non-floating-point) state delivered to the trap-handler or supervisor, is the state at the time the trap is taken. It is then left to the trap-handler or supervisor to deliver the correct result to an IEEE handler. (See section 11.13.)

Trapping due to non-floating-point errors – non-restartable

In general it is not possible to recover from an instruction that traps due to an error that isn't a floating-point exception (but see 'Trapping due to non-floating-point errors – restartable'). For such traps the contents of the following registers are not saved: integer stack registers, floating-point registers, block move registers and internal registers. But the trap does deliver to the trap-handler or supervisor, the remaining state at the time the trap is taken – i.e. the status bits, the workspace descriptor, the instruction pointer, the error pointer and the trap-handler pointer. If an error occurs during the fetch stage of instruction execution, then the next instruction due to be executed is still the instruction that is being fetched, and so the error pointer and instruction pointer will have the same value.

Trapping due to non-floating-point errors – restartable

For traps caused by certain errors, it is possible to restart the trapped process. When such errors occur, the process state delivered to the supervisor (or trap-handler) is the state as it was *before* the instruction was executed. However the instruction pointer is not saved. The following gives details for this class of error.

- Section 9.3 explains that when instructions *call*, *ajw* or *gajw* are executed in protected mode, *AccessViolation* is signalled if the new workspace pointer is not writable. When the supervisor has taken corrective action, it can re-execute the trapped instruction by restarting the P-process using *goprot*. Before executing *goprot* the supervisor must copy the error instruction pointer, which has been saved in **ps.Eptr**, into **ps.Iptr**.

- Section 9.2 explains that some instructions cannot be executed when running under protection. An attempt to execute any one of these 'privileged' instructions causes *PrivInstruction* to be signalled and a trap to be taken. This enables the supervisor to take corrective action (for example execute the instruction on behalf of the P-process) and to restart the P-process from the instruction following the privileged instruction using *goprot*. Before executing *goprot* the supervisor must copy into **ps.Iptr**, the error instruction pointer, which has been saved in **ps.Eptr**. N.B. When a privileged instruction is trapped, the stored error pointer is the address of the *next* instruction to be executed.

- Similarly if a process attempts to execute an an illegal instruction (i.e. an invalid opcode), then the process traps. This enables the trap-handler (or supervisor) to take corrective action and restart the trapped process from the following instruction using *tret* or *goprot*. Before executing *tret* (or *goprot)* the trap-handler (or supervisor) must add the instruction length (in bytes) to the error instruction pointer, which has been saved in **th.Eptr** (or **ps.Eptr**,) and write the result into **th.Iptr** (or **ps.Iptr**).

Instructions that explicitly cause traps

The instructions *j 0* (breakpoint), *syscall* and *causeerror,* which explicitly force a trap to be taken, deliver to the trap-handler or supervisor, the state *after* the instruction is executed. Since these instructions do no more than take a trap, the only change that these instructions make to the current process state during their execution, is to adjust the instruction pointer register to the address of the next instruction to be executed.

Context traps

If single-step is enabled or a watchpoint occurs when *goprot*, *selth*, or *restart* is executed, then a 'context' trap is taken, as discussed in section 10.3.1. Under these conditions therefore, the state is saved as for 'Instructions that explicitly cause traps' (described above).

Single-step and watchpoint traps

Single-step and watchpoint traps are usually taken between instructions. They therefore deliver the state exactly as it is *after* an instruction has been executed.

An L-process can *only* trap at the end of an instruction†. However, a timeslice may cause a P-process to trap in the middle of an instruction, delivering to the supervisor, the current process state at the time the trap is taken. Hence, if a timeslice is due in the middle of an interruptible instruction, when single-stepping a P-process, then a single-step and timeslice trap is taken (**t.StepTIme** is presented to the supervisor). Similarly if a timeslice is due in the middle of an interruptible instruction when a watchpoint is pending, then a watchpoint and timeslice trap is taken (**t.WatchTIme** is presented to the supervisor). A watchpoint trap is in this case *not* taken at the end of the instruction unless another watchpoint is detected after the instruction resumes execution.

When a watchpoint is detected, the watchpoint trap pending flag (**sb.WtchPntPend**) is set in the status register. When a trap is taken, this flag is cleared before saving to the control word of the THDS or PDS. If a watchpoint is pending when an instruction is interrupted then **sb.WtchPntPend** is saved in the shadow state, and this is reloaded into status register when the instruction resumes. Hence, even if an instruction is interrupted, a pending watchpoint causes a watchpoint trap at the end of that instruction (or at the next timeslice).

Memory semantics error

When a memory semantics error occurs (see page 120), it is not possible for the machine to determine the exact address of the instruction that caused the error. The address that it loads into the error pointer slot of the THDS or PDS points to a location that is near to and not before the error causing instruction, but is not guaranteed to point to the actual location. The instruction pointer register, the integer stack registers, the floating-point registers, the block move registers and the internal registers are not saved. It is not possible to restart a process that has caused a memory semantics error.

13.2.3 Instructions that are used to store and retrieve additional state

A point to note from tables 13.3 and 13.5 is that the floating-point registers and block move registers contain valid state of the trapped process *after* a trap has been taken. These registers are not used to establish a new environment for the trap-handler or supervisor, so there is no need for the mechanism to deliver this state in a data structure. However instructions are provided for a program to store this state in data structures for later use, and complementary instructions are available for reloading the registers with the preserved state. These are listed in table 13.7 (*fpstall* and *fpldall* are introduced in section 11.12.3.). Typically, the trap-handler or the supervisor executes *fpstall* and *stmove2dinit* immediately after a trap. Similarly *fpldall* and *move2dinit* are executed prior to restarting the trapped process. (A convenient place to store this extra state, might be immediately after the THDS or PDS. The programmer is, however, given the freedom to choose the location.)

If it is known that the trapped process does not have any floating-point or block move state, then execution of these instructions may be omitted.

mnemonic	name
fpstall	floating-point store all
fpldall	floating-point load all
stmove2dinit	store *move2dinit* data
move2dinit	initialize data for 2D block move

Table 13.7 Instructions for storing/retrieving extra state

† although it may complete execution of an instruction prematurely if an error is detected

fpstall and *fpldall*

These instructions are described in section 11.12.3.

stmove2dinit

The instruction *stmove2dinit* stores the block move register values into the empty data structure addressed by the pointer in **Areg**. The integer stack is popped leaving **Creg** undefined.

If the content of **Areg** is not word aligned, then *Unalign* is signalled. When executed in a P-process, if any part of the vector is protected, then *AccessViolation* is signalled.

The data structure pointed to by **Areg** should be able to store three words of data. The instruction stores the contents of the block move registers: **BMreg0**, **BMreg1** and **BMreg2** in the three slots of this structure This is summarized in table 13.8.

word offset	slot name	purpose
2	**bmr.DeltaS**	*stmove2dinit* copies the content of 2D block move control register 2 into this slot
1	**bmr.DeltaD**	*stmove2dinit* copies the content of 2D block move control register 1 into this slot
0	**bmr.Count**	*stmove2dinit* copies the content of 2D block move control register 0 into this slot

Table 13.8 Block move data structure

move2dinit

move2dinit is documented in section 7.15.1. Observe that it is complementary to *stmove2dinit* in much the same way that *fpldall* is complementary to *fpstall*. However a data structure is not used. Instead the block move registers are loaded with the contents of each of the integer stack registers. The integer stack is undefined after execution.

an example

For example the following extract of code may be used in a supervisor, where *BMDS_ptr* is a variable holding the address of a three word block move data structure (BMDS) of the form shown in table 13.8, and *FPDS_ptr* is a variable holding the address of a seven word floating-point data structure (FPDS) of the form shown in table 11.26. These and other variables used in this sequence are local variables.

```
ldl BMDS_ptr; ldnl bmr.DeltaS;
ldl BMDS_ptr; ldnl bmr.DeltaD;
ldl BMDS_ptr; ldnl bmr.Count;
move2dinit;

ldl FPDS_ptr;
fpldall;

ldl RDDS_ptr; ldl PDS_ptr;
goprot;
stl TrapReason; stl ErrorType;

ldl FPDS_ptr;
fpstall;

ldl BMDS_ptr;
stmove2dinit
```

Initially this sequence loads the block move state. To achieve this it loads the data in each slot of the BMDS into the integer stack, prior to executing *move2dinit*. It then loads the floating-point state more simply by

loading the address of the FPDS into **Areg** and executing *fpldall*. The RDDS and PDS pointers are loaded into the integer stack as described in chapter 9 prior to executing *goprot* which runs a P-process under protection. When that P-process traps, control returns to the instruction following *goprot*. The trap and error reasons, which are left in the integer stack by the trap mechanism, are saved in local variables. The floating-point and block move state is then saved back into their data structures using *fpstall* and *stmove2dinit* respectively.

13.3 Full context switch – interruption

An interrupt may occur at any interrupt point (see section 8.2.5). When a high priority process interrupts a low priority process, the entire state of the low priority process is saved. The value in each state register is loaded into its shadow register at the time of interrupt. This is known as a full context switch.

The state of the new high priority process is installed according to table 13.2.

When there are no more high priority processes to be executed, the transputer copies the contents of the shadow registers back into the main registers and restarts the interrupted low priority process. The complete state of the interrupted process is reloaded by this operation, and as far as the low priority processes are concerned nothing has changed, unless a high priority process has explicitly changed the shadow state.

Instructions are provided to enable the high priority process to inspect, save, and/or manipulate the values held in these shadow registers. A debugging tool for example may need these instructions to examine the state of the interrupted process. Also an operating system kernel may want to save the interrupted state for reloading later, and in the meantime restart a different process at low priority. More details are given on the latter application in section 13.4.

Instructions for saving and reloading shadow state

The instructions that give access to the shadow registers are listed in table 13.9. *stshadow* makes the shadow state visible by storing it in memory. This may, for example, need to be inspected or saved for later use. *ldshadow* is complementary to *stshadow* in that it loads the shadow registers from memory. This allows the state of the low priority process to be modified, or even a different low priority process to be started on return from interrupt. These instructions are only meaningful when executed in a high priority process.

mnemonic	**name**
ldshadow	load shadow registers
stshadow	store shadow registers

Table 13.9 Shadow register instructions

Figure 13.1 groups the shadow registers into the order in which they are saved with the *stshadow* instruction, and the order in which they are loaded with the *ldshadow* instruction. The shadow register index markers shown in the figure, are used by the instructions to specify which blocks of registers are to be saved or loaded.

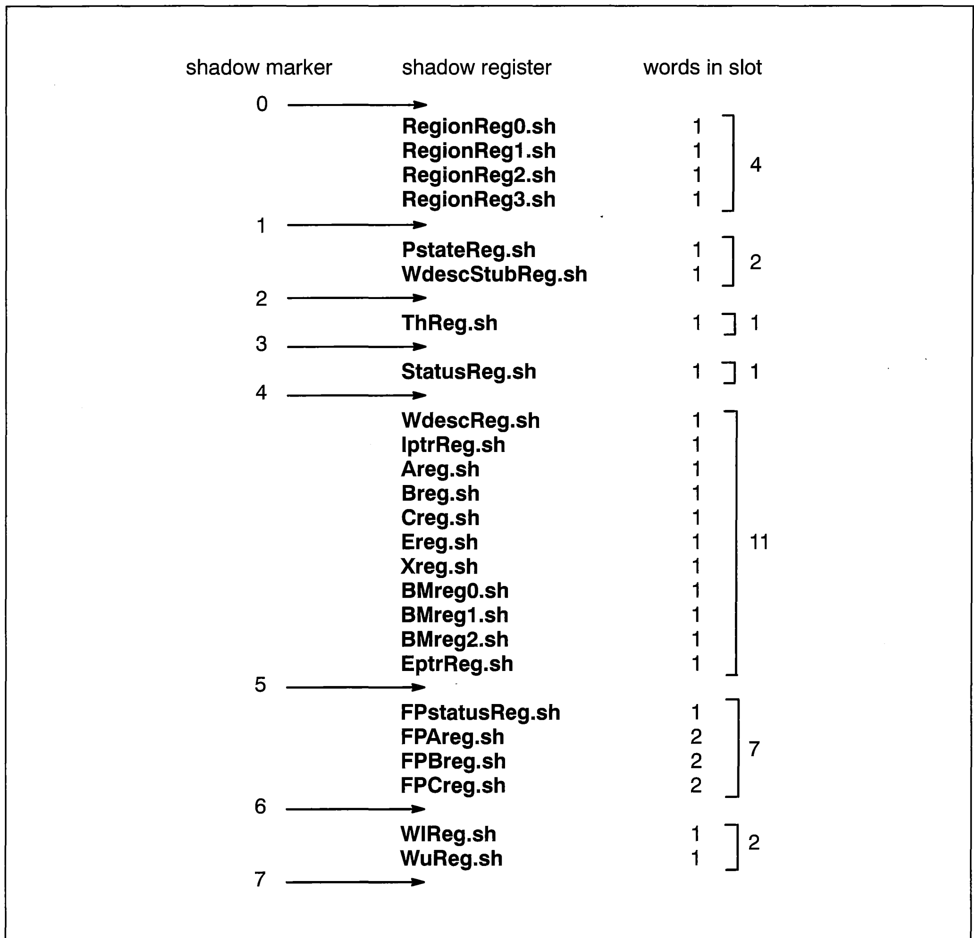


Figure 13.1 Order in which shadow registers are stored

Prior to executing either of these instructions, the integer stack registers are set up as follows.

Areg	marker specifying start of first shadow group to be loaded or stored
Breg	marker specifying end of last shadow group to be loaded or stored
Creg	pointer to the first location to be read from or written to

The range of registers is specified by the shadow markers held in **Areg** and **Breg**. The registers specified by the instructions are those grouped between these markers according to figure 13.1. (Note that the markers are between register groups to avoid ambiguity.) Hence if the value in **Areg** is 5 and the value in **Breg** is 6, then this specifies the floating-point shadow registers.

The shadow register instructions are privileged. Their behavior is undefined when executed by a low priority process. Their behavior is also undefined if the values held in **Areg** and **Breg** do not satisfy the certain criteria – namely, **Areg** should be greater than or equal to 0 and should less than **Breg**, **Breg** should be less than or equal to 7.

stshadow

stshadow stores the contents of the shadow registers specified by the shadow markers held in **Areg** and **Breg,** into the contiguous block of memory beginning at the address in **Creg**. The integer and floating-point stack registers are undefined after execution of *stshadow.*

The contents of the shadow registers are stored into the contiguous block of store that starts at the address stored in **Creg**. Data is stored into this block according to the order shown in figure 13.1. For example if the region descriptor registers are specified, then the content of **RegionReg0.sh** is stored at the first word location in the block, the content of **RegionReg1.sh** is stored at the second word location in the block, etc. The number of words required for storing the data in each register is also shown figure 13.1. Observe that, the floating-point stack registers are double word registers and so consume two word locations each in the data block. All other registers are single word registers.

Where the interrupted process is the null process (i.e. no low priority process had been executing), this in indicated by the workspace descriptor shadow register having the value *NotProcess.p* and the status shadow register having the value 0.

ldshadow

ldshadow loads the contents of the locations within the contiguous block of memory beginning at the address in **Creg,** into the shadow registers specified by the shadow markers held in **Areg** and **Breg**. The integer and floating-point stack registers are undefined after execution of *ldshadow.*

The shadow registers are loaded from the contiguous block of store that starts at the address stored in **Creg**. The order of the data held in this block determines the resultant shadow state, according to figure 13.1. For example if only the watchpoint registers are specified, then **WlReg.sh** is loaded with the content of the first word location in the block, and **WuReg.sh** is loaded with the content of the second word location in the block. The floating-point stack registers are double word registers and so consume two words of data each from the data block. All other registers are single word registers.

If any of the shadow registers **WdescReg.sh**, **ThReg.sh**, **PstateReg.sh**, or **WdescStubReg** is not word aligned, or the priority bit of **WdescReg.sh** or **WdescStubReg.sh** is not '1' (low), then undefined behavior will occur.

13.4 Restarting an interrupted process

It has previously been described how the instructions *tret* and *goprot* are respectively used to restart a trapped L-process and a trapped P-process. This section explains how to restart an interrupted process. If a process that has just been interrupted, is to be restarted, then there is no problem because its state is stored in the shadow registers and will be automatically reloaded when the current high priority process deschedules or terminates. However, if it is required to restart a process that is not the currently interrupted process, then one of the two methods described in this section can achieve this.

Use of *ldshadow*

One way of restarting an interrupted process (either P-process or L-process), is via starting up another high priority process (using *runp*). The job of this process is then to reload the shadow state from the saved data structure of the interrupted process. For this it uses the *ldshadow* instruction. It should then terminate itself (using *stopp*) hence allowing the low priority state to be reloaded by the processor.

This method of reloading interrupted state is in one way more straightforward than the method described next, because the *ldshadow* instruction complements *stshadow.* The saved data need not be manipulated between these two instructions.

The disadvantage is that it is necessary to run an extra process at high priority, to force the transputer to reload the interrupted state from the shadow registers.

Use of *goprot* and *restart*

An alternative way of restarting an interrupted P-process or L-process is by using *goprot* and *restart* (table 13.10) respectively. They use the P-state data structure (PDS), which is described in section 9.7.1.

mnemonic	name
goprot	go protected
restart	restart

Table 13.10 Instructions for restarting interrupted processes

goprot

goprot is fully described in section 9.8. As well a (re)starting a trapped P-process, it can be used to restart an *interrupted* P-process.

Note that when a subsequent trap occurs, control will be returned to the process that executed *goprot*, and the process state will be saved in the P-state data structure specified by that instruction. Thus if it is required to return control to the same supervisor that would have handled traps prior to the interrupt, then this supervisor must execute the *goprot*. A separate code sequence can be used for setting up the restart, but a jump must be made to the supervisor for the actual restart. The following example demonstrates.

```
            .
            .
supervisor: ldl RDDS_ptr    — load pointer to region descriptor data structure – RDDS
            ldl PDS_ptr     — load pointer to P-state data structure – PDS
            goprot          — restart P-process
            .
            .

restarter:  ... set up PDS and RDDS
            ... load WdescStubReg of interrupted process into Areg (from shadow state)
            ldc (–4)        — load workspace pointer mask
            and             — extract workspace pointer from process descriptor
            gajw            — set to supervisor's workspace
            j supervisor    — jump to supervisor
```

The 'restarter' is the code responsible for restarting the interrupted P-process. It firstly loads the PDS with the shadow state for the P-process. It then adjusts the workspace pointer to that of the 'supervisor', and makes a jump to the entry point in the code of the supervisor where the P-process is restarted. *PDS_ptr* and *RDDS_ptr* are local variables that hold pointers to the P-state data structure and region descriptor data structure respectively. The supervisor loads these into the integer stack and restarts the P-process using *goprot*.

restart

The *restart* instruction restarts an interrupted L-process in the same way that *goprot* restarts an interrupted P-process (except that there is no requirement for **Breg** to hold an RDDS pointer because relocation and protection are not applicable to an L-process). The state of the process should therefore be loaded into a PDS in exactly the same way as for a P-process, and a pointer to this should be held in **Areg** when the instruction is executed. Note that this is a secondary use for the PDS because in this case it is used to hold L-process state rather than P-process state.

This instruction saves the current state according to table 13.1. The new state is loaded according to table 13.6 except that **PstateReg**, **WdescStubReg**, and the region descriptor registers are not applicable for restarting an L-process.

restart does not change the value of the trap-handler register, and so an L-process started with *restart* has the same trap-handler as the process that starts it. It may therefore be necessary to install the trap-handler for the interrupted L-process (using *selth*) before starting it with *restart*.

If *restart* triggers a single-step or watchpoint trap, then it does not restart the L-process. The trap delivers in **Areg**, one of the values **t.WatchContext**, **t.StepContext** or **t.StepWatchContext**.

If an interrupted process is to be restarted using one of these instructions, then the high priority process that caused the interruption must extract the state from the shadow registers using *stshadow*. At some later stage, this state must be written into the PDS that is specified in **Areg** by *goprot* or *restart*.

It may be necessary to reload the floating-point and 2D block move state as described earlier (using *fpstall* etc.). This state should be reloaded immediately before execution of *goprot* or *restart* and it should be saved when an interrupt occurs.

13.5 Enabling and disabling interruption and timeslicing, and forcing a timeslice

By default, a low priority process is timesliced (section 8.2.4), and can be interrupted (section 8.2.5). However, timeslicing and interruption can be explicitly disabled.

mnemonic	name
settimeslice	set timeslicing status
timeslice	timeslice
intdis	interrupt disable
intenb	interrupt enable

Table 13.11 Timeslice and interrupt instructions

Instructions used to disable and enable timeslicing and interruption, and for forcing a timeslice, are shown in table 13.11, and described below, but firstly the following general points should be noted.

When the currently executing process enables (or disables) timeslicing, this action also enables (or disables) timeslicing for all the other processes that share the same trap-handler. The timeslice enable/disable state is indicated by the timeslice disable bit (**sb.TimesliceDisabledBit**) in the status register – i.e. when timeslicing has been disabled, this bit is set. This is a process control bit and so gets saved when a trap occurs.

When interruption has ben disabled, *neither* interrupts *nor* timeslices are allowed to occur. Interruption is automatically re-enabled when a process is descheduled. Interruption or timeslicing cannot be disabled while running under protection.

settimeslice

settimeslice can be used both to prevent timeslicing and to allow timeslicing. The value passed in **Areg** specifies which is required. If **Areg** is *false*, timeslicing is disabled, and if **Areg** is *true*, timeslicing is enabled. After execution of the instruction, **Areg** holds a value to indicate whether or not timeslicing was previously enabled (again *false* means disabled and *true* means enabled). **Breg** and **Creg** are unaffected by this instruction.

This instruction is privileged and is only meaningful when run in a low-priority process. If **Areg** does not contain *false* or *true* when it is executed, the behavior is undefined.

If prior to execution, the **sb.TimesliceDisabledBit** was set then **Areg** is loaded with *false* to indicate that timeslicing was disabled. If prior to execution, the **sb.TimesliceDisabledBit** was reset then **Areg** is loaded with *true* to indicate that timeslicing was enabled. If prior to execution, the value *false* was present in **Areg**, then **sb.TimesliceDisabledBit** is set to 1 to indicate that timeslicing is now disabled. If prior to execution, the value *true* was present in **Areg**, then **sb.TimesliceDisabledBit** is set to 0 to indicate that timeslicing is no longer disabled.

timeslice

timeslice forces a timeslice to be taken, regardless of whether timeslicing or interruption is enabled. The instruction requires no stack parameters.

This can be executed from a high or low priority process. It is a descheduling point. A side-effect of this is that when executed from a low priority L-process, it will cause interruption to be enabled (because the current process is descheduled).

When executed from an L-process at either low or high priority, the current process is descheduled and rescheduled, hence placing it at the back of the current priority scheduling list. The integer and floating-point stacks are not preserved. When executed from a P-process at either low or high priority, a trap is taken and a special value is presented to the supervisor in **Areg**, indicating that the reason for the trap was a timeslice.

intdis

intdis disables interruption and timeslicing. It requires no parameters and does not affect the integer stack.

This instruction is privileged and is only meaningful when run in a low-priority L-process.

intenb

intenb enables interruption. It requires no parameters and does not affect the integer stack.

This instruction is privileged and is only meaningful when run in low-priority L-process.

Remember that disabling interruption forces timeslicing to be disabled; but provided that the timeslice disable bit is 0, *intenb* also re-enables timeslicing.

Additional timeslicing considerations

When a low priority L-process is executed with a null trap-handler, timeslicing is enabled by default. It can be explicitly disabled, but if the process is descheduled or trapped, timeslicing will again be enabled when it restarts. This implies that a trap-hander will always start executing with timeslicing enabled, because it must itself have a null trap-handler. But if it subsequently installs a trap-handler for itself (using *selth*) it will inherit the timeslice enable/disable state specified by that trap-handler.

If an interrupt occurs, the entire content of the status register is preserved via its shadow register. Therefore, when the low priority process resumes, timeslicing is as it was when the interrupt occurred (unless explicitly manipulated by a high priority process).

The timeslice clock continues to count while timeslicing is disabled. If a timeslice is due when timeslicing is enabled, a timeslice will be taken at the next timeslicing point (or at the next interrupt point if running under protection). Furthermore, a timeslice trap does not reset the timeslice clock, and so if the timeslice clock indicates that a timeslice is due while executing a P-process, it will still indicate that a timeslice is due after the timeslice trap has been taken. Assuming therefore that it has timeslicing enabled, the supervisor will timeslice at the next timeslicing point. Note however that this may not be the case when a timeslice trap is forced by execution of *timeslice*.

Alternative means of disabling timeslicing

Where a process requires a short sequence of its code to execute without descheduling, the *settimeslice* instruction described above should be used. However in applications that require an entire process to be executed without timeslicing, there is an alternative method.

The trap-handler (or supervisor) can disable timeslicing in a subordinate L-process (or P-process) by setting the timeslice disable bit (**sb.TimesliceDisabledBit**) in the control word prior to executing *tret* (or *goprot*). When the processor executes an L-process (or P-process), it loads the timeslice disable bit from the control word of the process's THDS (or PDS) into the status register.

13.6 Scheduling list and timer list queue manipulation

Before reading this section, the reader should be familiar with sections 8.2.1, 8.3 and 8.5.

As well as the instructions described for initiation and termination of processes (section 8.3), the IMS T9000 provides two instructions for explicit manipulation of the scheduling lists. These provide more control over the scheduling mechanism. This is of particular interest to the operating system programmer.

Similarly, as well as the timer instructions described in section 8.5, the IMS T9000 also provides instructions to manipulate the timer lists and restart the clocks.

It is safe to manipulate either scheduling list or either timer list from either priority. It is usual to manipulate the lists from the same or higher priority. It would not be normal practice for a low priority process to manipulate the high priority list. However, bear in mind that if a high-priority queue is installed by a low priority process, an interrupt will result as soon as the list has been modified (unless interruption has been disabled).

The programmer must be aware that the scheduling or timer lists may asynchronously change as a result of the communication or timer mechanism recognizing ready processes, but it is guaranteed that while a list manipulation instruction is being executed, the list specified by the instruction cannot be changed by these mechanisms. However between instructions, there is the potential for interruption or timeslicing. If a sequence of manipulation instructions is being executed, it may be necessary to disable interruption and timeslicing (using the instructions described in section 13.5). Note however that queues can still change while interruption is disabled.

The instructions that can be used to manipulate the scheduling lists and the timer lists, are shown in table 13.12, and are described below. For each of these instructions, if a specified queue has a front pointer value *NotProcess.p* (the null process), then the queue is empty.

mnemonic	name
swapqueue	swap scheduler queue
insertqueue	insert at front of scheduler queue
swaptimer	swap timer queue
ldtimer	load timer
sttimer	store timer

Table 13.12 Instructions used to manipulate scheduling and timer lists

swapqueue

The instruction *swapqueue* takes the following parameters. **Areg** indicates the priority of the scheduling list to be swapped (0 for high and 1 for low). **Breg** contains a pointer to the workspace of the first process in the new queue. **Creg** contains a pointer to the workspace of the last process in the new queue. After execution, the new queue is installed as the scheduling list for the specified priority and pointers of the old scheduling list are held in **Areg** and **Breg**. **Creg** is left undefined.

This instruction is privileged. If **Areg** initially has a value other than 0 or 1, then the behavior is undefined. Unless **Breg** specifies an empty queue (*NotProcess.p*), the instruction has an undefined effect if the pointers in **Breg** or **Creg** are not word aligned. Note that the queue pointers are guaranteed to be word aligned for all queues generated by the processor.

The front pointer register of the appropriate priority takes the value initially in **Breg**, and **Areg** is assigned the initial value of the front pointer register. Similarly the back pointer register of the appropriate priority takes the value initially in **Creg**, and **Breg** is assigned the initial value of the back pointer register.

insertqueue

The instruction *insertqueue* takes the following parameters. **Areg** indicates the priority of the scheduling list to be extended. **Breg** contains a pointer to the workspace of the first process in the queue to be inserted. **Creg** contains a pointer to the workspace of the last process in the queue to be inserted. After execution, the queue is inserted at the front of the scheduling list for the specified priority. The integer stack is left undefined.

This instruction is privileged. If **Areg** initially has a value other than 0 or 1, then the behavior is undefined. If **Breg** initially contains *NotProcess.p*, then the insert queue is empty and the instruction has no effect. Unless **Breg** specifies an empty queue (*NotProcess.p*), the instruction has an undefined effect if the pointers in **Breg** or **Creg** are not word aligned. Note that the queue pointers are guaranteed to be word aligned for all queues generated by the processor.

The front pointer register of the appropriate priority takes the value initially in **Breg**. If the initial value of the front pointer register was *NotProcess.p*, then the initial scheduling list was empty, and the back pointer register is assigned the initial value of **Creg**. If the initial scheduling list was not empty, then the initial value of the front pointer register is stored in the **pw.Link** slot of the process workspace pointed to by **Creg**, thereby linking the original list on to the end of the newly inserted queue.

swaptimer

The instruction *swaptimer* takes the following parameters. **Areg** indicates the priority of the timer list to be swapped (0 for high and 1 for low). **Breg** contains a pointer to the workspace of the first process in the new queue. After execution, the new queue is installed as the timer list for the specified priority and the front pointer of the old timer list is held in **Areg**. **Breg** and **Creg** are left undefined.

This instruction is privileged. If **Areg** initially has a value other than 0 or 1, then the behavior is undefined. Unless **Breg** specifies an empty queue (*NotProcess.p*), the instruction has an undefined effect if the pointer in **Breg** is not word aligned. Note that queue pointers are guaranteed to be word aligned for all queues generated by the processor.

The front pointer register of the appropriate priority takes the value initially in **Breg**, and **Areg** is assigned the initial value of the front pointer register. The timeout register of the appropriate priority is loaded with the first time on the new list, which is obtained from the **pw.Time** slot of the workspace of the process at the front of the list.

note: The timer mechanism will only work correctly if the timer list is linked in chronological order. This can be guaranteed if the newly installed list has been previously created by the processor using *tin* etc. (section 8.5).

ldtimer

The *ldtimer* instruction reads the time of the current priority clock, where the current priority is the priority of the executing process. The value in the appropriate clock register is pushed into the integer stack.

sttimer

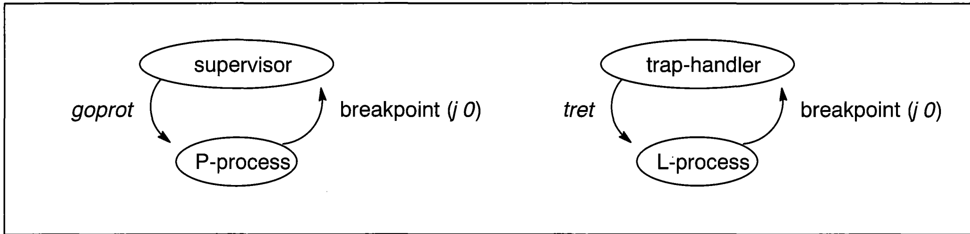
The *sttimer* instruction starts both high and low priority clocks. The value in **Areg** at the time of execution, is assigned to both clock registers. **Areg** is then popped from the integer stack, leaving **Creg** undefined. This is a privileged instruction.

This needs to be executed before any timer operations are executed, and furthermore because the timeslicing mechanism uses these clocks, no timeslicing will occur before it is executed. For these reasons the instruction is usually executed by the bootstrap code. However, execution of this instruction can be repeated, hence resetting the clocks to the value in **Areg**.

14 Debugging mechanisms

Section 10.3.1 introduced seven trap causes. Three of these are specifically provided to support program debugging – namely: breakpoints, watchpoints, and single-stepping. This chapter discusses how each of these are used to monitor both P-processes and L-processes. It also considers some subtleties that occur when a process is due to deschedule or change context at the time of a trap.

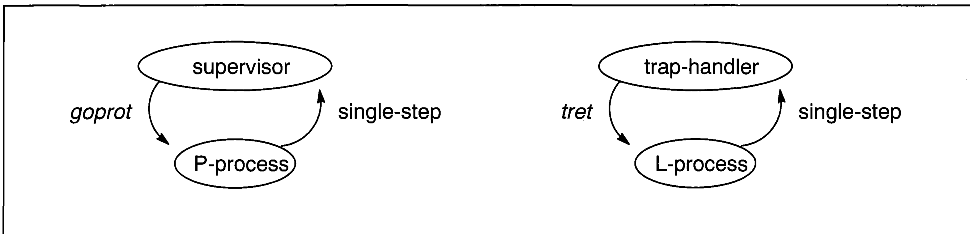
14.1 Breakpoints



A breakpoint can be used to debug a program by deliberately diverting control from the process in which it appears. It is normally inserted by a debugger, in place of an existing instruction, which should be replaced at a later time. A breakpoint instruction is implemented by a dual use of the *jump* instruction. Thus, the otherwise redundant instruction *j 0*, is interpreted as a breakpoint. This is coded in a single byte and hence can always substitute any single instruction. The action is to cause a trap to be taken to its supervisor if the process is a P-process or to its trap-handler if the process is an L-process.

The only other trap that can occur at the same time as a breakpoint trap, is a single-step trap. The trap delivers a reason in **Areg** for the supervisor or trap-handler to inspect. This will be one of the trap reasons **t.Break** or **t.StepBreak** (see table 10.5). It also delivers **et.NoError** to **Breg**.

14.2 Single-stepping



14.2.1 Single-stepping a P-process

A supervisor can single-step a P-process by setting to '1' the single-stepping trap enable bit (**sb.StepBit**) of control word in the PDS for that P-process. Provided this bit is set, when the supervisor executes *goprot* to (re)start the P-process, the latter will trap back to the supervisor when it has executed a single instruction.

When the P-process trap saves the process state, it writes the address of the next instruction to be executed into the **ps.slptr** slot of the PDS. Successive executions of *goprot* by the supervisor thus allow a sequence of instructions to be executed in the P-process.

The trap delivers a reason in **Areg** for the supervisor to inspect. It may be that there is more than one cause. The instruction being single-stepped may have caused an error, may have indicated that a timeslice is due, or may have written to a watchpointed location (see section 14.3); or it could be one of the instructions *syscall*, *j 0* (breakpoint), or *causeerror*, which all force a trap to be taken. If one of more of these trap conditions has been detected as well as the single-step, then these are encoded in the trap reason.

The trap also delivers the error type in **Breg**. For example if an *add* instruction causes an integer overflow, it delivers **t.StepError** to **Areg** and **et.IntegerOverflow** to **Breg** (see tables 10.5 and 10.10). If no error occurs during execution of a single-stepped instruction, the trap delivers the type code **et.NoError**.

14.2.2 Single-stepping an L-process

A trap-handler can single-step an L-process by setting to '1' **sb.StepBit** of control word in the THDS. Provided this bit is set, when the trap-handler executes *tret* to restart the L-process, the latter will trap back to the trap-handler when it has executed a single instruction.

When the L-process trap saves the process state, it writes the address of the next instruction to be executed into the **th.slptr** slot of the THDS. Successive executions of *tret* by the trap-handler thus allow a sequential sequence of instructions to be executed in the L-process.

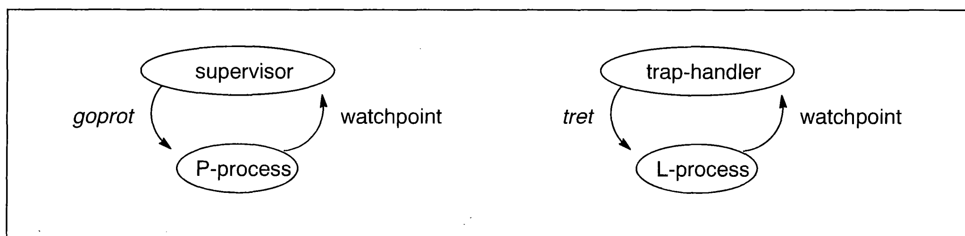
The trap delivers a reason in **Areg** for the trap-handler to inspect. It may be that there is more than one cause. The instruction being single-stepped may have caused an error, or may have written to a watchpointed location (see section 14.3); or it could be one of the instructions *syscall*, *j 0* (breakpoint), or *causeerror*, which all force a trap to be taken. If one of more of these trap conditions has been detected as well as the single-step, then these are encoded in the trap reason. Furthermore, it may be that certain instructions are due to deschedule or context change and so this information is also encoded (this is discussed in section 14.4).

The trap also delivers the error type in **Breg**. For example if an *fpsub* instruction causes floating-point underflow, it delivers **t.StepError** to **Areg** and **et.FPUnderflow** to **Breg** (see tables 10.5 and 10.10). If no error occurs during execution of a single-stepped instruction, the trap delivers the type code **et.NoError**.

14.2.3 Early 'single-step' trap

If single-stepping is enabled then a trap occurring for any other reason, after the single-stepped instruction has started execution, is also coded as a single-step trap. For example if a timeslice trap is taken from a P-process in the middle of an instruction that is being single-stepped, then the reason for the trap is **t.StepTime** (see table 10.5). Provided single-stepping is still enabled when this instruction is restarted, a single-step trap will still be taken when the instruction has finished execution. However if a P-process timeslice trap is taken after the completion of *goprot* but before the first instruction of the P-process starts execution, then this is *not* coded as a 'single-step' trap.

14.3 Watchpoints



A trap-handler or supervisor can monitor a specified region of the address space for write operations made by its subordinate L-process or P-process. This 'watchpointed region' is specified by a lower bound watchpoint register (**WlReg**) and an upper bound watchpoint register (**WuReg**). The addresses in these registers must be word-aligned, and the address in **WuReg** must be greater than or equal to the address in **WlReg**. If the watchpoint trap enable bit (**sb.WtchPntEnbl**) in the status register is set, then any instruction, attempting to write to an address between and including those addresses in these registers, causes a trap to occur. Hence any instruction that performs a write operation can cause a watchpoint trap to be taken.

This section explains how a trap-handler or supervisor can set up this mechanism.

14.3.1 Watchpointing a P-process

The watchpointed region is specified for a P-process by loading **ps.eWu** and **ps.eWl** with the upper and lower addresses respectively. A supervisor can then watchpoint a P-process by setting to '1' the watchpoint trap enable bit of control word in the PDS for that P-process. Provided this bit is set, when the supervisor executes *goprot* to (re)start the P-process, the latter will trap back to the supervisor when an instruction makes a write to the watchpointed region. Because the processor is operating under protection, the addresses in **ps.eWu** and **ps.eWl** are logical addresses, and the specified watchpointed region is a logical address region.

The trap delivers a reason in **Areg** for the supervisor to inspect. It may be that there is more than one cause. The instruction that caused the watchpoint also may have caused an error, may have indicated that a timeslice is due, or may have been single-stepped (see section 14.2). If one of more of these trap conditions has been detected as well as the watchpoint, then these are encoded in the trap reason.

14.3.2 Watchpointing an L-process

The watchpointed region is specified for an L-process by loading **th.eWu** and **th.eWl** with the upper and lower addresses respectively. A trap-handler can then watchpoint an L-process by setting to '1' the watchpoint trap enable bit of control word in the THDS for that L-process. Provided this bit is set, when the trap-handler executes *tret* to (re)start the L-process, the latter will trap back to the trap-handler when an instruction makes a write to the watchpointed region.

Since a number of L-processes can share the same trap-handler, it is possible for the same trap-handler to watchpoint a collection of L-processes.

The trap delivers a reason in **Areg** for the trap-handler to inspect. It may be that there is more than one cause. The instruction that caused the watchpoint also may have caused an error, or may have been single-stepped (see section 14.2). If one of more of these trap conditions has been detected as well as the watchpoint, then these are encoded in the trap reason. Furthermore, it may be that certain instructions are due to deschedule or context change and so this information is also encoded (this is discussed in section 14.4).

When a watchpoint is detected, the trap always occurs after the instruction has been executed, unless one of the other trap conditions has caused a trap to occur earlier than this. For example if a timeslice trap is taken from a P-process in the middle of an instruction, when a watchpoint is pending, then the reason for the trap is **t.WatchTime** (see table 10.5). When the instruction is restarted, there will no longer be a watchpoint pending, and so a watchpoint trap will not occur at the end of the instruction, unless another watchpoint is detected before completion of execution.

The trap also delivers the error type in **Breg**. Even when no error occurs at the same time as a watchpoint, it delivers the type code **et.NoError**.

14.4 Single-stepping and watchpointing an L-process – some special considerations

A trap-handler that is monitoring an L-process using single-stepping or watchpointing, needs to take account of the special behavior of certain instructions. Some instructions cause a process to timeslice (see section 8.2.4), some instructions cause a process to deschedule (see section 8.2.2), and some instructions cause other context swaps. Each of these are considered in this section.

Note that for a P-process, this is not a problem. If a timeslice is due, this causes a trap to be taken at the next interruptible point, but otherwise a P-process does not deschedule or context change.

Timeslicing

If a timeslice is due prior to taking a watchpoint or single-step trap, then it is still due after the trap – i.e. a timeslice is taken at the next timeslice point.

Note that timeslicing is always enabled by the trap mechanism – see section 13.5. If for example there is a timeslice due but timeslicing is disabled, then a trap will re-enable timeslicing and the trap-handler will by default timeslice at the next timeslicing point. If this is not required then the trap-handler must explicitly disable timeslicing before executing a timeslicing point.

Descheduling

When a descheduling point (other than a timeslice point) takes a watchpoint or single-step trap, then this information is encoded into the trap reason (e.g. **t.StepDesch**). The trap-handler may then deal with this accordingly. Typically it will load '1' into **Areg** prior to executing trap return to prevent the process from restarting.

Context changes

When single-stepping or watchpointing an L-process, the instructions *goprot*, *restart* and *selth* must be treated with care, because under normal circumstances these instructions would cause a change of context and so it would be unclear as to where control is transferred when the trap is taken.

For this reason, when *goprot*, *restart* and *selth* cause a single-step or watchpoint trap, these instructions are not executed in the normal way. They simply deliver the state to the current trap-handler, indicating in the trap reason code that a context change was due to occur. It is thus left to the trap-handler to act on this information. Note that a watchpoint trap is caused if the control word of the current trap-handler is being watchpointed.

When an L-process traps while executing *selth*, it traps to the trap-handler that it had before the instruction was executed. The new trap-handler is not selected. If the trap-handler is required to install a new trap-handler on behalf of the trapped process, then it must change the context of that process. It can do this by adjusting the trapped process's workspace data structure and restarting the process. The instruction pointer should therefore be copied from **th.sIptr** to **pw.Iptr** of the workspace data structure, and the new trap-handler should be copied from **th.sAreg** to **pw.TrapHandler**. The address of the workspace data structure is held in **th.sWptr**. This address should then be loaded into **Areg** prior to execution of *runp* which will restart the process with the new trap-handler. Note that this procedure forces the process executing *selth* to be descheduled, whereas this is not necessarily the case when no trap is taken. When the trap-handler executes *tret*, it should ensure that '1' is loaded into **Areg** to prevent the process continuing execution with the original context.

When an L-process traps while executing *goprot*, it traps to the trap-handler that it has before the instruction is executed. The P-process does not start to execute. There are a number of ways that the trap-handler might wish to handle this depending on the full reasons for the trap and the requirements of the code behavior. The following discusses two options that can be implemented for a single-step trap: (i) it might be required to single-step the P-process, (ii) it might be required to let the P-process execute without stepping and for stepping the resume when control has returned to the L-process (the supervisor).

In order to single step the P-process, the trap-handler can be made to act as its supervisor. The address of the PDS for the P-process can be retrieved from **th.sAreg**. **sb.StepBit** should be set to '1' in the control word of the PDS. The address of the PDS should be loaded into **Areg**, and the address of the RDDS (region descriptor data structure) should be loaded into **Breg** from **th.sBreg**. The trap-handler is then ready to execute *goprot*. This can then be repeated to single-step sequential instructions in the P-process as described in section 14.2.1.

In order to bypass single-stepping of the P-process, is necessary to clear **sb.StepBit** in the control word of the THDS. To re-execute *goprot* in the trapped process, the address of the trapped instruction should be specified as the next instruction by copying **th.Eptr** into **th.sIptr**, prior to executing *tret*. In addition to this it is also necessary to provide an indication to the trap-handler of when the P-process traps back to the supervisor, so that single-stepping can be restarted. A breakpoint inserted at the instruction following *goprot* provides this indication.

15 Cache instructions

The performance of a computer system can be improved by providing a memory 'cache'. A cache is a fast access memory separate from main memory. It contains a copy of the data stored in selected locations of main memory, and is designed such that at any time, the contents of the most frequently accessed locations are held in cache. Access to these locations can then be made to the cache rather than to main memory, thus reducing access time.

When a location is accessed for reading or writing, the processor examines the cache to determine whether or not there is a copy of the data for that location currently in the cache. If not this is known as a cache 'miss'. On a miss, the data is read from the external memory location, and if the location being accessed is one that can be associated with the cache – i.e. it is 'cacheable' – then the contents of a fixed area of memory (often called a 'line') containing that location are copied into the cache.

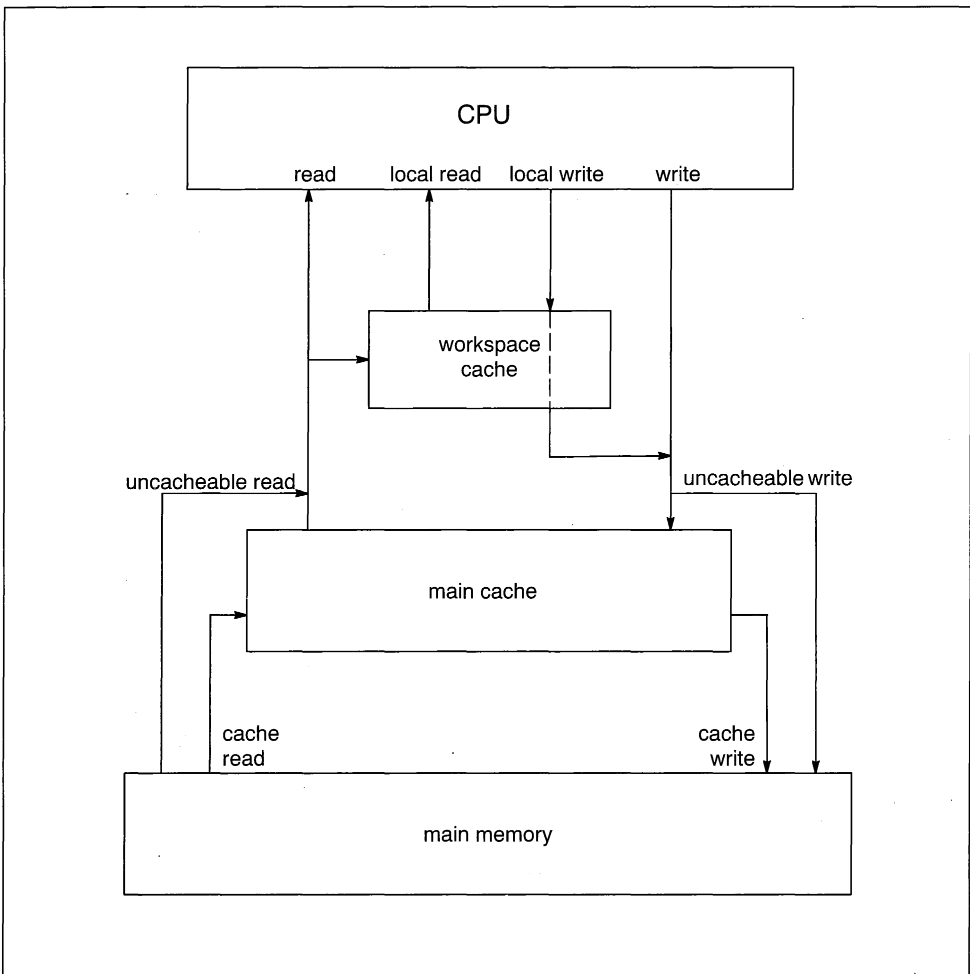


Figure 15.1 Memory architecture

The IMS T9000 has two memory caches: a workspace cache, and a main cache. Figure 15.1 shows the memory architecture. The workspace cache is a 'write-through' cache, whereas the main cache is a 'write-back' cache. This chapter explains these two different schemes, and describes the instructions that are

used to flush and invalidate the main cache. Refer to the *Instruction and data cache* and *Programmable memory interface* chapters of *The T9000 Hardware Reference Manual* for a more detailed presentation of the memory architecture.

15.1 Workspace cache

The workspace cache is used to provide fast access to local variables. It can cache (i.e. hold a copy of) the first 32 words relative to the current workspace pointer (i.e. memory in the range **Wptr** @ 0 to **Wptr** @ 31 – see section A.2.5 for an explanation of this notation). Each cache location is marked as invalid (i.e. empty) until an access is made to the associated memory location. When the workspace is adjusted, all new locations in the 32 word range are marked as invalid.

If the memory location being read is within the first 32 words in the local workspace then *ldl* operates as follows:–

- If the data for the specified location is not already cached, it copies the data to the workspace cache and marks this data as valid, before loading into the integer stack.
- If the data is already cached it loads directly from the workspace cache.

When the processor writes a whole word (e.g. *stl, stnl*, *move*) to any of the first 32 words in the local workspace, it writes this data to both the workspace cache and the main cache / main memory. This data is marked as valid in the workspace cache after this operation. The workspace cache is thus *always* consistent with main memory, hence whenever data is read from either main memory or cache it is known that this data will be correct. This is known as a 'write-through' cache.

A location that is cached by the workspace cache becomes uncached and its associated cache slot is marked as invalid, when an instruction (*sb* or *ss*) writes only part of a word.

Note that the load instructions *ldnl*, *lb*, *lbx*, *ls*, *lsx* always load from main cache / main memory – i.e. they do not read data from the workspace cache.

15.2 Main Cache

The programmable memory interface (PMI) specifies which parts of memory can be cached by the main cache.

The main cache comprises a collection of cache 'lines', where each line holds data for 4 contiguous word locations. A cache line comprises (i) a 4-word aligned physical address specifying the 4-word block represented by the line and (ii) data for those 4 words.

When a location is accessed (for either reading or writing), the processor searches the main cache. If it finds a line specifying the location's address, then the access is made to the cache rather than main memory. If it does not find a valid cache line for that location – i.e. the processor performs a cache 'miss' – then provided that location is cacheable, it copies the contents of the 4-word aligned block that contains the accessed location from main memory into an invalid cache line, which is then marked as valid. This and subsequent memory accesses to locations with addresses within this 4-word block are then performed to/from the cache rather than the main memory, as long as the line remains valid. After marking a new line as valid, the processor invalidates one line of the cache (chosen at random), ensuring that there is always at least one empty cache line for the next cache miss. When the cache has just been set up, there is guaranteed to be one cache line, chosen at random, marked as invalid.

When the processor writes to a cached location, it writes the new data to the appropriate cache line, but does not immediately record the change in main memory. Therefore unlike the workspace cache, the cache data can be inconsistent with main memory. A cache line with such data is known as 'dirty'. If a cache is dirty when it is being invalidated, then the processor writes its data back to main memory (although explicit invalidation using *ica* or *icl* – see later – does not cause write-back). This type of cache is known as a 'write-back' cache. Note that provided the host processor is the only machine accessing

memory, it will never access incorrect data. If the memory is already cached then it will read from the cache, which is guaranteed to hold the most recent data. If the memory is not cached, then it either will have never been cached, or will have been written back with the most recent data at the last invalidation.

15.3 Instructions

Where main memory is read by an external device (e.g. a DMA controller or another processor), the programmer must ensure that the main memory data is consistent with the cache data. Since the main cache is write-back rather than write-through, this cannot automatically be guaranteed. The IMS T9000 provides cache flushing instructions for this purpose.

Similarly, if an external device changes main memory, this is not automatically reflected in the workspace or main cache. The programmer must therefore be aware that any such external memory change may cause the information in the caches to be out-of-date. The IMS T9000 provides cache invalidation instructions, which can be used to ensure that the next read from specified locations is not taken from the main cache but is read from external memory. However there are no such instructions for the workspace cache. Hence to ensure that this never causes a problem, the local workspace should never be placed in an area of memory that is likely to be modified by an external device – irrespective of whether this memory is cacheable.

The instructions listed in table 15.1 operate on the main cache rather than the workspace cache. They are provided to support the use of shared external memory.

mnemonic	**name**
ica	invalidate cache address
icl	invalidate cache line
fdca	flush dirty cache address
fdcl	flush dirty cache line

Table 15.1 Cache instructions

The only times that a cache line gets written back to main memory, are either when a line is replaced following a cache miss, or by explicit execution of *fdca* or *fdcl*. These instructions do *not* invalidate cache lines.

If there is a valid cache line in the main cache that holds data for the address specified in **Areg** and this line is dirty, then *fdca* writes this cache line back to main memory and marks it as not dirty. **Areg** is incremented by *BytesPerLine* making this instruction convenient for sequential use where a contiguous block of memory needs to be flushed. **Breg** and **Creg** are unaffected. Note that when a P-process executes *fdca*, it interprets the content of **Areg** as a logical address.

If the cache line specified in **Areg** is valid and dirty and the address associated with that line is in the range of addresses specified by **Breg** to **Creg**, then *fdcl* writes it back to main memory and marks it as not dirty. **Areg** is incremented by one, making this instruction convenient for sequential use where a number of lines need to be flushed. This is a privileged instruction. **Breg** and **Creg** are unaffected. The cache line specified should be in the range 0..1023, unless half of the cache is configured as internal RAM in which case the line should be in the range 512..1023.

When only a small area of memory† needs to be flushed, this can be achieved by applying a short sequence of *fdca* instructions – one application for each block of four words. Conversely when a large area of memory needs to be flushed, it is more efficient to apply a sequence of *fdcl* instructions. This instruction would normally be used in a loop incrementing the line in **Areg** from 0 through to 1023, thus ensuring that all memory addresses in the specified range are consistent with the data in the cache. In some circumstances it may be desirable to flush just part of the cache. For example if half of the cache lines are configured as internal memory, only those relating to external memory should be flushed. Also for flushing a very large range of memory, it may be better to flush this by parts to avoid consuming a large amount of time doing the flush in one go.

† As a rule of thumb, if a contiguous block of data that is less than 4000 words is to be flushed, then a sequence of *fdca* instructions should be used in preference to a sequence of *fdcl* instructions.

Under ongoing operating conditions, there will normally be one and only one invalid line in the main cache. However a program can explicitly invalidate cache lines using one of the instructions *ica* or *icl*. Be aware that these instructions do *not* write dirty lines back to main memory before invalidating. It is therefore important that if variables are shared between processes, these should be explicitly flushed (using *fdcl* or *fdca*) prior to being explicitly invalidated. This is because when one process invalidates a cache line, it cannot be certain that another process has not made a write prior to the invalidation.

If there is a cache line in the main cache that holds data for the address specified in **Areg**, then *ica* invalidates this cache line. **Areg** is incremented by *BytesPerLine*, making this instruction convenient for repeated use where a contiguous block a memory needs to be invalidated. **Breg** and **Creg** are unaffected.

icl invalidates the cache line specified in **Areg**, if the address associated with that line is in the range of addresses specified by **Breg** to **Creg**. **Areg** is incremented by one making this instruction convenient for repeated use where a number of lines need to be invalidated. **Breg** and **Creg** are unaffected. The cache line specified should be in the range 0..1023, unless half of the cache is configured as internal RAM in which case the line should be the range 512..1023.

When only a small area‡ of memory needs to be invalidated, this can be achieved by applying a short sequence of *ica* instructions – one application for each block of four words. Conversely when a large area of memory needs to be invalidated, it is more efficient to apply a sequence *icl* instructions. These instructions would normally be used in a loop incrementing the line in **Areg** from 0 through to 1023, thus ensuring that all memory addresses in the specified range are uncached. In some circumstances it may be desirable to invalidate just part of the cache. For example if half of the cache lines are configured as internal memory, only those relating to external memory should be invalidated. Also for invalidating a very large range of memory, it may be better to invalidate this by parts to avoid consuming a large amount of time doing the invalidate in one go.

‡ As a rule of thumb, if a contiguous block of data that is less than 4000 words is to be invalidated, then a sequence of *ica* instructions should be used in preference to a sequence of *icl* instructions.

A T9000 instruction set reference guide

A.1 Introduction

This reference section provides a summary of the T9000 instruction set. The instructions are listed in alphabetical order, one to a page. Each page has the instruction mnemonic and full name at the top and then the following categories of information:

- **Code:** the instruction code;
- **Description:** a brief summary of the purpose and behavior of the instruction;
- **Definition:** a more complete description of the instruction, using the notation described below in section A.2;
- **Error signals:** a list of errors and other signals which can occur;
- **Comments:** a list of other important features of the instruction;
- **See chapter:** a reference to the chapter in the main body of this book where the use of the instruction is described;
- **See also:** for some instructions, a cross reference is provided to other instructions with a related function.

These categories are explained in more detail below, using the *add* instruction as an example.

A.1.1 Instruction name

The header at the top of each page shows the instruction mnemonic and, on the right, the full name of the instruction. For primary instructions the mnemonic is followed by '*n*' to indicate the operand to the instruction — this is used in the description to show how the operand is used.

A.1.2 Code

For secondary instructions the instruction op-code is shown as the 'memory code' — the actual bytes, including any prefixes etc., which are stored in memory. The value is given as a sequence of bytes in hexadecimal, most significant byte first. The codes are stored in memory in 'little-endian' format — with the least significant byte at the lowest address.

For primary instructions the code stored in memory is determined partly by the value of the operand to the instruction. In this case the op-code is shown as 'Function *x*' where *x* is the function code in the last byte of the instruction. For example, *adc* (*add constant*) is shown as 'Function 8'. See Chapter 6 for more details of instruction and operand encoding.

Example

The entry for the *add* instruction is:

Code: F5

A.1.3 Description

The description section provides an indication of the purpose of the instruction as well as a summary of the behavior. This includes the meaning of the values in the registers which are used as parameters and results of the instruction.

Example

The *add* instruction contains the following description:

Description: Add **Areg** and **Breg**.

A.1.4 Definition

The 'definition' section is intended to provide a more complete description of the behavior of the instruction. The behavior is defined in terms of its effect on the state of the processor (i.e. the values in registers and memory before and after the instruction has executed).

The effects of the instruction on registers, etc. are given as relationships of the following form:

`register'` ← *expression involving registers, etc.*

Where primed names (e.g. `Areg'`) represent values after instruction execution, while unprimed names represent values when instruction execution starts. For example, `Areg` represents the value in **Areg** before the execution of the instruction while `Areg'` represents the value in **Areg** afterwards. So, the example above states that the register on the left hand side becomes equal to the value of the expression on the right hand side after the instruction has been executed.

The description is written with the main function of the instruction stated first (e.g. the main function of the *add* instruction is to put the sum of **Areg** and **Breg** into **Areg**). This is followed by the other effects of the instruction (e.g. popping the stack). There is no temporal ordering implied by the order in which the statements are written.

The notation is described more fully below, in section A.2.

Example

The *add* instruction contains the following description:

Definition:

`Areg'` ← `Breg` $+_{checked}$ `Areg`

`Breg'` ← `Creg`
`Creg'` ← *undefined*

This says that the integer stack is popped and **Areg** assigned the sum of the values that were initially in **Breg** and **Areg**. After the instruction has executed **Breg** contains the value that was originally in **Creg**, and **Creg** is undefined.

A.1.5 Error signals

This section lists the errors and other exceptional conditions that can be signalled by the instruction. This only indicates the error signal, not the action that will be taken by the processor – this will depend on the the trap enable bits which are set, the value in the trap handler register, etc. The effects of these signals are fully explained in chapter 10. The order of the error signals listed is significant in that if a particular error is signalled then errors later in the list may not be signalled. The errors that may be signalled are as follows:

AccessViolation indicates that an attempt was made to access a non-existent or protected memory or device address. Can only occur in P-processes.

IntegerError indicates a variety of general errors such as a value out of range and misuse of channels.

IntegerOverflow indicates that an overflow occurred during an integer arithmetic operation.

PrivInstruction indicates that an attempt was made to execute a privileged instruction. Can only occur in P-processes.

Unalign indicates that an access to an incorrectly aligned data object was attempted.

FPDivideByZero, *FPInexact*, *FPInvalidOp*, *FPOverflow*, and *FPUnderflow* — these floating point errors correspond directly to the exceptional events specified in IEEE standard 754-1985: "divide by zero", "inexact result", "invalid operation", "overflow", and "underflow" respectively.

FPError indicates a general floating point error — it is signalled by a not-a-number (NaN) or an infinity as an operand to the instruction or whenever *FPDivideByZero*, *FPInvalidOp* or *FPOverflow* are signalled. This provides a simpler error handling mechanism and compatibility with the T800 series of transputers.

Example

As an example, the error signals listed for the *add* instruction are:

Error signals:
IntegerOverflow can be caused by $+_{checked}$

So, the only error that can be caused by *add* is an integer overflow during the addition of **Areg** and **Breg**.

A.1.6 Comments

This section is used for listing other information about the instructions that may be of interest. Firstly, there is an indication of the type of the instruction. These are:

"Primary instruction" — indicates one of the 13 functions which are directly encoded in a single byte instruction.

"Secondary instruction" — indicates an instruction which is encoded using *opr.*

Then there is information concerning the scheduling of the process:

"Instruction is a descheduling point" — an L-process may be descheduled after executing this instruction.

"Instruction is a timeslicing point" — an L-process may be timesliced after executing this instruction.

"Instruction is interruptible" — the execution of this instruction may be interrupted by a high priority process.

"Instruction is privileged" — the instruction cannot be executed when running under protection.

This section also describes any situations where the operation of the instruction is undefined or invalid.

Example

Using the *add* instruction as an example again, the comments listed are:

Comments:
Secondary instruction

This says that *add* is a secondary instruction (it is encoded as *operate 5*).

A.2 Notation

The following sections gives a full description of the notation used in the 'definition' section of the instruction descriptions.

A.2.1 The transputer state

The transputer state consists of the registers (mainly **Areg**, **Breg**, **Creg**, **IptrReg**, **WdescReg**, **FPAreg**, **FPBreg**, and **FPCreg**), the contents of memory, and various flags and special registers (such as the error flags, process queue pointers, clock registers, etc.). See chapter 5 for more information about process state.

The two names `Wptr` and `WdescReg`, in the description, represent different values from the same T9000 register, **WdescReg**. `Wptr` is used for the address of the process workspace — this address is word aligned and therefore has the two least significant bits set to zero. `WdescReg` is used for the 'process descriptor' — the value that is actually held in the workspace descriptor register. This value is composed of

the workspace address and the process priority, stored in bit 0 of the word. Bit 0 is set to 0 for high priority processes and is set to 1 for low priority processes. Bit 1 of the process descriptor is always 0.

The floating point rounding mode is represented by a variable called `RoundMode`.

A.2.2 General

The instruction descriptions are not intended to describe the way the instructions are implemented, but only their effect on the state of the processor. So, for example, the block move instructions are described in terms of a sequence of *byte* reads and writes even though the instructions are implemented to perform the minimum number of *word* reads and writes.

Comments (in *italics*) are used to both clarify the description and to describe things that cannot easily be represented by the notation used here; e.g. *start next process*. These actions may be performed in another subsystem in the device such as the VCP or the scheduler and so any changes to machine state are not necessarily completely synchronized with the execution of the instruction (as the different subsystems work independently and in parallel).

Ellipses are used to show a range of values; e.g. '`i = 0..31`' means that `i` has values from 0 to 31, inclusive.

Subscripts are used to indicate particular bits in a word; e.g. $\mathtt{Areg}_{\mathtt{i}}$ for bit `i` of `Areg`; and $\mathtt{Areg}_{\mathtt{0..7}}$ for the least significant byte of `Areg`. Note that bit 0 is the least significant bit in a word, and bit 31 is the most significant bit.

Instruction pointer

Generally, if the description does not mention the state of a register or memory location after the instruction, then the value will not be changed by the instruction.

One exception to this general rule is **IptrReg**, which is assigned the address of the next instruction in the code *before* every instruction execution starts. The **IptrReg** is included in the description only when it is *directly* affected by the instruction (e.g. in the *jump* instruction). In these cases the address of the next instruction is indicated by the comment "*next instruction*".

Scheduling operations

Some registers, such as the timer and scheduling list pointers, and some special workspace locations can be changed at any time by scheduling operations. Changes to these are included in the description only when they are *directly* caused by the instruction, and not just as an effect of any scheduling operation which might take place.

A.2.3 Undefined values

Many instructions leave the contents of a register or memory location in an undefined state. This means that the value of the location may be changed by the instruction, but the new value cannot be easily defined, or is not a meaningful result of the instruction. For example, when the integer stack is popped, **Creg** becomes undefined, i.e. it does not contain any meaningful data. An undefined value is represented by the name *undefined*. The values of registers which become undefined by an instruction are implementation dependent and are not guaranteed to be the same on different transputer versions, or on future implementations of the T9000.

A.2.4 Data types

The transputer instruction set includes operations on four sizes of data: 8, 16, 32 and 64-bit objects. 8-bit and 16-bit data can represent signed or unsigned integers; 32-bit data can represent addresses, signed or unsigned integers, or single length floating point numbers; and 64-bit data can represent signed or unsigned integers, or double length floating point values. Normally it is clear from the context (e.g. from the

operators used) whether a particular object represents a signed, unsigned or floating point number. A subscripted label is added (e.g. $\texttt{Areg}_{unsigned}$) to clarify where necessary.

A.2.5 Representing memory

The transputer memory is represented by arrays of each data type. These are indexed by a value representing a byte address. Access to the four data types is represented in the instruction descriptions in the following way:

- `byte[`*address*`]` references a byte in memory at the given address
- `sixteen[`*address*`]` references a 16-bit object in memory
- `word[`*address*`]` references a 32-bit word in memory
- `double[`*address*`]` references a 64-bit object in memory

For all of these ,the state of the machine referenced is that *before* the instruction if the function is used without a prime (e.g. `word[ ]`), and that *after* the instruction if the function is used with a prime (e.g. `word'[ ]`).

For example, writing a value given by an expression, *expr*, to the word in memory at address *addr* is represented by:

`word'[`*addr*`]` ← *expr*

and reading a word from a memory location is achieved by:

`Areg'` ← `word[`*addr*`]`

Writing to memory in any of these ways will update the contents of memory, and these updates will be consistently visible to the other representations of the memory – i.e. writing a byte at address 0 will modify the least significant byte of the word at address 0.

Data alignment

Each of these data items have restrictions on their alignment in memory. Byte values can be accessed at any byte address, i.e. they are byte aligned. 16-bit objects can only be accessed at even byte addresses, i.e. the least significant bit of the address must be 0. 32-bit and 64-bit objects must be word aligned, i.e. the 2 least significant bits of the address must be zero.

Unalign may be signalled by an instruction accessing an object with the wrong alignment.

Address calculation

An address identifies a particular byte in memory. Addresses are frequently calculated from a base address and an offset. For different instructions the offset may be given in units of bytes, words or double words depending on the data type being accessed. In order to calculate the address of the data, the offset must be converted to a byte offset before being added to the base address. This is done by multiplying the offset by the number of bytes in the particular units being used. So, for example, a word offset is converted to a byte offset by multiplying it by the number of bytes in a word (4 in the case of the T9000).

As there are many accesses to memory at word offsets, a shorthand notation is used to represent the calculation of a word address. The notation *register* `@` *x* is used to represent an address which is offset by *x* words ($4 \times x$ bytes) from *register*. For example, in the specification of *load non-local* there is:

`Areg'` ← `word[Areg @ n]`

Here, `Areg` is loaded with the contents of the word that is n words from the address pointed to by **Areg** (i.e. **Areg** + 4 × n), where n is the operand to the instruction —e.g. *ldnl n*. As another example, the specification of *fpldnldbi (floating point load non-local double indexed)* instruction includes the line:

`FPAreg'` ← `double[Areg @ (2 × Breg)]`

In this case, `FPAreg` is loaded with the 64-bit value that is offset from **Areg** by the number of double words in **Breg** — i.e. at address **Areg** + (8 × **Breg**).

In all cases, if the given base address has the correct alignment then any offset used will also give a correctly aligned address.

A.2.6 The configuration subsystem

The configuration registers are also represented as an array called `ConfigReg`. An error is caused if an access is made to an illegal configuration register address. An error is also signalled if the CPU attempts to write (using *stconf*) to a register which has been locked.

In addition, some of the commonly used configuration registers are referenced directly. These are:

`ExternalRCBase`

`HdrAreaBase`

`MemStart`

A.2.7 Constants

A number of data structures have been defined in this book. Each comprises a number of data slots that are referenced by name in the text and the following instructions descriptions. The complete set of these data structures is repeated below for convenience.

word offset	slot name	purpose
0	**le.Index**	contains the loop control variable
1	**le.Count**	contains number of iterations left to perform

Table A.1 Loop end data structure

word offset	slot name	purpose
0	**pw.Temp**	slot used by some instructions for storing temporary values
–1	**pw.Iptr**	the instruction pointer of a descheduled process
–2	**pw.Link**	the address of the workspace of the next process in scheduling list
	pw.Count	message length in variable length communication
–3	**pw.TrapHandler**	pointer to trap-handler data structure (THDS)
–4	**pw.Pointer**	saved pointer to communication data area
	pw.State	saved alternative state
	pw.Length	length of message received in variable length communication
–5	**pw.TLink**	address of the workspace of the next process on the timer list
–6	**pw.Time**	time that a process on a timer list is waiting for

Table A.2 Word offsets and names for data slots in a L-process workspace

word offset	slot name	purpose
1	**pp.Count**	contains unsigned count of parallel processes
0	**pp.IptrSucc**	contains pointer to first instruction of successor process

Table A.3 Parallel process data structure

word offset	slot name	purpose
2	**s.Back**	back of waiting queue
1	**s.Front**	front of waiting queue
0	**s.Count**	number of extra processes that the semaphore will allow to continue running on a *wait* request

Table A.4 Word offsets and names for data slots in a semaphore data structure

word offset	slot name	purpose	initial value
2	**rds.Back**	pointer to back of resource channel queue	any
1	**rds.Front**	pointer to front of resource channel queue	*NotProcess.p*
0	**rds.Proc**	process descriptor of server	*NotProcess.p*

Table A.5 Resource data structure (RDS)

word offset	slot name	purpose	initial value
1	**rc.Id**	resource channel identifier / mode indicator	*NotProcess.p*
0	**rc.Ptr**	pointer to RDS or next resource channel	any

Table A.6 Resource channel data structure

word offset	slot name	purpose
10	**ps.sXreg**	internal state – loaded into / stored from **Xreg** when protected mode entered/exited in the middle of executing an interruptible instruction
9	**ps.sEreg**	internal state – loaded into / stored from **Ereg** when protected mode entered/exited in the middle of executing an interruptible instruction
8	**ps.sCreg**	P-process C register – loaded into / stored from integer stack C register when protected mode entered/exited
7	**ps.sBreg**	P-process B register – loaded into / stored from integer stack B register when protected mode entered/exited
6	**ps.sAreg**	P-process A register – loaded into / stored from integer stack A register when protected mode entered/exited
5	**ps.sIptr**	P-process instruction pointer – loaded into / stored from instruction pointer register when protected mode entered/exited
4	**ps.sWptr**	P-process workspace pointer – loaded into / stored from **Wptr** when protected mode entered/exited
3	**ps.eWu**	upper bound of P-process watchpoint region – may be loaded into upper watchpoint register when protected mode entered
2	**ps.eWl**	lower bound of P-process watchpoint region – may be loaded into lower watchpoint register when protected mode entered
1	**ps.Eptr**	pointer to instruction causing trap – loaded into / stored from error pointer register when protected mode entered/exited
0	**ps.Cntl**	control word

Table A.7 P-state data structure (or PDS)

word offset	slot name	purpose
3	**pc.RegionReg3**	loaded into region descriptor register 3 when protected mode entered
2	**pc.RegionReg2**	loaded into region descriptor register 2 when protected mode entered
1	**pc.RegionReg1**	loaded into region descriptor register 1 when protected mode entered
0	**pc.RegionReg0**	loaded into region descriptor register 0 when protected mode entered

Table A.8 Region descriptor data structure

word offset	slot name	purpose
11	**th.sCreg**	L-process C register – stored from / loaded into integer stack C register when trap-handler entered/exited
10	**th.sBreg**	L-process B register – stored from / loaded into integer stack B register when trap-handler entered/exited
9	**th.sAreg**	L-process A register – stored from / loaded into integer stack A register when trap-handler entered/exited
8	**th.sIptr**	L-process instruction pointer – stored from / loaded into instruction pointer register when trap-handler entered/exited
7	**th.sWptr**	L-process workspace pointer – stored from / loaded into Wptr when trap-handler entered/exited
6	**th.eWu**	upper bound of L-process watchpoint region – may be loaded into upper watchpoint register when an L-process is executed[1]
5	**th.eWl**	lower bound of L-process watchpoint region – may be loaded into lower watchpoint register when an L-process is executed[1]
4	**th.Eptr**	pointer to instruction causing trap – stored from error pointer register when trap-handler entered
3	**th.Bptr**	back of trap sharing process queue
2	**th.Fptr**	front of trap sharing process queue
1	**th.Iptr**	trap-handler instruction pointer – loaded into instruction pointer register when trap-handler entered
0	**th.Cntl**	control word

Table A.9 Trap-handler data structure (or THDS)

word offset	slot name	purpose
5	**fp.FPCreg**	loaded into / stored from floating-point stack register C by *fpldall* / *fpstall*
3	**fp.FPBreg**	loaded into / stored from floating-point stack register B by *fpldall* / *fpstall*
1	**fp.FPAreg**	loaded into / stored from floating-point stack register A by *fpldall* / *fpstall*
0	**fp.FPstatusReg**	loaded into / stored from floating-point status register by *fpldall* / *fpstall*

Table A.10 Floating-point state data structure

word offset	slot name	purpose
2	**bmr.DeltaS**	*stmove2dinit* copies the content of 2D block move control register 2 into this slot
1	**bmr.DeltaD**	*stmove2dinit* copies the content of 2D block move control register 1 into this slot
0	**bmr.Count**	*stmove2dinit* copies the content of 2D block move control register 0 into this slot

Table A.11 Block move data structure

In addition, a number of constants are used to identify word length related values, etc. Note that, although the T9000 is a 32-bit processor, these names are used for clarity and for consistency with the descriptions of other transputers. These constants are listed in table A.12.

There are a number of values that are used by the transputer to indicate the state of a process and other conditions. These are shown in table A.13.

Name	Value	Meaning
BitsPerByte	8	The number of bits in a byte.
BitsPerWord	32	The number of bits in a word.
ByteSelectMask	#00000003	Used to select the byte select bits of an address.
WordSelectMask	#FFFFFFFC	Used to select the byte select bits of an address.
BytesPerWord	4	The number of bytes in a word.
BytesPerLine	16	The number of bytes in each cache line.
BytesPerVLCB	32	The number of bytes that are allocated for each VLCB
MaxHeaderOffset	#FFFE	The maximum offset allowed into the VLCB header area.
MaxLink	3	The maximum link number.
MaxPacketLength	32	The maximum length of a virtual channel packet.
MinLink	0	The minimum link number.
MinVirtualChannel	*MostNeg* + #40 #80000040	The lowest virtual channel address.
MinEventChannel	*MostNeg* + #20 #80000020	The lowest event channel address.
MostNeg	#80000000	The most negative integer value.
MostPos	#7FFFFFFF	The most positive signed integer value.
MostPosUnsigned	#FFFFFFFF	The most positive unsigned integer value.

Table A.12 Constants used in the instruction descriptions

Name	Value	Meaning
Deactivated.p	*MostNeg* + #01 #80000001	Stored in reverse channel word of a deactivated event channel.
DeviceId	Depends on transputer type. See table A.14.	A value used to identify the type and revision of transputer. Returned by the *ldprodid* and *lddevid* instructions.
Disabling.p	*MostNeg* + #03 #80000003	Stored in the **pw.State** location while an alternative is being disabled.
Enabling.p	*MostNeg* + #01 #80000001	Stored in the **pw.State** location while an alternative is being enabled.
false	0	The boolean value 'false'.
LengthError.p	*MostPosUnsigned* #FFFFFFFF –1	Stored in the **pw.Length** slot of a process' workspace to indicate that the number of bytes received by a *vin* was more than the maximum specified.
NoneSelected.o	–1 #FFFFFFFF	Stored in the **pw.Temp** slot of a process' workspace while no branch of an alternative has yet been selected during the waiting and disabling phases.
NotProcess.p	*MostNeg* #80000000	Used, wherever a process descriptor is expected, to indicate that there is no process.
NullHeader	–1 #FFFFFFFF	Used where a null virtual channel header is needed.
NullOffset	–1 #FFFFFFFF	Used where a null virtual channel header offset is needed.
Ready.p	*MostNeg* + #03 #80000003	Stored in the **pw.State** location during the enabling phase of an alternative, to indicate that a guard is ready.
ResChan.p	*MostNeg* + #02 #80000002	Stored in a channel word to indicate that it is in resource channel mode.
Stopping.p	*MostNeg* + #03 #80000003	Stored in a channel word to indicate that the channel is stopping.
TimeNotSet.p	*MostNeg* + #02 #80000002	Stored in **pw.TLink** location during enabling of a timer alternative after a time to wait for has been encountered.
TimeSet.p	*MostNeg* + #01 #80000001	Stored in **pw.TLink** location during enabling of a timer alternative after a time to wait for has been encountered.
true	1	The boolean value 'true'.
Waiting.p	*MostNeg* + #02 #80000002	Stored in the **pw.State** location by *altwt* and *taltwt* to indicate that the alternative is waiting.

Table A.13 Constants used within the T9000

Device identity values

The following table lists the values returned by the *lddevid* and *ldprodid* instructions.

Device	Revision	Value
T9000		300 .. 319

Table A.14 Device identity values

A.2.8 Operators

Modulo operators

Arithmetic on addresses is done using *modulo* arithmetic — i.e. there is no checking for errors and, if the calculation overflows, the result 'wraps around' the range of values representable in the word length of

the processor — e.g. adding 1 to the address at the top of the address map produces the address of the byte at the bottom of the address map. There are also a number of instructions for performing modulo arithmetic, such as *sum*, *prod*, etc. These operators are represented by the symbols '+', '−', etc.

Error conditions

Any errors that can occur in instructions which are defined in terms of the modulo operators are indicated explicitly in the instruction description. For example the *div* (*divide*) instruction indicates the cases that can cause overflow, independently of the actual division:

```
if (Areg = 0) or ((Breg = MostNeg) and (Areg = −1))
{
      Areg ← undefined
      IntegerOverflow
}
else
      Areg′ ← Breg / Areg

Breg′ ← Creg
Creg′ ← undefined
```

Checked operators

To simplify the description of *checked* arithmetic, the operators '$+_{checked}$', '$-_{checked}$', etc. are used to indicate operations that perform checked arithmetic on signed integers. These operators signal an *IntegerOverflow* if an overflow, divide by zero, or other arithmetic error occurs. If no trap is taken, the operators also deliver the modulo result.

To indicate floating point arithmetic performed to the IEEE standard, the operators '$+_{IEEE}$', '$-_{IEEE}$', etc. are used.

A number of comparison operators are also used and there are versions of some of these that treat the operands as unsigned integers.

A full list of the operators used is given in table A.15.

Symbol	Meaning
Integer arithmetic with overflow checking	
$+_{checked}$ $-_{checked}$ $\times_{checked}$	Add, subtract, and multiply of signed integers. If the computation overflows an *IntegerOverflow* is signalled and the result of the operation is truncated to the word length.
Unchecked (modulo) integer arithmetic	
$+$ $-$ $\times$ $/$ **rem**	Integer add, subtract, multiply, divide and remainder. If the computation overflows the result of the operation is truncated to the word length. If a divide or remainder by zero occurs the result of the operation is undefined. No errors are signalled. The operator '$-$' is also used as a monadic operator.
Floating point arithmetic to IEEE standard 754-1988	
$+_{IEEE}$ $-_{IEEE}$ $\times_{IEEE}$ $/_{IEEE}$ **rem**$_{IEEE}$	These perform IEEE floating point arithmetic, using the current rounding mode, on the operands. Both operands must be of the same precision. If an error occurs, the appropriate exception is signalled and the result depends on the particular instruction and the type of the error (see chapter 11 for more details).
Signed comparison operators	
$<$ $>$ $\leq$ $\geq$ $=$ $\neq$	Comparisons of signed integer and floating point values: 'less than', 'greater than', 'less than or equal', 'greater than or equal', 'equal' and 'not equal'.
Unsigned comparison operators	
$<_{unsigned}$ $>_{unsigned}$ $\geq_{unsigned}$ **after**	Comparisons of unsigned integer values: 'less than', 'greater than', 'greater than or equal', and 'after' (for comparison of times — defined in chapter 8).
Logical bitwise operations	
$\sim$ (or BITNOT) $\wedge$ (or BITAND) $\vee$ (or BITOR) $\odot$ (or BITXOR) $\ll$ $\gg$	'Not' (1's complement), 'and', 'or', 'exclusive or', and logical left and right shift operations on bits in words.
Boolean operators	
not **and** **or**	Boolean combination in conditionals.

Table A.15 Operators used in the instruction descriptions

A.2.9 Functions

Type conversions

A number of functions are used to indicate type conversions. These are:

`fpint(x)` converts `x` (a floating point number) to an integral value, in the same floating point format, using the current rounding mode.

`int64(x)` converts the floating point value, `x`, to a 64-bit signed integer, rounding towards $-\infty$ (note that this operation ignores the current rounding mode).

`real32(x)` converts `x` (which can be either an integer or a double precision floating point value) to a single precision (32-bit) floating point number, using the current rounding mode. If `x` is a NaN then the result is *R64ToR32NaN*.

`real64(x)` converts `x` (which can be either an integer or a single precision floating point value) to a double precision (64-bit) floating point number, using the current rounding mode. See chapter 11 for details of the conversion of NaNs.

`unsign(x)` causes the bit-pattern in `x` to be interpreted as an unsigned integer.

In addition the following functions are used to simplify the descriptions:

`sqroot(x)` returns the square root of `x` using the current rounding mode.

`Q(x)` converts a signalling NaN, `x`, into the equivalent quiet NaN.

See chapter 11 for more details of the floating point operations.

A.2.10 Conditions to instructions

In many cases, the action of an instruction depends on the current state of the processor. In these cases the conditions are shown by an **if** clause; this can take one of the following forms:

```
if (condition)
      statement
```

```
if (condition)
      statement
else
      statement
```

These conditions can be nested. Braces, {}, are used to group statements which are dependent on a condition. For example, the *cj* (*conditional jump*) instruction contains the following lines:

```
if (Areg = 0)
      IptrReg' ← next instruction + n
else
{
      IptrReg' ← next instruction

      Areg' ← Breg
      Breg' ← Creg
      Creg' ← undefined
}
```

This says that if the value in **Areg** is zero, then the jump is taken (the instruction operand, *n*, is added to the instruction pointer), otherwise the stack is popped and execution continues with the next instruction.

adc n add constant

Code: Function 8

Description: Add a constant to **Areg**, with checking for overflow.

Definition:

$\texttt{Areg}' \leftarrow \texttt{Areg} +_{checked} \texttt{n}$

Error signals:

IntegerOverflow can be signalled by $+_{checked}$

Comments:

Primary instruction

See chapter: 7

See also: *ldnlp*

add add

Code: F5

Description: Add **Areg** and **Breg**, with checking for overflow.

Definition:

Areg′ ← Breg $+_{checked}$ Areg

Breg′ ← Creg
Creg′ ← *undefined*

Error signals:
IntegerOverflow can be signalled by $+_{checked}$

Comments:
Secondary instruction

See chapter: 7

See also: *adc sum*

ajw n	adjust workspace

Code: Function B

Description: Move the workspace pointer by the number of words specified in the operand, in order to allocate/deallocate workspace stack. In protected mode a (recoverable) error occurs if the new workspace address is not writable. The state delivered to the trap-handler in this case is the state *before* the instruction started.

Definition:

```
Wptr' ← Wptr @ n
```

Error signals:

AccessViolation signalled in a P-process if new workspace address is not writable

Comments:

Primary instruction

See chapter: 7

See also: *call gajw*

alt	alt start

Code: 24 F3

Description: Start of a non-timer alternative sequence. The **pw.State** location of the workspace is set to *Enabling.p*.

Definition:

```
word'[Wptr @ pw.State] ← Enabling.p
```

enter alternative sequence

Error signals:

PrivInstruction signalled if executed by a P-process

Comments:

Instruction is privileged
Secondary instruction

See chapter: 8

See also: *altend altwt disc disg diss dist enbc enbg enbs enbt talt taltwt*

altend	alt end

Code: 24 F5

Description: End of alternative sequence. Jump to start of selected process.

Definition:

terminate alternative sequence
`IptrReg'` ← *next instruction* + `word[Wptr @ pw.Temp]`

Error signals:

PrivInstruction signalled if executed by a P-process

Comments:

Instruction is privileged
Secondary instruction
Uses the **pw.Temp** slot in the process workspace

See chapter: 8

See also: *alt altwt disc disg diss dist enbc enbg enbs enbt talt taltwt*

altwt alt wait

Code: 24 F4

Description: Wait until one of the enabled guards of an alternative has become ready, and initialize workspace for use during the disabling sequence.

Definition:

```
if (word[Wptr @ pw.State] ≠ Ready.p)
{
  word'[Wptr @ pw.State] ← Waiting.p
  deschedule process and wait for one of the guards to become ready
}

word'[Wptr @ pw.Temp] ← NoneSelected.o

FPAreg' ← undefined
FPBreg' ← undefined
FPCreg' ← undefined
Areg'   ← undefined
Breg'   ← undefined
Creg'   ← undefined
```

Error signals:
PrivInstruction signalled if executed by a P-process

Comments:
Instruction is privileged
Secondary instruction
Instruction is a descheduling point
Uses the **pw.Temp** slot in the process workspace

See chapter: 8

See also: *alt altend disc disg diss dist enbc enbg enbs enbt talt taltwt*

and	and

Code: 24 F6

Description: Bitwise and of **Areg** and **Breg**.

Definition:

```
Areg′ ← Breg ∧ Areg

Breg′ ← Creg
Creg′ ← undefined
```

Error signals: *none*

Comments:
Secondary instruction

See chapter: 7

bcnt	byte count

Code: 23 F4

Description: Produces the length, in bytes, of a multiword data object. Converts the value in **Areg**, representing a number of words, to the equivalent number of bytes.

Definition:

```
Areg' ← Areg × BytesPerWord
```

Error signals: *none*

Comments:
Secondary instruction

See chapter: 7

bitcnt	count bits set in word

Code: 27 F6

Description: Count the number of bits set in **Areg** and add this to the value in **Breg**.

Definition:

`Areg'` ← `Breg` + *number of bits set to* `1` *in* `Areg`

`Breg'` ← `Creg`
`Creg'` ← *undefined*

Error signals: *none*

Comments:
Secondary instruction

See chapter: 7

bitrevnbits reverse bottom n bits in word

Code: 27 F8

Description: Reverse the order of the bottom **Areg** bits of **Breg**.

Definition:

```
if (0 ≤ Areg) and (Areg ≤ BitsPerWord)
{
  Areg'0..Areg–1                ← reversed Breg0..Areg–1
  Areg'Areg..BitsPerWord–1      ← 0
}
else
  undefined effect

Breg' ← Creg
Creg' ← undefined
```

Error signals: *none*

Comments:
Secondary instruction
The effect of the instruction is undefined if the number of bits specified is more than the word length

See chapter: 7

See also: *bitrevword*

bitrevword reverse bits in word

Code: 27 F7

Description: Reverse the order of all the bits in **Areg**.

Definition:

`Areg' ←` *reversed* `Areg`

Error signals: *none*

Comments:

Secondary instruction

See chapter: 7

See also: *bitcnt bitrevnbits*

bsub byte subscript

Code: F2

Description: Generate the address of the element which is indexed by **Breg**, in the byte array pointed to by **Areg**.

Definition:

```
Areg' ← Areg + Breg

Breg' ← Creg
Creg' ← undefined
```

Error signals: *none*

Comments:
Secondary instruction

See chapter: 7

See also: *ssub sum wsub wsubdb*

call n call

Code: Function 9

Description: Adjust workspace pointer, save evaluation stack, and call subroutine at specified byte offset. In protected mode a (recoverable) error occurs if the new workspace address is not writable. The state delivered to the trap-handler in this case is the state *before* the instruction started.

Definition:

```
Wptr' ← Wptr @ −4

word'[Wptr' @ 0] ← IptrReg
word'[Wptr' @ 1] ← Areg
word'[Wptr' @ 2] ← Breg
word'[Wptr' @ 3] ← Creg

IptrReg' ← next instruction + n

Areg' ← IptrReg
Breg' ← undefined
Creg' ← undefined
```

Error signals:

AccessViolation signalled in a P-process if new workspace address is not writable

Comments:

Primary instruction

See chapter: 7

See also: *ajw gcall ret*

causeerror cause error

Code: 62 FF

Description: Take a trap with a reason of *causeerror* and with the error type set to the value in **Areg**. If the error code is not in the range 0 through 13, an integer error is signalled. Note that this instruction forces a trap even when the corresponding trap enable flags are not set.

Definition:

```
if (0 ≤ Areg) and (Areg ≤ 13)
  take a 'causeerror' trap with error value given in Areg
else
  IntegerError
```

Error signals:
IntegerError signalled if **Areg** is not in range

Comments:
CauseError trap taken if **Areg** is in range

See chapter: 10

cb	check byte

Code: 2B FA

Description: Check that the value in **Areg** can be represented as an 8 bit signed integer.

Definition:

if (Areg $<$ -2^7) **or** (Areg $\geq$ 2^7)
IntegerError

Error signals:
IntegerError signalled if **Areg** is not in range

Comments:
Secondary instruction

See chapter: 7

See also: *cbu cir ciru cs csu*

cbu check byte unsigned

Code: 2B FB

Description: Check that the value in **Areg** can be represented as an 8 bit unsigned integer.

Definition:

```
if (Areg < 0) or (Areg ≥ 2^8)
    IntegerError
```

Error signals:
IntegerError signalled if **Areg** is not in range

Comments:
Secondary instruction

See chapter: 7

See also: *cb cir ciru cs csu*

ccnt1	check count from 1

Code: 24 FD

Description: Check that **Breg** is the range 1..**Areg**, interpreting **Areg** and **Breg** as unsigned numbers.

Definition:

```
if (Breg = 0) or (Breg_unsigned > Areg_unsigned)
   IntegerError

Areg' ← Breg
Breg' ← Creg
Creg' ← undefined
```

Error signals:
IntegerError signalled if **Areg** is not in range

Comments:
Secondary instruction

See chapter: 7

See also: *csub0*

chantype channel type

Code: 2C F9

Description: Test if channel pointed to by **Areg** is an internal channel.

Definition:

```
if (Areg does not cause Unalign trap)
{
  if (internal channel)
    Areg' ← true

  else if (external channel)
    Areg' ← false

  else
    IntegerError
}
```

Error signals:
PrivInstruction signalled if executed by a P-process
Unalign signalled if **Areg** is not word aligned
IntegerError signalled if **Areg** is not a legal channel address

Comments:
Instruction is privileged
Secondary instruction

See chapter: 12

cir check in range

Code: 2C F7

Description: Check that **Creg** is in the range **Areg**..**Breg**.

Definition:

```
if (Creg < Areg) or (Creg > Breg)
   IntegerError

Areg′ ← Creg
Breg′ ← undefined
Creg′ ← undefined
```

Error signals:

IntegerError signalled if **Areg** is not in range

Comments:

Secondary instruction

See chapter: 7

See also: *ciru*

ciru check in range unsigned

Code: 2C FC

Description: Check that **Creg** is the range **Areg**..**Breg**, treating all as unsigned values.

Definition:

if (Creg$_{unsigned}$ < Areg$_{unsigned}$) **or** (Creg$_{unsigned}$ > Breg$_{unsigned}$)
IntegerError

Areg′ ← Creg
Breg′ ← *undefined*
Creg′ ← *undefined*

Error signals:
IntegerError signalled if **Areg** is not in range

Comments:
Secondary instruction

See chapter: 7

See also: *cir*

cj n conditional jump

Code: Function A

Description: Jump if **Areg** is 0 (i.e. jump if *false*). The destination of the jump is expressed as a byte offset from the instruction following.

Definition:
```
if (Areg = 0)
  IptrReg' ← next instruction + n
else
{
  IptrReg' ← next instruction
  Areg'    ← Breg
  Breg'    ← Creg
  Creg'    ← undefined
}
```

Error signals: *none*

Comments:
Primary instruction

See chapter: 7

See also: *j lend*

crcbyte calculate CRC on byte

Code: 27 F5

Description: Generate a CRC (cyclic redundancy check) checksum from the most significant byte of **Areg**. **Breg** contains the previously accumulated checksum and **Creg** the polynomial divisor (or 'generator'). The new CRC checksum, the polynomial remainder, is calculated by repeatedly (8 times) shifting the accumulated checksum left, shifting in successive bits from the **Areg** and if the bit shifted out of the checksum was a 1, then the generator is exclusive-ored into the checksum.

Definition:

```
Areg' ← temp(8)
Breg' ← Creg
Creg' ← undefined

where
  temp(0) = Breg
  for i = 1 .. 8
    temp(i) = (temp(i−1) << 1) + Areg_{BitsPerWord−i})
                      ⊙ (Creg * temp(i−1)_{BitsPerWord−1})
```

Error signals: *none*

Comments:
Secondary instruction

See chapter: 7

See also: *crcword*

crcword calculate CRC on word

Code: 27 F4

Description: Generate a CRC (cyclic redundancy check) checksum from **Areg**. **Breg** contains the previously accumulated checksum and **Creg** the polynomial divisor (or 'generator'). The new CRC checksum, the polynomial remainder, is calculated by repeatedly (BitsPerWord times) shifting the accumulated checksum left, shifting in successive bits from the **Areg** and if the bit shifted out of the checksum was a 1, then the generator is exclusive-ored into the checksum.

Definition:

```
Areg' ← temp(BitsPerWord)
Breg' ← Creg
Creg' ← undefined

where
  temp(0)  = Breg
  for i = 1 .. BitsPerWord
    temp(i)  = (temp(i−1) << 1) + Areg_{BitsPerWord−i})
                              ⊙ (Creg * temp(i−1)_{BitsPerWord−1})
```

Error signals: *none*

Comments:
Secondary instruction

See chapter: 7

See also: *crcbyte*

cs check sixteen

Code: 2F FA

Description: Check that the value in **Areg** can be represented as a 16 bit signed integer.

Definition:

if (Areg < -2^{15}) **or** (Areg ≥ 2^{15})
  IntegerError

Error signals:
IntegerError signalled if **Areg** is not in range

Comments:
Secondary instruction

See chapter: 7

See also: *cb cbu cir ciru csngl csu cword*

csngl	check single

Code: 24 FC

Description: Check that the two word signed value in **Areg** and **Breg** (most significant word in **Breg**) can be represented as a single length signed integer.

Definition:

```
if ((Areg ≥ 0) and (Breg ≠ 0)) or ((Areg < 0) and (Breg ≠ −1))
   IntegerError

Breg′ ← Creg
Creg′ ← undefined
```

Error signals:
IntegerError signalled if **Areg** is not in range

Comments:
Secondary instruction

See chapter: 7

See also: *cb cbu cir ciru cs csu cword*

csu	check sixteen unsigned

Code: 2F FB

Description: Check that the value in **Areg** can be represented as a 16 bit unsigned integer.

Definition:

```
if (Areg < 0) or (Areg ≥ 2^16)
    IntegerError
```

Error signals:
IntegerError signalled if **Areg** is not in range

Comments:
Secondary instruction

See chapter: 7

See also: *cb cbu cir ciru cs csngl cword*

csub0 check subscript from 0

Code: 21 F3

Description: Check that **Breg** is in the range 0..(**Areg**-1), interpreting **Areg** and **Breg** as unsigned numbers.

Definition:

```
if (Breg_unsigned ≥ Areg_unsigned)
   IntegerError

Areg' ← Breg
Breg' ← Creg
Creg' ← undefined
```

Error signals:
IntegerError signalled if **Areg** is not in range

Comments:
Secondary instruction

See chapter: 7

See also: *ccnt1*

cword check word

Code: 25 F6

Description: Check that the value in **Breg** can be represented as an *N* bit signed integer. **Areg** contains 2^{N-1} to indicate the value of *N* (i.e. bit *N*–1 of **Areg** is set to 1 and all other bits are set to zero).

Definition:

```
if (Breg < –Areg) or (Breg ≥ Areg)
   IntegerError

Areg′ ← Breg
Breg′ ← Creg
Creg′ ← undefined
```

Error signals:
IntegerError signalled if **Areg** is not in range

Comments:
The result of the instruction is undefined if **Areg** is not a power of 2
Secondary instruction

See chapter: 7

See also: *cb cs csngl xword*

devlb device load byte

Code: 2F F0

Description: Perform a device read from memory or a memory-mapped device. The byte addressed by **Areg** is read into **Areg** as an unsigned value. The memory access performed by this instruction is guaranteed to be correctly sequenced with respect to other device-access instructions. Also the instruction is guaranteed to be executed *after* all normal memory load instructions that appear before it in the code sequence, and *before* all normal memory loads that appear later.

Definition:

$Areg'_{0..7} \leftarrow byte[Areg]$

$Areg'_{8..BitsPerWord-1} \leftarrow 0$

Error signals:

AccessViolation signalled in a P-process if the address in **Areg** is protected

Comments:

Secondary instruction

See chapter: 7

See also: *devls devlw devsb lb*

devls device load sixteen

Code: 2F F2

Description: Perform a device read from memory or a memory-mapped device. The 16 bit object addressed by **Areg** is read into **Areg** as an unsigned value. The memory access performed by this instruction is guaranteed to be correctly sequenced with respect to other device-access instructions. Also the instruction is guaranteed to be executed *after* all normal memory load instructions that appear before it in the code sequence, and *before* all normal memory loads that appear after it.

Definition:

$\texttt{Areg}'_{0..15} \leftarrow \texttt{sixteen[Areg]}$

$\texttt{Areg}'_{16..\texttt{BitsPerWord}-1} \leftarrow 0$

Error signals:

Unalign signalled if **Areg** is not 16 bit aligned
AccessViolation signalled in a P-process if the address in **Areg** is protected

Comments:

Secondary instruction

See chapter: 7

See also: *devlb devlw devsb ls*

devlw	device load word

Code: 2F F4

Description: Perform a device read from memory or a memory-mapped device. The word addressed by **Areg** is read into **Areg**. The memory access performed by this instruction is guaranteed to be correctly sequenced with respect to other device-access instructions. Also the instruction is guaranteed to be executed *after* all normal memory load instructions that appear before it in the code sequence, and *before* all normal memory loads that appear after it.

Definition:

```
Areg' ← word[Areg]
```

Error signals:

Unalign signalled if **Areg** is not word aligned
AccessViolation signalled in a P-process if the address in **Areg** is protected

Comments:

Secondary instruction

See chapter: 7

See also: *devlb devls devsw ldnl*

devmove device move

Code: 62 F4

Description: Perform a device copy between memory or memory-mapped devices. Copies **Areg** bytes to address **Breg** from address **Creg**. Only the minimum number of reads and writes required to copy the data will be performed. Each read will be to a strictly higher (more positive) address than the one before and each write will be to a strictly higher address than the one before. There is no guarantee of the relative ordering of read and write cycles, except that a write cannot occur until the corresponding read has been performed. The memory accesses performed by this instruction are guaranteed to be correctly sequenced with respect to other device-access instructions. Also the instruction is guaranteed to be executed *after* all normal memory access instructions that appear before it in the code sequence, and *before* all normal memory accesses that appear after it.

Definition:

```
if (source and destination overlap)
  undefined effect
else for i = 0..(unsign(Areg) − 1)
  byte'[Breg + i] ← byte[Creg + i]

Areg' ← undefined
Breg' ← undefined
Creg' ← undefined
```

Error signals:

AccessViolation signalled in a P-process if any address accessed is protected

Comments:

Secondary instruction
The effect of the instruction is undefined if the source and destination overlap
Instruction is interruptible

See chapter: 7

See also: *move*

devsb device store byte

Code: 2F F1

Description: Perform a device write from memory or a memory-mapped device. Store least significant byte of **Breg** into the byte addressed by **Areg**. The memory access performed by this instruction is guaranteed to be correctly sequenced with respect to other device-access instructions. Also the instruction is guaranteed to be executed *after* all normal memory store instructions that appear before it in the code sequence, and *before* all normal memory stores that appear after it.

Definition:

byte'[Areg] ← $Breg_{0..7}$

Areg' ← Creg
Breg' ← *undefined*
Creg' ← *undefined*

Error signals:
AccessViolation signalled in a P-process if the address in **Areg** is not writable

Comments:
Secondary instruction

See chapter: 7

See also: *devlb devss devsw sb*

devss	device store sixteen

Code: 2F F3

Description: Perform a device write from memory or a memory-mapped device. Store bits 0..15 of **Breg** into the sixteen bits addressed by **Areg**. The memory access performed by this instruction is guaranteed to be correctly sequenced with respect to other device-access instructions. Also the instruction is guaranteed to be executed *after* all normal memory store instructions that appear before it in the code sequence, and *before* all normal memory stores that appear after it.

Definition:

sixteen'[Areg] ← $Breg_{0..15}$

Areg' ← Creg
Breg' ← *undefined*
Creg' ← *undefined*

Error signals:

Unalign signalled if **Areg** is not 16 bit aligned
AccessViolation signalled in a P-process if the address in **Areg** is not writable

Comments:

Secondary instruction

See chapter: 7

See also: *devls devsb devsw ss*

devsw device store word

Code: 2F F5

Description: Perform a device write from memory or a memory-mapped device. Store **Breg** into the word of memory addressed by **Areg**. The memory access performed by this instruction is guaranteed to be correctly sequenced with respect to other device-access instructions. Also the instruction is guaranteed to be executed *after* all normal memory store instructions that appear before it in the code sequence, and *before* all normal memory stores that appear after it.

Definition:

```
word'[Areg] ← Breg

Areg' ← Creg
Breg' ← undefined
Creg' ← undefined
```

Error signals:

Unalign signalled if **Areg** is not word aligned
AccessViolation signalled in a P-process if the address in **Areg** is not writable

Comments:

Secondary instruction

See chapter: 7

See also: *devlw devsb devss stnl*

diff difference

Code: F4

Description: Subtract **Areg** from **Breg**, with checking for overflow.

Definition:

```
Areg' ← Breg − Areg

Breg' ← Creg
Creg' ← undefined
```

Error signals: *none*

Comments:
Secondary instruction

See chapter: 7

See also: *sub*

disc	disable channel

Code: 22 FF

Description: Disable a channel guard in an alternative sequence. **Areg** is the offset from the byte following the *altend* to the start of the guarded process, **Breg** is the boolean guard and **Creg** is a pointer to the channel. If this is the first ready guard then the value in **Areg** is stored in workspace and **Areg** is set to *true*, otherwise **Areg** is set to *false*. Note that this instruction should be used as part of an alternative sequence following an *altwt* or *taltwt* instruction.

Definition:

```
if (Breg = false)                                      — boolean guard is false
  Areg' ← false
else if (Creg does not cause Unalign trap)
{
  if (Creg is internal channel)
  {
    if (word[Creg] = NotProcess.p)                     — guard already disabled
      Areg' ← false
    else if (word[Creg] = WdescReg)                    — this guard is not ready
    {
      word'[Creg] ← NotProcess.p
      Areg'       ← false
    }
    else if (word[Wptr @ pw.Temp] = NoneSelected.o)— this is the first ready guard
    {
      word'[Wptr @ pw.Temp] ← Areg
      Areg'                 ← true
    }
    else                                               — a previous guard was selected
      Areg' ← false
  }
  else if (Creg is external channel)
  {
    request VCP to disable external channel

    if (channel not ready)                             — determined by VCP
      Areg' ← false
    else if (word[Wptr @ pw.Temp] = NoneSelected.o)— this is the first ready guard
    {
      word'[Wptr @ pw.Temp] ← Areg
      Areg'                 ← true
    }
    else                                               — a previous guard was selected
      Areg' ← false
  }
  else
    IntegerError
}

Breg' ← undefined
Creg' ← undefined
```

(continued)

Error signals:

PrivInstruction signalled if executed by a P-process
Unalign signalled if **Creg** is not word aligned and **Breg** is not *false*
IntegerError signalled if **Creg** is not a legal channel address and **Breg** is not *false*

Comments:

Instruction is privileged
Secondary instruction
Uses the **pw.Temp** slot in the process workspace

See chapter: 8

See also: *alt altend altwt enbc talt taltwt*

disg disable grant

Code: 61 F3

Description: Disable a resource channel guard in an alternative sequence. Initially **Areg** is the offset from the byte following the *altend* to the start of the guarded process, **Breg** is the boolean guard and **Creg** is a pointer to the resource data structure. If this is the first ready guard then the value in **Areg** is stored in **pw.Temp**, and **Areg** is set to *true*. Note that this instruction should be used as part of an alternative sequence following an *altwt* or *taltwt* instruction.

Definition:

```
if (Breg = false)                                    — boolean guard is false
  Areg′ ← false
else if (Creg does not cause Unalign trap)
{
  if (word[Creg @ rds.Proc] = NotProcess.p)          — guard already disabled
    Areg′ ← false
  else
  {
    word′[Creg @ rds.Proc] ← NotProcess.p

    if (word[Creg @ rds.Front] = NotProcess.p)       — this guard is not ready
      Areg′ ← false
    else if (word[Wptr @ pw.Temp] = NoneSelected.o)— this is the first ready guard
    {
      Areg′ ← true
      word′[Wptr @ pw.Temp] ← Areg
    }
    else                                             — a previous guard was selected
      Areg′ ← false
  }
}

Breg′    ← undefined
Creg′    ← undefined
```

Error signals:

PrivInstruction signalled if executed by a P-process
Unalign signalled if **Creg** is not word aligned and **Breg** is not *false*

Comments:

Instruction is privileged
Secondary instruction
Uses the **pw.Temp** slot in the process workspace

See chapter: 8

See also: *alt altend altwt enbg talt taltwt*

diss disable skip

Code: 23 F0

Description: Disable a 'skip' guard in an alternative sequence. **Areg** is the offset from the byte following the *altend* to the start of the guarded process and **Breg** is the boolean guard. If this is the first ready guard then the value in **Areg** is stored in workspace and **Areg** is set to *true*, otherwise **Areg** is set to *false*. Note that this instruction should be used as part of an alternative sequence following an *altwt* or *taltwt* instruction.

Definition:

```
if (Breg = false)                                  — boolean guard is false
  Areg' ← false
else if (word[Wptr @ pw.Temp] ≠ NoneSelected.o)    — this is the first ready guard
  Areg' ← false
else                                               — another guard was selected
{
  word'[Wptr @ pw.Temp] ← Areg
  Areg'                 ← true
}

Breg' ← Creg
Creg' ← undefined
```

Error signals:
PrivInstruction signalled if executed by a P-process

Comments:
Instruction is privileged
Secondary instruction
Uses the **pw.Temp** slot in the process workspace

See chapter: 8

See also: *alt altend altwt enbs talt taltwt*

dist disable timer

Code: 22 FE

Description: Disable a timer guard in an alternative sequence. **Areg** is the offset from the byte following the *altend* to the start of the guarded process, **Breg** is the boolean guard and **Creg** is the time after which this guard will be ready. If this is the first ready guard then the value in **Areg** is stored in **pw.Temp**, and **Areg** is set to *true*. Note that this instruction should be used as part of an alternative sequence following a *taltwt* instruction.

Definition:

```
if (Breg = false)                                       — boolean guard is false
  Areg' ← false
else if (word[Wptr @ pw.TLink] = TimeNotSet.p)          — no timer is ready
  Areg' ← false
else if (word[Wptr @ pw.TLink] = TimeSet.p)             — a timer is ready
{
  if not (word[Wptr @ pw.Time] after Creg)              — but not this one
    Areg' ← false

  else if (word[Wptr @ pw.Temp] = NoneSelected.o)       — this is the first ready guard
  {
    word'[Wptr @ pw.Temp] ← Areg
    Areg' ← true
  }
  else                                                  — a previous guard was selected
      Areg' ← false
}
else
  Areg' ← false

remove this process from timer list
Breg' ← undefined
Creg' ← undefined
```

Error signals:

PrivInstruction signalled if executed by a P-process

Comments:

Instruction is privileged
Secondary instruction
Instruction is interruptible
Uses the **pw.Temp** slot in the process workspace

See chapter: 8

See also: *altend enbt talt taltwt*

div divide

Code: 22 FC

Description: Divide **Breg** by **Areg**, with checking for overflow. The result when not exact is rounded towards zero.

Definition:

```
if (Areg = 0) or ((Breg = MostNeg) and (Areg = −1))
{
  Areg ← undefined
  IntegerOverflow
}
else
  Areg′ ← Breg / Areg

Breg′ ← Creg
Creg′ ← undefined
```

Error signals:
IntegerOverflow can be signalled

Comments:
Secondary instruction

See chapter: 7

See also: *rem*

dup duplicate top of stack

Code: 25 FA

Description: Duplicate the top of the integer stack.

Definition:

```
Areg' ← Areg
Breg' ← Areg
Creg' ← Breg
```

Error signals: *none*

Comments:
Secondary instruction

See chapter: 7

See also: *fpdup pop rev*

enbc enable channel

Code: 24 F8

Description: Enable a channel guard in an alternative sequence. **Areg** is the boolean guard and **Breg** is a pointer to the channel. Note that this instruction should only be used as part of an alternative sequence following an *alt* or *talt* instruction.

Definition:

```
if (Areg ≠ false)
{
  if (Breg does not cause Unalign trap)
  {
    if (Breg is internal channel)
    {
      if (word[Breg] = NotProcess.p)                  — not ready
        word'[Breg] ← WdescReg

      else if (word[Creg] ≠ WdescReg)                 — not previously enabled
        word'[Wptr @ pw.State] ← Ready.p
    }
    else if (Breg is external channel)
    {
      request VCP to enable external channel

      if (channel ready)                              — determined by VCP
        word'[Wptr @ pw.State] ← Ready.p

    }
    else
      IntegerError
  }
}

    Breg' ← Creg
    Creg' ← undefined
```

Error signals:

PrivInstruction signalled if executed by a P-process
Unalign signalled if **Breg** is not word aligned and **Areg** is not *false*
IntegerError signalled if **Breg** is not a legal channel address and **Areg** is not *false*

Comments:

Instruction is privileged
Secondary instruction

See chapter: 8

See also: *alt altend altwt disc talt taltwt*

enbg	enable grant

Code: 61 F2

Description: Enable a resource channel guard in an alternative sequence. **Areg** is the boolean guard and **Breg** is a pointer to the resource data structure. Note that this instruction should only be used as part of an alternative sequence following an *alt* or *talt* instruction.

Definition:

```
if (Areg ≠ false)
{
  word'[Breg @ rds.Proc] ← WdescReg

  if (word[Breg @ rds.Front] ≠ NotProcess.p)
    word'[Wptr @ pw.State] ← Ready.p
}

Breg' ← Creg
Creg' ← undefined
```

Error signals:
PrivInstruction signalled if executed by a P-process
Unalign signalled if the address in **Breg** is not word aligned and **Areg** is not *false*

Comments:
Instruction is privileged
Secondary instruction

See chapter: 8 and 12

See also: *alt altend altwt disg talt taltwt*

enbs enable skip

Code: 24 F9

Description: Enable a 'skip' guard in an alternative sequence. **Areg** is the boolean guard. Note that this instruction should only be used as part of an alternative sequence following an *alt* or *talt* instruction.

Definition:
```
if (Areg ≠ false)
  word'[Wptr @ pw.State] ← Ready.p
```

Error signals:
PrivInstruction signalled if executed by a P-process

Comments:
Instruction is privileged
Secondary instruction

See chapter: 8

See also: *alt altend altwt diss talt taltwt*

enbt enable timer

Code: 24 F7

Description: Enable a timer guard in an alternative sequence. **Areg** is the boolean guard and **Breg** is the time after which the guard may be selected. Note that this instruction should only be used as part of an alternative sequence following a *talt* instruction; in this case the location **pw.State** will have been initialized to *Enabling.p* and the **pw.Tlink** slot initialized to *TimeNotSet.p*.

Definition:

```
if (Areg ≠ false)
{
  if (word[Wptr @ pw.TLink] = TimeNotSet.p)            — this is the first enbt
  {
    word'[Wptr @ pw.TLink] ← TimeSet.p
    word'[Wptr @ pw.Time]  ← Breg
  }
  else if (word[Wptr @ pw.TLink] = TimeSet.p)          — this is not the first enbt
  {
    if (word[Wptr @ pw.Time] after Breg)               — this enbt has an earlier time
      word'[Wptr @ pw.Time] ← Breg
  }
}

Breg' ← Creg
Creg' ← undefined
```

Error signals:
PrivInstruction signalled if executed by a P-process

Comments:
Instruction is privileged
Secondary instruction

See chapter: 8

See also: *altend dist talt taltwt*

endp — end process

Code: F3

Description: Synchronize the termination of a parallel construct. When all branches have executed an *endp* instruction a 'successor' process then executes. **Areg** points to the workspace of this successor process. This workspace contains a data structure which holds the instruction pointer of the successor process and the number of processes still active.

Definition:

```
if (Areg does not cause Unalign trap)
{
  word'[Areg @ pp.Count] ← word[Areg @ pp.Count] − 1

  if (word'[Areg @ pp.Count] = 0)
  {
    IptrReg' ← word[Areg @ pp.IptrSucc]
    Wptr'    ← Areg
  }
  else
    start next process
}

FPAreg' ← undefined
FPBreg' ← undefined
FPCreg' ← undefined
Areg'   ← undefined
Breg'   ← undefined
Creg'   ← undefined
```

Error signals:
PrivInstruction signalled if executed by a P-process
Unalign signalled if **Areg** is not word aligned

Comments:
Instruction is privileged
Secondary instruction
Instruction is a descheduling point

See chapter: 8

See also: *startp stopp*

eqc n equals constant

Code: Function C

Description: Compare **Areg** to a constant.

Definition:

```
if (Areg = n)
  Areg′ ← true
else
  Areg′ ← false
```

Error signals: *none*

Comments:
Primary instruction

See chapter: 7

erdsq	empty resource data structure queue

Code: 62 FA

Description: Load the list pointers from the resource data structure (RDS) into **Areg** and **Breg**, and set the RDS list to empty. **Areg** is a pointer to the RDS.

Definition:

```
Areg′ ← word[Areg @ rds.Front]
Breg′ ← word[Areg @ rds.Back]

word′[Areg @ rds.Front] ← NotProcess.p

Creg′ ← Breg
```

Error signals:
PrivInstruction signalled if executed by a P-process
Unalign signalled if **Areg** is not word aligned

Comments:
Instruction is privileged
Secondary instruction

See chapter: 12

See also: *grant irdsq ldresptr mkrc stresptr unmkrc*

fdca flush dirty cache address

Code: 62 F0

Description: If the cache line which contains the logical address specified in **Areg** is dirty, write the line back to memory and mark it as clean; the line remains valid. **Areg** is incremented by the number of bytes in a line. If executed by a P-process the address specified must be writable.

Definition:

```
if (line containing logical address Areg is valid and dirty)
{
    write line back to memory
    mark line as clean
}

Areg' ← Areg + BytesPerLine
```

Error signals:

AccessViolation signalled in a P-process if the address in **Areg** is protected

Comments:

Secondary instruction

See chapter: 15

See also: *fdcl ica icl*

fdcl flush dirty cache line

Code: 62 F2

Description: If the cache line specified by **Areg** contains an address in the range specified by **Breg** and **Creg**, and the cache line is dirty, write the line back to memory; the line remains valid. Increment **Areg** by 1.

Definition:

```
if ((Breg ≤ address in line Areg ≤ Creg) and (line Areg is valid and dirty)
{
    write line back to memory
    mark line as clean
}

Areg' ← Areg + 1
```

Error signals:
PrivInstruction signalled if executed by a P-process

Comments:
Instruction is privileged
Secondary instruction

See chapter: 15

See also: *fdca ica icl*

fmul fractional multiply

Code: 27 F2

Description: Multiply **Areg** by **Breg** treating the values as fractions, rounding the result. The values in **Areg** and **Breg** are interpreted as fractions in the range [–1, 1) — i.e. the integer values divided by $2^{BitsPerWord-1}$. The result is rounded: the rounding mode used is analogous to IEEE round nearest; that is the result produced is the fraction which is nearest to the exact product, and, in the event of the product being equidistant between two factions, the fraction with least significant bit 0 is produced.

Definition:

```
if (Areg = MostNeg) and (Breg = MostNeg)          — MostNeg interpreted as –1
{
  Areg′ ← undefined
  IntegerOverflow
}
else
  Areg′ ← (Breg × Areg) / 2^(BitsPerWord–1)

Breg′ ← Creg
Creg′ ← undefined
```

Error signals:
IntegerOverflow can occur

Comments:
Secondary instruction

See chapter: 7

fpabs floating point absolute

Code: 2D FB

Description: Make **FPAreg** positive.

Definition:

```
if (FPAreg is a signalling NaN)
  FPAreg′ ← Q(FPAreg)
else if (FPAreg is a quiet NaN)
  FPAreg′ ← FPAreg
else if (FPAreg = −0.0)
  FPAreg′ ← 0.0
else if (FPAreg < 0.0)
  FPAreg′ ← 0.0 −IEEE FPAreg
else
  FPAreg′ ← FPAreg

RoundMode′ ← 'round to nearest'
```

Error signals:

FPError signalled by a NaN or an infinity as an operand
FPInvalidOp signalled by a signalling NaN as an operand

Comments:

Secondary instruction

See chapter: 11

fpadd floating point add

Code: 28 F7

Description: Add **FPAreg** to **FPBreg**.

Definition:

```
if (FPAreg is a signalling NaN)
  FPAreg' ← Q(FPAreg)
else if (FPBreg is a signalling NaN)
  FPAreg' ← Q(FPBreg)
else if (FPAreg is a quiet NaN)
  FPAreg' ← FPAreg
else if (FPBreg is a quiet NaN)
  FPAreg' ← FPBreg
else if (FPAreg and FPBreg are infinities of opposite sign)
  FPAreg' ← AddOpInfsNaN
else
  FPAreg' ← FPBreg +IEEE FPAreg

FPBreg'     ← FPCreg
FPCreg'     ← undefined
RoundMode'  ← 'round to nearest'
```

Error signals:

FPError signalled by a NaN or an infinity as an operand
FPInvalidOp signalled by:
  a signalling NaN as an operand
  adding infinities of opposite sign
FPInexact, *FPOverflow*, *FPUnderflow* can be signalled by $+_{IEEE}$
FPError also signalled whenever *FPInvalidOp* or *FPOverflow* are signalled

Comments:

Secondary instruction
For the operation to be defined the operands must both be the same precision

See chapter: 11

fpadddbsn floating point add double producing single

Code: 2D FD

Description: Add **FPAreg** to **FPBreg**, rounding the result to single precision. The result is put into **FPAreg** and **FPCreg** is popped into **FPBreg**.

Definition:

```
if (FPAreg is a NaN) or (FPBreg is a NaN)
  FPAreg' ← R64ToR32NaN
else if (FPAreg and FPBreg are infinities of opposite sign)
  FPAreg' ← AddOpInfsNaN
else
  FPAreg' ← single precision result of (FPBreg +IEEE FPAreg)

FPBreg'     ← FPCreg
FPCreg'     ← undefined
RoundMode'  ← 'round to nearest'
```

Error signals:

FPError signalled by a NaN or an infinity as an operand
FPInvalidOp signalled by:
a signalling NaN as an operand
adding infinities of opposite sign
FPInexact, *FPOverflow*, *FPUnderflow* can be signalled by $+_{IEEE}$
FPError also signalled whenever *FPInvalidOp* or *FPOverflow* are signalled

Comments:

Secondary instruction
For the operation to be defined the operands must both be double precision
Any NaN produced by $+_{IEEE}$ will be converted to *R64ToR32NaN*

See chapter: 11

fpb32tor64 BIT32 to REAL64

Code: 29 FA

Description: Load an unsigned 32 bit integer, from the word addressed by **Areg**, into **FPAreg** and convert it to a double precision floating-point number.

Definition:

```
FPAreg' ← real64(unsigned(word[Areg]))

Areg'      ← Breg
Breg'      ← Creg
Creg'      ← undefined
FPBreg'    ← FPAreg
FPCreg'    ← FPBreg
RoundMode' ← 'round to nearest'
```

Error signals:

Unalign signalled if address in **Areg** is not word aligned
AccessViolation signalled in a P-process if the address in **Areg** is protected

Comments:

Secondary instruction

See chapter: 11

fpchki32 floating point check in range of `INT32`

Code: 2D FE

Description: Check that the floating point integer value in **FPAreg** lies within the range of a signed 32 bit integer.

Definition:

if ($-2^{31} \le$ FPAreg) **and** (FPAreg $< 2^{31}$)
FPInvalidOp

RoundMode′ ← *'round to nearest'*

Error signals:

FPError signalled by a NaN or an infinity as an operand
FPInvalidOp signalled if **FPAreg** is not in range
FPError also signalled whenever *FPInvalidOp* signalled

Comments:

Secondary instruction
For the operation to be defined the initial value of **FPAreg** must be an integral value

See chapter: 11

See also: *fpint fprtoi32*

fpchki64 floating point check in range of `INT64`

Code: 2D FF

Description: Check that the floating point integer value in **FPAreg** lies within the range of a signed 64 bit integer.

Definition:

if ($-2^{63} \leq$ FPAreg) **and** (FPAreg $< 2^{63}$)
FPInvalidOp

RoundMode′ ← *'round to nearest'*

Error signals:
FPError signalled by a NaN or an infinity as an operand
FPInvalidOp signalled if **FPAreg** is not in range
FPError also signalled whenever *FPInvalidOp* signalled

Comments:
Secondary instruction
For the operation to be defined the initial value of **FPAreg** must be an integral value

See chapter: 11

fpdiv floating point divide

Code: 28 FC

Description: Divide **FPAreg** into **FPBreg**. Note that dividing by zero (infinity) will produce a result of infinity (zero) with the correct sign.

Definition:

```
if (FPAreg is a signalling NaN)
  FPAreg' ← Q(FPAreg)
else if (FPBreg is a signalling NaN)
  FPAreg' ← Q(FPBreg)
else if (FPAreg is a quiet NaN)
  FPAreg' ← FPAreg
else if (FPBreg is a quiet NaN)
  FPAreg' ← FPBreg
else if (FPAreg = 0.0) and (FPBreg = 0.0)
  FPAreg' ← DivZeroByZeroNaN
else if (FPAreg = ∞) and (FPBreg = ∞)
  FPAreg' ← DivInfByInfNaN
else
  FPAreg' ← FPBreg /IEEE FPAreg

FPBreg'     ← FPCreg
FPCreg'     ← undefined
RoundMode'  ← 'round to nearest'
```

Error signals:

FPError signalled by a NaN or an infinity as an operand
FPInvalidOp signalled by a signalling NaN as an operand, by (0.0 $/_{IEEE}$ 0.0) and (∞ $/_{IEEE}$ ∞)
FPInexact, *FPDivideByZero*, *FPOverflow*, *FPUnderflow* can be signalled by $/_{IEEE}$
FPError also signalled whenever *FPInvalidOp*, *FPOverflow*, or *FPDivideByZero* are signalled

Comments:

Secondary instruction
For the operation to be defined the operands must both be the same precision

See chapter: 11

See also: *fpdivby2 fpexpdec32*

fpdivby2	floating point divide by 2.0

Code: 2D F1

Description: Divide **FPAreg** by two.

Definition:

```
if (FPAreg is a signalling NaN)
  FPAreg' ← Q(FPAreg)
else if (FPAreg is a quiet NaN)
  FPAreg' ← FPAreg
else
  FPAreg' ← FPAreg /IEEE 2.0

RoundMode' ← 'round to nearest'
```

Error signals:

FPError signalled by a NaN or an infinity as an operand
FPInvalidOp signalled by a signalling NaN as an operand
FPInexact, *FPUnderflow* can be signalled by $/_{IEEE}$

Comments:

Secondary instruction

See chapter: 11

fpdup floating point duplicate

Code: 2A F3

Description: Duplicate top of floating point stack.

Definition:

```
FPAreg' ← FPAreg
FPBreg' ← FPAreg
FPCreg' ← FPBreg

RoundMode' ← 'round to nearest'
```

Error signals: *none*

Comments:
Secondary instruction

See chapter: 11

See also: *dup fprev*

fpeq floating point equality

Code: 29 F5

Description: Implements (**FPBreg** = **FPAreg**) as per ANSI/IEEE Std 754-1985 (Page 13, Table 4, line 1). The result of the comparison is loaded into **Areg**.

Definition:

```
if (FPAreg is a NaN) or (FPBreg is a NaN)
  Areg' ← false
else if (FPBreg = FPAreg)
  Areg' ← true
else
  Areg' ← false

FPAreg'     ← FPCreg
FPBreg'     ← undefined
FPCreg'     ← undefined
Breg'       ← Areg
Creg'       ← Breg
RoundMode'  ← 'round to nearest'
```

Error signals:

FPError signalled by a NaN or an infinity as an operand
FPInvalidOp signalled by a signalling NaN as an operand

Comments:

Secondary instruction
For the operation to be defined the operands must both be the same precision

See chapter: 11

fpexpdec32 floating point divide by 2^{32}

Code: 2D F9

Description: Multiply **FPAreg** by 2^{-32}.

Definition:

```
if (FPAreg is a signalling NaN)
  FPAreg' ← Q(FPAreg)
else if (FPAreg is a quiet NaN)
  FPAreg' ← FPAreg
else
  FPAreg' ← FPAreg ×IEEE 2^-32

RoundMode' ← 'round to nearest'
```

Error signals:

FPError signalled by a NaN or an infinity as an operand
FPInvalidOp signalled by a signalling NaN as an operand
FPError also signalled whenever *FPInvalidOp* signalled
FPInexact, *FPUnderflow* can be signalled by $\times_{IEEE}$

Comments:

Secondary instruction

See chapter: 11

fpexpinc32 floating point multiply by 2^{32}

Code: 2D FA

Description: Multiply **FPAreg** by 2^{32}.

Definition:

```
if (FPAreg is a signalling NaN)
   FPAreg' ← Q(FPAreg)
else if (FPAreg is a quiet NaN)
   FPAreg' ← FPAreg
else
   FPAreg' ← FPAreg ×IEEE 2^32

RoundMode' ← 'round to nearest'
```

Error signals:

FPError signalled by a NaN or an infinity as an operand
FPInvalidOp signalled by a signalling NaN as an operand
FPInexact, *FPOverflow*, *FPUnderflow* can be signalled by $\times_{IEEE}$
FPError also signalled whenever *FPInvalidOp* or *FPOverflow* are signalled

Comments:

Secondary instruction

See chapter: 11

fpge floating point greater than or equals

Code: 29 F7

Description: Implements (**FPBreg** >= **FPAreg**) as per ANSI/IEEE Std 754-1985 (Page 13, Table 4, line 4). The result of the comparison is loaded into **Areg**.

Definition:

```
if (FPAreg is a NaN) or (FPBreg is a NaN)
  Areg' ← false
else if (FPBreg ≥ FPAreg)
  Areg' ← true
else
  Areg' ← false

FPAreg'    ← FPCreg
FPBreg'    ← undefined
FPCreg'    ← undefined
Breg'      ← Areg
Creg'      ← Breg
RoundMode' ← 'round to nearest'
```

Error signals:

FPError signalled by a NaN or an infinity as an operand
FPInvalidOp signalled by a NaN as an operand

Comments:

Secondary instruction
For the operation to be defined the operands must both be the same precision

See chapter: 11

fpgt floating point greater than

Code: 29 F4

Description: Implements (**FPBreg** > **FPAreg**) as per ANSI/IEEE Std 754-1985 (Page 13, Table 4, line 3). The result of the comparison is loaded into **Areg**.

Definition:

```
if (FPAreg is a NaN) or (FPBreg is a NaN)
  Areg' ← false
else if (FPBreg > FPAreg)
  Areg' ← true
else
  Areg' ← false

FPAreg'     ← FPCreg
FPBreg'     ← undefined
FPCreg'     ← undefined
Breg'       ← Areg
Creg'       ← Breg
RoundMode'  ← 'round to nearest'
```

Error signals:

FPError signalled by a NaN or an infinity as an operand
FPInvalidOp signalled by NaN as an operand

Comments:

Secondary instruction
For the operation to be defined the operands must both be the same precision

See chapter: 11

fpi32tor32 `INT32` to `REAL32`

Code: 29 F6

Description: Load signed 32 bit value from the word addressed by **Areg** into **FPAreg**, converting it to a single precision floating-point number. If this instruction is not preceded by an instruction to set the rounding mode then it is equivalent to the sequence *fpi32tor64; fpr64tor32*. If the instruction is preceded by an instruction to set the rounding mode, then the rounding mode is set after the load and before the conversion to single precision (i.e. *fprx; fpi32tor32* is equivalent to *fpi32tor64; fprx; fpr64tor32*).

If an error occurs then the behavior is as if the equivalent instruction sequence had been executed, as follows. A trap may occur during the load, in which case the state delivered to the trap-handler is the same as if a trap had occurred during the execution of *fpi32tor64*. If the load is successful then a trap may occur during the conversion to single precision, in which case the state delivered to the trap-handler is the same as if *fpi32tor64* (and possibly *fprx*) had been successful and a trap had occurred during a subsequent *fpr64tor32*.

Definition:

```
FPAreg' ← real32(word[Areg])

Areg'      ← Breg
Breg'      ← Creg
Creg'      ← undefined
FPBreg'    ← FPAreg
FPCreg'    ← FPBreg
RoundMode' ← 'round to nearest'
```

Error signals:

Unalign signalled if the address in **Areg** is not word aligned
AccessViolation signalled in a P-process if the address in **Areg** is protected
FPInexact can be signalled by the conversion

Comments:

Secondary instruction

See chapter: 11

See also: *fpi32tor64 fpr64tor32*

fpi32tor64	INT32 to REAL64

Code: 29 F8

Description: Load signed 32 bit value from the word addressed by **Areg** into **FPAreg**, converting it to a double precision floating-point number.

Definition:

```
FPAreg′ ← real64(word[Areg])

Areg′      ← Breg
Breg′      ← Creg
Creg′      ← undefined
FPBreg′    ← FPAreg
FPCreg′    ← FPBreg
RoundMode′ ← 'round to nearest'
```

Error signals:

Unalign signalled if the address in **Areg** is not word aligned
AccessViolation signalled in a P-process if the address in **Areg** is protected

Comments:

Secondary instruction

See chapter: 11

fpint round to floating integer

Code: 2A F1

Description: Convert **FPAreg** to an integer in floating point format using the current rounding mode.

Definition:

```
if (FPAreg is a signalling NaN)
  FPAreg' ← Q(FPAreg)
else if (FPAreg is a quiet NaN) or (FPAreg = ∞)
  FPAreg' ← FPAreg
else
  FPAreg' ← fpint(FPAreg)

RoundMode' ← 'round to nearest'
```

Error signals:

FPError signalled by a NaN or an infinity as an operand
FPInvalidOp signalled by a signalling NaN as an operand
FPInexact can be signalled by the conversion

Comments:

Secondary instruction

See chapter: 11

See also: *fprtoi32*

fpldall floating point load all

Code: 60 FE

Description: Load the entire FPU state from a 7 word block of memory (the floating-point state data structure) pointed to by **Areg**. Note that this instruction may change the rounding mode.

Definition:

```
FPstatusReg' ← word[Areg @ 0]

if (FPstatusReg'3 = 1)
  FPAreg' ← {undefined)
else if (FPstatusReg'2 = 0)
  FPAreg' ← word[Areg @ 1]            — single precision
else
  FPAreg' ← double[Areg @ 1]          — double precision

if (FPstatusReg'5 = 1)
  FPBreg' ← {undefined)
else if (FPstatusReg'4 = 0)
  FPBreg' ← word[Areg @ 3]            — single precision
else
  FPBreg' ← double[Areg @ 3]          — double precision

if (FPstatusReg'7 = 1)
  FPCreg' ← {undefined)
else if (FPstatusReg'6 = 0)
  FPCreg' ← word[Areg @ 5]            — single precision
else
  FPCreg' ← double[Areg @ 5]          — double precision

if ((FPstatusReg' ∧ 3) = 0)
  RoundMode' ← 'round to zero'
else if ((FPstatusReg' ∧ 3) = 1)
  RoundMode' ← 'round to nearest'
else if ((FPstatusReg' ∧ 3) = 2)
  RoundMode' ← 'round to plus infinity'
else if ((FPstatusReg' ∧ 3) = 3)
  RoundMode' ← 'round to minus infinity'

Areg' ← Breg
Breg' ← Creg
Creg' ← undefined
```

Error signals:

Unalign signalled if the address in **Areg** is not word aligned
AccessViolation signalled in a P-process if any address accessed is protected

Comments:

Secondary instruction

See chapter: 11

See also: *fpstall*

fpldnladddb floating point load non-local and add double

Code: 2A F6

Description: Add the double precision floating point number addressed by **Areg** to **FPAreg**. **FPCreg** becomes undefined. If this instruction is not preceded by an instruction to set the rounding mode then it is equivalent to the sequence *fpldnldb; fpadd*. If the instruction is preceded by an instruction to set the rounding mode, then the rounding mode is set after the load and before the add operation (i.e. *fprx; fpldnladddb* is equivalent to *fpldnldb; fprx; fpadd*).

If an error occurs then the behavior is as if the equivalent instruction sequence had been executed, as follows. A trap may occur during the load, in which case the state delivered to the trap-handler is the same as if a trap had occurred during the execution of *fpldnldb.* If the load is successful then a trap may occur during the add, in which case the state delivered to the trap-handler is the same as if *fpldnldb* (and possibly *fprx*) had been successful and a trap had occurred during a subsequent *fpadd*.

Definition:

```
if (double[Areg] is a signalling NaN)
  FPAreg' ← Q(double[Areg])
else if (FPAreg is a signalling NaN)
  FPAreg' ← Q(FPAreg)
else if (double[Areg] is a quiet NaN)
  FPAreg' ← double[Areg]
else if (FPAreg is a quiet NaN)
  FPAreg' ← FPAreg
else if (double[Areg] and FPAreg are infinities of opposite sign)
  FPAreg' ← AddOpInfsNaN
else
  FPAreg' ← FPAreg +IEEE double[Areg]

FPCreg'     ← undefined
Areg'       ← Breg
Breg'       ← Creg
Creg'       ← undefined
RoundMode'  ← 'round to nearest'
```

Error signals:

Unalign signalled if the address in **Areg** is not word aligned
AccessViolation signalled in a P-process if any address accessed is protected
FPError signalled by a NaN or an infinity as an operand of the add
FPInvalidOp signalled by:
  a signalling NaN as an operand
  adding infinities of opposite sign
FPInexact, *FPOverflow*, *FPUnderflow* can be signalled by $+_{IEEE}$
FPError also signalled whenever *FPInvalidOp* or *FPOverflow* are signalled

Comments:

Secondary instruction
For the operation to be defined the initial value of **FPAreg** must be double precision

See chapter: 11

See also: *fpldnldb fpadd*

fpldnladdsn floating point load non-local and add single

Code: 2A FA

Description: Add the single precision floating point number addressed by **Areg** to **FPAreg**. **FPCreg** becomes undefined. If this instruction is not preceded by an instruction to set the rounding mode then it is equivalent to the sequence *fpldnlsn; fpadd*. If the instruction is preceded by an instruction to set the rounding mode, then the rounding mode is set after the load and before the add operation (i.e. *fprx; fpldnladdsn* is equivalent to *fpldnlsn; fprx; fpadd*).

If an error occurs then the behavior is as if the equivalent instruction sequence had been executed, as follows. A trap may occur during the load, in which case the state delivered to the trap-handler is the same as if a trap had occurred during the execution of *fpldnlsn.* If the load is successful then a trap may occur during the add, in which case the state delivered to the trap-handler is the same as if *fpldnlsn* (and possibly *fprx*) had been successful and a trap had occurred during a subsequent *fpadd*.

Definition:

```
if (word[Areg] is a signalling NaN)
   FPAreg' ← Q(word[Areg])
else if (FPAreg is a signalling NaN)
   FPAreg' ← Q(FPAreg)
else if (word[Areg] is a quiet NaN)
   FPAreg' ← word[Areg]
else if (FPAreg is a quiet NaN)
   FPAreg' ← FPAreg
else if (word[Areg] and FPAreg are infinities of opposite sign)
   FPAreg' ← AddOpInfsNaN
else
   FPAreg' ← FPAreg +IEEE word[Areg]

FPCreg'     ← undefined
Areg'       ← Breg
Breg'       ← Creg
Creg'       ← undefined
RoundMode'  ← 'round to nearest'
```

Error signals:

Unalign signalled if the address in **Areg** is not word aligned
AccessViolation signalled in a P-process if the address in **Areg** is protected
FPError signalled by a NaN or an infinity as an operand of the add
FPInvalidOp signalled by:
  a signalling NaN as an operand
  adding infinities of opposite sign
FPInexact, *FPOverflow*, *FPUnderflow* can be signalled by $+_{IEEE}$
FPError also signalled whenever *FPInvalidOp* or *FPOverflow* are signalled

Comments:

Secondary instruction
For the operation to be defined the initial value of **FPAreg** must be single precision

See chapter: 11

See also: *fpldnlsn fpadd*

fpldnldb floating point load non-local double

Code: 28 FA

Description: Load double precision floating point number into **FPAreg** from the pair of words addressed by **Areg**.

Definition:

```
FPAreg' ← double[Areg]

FPBreg'     ← FPAreg
FPCreg'     ← FPBreg
Areg'       ← Breg
Breg'       ← Creg
Creg'       ← undefined
RoundMode'  ← 'round to nearest'
```

Error signals:

Unalign signalled if the address in **Areg** is not word aligned
AccessViolation signalled in a P-process if any address accessed is protected

Comments:

Secondary instruction

See chapter: 11

fpldnldbi floating point load non-local indexed double

Code: 28 F2

Description: Load double precision floating point number into **FPAreg** from the pair of words of memory offset by (2 × **Breg**) words from the address in **Areg**.

Definition:

```
FPAreg' ← double[Areg @ (2 × Breg)]

FPBreg'     ← FPAreg
FPCreg'     ← FPBreg
Areg'       ← Creg
Breg'       ← undefined
Creg'       ← undefined
RoundMode'  ← 'round to nearest'
```

Error signals:

Unalign signalled if the address in **Areg** is not word aligned
AccessViolation signalled in a P-process if any address accessed is protected

Comments:

Secondary instruction

See chapter: 11

fpldnlmuldb	floating point load non-local and multiply double

Code: 2A F8

Description: Multiply the double precision floating point number addressed by **Areg** to **FPAreg**. **FPCreg** becomes undefined. If this instruction is not preceded by an instruction to set the rounding mode then it is equivalent to the sequence *fpldnldb; fpmul*. If the instruction is preceded by an instruction to set the rounding mode, then the rounding mode is set after the load and before the multiply operation (i.e. *fprx; fpldnlmuldb* is equivalent to *fpldnldb; fprx; fpmul*).

If an error occurs then the behavior is as if the equivalent instruction sequence had been executed as follows. A trap may occur during the load, in which case the state delivered to the trap-handler is the same as if a trap had occurred during the execution of *fpldnldb.* If the load is successful then a trap may occur during the multiply, in which case the state delivered to the trap-handler is the same as if *fpldnldb* (and possibly *fprx*) had been successful and a trap had occurred during a subsequent *fpmul*.

Definition:

```
if (double[Areg] is a signalling NaN)
  FPAreg' ← Q(double[Areg])
else if (FPAreg is a signalling NaN)
  FPAreg' ← Q(FPAreg)
else if (double[Areg] is a quiet NaN)
  FPAreg' ← double[Areg]
else if (FPAreg is a quiet NaN)
  FPAreg' ← FPAreg
else if (double[Areg] = 0.0) and (FPAreg = ∞)
     or (double[Areg] = ∞) and (FPAreg = 0.0)
  FPAreg' ← ZeroMulInfNaN
else
  FPAreg' ← double[Areg] ×IEEE FPAreg

FPCreg'     ← undefined
Areg'       ← Breg
Breg'       ← Creg
Creg'       ← undefined
RoundMode'  ← 'round to nearest'
```

Error signals:

Unalign signalled if the address in **Areg** is not word aligned
AccessViolation signalled in a P-process any address in accessed is protected
FPError signalled by a NaN or an infinity as an operand of the multiply
FPInvalidOp signalled by a signalling NaN as an operand and by ($\infty \times_{IEEE}$ 0.0)
FPInexact, *FPOverflow*, *FPUnderflow* can be signalled by $\times_{IEEE}$
FPError also signalled whenever *FPInvalidOp* or *FPOverflow* are signalled

Comments:

Secondary instruction
For the operation to be defined the initial value of **FPAreg** must be double precision

See chapter: 11

See also: *fpldnldb fpmul*

fpldnlmulsn	floating point load non-local and multiply single

Code: 2A FC

Description: Multiply the single precision floating point number addressed by **Areg** to **FPAreg**. **FPCreg** becomes undefined. If this instruction is not preceded by an instruction to set the rounding mode then it is equivalent to the sequence *fpldnlsn; fpmul*. If the instruction is preceded by an instruction to set the rounding mode, then the rounding mode is set after the load and before the multiply operation (i.e. *fprx; fpldnlmulsn* is equivalent to *fpldnlsn; fprx; fpmul*).

If an error occurs then the behavior is as if the equivalent instruction sequence had been executed, as follows. A trap may occur during the load, in which case the state delivered to the trap-handler is the same as if a trap had occurred during the execution of *fpldnlsn*. If the load is successful then a trap may occur during the multiply, in which case the state delivered to the trap-handler is the same as if *fpldnlsn* (and possibly *fprx*) had been successful and a trap had occurred during a subsequent *fpmul*.

Definition:

```
if (word[Areg] is a signalling NaN)
  FPAreg' ← Q(word[Areg])
else if (FPAreg is a signalling NaN)
  FPAreg' ← Q(FPAreg)
else if (word[Areg] is a quiet NaN)
  FPAreg' ← word[Areg]
else if (FPAreg is a quiet NaN)
  FPAreg' ← FPAreg
else if (word[Areg] = 0.0) and (FPAreg = ∞)
     or (word[Areg] = ∞) and (FPAreg = 0.0)
  FPAreg' ← ZeroMulInfNaN
else
  FPAreg' ← word[Areg] ×IEEE FPAreg

FPCreg'     ← undefined
Areg'       ← Breg
Breg'       ← Creg
Creg'       ← undefined
RoundMode'  ← 'round to nearest'
```

Error signals:

Unalign signalled if the address in **Areg** is not word aligned
AccessViolation signalled in a P-process if the address in **Areg** is protected
FPError signalled by a NaN or an infinity as an operand of the multiply
FPInvalidOp signalled by a signalling NaN as an operand and by ($\infty \times_{IEEE} 0.0$)
FPInexact, *FPOverflow*, *FPUnderflow* can be signalled by $\times_{IEEE}$
FPError also signalled whenever *FPInvalidOp* or *FPOverflow* are signalled

Comments:

Secondary instruction
For the operation to be defined the initial value of **FPAreg** must be single precision

See chapter: 11

See also: *fpldnlsn fpmul*

fpldnlsn	floating point load non-local single

Code: 28 FE

Description: Load single precision floating point number into **FPAreg** from the word addressed by **Areg**.

Definition:

```
FPAreg' ← word[Areg]

FPBreg'    ← FPAreg
FPCreg'    ← FPBreg
Areg'      ← Breg
Breg'      ← Creg
Creg'      ← undefined
RoundMode' ← 'round to nearest'
```

Error signals:

Unalign signalled if the address in **Areg** is not word aligned
AccessViolation signalled in a P-process if the address in **Areg** is protected

Comments:

Secondary instruction

See chapter: 11

fpldnlsni floating point load non-local indexed single

Code: 28 F6

Description: Load single precision floating point number into **FPAreg** from the word offset by **Breg** words from the address in **Areg**.

Definition:

```
FPAreg' ← word[Areg @ Breg]

FPBreg'     ← FPAreg
FPCreg'     ← FPBreg
Areg'       ← Creg
Breg'       ← undefined
Creg'       ← undefined
RoundMode'  ← 'round to nearest'
```

Error signals:

Unalign signalled if the address in **Areg** is not word aligned
AccessViolation signalled in a P-process if the address accessed is protected

Comments:

Secondary instruction

See chapter: 11

fpldzerodb	load zero double

Code: 2A F0

Description: Load zero as a 64 bit floating point number into **FPAreg**.

Definition:

```
FPAreg' ← 0.0                                   — single precision

FPBreg' ← FPAreg
FPCreg' ← FPBreg
RoundMode' ← 'round to nearest'
```

Error signals: *none*

Comments:
Secondary instruction

See chapter: 11

fpldzerosn load zero single

Code: 29 FF

Description: Load zero as a 32 bit floating point number into **FPAreg**.

Definition:

```
FPAreg' ← 0.0                                   — single precision

FPBreg' ← FPAreg
FPCreg' ← FPBreg
RoundMode' ← 'round to nearest'
```

Error signals: *none*

Comments:
Secondary instruction

See chapter: 11

fplg floating point less than or greater than

Code: 29 FB

Description: Implements (**FPBreg** < > **FPAreg**) as per ANSI/IEEE Std 754-1985 (Page 13, Table 4, line 8). The result of the comparison is loaded into **Areg**.

Definition:

```
if (FPAreg is a NaN) or (FPBreg is a NaN)
  Areg' ← false
else if (FPBreg <> FPAreg)
  Areg' ← true
else
  Areg' ← false

FPAreg'    ← FPCreg
FPBreg'    ← undefined
FPCreg'    ← undefined
Breg'      ← Areg
Creg'      ← Breg
RoundMode' ← 'round to nearest'
```

Error signals:

FPError signalled by a NaN or an infinity as an operand
FPInvalidOp signalled by a NaN as an operand

Comments:

Secondary instruction
For the operation to be defined the operands must both be the same precision

See chapter: 11

fpmul — floating point multiply

Code: 28 FB

Description: Multiply **FPAreg** by **FPBreg**.

Definition:

```
if (FPAreg is a signalling NaN)
  FPAreg' ← Q(FPAreg)
else if (FPBreg is a signalling NaN)
  FPAreg' ← Q(FPBreg)
else if (FPAreg is a quiet NaN)
  FPAreg' ← FPAreg
else if (FPBreg is a quiet NaN)
  FPAreg' ← FPBreg
else if (FPAreg = 0.0) and (FPBreg = ∞)
     or (FPAreg = ∞) and (FPBreg = 0.0)
  FPAreg' ← ZeroMulInfNaN
else
  FPAreg' ← FPBreg ×IEEE FPAreg

FPBreg' ← FPCreg
FPCreg' ← undefined
RoundMode' ← 'round to nearest'
```

Error signals:

FPError signalled by a NaN or an infinity as an operand
FPInvalidOp signalled by a signalling NaN as an operand and by ($\infty \times_{IEEE} 0.0$)
FPInexact, *FPOverflow*, *FPUnderflow* can be signalled by $\times_{IEEE}$
FPError also signalled whenever *FPInvalidOp* or *FPOverflow* are signalled

Comments:

Secondary instruction
For the operation to be defined the operands must both be the same precision

See chapter: 11

See also: *fpexpinc32 fpmulby2*

fpmulby2 floating point multiply by 2.0

Code: 2D F2

Description: Multiply **FPAreg** by 2.

Definition:

```
if (FPAreg is a signalling NaN)
  FPAreg' ← Q(FPAreg)
else if (FPAreg is a quiet NaN)
  FPAreg' ← FPAreg
else
  FPAreg' ← FPAreg ×IEEE 2.0

RoundMode' ← 'round to nearest'
```

Error signals:

FPError signalled by a NaN or an infinity as an operand
FPInvalidOp signalled by a signalling NaN as an operand
FPInexact, *FPOverflow*, *FPUnderflow* can be signalled by $\times_{IEEE}$
FPError also signalled whenever *FPInvalidOp* or *FPOverflow* are signalled

Comments:

Secondary instruction

See chapter: 11

fpnan floating point NaN

Code: 29 F1

Description: Test whether **FPAreg** is a NaN. The result of the test is loaded into **Areg**.

Definition:

```
if (FPAreg is a NaN)
  Areg' ← true
else
  Areg' ← false

Breg' ← Areg
Creg' ← Breg
RoundMode' ← 'round to nearest'
```

Error signals: *none*

Comments:
Secondary instruction

See chapter: 11

fpnotfinite floating point not finite

Code: 29 F3

Description: Test whether **FPAreg** is finite (not a NaN and not an infinity). The result of the test is loaded into **Areg**.

Definition:

```
if (FPAreg is a NaN) or (FPAreg = ∞)
  Areg' ← true
else
  Areg' ← false

Breg' ← Areg
Creg' ← Breg
RoundMode' ← 'round to nearest'
```

Error signals: *none*

Comments:
Secondary instruction

See chapter: 11

fpordered floating point orderability

Code: 29 F2

Description: Test if **FPAreg** and **FPBreg** can be ordered; i.e. neither operand is a NaN. Implements **not(FPBreg ? FPAreg)** as per ANSI/IEEE Std 754-1985 (Page 13, Table 4, line 19). The result of the comparison is loaded into **Areg**.

Definition:

```
if (FPAreg is a NaN) or (FPBreg is a NaN)
  Areg' ← false
else
  Areg' ← true

Breg' ← Areg
Creg' ← Breg
RoundMode' ← 'round to nearest'
```

Error signals:

FPError signalled by a NaN or an infinity as an operand
FPInvalidOp signalled by a signalling NaN as an operand

Comments:

Secondary instruction
For the operation to be defined the operands must both be the same precision

See chapter: 11

fpr32tor64 floating point REAL32 to REAL64

Code: 2D F7

Description: Extend single precision value in **FPAreg** to double precision.

Definition:

```
if (FPAreg is a signalling NaN)
  FPAreg' ← real64(Q(FPAreg))
else
  FPAreg' ← real64(FPAreg)

RoundMode' ← 'round to nearest'
```

Error signals:

FPError signalled by a NaN or an infinity as an operand
FPInvalidOp signalled by a signalling NaN as an operand

Comments:

Secondary instruction
For the operation to be defined the initial value of **FPAreg** must be single precision

See chapter: 11

fpr64tor32 floating point `REAL64` to `REAL32`

Code: 2D F8

Description: Round double precision value in **FPAreg** to single precision.

Definition:

```
if (FPAreg is a NaN)
  FPAreg' ← R64ToR32NaN
else
  FPAreg' ← real32(FPAreg)

RoundMode' ← 'round to nearest'
```

Error signals:

FPError signalled by a NaN or an infinity as an operand
FPInvalidOp signalled by a signalling NaN as an operand
FPInexact, *FPOverflow*, *FPUnderflow* can be signalled by the conversion
FPError also signalled whenever *FPOverflow* signalled

Comments:

Secondary instruction
For the operation to be defined the initial value of **FPAreg** must be double precision

See chapter: 11

fprange floating point range reduce

Code: 28 FD

Description: Calculate the values needed to perform range reduction on an argument. Remainder **FPBreg** by **FPAreg** putting the remainder in **FPAreg** and the corresponding integer quotient (in floating point format) into **FPBreg**. The remainder, r = x **rem** y, is defined by r = x − (y × n), where n is the integer quotient (the nearest integer to the exact value x / y — calculated using 'round to nearest' mode).

Definition:

```
if (FPAreg is a signalling NaN)
{
  FPAreg' ← Q(FPAreg)
  FPBreg' ← undefined
}
else if (FPBreg is a signalling NaN)
{
  FPAreg' ← Q(FPBreg)
  FPBreg' ← undefined
}
else if (FPAreg is a quiet NaN)
{
  FPAreg' ← FPAreg
  FPBreg' ← undefined
}
else if (FPBreg is a quiet NaN)
{
  FPAreg' ← FPBreg
  FPBreg' ← undefined
}
else if (FPBreg = ∞)
{
  FPAreg' ← RemainderFromInfNaN
  FPBreg' ← undefined
}
else if (FPAreg = 0.0)
{
  FPAreg' ← RemainderByZeroNaN
  FPBreg' ← undefined
}
else if (quotient is out of range)                    — see [Error signals] below
{
  FPAreg' ← RangeQuotErrorNaN
  FPBreg' ← undefined
}
else
{
  FPAreg' ← FPBreg rem_IEEE FPAreg
  FPBreg' ← fpint(FPBreg / FPAreg)
}

RoundMode' ← 'round to nearest'
```

(continued)

Error signals:

FPError signalled by a NaN or an infinity as an operand
FPInvalidOp signalled by:
  FPAreg = 0.0;
  FPBreg = ∞;
  signalling Nans;
  magnitude of the quotient exceeding $2^{53}-1$ for double precision, or $2^{24}-1$ for single precision
FPError also signalled whenever *FPInvalidOp* signalled
FPUnderflow can be signalled by $\mathbf{rem}_{IEEE}$

Comments:

Secondary instruction
For the operation to be defined the operands must both be the same precision
Unlike *fprem* this instruction is **not** interruptible

See chapter: 11

See also: *fprem*

fprem floating point remainder

Code: 2C FF

Description: Remainder **FPBreg** by **FPAreg**; calculates the remainder when evaluating the integer quotient of **FPBreg** divided by **FPAreg**. The remainder, r = x **rem** y, is defined by r = x − (y × n), where n is the nearest integer (calculated using 'round to nearest' mode) to the exact value x / y.

Definition:

```
if (FPAreg is a signalling NaN)
  FPAreg' ← Q(FPAreg)
else if (FPBreg is a signalling NaN)
  FPAreg' ← Q(FPBreg)
else if (FPAreg is a quiet NaN)
  FPAreg' ← FPAreg
else if (FPBreg is a quiet NaN)
  FPAreg' ← FPBreg
else if (FPBreg = ∞)
  FPAreg' ← RemainderFromInfNaN
else if (FPAreg = 0.0)
  FPAreg' ← RemainderByZeroNaN
else
  FPAreg' ← FPBreg remIEEE FPAreg

FPBreg' ← FPCreg
FPCreg' ← undefined
RoundMode' ← 'round to nearest'
```

Error signals:

FPError signalled by a NaN or an infinity as an operand
FPInvalidOp signalled by:
  FPAreg = 0.0;
  FPBreg = ∞;
  signalling Nans;
FPError also signalled whenever *FPInvalidOp* signalled
FPUnderflow can be signalled by $\mathbf{rem}_{IEEE}$ (if the underflow trap is enabled)

Comments:

Secondary instruction
For the operation to be defined the operands must both be the same precision
Instruction is interruptible

See chapter: 11

See also: *fprange*

fprev floating point reverse

Code: 2A F4

Description: Swap the top two elements of the floating point stack.

Definition:

```
FPAreg' ← FPBreg
FPBreg' ← FPAreg

RoundMode' ← 'round to nearest'
```

Error signals: *none*

Comments:
Secondary instruction

See chapter: 11

See also: *fpdup rev*

fprm set rounding mode to round minus

Code: 2D F5

Description: Set rounding mode to 'round towards minus infinity' for the next floating point instruction.

Definition:
`RoundMode'` ← *'round to minus infinity'*

Error signals: *none*

Comments:
Secondary instruction

See chapter: 11

See also: *fprn fprp fprz*

fprn set rounding mode to round nearest

Code: 2D F0

Description: Set rounding mode to 'round towards nearest' for the next floating point instruction.

Definition:
`RoundMode'` ← *'round to nearest'*

Error signals: *none*

Comments:
Secondary instruction

See chapter: 11

See also: *fprm fprp fprz*

fprp set rounding mode to round plus

Code: 2D F4

Description: Set rounding mode to 'round towards plus infinity' for the next floating point instruction.

Definition:

`RoundMode'` ← *'round to plus infinity'*

Error signals: *none*

Comments:

Secondary instruction

See chapter: 11

See also: *fprm fprn fprz*

fprtoi32 — REAL to INT32

Code: 29 FD

Description: Round **FPAreg** into an integer value (in floating point format) and check the result lies within the range of a 32 bit signed integer. Unless an exception occurs, this instruction is equivalent to *fpint*; *fpchki32*

Definition:

if ($-2^{31} \leq$ FPAreg′) **and** (FPAreg′ $< 2^{31}$)
  FPInvalidOp

FPAreg′ ← fpint(FPAreg)
RoundMode′ ← *'round to nearest'*

Error signals:

FPError signalled by a NaN or an infinity as an operand
FPInvalidOp signalled by a signalling NaN or an infinity as an operand, or result not in range
FPError also signalled whenever *FPInvalidOp* signalled
FPInexact can be signalled by the conversion

Comments:

Secondary instruction

See chapter: 11

See also: *fpint fpchki32*

fprz set rounding mode to round zero

Code: 2D F6

Description: Set rounding mode to 'round towards zero' for the next floating point instruction.

Definition:
`RoundMode'` ← *'round to zero'*

Error signals: *none*

Comments:
Secondary instruction

See chapter: 11

See also: *fprm fprn fprp*

fpsqrt floating point square root

Code: 2D F3

Description: Take the square root of **FPAreg**.

Definition:

```
if (FPAreg is a signalling NaN)
  FPAreg' ← Q(FPAreg)
else if (FPAreg is a quiet NaN)
  FPAreg' ← FPAreg
else if (FPAreg = +∞)
  FPAreg' ← +∞
else if (FPAreg = −0.0)
  FPAreg' ← −0.0
else if (FPAreg < 0.0)
  FPAreg' ← NegSqrtNaN
else
  FPAreg' ← sqroot(FPAreg)

RoundMode' ← 'round to nearest'
```

Error signals:

FPError signalled by a NaN or an infinity as an operand
FPInvalidOp signalled by a signalling NaN as an operand and by (**FPAreg** $<$ 0)
FPError also signalled whenever *FPInvalidOp* signalled
FPInexact can be signalled when taking square root

Comments:

Secondary instruction

See chapter: 11

fpstall floating point store all

Code: 60 FF

Description: Store the entire FPU state in a 7 word block of memory (the floating-point state data structure) pointed to by **Areg**.

Definition:

```
word'[Areg @ 0]    ← FPstatusReg

if (FPstatusReg2 = 0)
  word'[Areg @ 1] ← FPAreg                — single precision
else
  double'[Areg @ 1] ← FPAreg              — double precision

if (FPstatusReg4 = 0)
  word'[Areg @ 3] ← FPBreg                — single precision
else
  double'[Areg @ 3] ← FPBreg              — double precision

else if (FPstatusReg6 = 0)
  word'[Areg @ 5] ← FPCreg                — single precision
else
  double'[Areg @ 5] ← FPCreg              — double precision

Areg' ← Breg
Breg' ← Creg
Creg' ← undefined

FPAreg' ← undefined
FPBreg' ← undefined
FPCreg' ← undefined

RoundMode' ← 'round to nearest'
```

Error signals:

Unalign signalled if the address in **Areg** is not word aligned
AccessViolation signalled in a P-process if any address accessed is not writable

Comments:

Secondary instruction

See chapter: 11

fpstnldb	floating point store non-local double

Code: 28 F4

Description: Store **FPAreg** as a double precision floating point number into the pair of words addressed by **Areg**.

Definition:

```
double'[Areg] ← FPAreg

Areg'      ← Breg
Breg'      ← Creg
Creg'      ← undefined
FPAreg'    ← FPBreg
FPBreg'    ← FPCreg
FPCreg'    ← undefined
RoundMode' ← 'round to nearest'
```

Error signals:

Unalign signalled if the address in **Areg** is not word aligned
AccessViolation signalled in a P-process if any address accessed is not writable

Comments:

Secondary instruction

See chapter: 11

fpstnli32 floating point store non-local `INT32`

Code: 29 FE

Description: Convert the floating-point number in **FPAreg** to an integer value, rounding towards $-\infty$ and then convert to a 64-bit twos-complement integer, storing the least significant 32-bits of this integer in the location pointed to by **Areg**.

Definition:

word'[Areg] ← int64(FPAreg) ∧ (2^{32}–1)

FPAreg'	←	FPBreg
FPBreg'	←	FPCreg
FPCreg'	←	*undefined*
Areg'	←	Breg
Breg'	←	Creg
Creg'	←	*undefined*
RoundMode'	←	*'round to nearest'*

Error signals:

Unalign signalled if the address in **Areg** is not word aligned
AccessViolation signalled in a P-process if the address in **Areg** is not writable

Comments:

Secondary instruction

See chapter: 11

fpstnlsn — floating point store non-local single

Code: 28 F8

Description: Store the single precision floating point number in **FPAreg** into the word addressed by **Areg**.

Definition:

```
word'[Areg] ← FPAreg

FPAreg'    ← FPBreg
FPBreg'    ← FPCreg
FPCreg'    ← undefined
Areg'      ← Breg
Breg'      ← Creg
Creg'      ← undefined
RoundMode' ← 'round to nearest'
```

Error signals:

Unalign signalled if the address in **Areg** is not word aligned
AccessViolation signalled in a P-process if the address in **Areg** is not writable

Comments:

Secondary instruction

See chapter: 11

fpsub floating point subtract

Code: 28 F9

Description: Subtract **FPAreg** from **FPBreg**.

Definition:

```
if (FPAreg is a signalling NaN)
  FPAreg' ← Q(FPAreg)
else if (FPBreg is a signalling NaN)
  FPAreg' ← Q(FPBreg)
else if (FPAreg is a quiet NaN)
  FPAreg' ← FPAreg
else if (FPBreg is a quiet NaN)
  FPAreg' ← FPBreg
else if (FPAreg and FPBreg are infinities of opposite sign)
  FPAreg' ← AddOpInfsNaN
else
  FPAreg' ← FPBreg −IEEE FPAreg

FPBreg' ← FPCreg
FPCreg' ← undefined
RoundMode' ← 'round to nearest'
```

Error signals:

FPError signalled by a NaN or an infinity as an operand
FPInvalidOp signalled by:
  a signalling NaN as an operand
  subtracting infinities of opposite sign
FPInexact, *FPOverflow*, *FPUnderflow* can be signalled by $-_{IEEE}$
FPError also signalled whenever *FPInvalidOp* or *FPOverflow* are signalled

Comments:

Secondary instruction
For the operation to be defined the operands must both be the same precision

See chapter: 11

gajw general adjust workspace

Code: 23 FC

Description: Set the workspace pointer to the address in **Areg**, saving previous value in **Areg**. In protected mode a (recoverable) error occurs if the new workspace address is not writable. The state delivered to the trap-handler in this case is the state *before* the instruction started.

Definition:

```
Wptr' ← Areg
Areg' ← Wptr
```

Error signals:

Unalign signalled if the new workspace address is not word aligned
AccessViolation signalled in a P-process if the new workspace address is not writable

Comments:

Secondary instruction

See chapter: 7

See also: *ajw call gcall*

gcall general call

Code: F6

Description: Jump to computed address in **Areg**, saving previous address in **Areg**.

Definition:

```
IptrReg′ ← Areg
Areg′    ← IptrReg
```

Error signals: *none*

Comments:
Secondary instruction

See chapter: 7

See also: *ajw call gajw ret*

goprot go protected

Code: 60 FA

Description: Start a P-process, continue execution of a P-process after it has trapped, or restart execution of an interrupted P-process. Initially, **Areg** is a pointer to a P-state data structure (PDS) and **Breg** is a pointer to a region descriptor data structure. If the instruction causes a 'watchpoint' or 'single-step' trap, then a 'context' trap is taken after the instruction has performed all its writes to memory; the stack state delivered to the trap handler is the state before the instruction started.

Definition:

```
if (Areg does not cause Unalign trap and Breg does not cause Unalign trap)
{
  if (ThReg ≠ NotProcess.p)
    word'[ThReg @ th.Cntl] ← StatusReg          — process status/control bits only

  word'[Wptr @ pw.Iptr] ← IptrReg

  if (not single step or watchpoint trap)
  {
    WdescStubReg' ← WdescReg
    PstateReg'    ← Areg
    Wptr'         ← word[Areg @ ps.sWptr]
    IptrReg'      ← word[Areg @ ps.sIptr]
    Areg'         ← word[Areg @ ps.sAreg]
    Breg'         ← word[Areg @ ps.sBreg]
    Creg'         ← word[Areg @ ps.sCreg]
    StatusReg'    ← sb.IsPprocessBit ∨
                              word[Areg @ ps.Cntl] — process status/control bits only

    if (restarted process was interrupted)
    {
      Ereg'    ← word[Areg @ ps.sEreg]
      EptrReg' ← word[Areg @ ps.Eptr]
      Xreg'    ← word[Areg @ ps.sXreg]
    }

    RegionReg0' ← word[Breg @ pc.RegionReg0]
    RegionReg1' ← word[Breg @ pc.RegionReg1]
    RegionReg2' ← word[Breg @ pc.RegionReg2]
    RegionReg3' ← word[Breg @ pc.RegionReg3]

    if (StatusReg'sb.WtchPntEnbl = 1)
    {
      WlReg' ← word[Areg @ ps.eWl]
      WuReg' ← word[Areg @ ps.eWu]
    }

    enable interrupts
  }
}
```

Error signals:

PrivInstruction signalled if executed by a P-process
Unalign signalled if the addresses in **Areg** or **Breg** are not word aligned

Comments:

Instruction is privileged
Secondary instruction
The *ps.sWptr* slot of the PDS must be word aligned
For stack extension on demand, the address read from the *ps.sWptr* slot of the PDS must be writable

See chapter: 9 and 10

See also: *restart syscall tret*

grant grant resource

Code: 61 F1

Description: Grant a claim on a resource (a claim is made by a client process performing an output to a resource channel). **Areg** is a pointer to the resource data structure and **Breg** points to a word where the resource channel identifier is to be stored. If no client has made a claim then the process descriptor of the server is saved and the server deschedules; otherwise the resource channel identifier of the first channel on the list is saved, the channel is set back to normal mode and then removed from the list.

Definition:

```
if (Areg does not cause Unalign trap and Breg does not cause Unalign trap)
{
   if (word[Areg @ rds.Front] = NotProcess.p)
   {
      word'[Areg @ rds.Proc] ← WdescReg
      word'[Wptr @ pw.Pointer] ← Breg
      start next process
   }
   else
   {
      word'[Breg] ← word[RC @ rc.Id]
      word'[RC @ rc.Id]  ← NotProcess.p
      word'[RC @ rc.Ptr] ← NotProcess.p
      remove RC from resource list

      where RC is the resource channel data structure of channel at word[Areg @ rds.Front]

   }
}

Areg'   ← undefined
Breg'   ← undefined
Creg'   ← undefined
FPAreg' ← undefined
FPBreg' ← undefined
FPCreg' ← undefined
```

Error signals:

PrivInstruction signalled if executed by a P-process
Unalign signalled if the addresses in **Areg** and **Breg** are not word aligned

Comments:

Instruction is privileged
Secondary instruction
Instruction is a descheduling point

See chapter: 12

See also: *disg enbg mkrc out unmkrc vout*

gt greater than

Code: F9

Description: Compare top two elements of stack, returning *true* if **Breg** is greater than **Areg**.

Definition:

```
if (Breg > Areg)
  Areg' ← true
else
  Areg' ← false

Breg' ← Creg
Creg' ← undefined
```

Error signals: *none*

Comments:
Secondary instruction

See chapter: 7

gtu greater than unsigned

Code: 25 FF

Description: Compare top two elements of stack, treating both as unsigned integers, returning *true* if **Breg** is greater than **Areg**.

Definition:

```
if (Breg_unsigned > Areg_unsigned)
  Areg' ← true
else
  Areg' ← false

Breg' ← Creg
Creg' ← undefined
```

Error signals: *none*

Comments:
Secondary instruction

See chapter: 7

ica invalidate cache address

Code: 62 F1

Description: Invalidate the cache line which contains the logical address specified in **Areg** then increment **Areg** by the number of bytes in a line. The cache line is not written back to memory, even if it is dirty. If executed by a P-process the address specified must be writable.

Definition:

```
if (line containing logical address Areg is valid)
  invalidate line

Areg' ← Areg + BytesPerLine
```

Error signals:
AccessViolation signalled in a P-process if the address in **Areg** is not writable

Comments:
Secondary instruction

See chapter: 15

See also: *fdca icl*

icl invalidate cache line

Code: 62 F3

Description: Invalidate cache line indicated by **Areg** if it contains an addresses between **Breg** and **Creg** inclusive, then increment **Areg** by 1. The cache line is not written back to memory, even if it is dirty.

Definition:

```
if ((Breg ≤ address in line Areg ≤ Creg) and line is valid)
   invalidate line

Areg′ ← Areg + 1
```

Error signals:

PrivInstruction signalled if executed by a P-process

Comments:

Secondary instruction

See chapter: 15

See also: *fdcl ica*

in input message

Code: F7

Description: Input a message. The corresponding output is performed by an *out*, *outword* or *outbyte* instruction, and must specify a message of the same length. **Areg** is the unsigned length in bytes, **Breg** is a pointer to the channel and **Creg** is a pointer to where the message is to be stored. The process executing *in* will be descheduled if the channel is external or is not ready, and is rescheduled when the communication is complete.

Definition:

```
if (Breg does not cause Unalign trap)
{
   if (Breg is not a legal channel address)
      IntegerError
   else
      synchronize, and input Areg_unsigned bytes from channel Breg to address Creg
}

FPAreg' ← undefined
FPBreg' ← undefined
FPCreg' ← undefined
Areg'   ← undefined
Breg'   ← undefined
Creg'   ← undefined
```

Error signals:

PrivInstruction signalled if executed by a P-process
Unalign signalled if the address in **Breg** is not word aligned
IntegerError signalled if the address in **Breg** is not a legal channel address

Comments:

Instruction is privileged
Secondary instruction
Instruction is a descheduling point
Instruction is interruptible

See chapter: 8

See also: *out vin*

initvlcb initialize VLCB

Code: 61 F6

Description: Initialize virtual link control block (VLCB) of the virtual channel in **Areg**. The VLCB implements a virtual link and either the input channel or the output channel may be used as the parameter in **Areg**. **Breg** is a pointer to the area of memory to be used as the packet buffer for the input channel.

Definition:

```
if (Areg does not cause Unalign trap and Breg does not cause Unalign trap)
{
    if (Areg is virtual channel)
    {
        set buffer pointer to Breg
        set header to null header
        deactivate input and output channels
        initialize input and output channels to 'empty'
    }
    else
        IntegerError

    Areg' ← Creg
    Breg' ← undefined
    Creg' ← undefined
}
```

Error signals:

PrivInstruction signalled if executed by a P-process
Unalign signalled if the addresses in **Areg** or **Breg** are not word aligned
IntegerError signalled if the address in **Areg** is not a virtual channel

Comments:

Instruction is privileged
Secondary instruction

See chapter: 12

insertqueue insert at front of scheduler queue

Code: 60 F2

Description: Insert a list of processes at the front of the scheduling list of priority indicated by **Areg**, where 0 indicates high priority and 1 indicates low priority. **Breg** and **Creg** are the front and back, respectively, of the list to be inserted.

Definition:

```
if (Breg ≠ NotProcess.p)
{
  word'[Creg @ pw.Link] ← FptrReg[Areg]
  FptrReg'[Areg] ← Breg

  if (FptrReg[Areg] = NotProcess.p)
    BptrReg'[Areg] ← Creg
}

Areg' ← undefined
Breg' ← undefined
Creg' ← undefined
```

Error signals:
PrivInstruction signalled if executed by a P-process

Comments:
Instruction is privileged
Secondary instruction
The effect of the instruction is undefined if **Areg** does not contain 0 or 1
Breg and **Creg** must be word aligned

See chapter: 13

See also: *swapqueue*

insphdr inspect header

Code: 2B FC

Description: Inspect the header of the virtual link one of whose channels is pointed to by **Areg** to determine the link number, header length and offset.

Definition:

```
if (Areg does not cause Unalign trap)
{
  if (Areg is virtual input channel or Areg is virtual output channel)
  {
    Areg' ← number (0..3) of physical link used by virtual channel Areg

    if (null header)
    {
      Breg' ← NullHeader
      Creg' ← NullOffset
    }
    else if (long header)
    {
      Breg' ← header length of channel
      Creg' ← header offset of channel
    }
    else
    {
      Breg' ← header length of channel
      Creg' ← NullOffset
    }
  }
  else
    IntegerError
}
```

Error signals:

PrivInstruction signalled if executed by a P-process
Unalign signalled if the address in **Areg** is not word aligned
IntegerError signalled if the address in **Areg** is not a virtual channel

Comments:

Instruction is privileged
Secondary instruction

See chapter: 12

See also: *readhdr sethdr writehdr*

intdis — interrupt disable

Code: 2C F4

Description: Disable interrupts until either an *intenb* instruction is executed or the process deschedules. Timeslicing does not occur while interrupts are disabled. This instruction is only meaningful for low priority processes.

Definition:
disable interrupts

Error signals:
PrivInstruction signalled if executed by a P-process

Comments:
Instruction is privileged
Secondary instruction

See chapter: 13

See also: *intenb settimeslice*

intenb interrupt enable

Code: 2C F5

Description: Enable interrupts. This instruction is only meaningful for low priority processes.

Definition:
enable interrupts

Error signals:
PrivInstruction signalled if executed by a P-process

Comments:
Instruction is privileged
Secondary instruction

See chapter: 13

See also: *intdis settimeslice*

irdsq	insert at front of RDS queue

Code: 62 FB

Description: Insert a list of resource channels at the front of a resource data structure (RDS) list. **Areg** is a pointer to the RDS, **Breg** is the address of the channel at the front of the resource channel list to be inserted, and **Creg** is the address of the channel at the back of the list.

Definition:

```
word'[Areg @ rds.Front] ← Breg

if (word[Areg @ rds.Front] = NotProcess.p)
  word'[Areg @ rds.Back] ← Creg
else
  word'[RC @ rc.Ptr] ← word[Areg @ rds.Front]

where
  for an external channel
    RC = ((Creg − MinEventChannel) + ExternalRCbase)

  for an internal channel
    RC = (Creg @ 1)

FPAreg' ← undefined
FPBreg' ← undefined
FPCreg' ← undefined
Areg'   ← undefined
Breg'   ← undefined
Creg'   ← undefined
```

Error signals:
PrivInstruction signalled if executed by a P-process
Unalign signalled if the address in **Areg** is not word aligned

Comments:
Instruction is privileged
Secondary instruction
Breg and **Creg** must be word aligned

See chapter: 12

See also: *erdsq grant irdsq ldresptr mkrc stresptr unmkrc*

j n	jump

Code: Function 0

Description: Unconditional relative jump. The destination of the jump is expressed as a byte offset from the first byte after the current instruction. *j 0* causes a breakpoint.

Definition:

```
if (n = 0)
   take a 'breakpoint' trap
else
   IptrReg' ← next instruction + n

FPAreg' ← undefined
FPBreg' ← undefined
FPCreg' ← undefined
Areg'   ← undefined
Breg'   ← undefined
Creg'   ← undefined
```

Error signals: *none*

Comments:
Primary instruction
Instruction is a descheduling point
Instruction is a timeslicing point

See chapter: 7

See also: *cj lend*

ladd long add

Code: 21 F6

Description: Add with carry in and check for overflow. The result of the operation is the sum of **Areg**, **Breg** and bit 0 of **Creg**.

Definition:

```
if (sum > MostPos)
{
  Areg' ← sum − 2^BitsPerWord
  IntegerOverflow
}
else if (sum < MostNeg)
{
  Areg' ← sum + 2^BitsPerWord
  IntegerOverflow
}
else
  Areg' ← sum

Breg' ← undefined
Creg' ← undefined

where sum = Areg + Breg + Creg_0
        — the value of sum is calculated to unlimited precision
```

Error signals:
IntegerOverflow can be signalled

Comments:
Secondary instruction

See chapter: 7

See also: *lsum*

lb	load byte

Code: F1

Description: Load the unsigned byte addressed by **Areg** into **Areg**.

Definition:

$\texttt{Areg}'_{0..7} \leftarrow \texttt{byte[Areg]}$

$\texttt{Areg}'_{8..\texttt{BitsPerWord}-1} \leftarrow 0$

Error signals:

AccessViolation signalled in a P-process if the address in **Areg** is protected

Comments:

Secondary instruction

See chapter: 7

See also: *bsub devlb lbx*

lbx load byte and sign extend

Code: 2B F9

Description: Load the byte addressed by **Areg** into **Areg** and sign extend to a word.

Definition:

$Areg'_{0..7} \leftarrow byte[Areg]$

$Areg'_{8..BitsPerWord-1} \leftarrow Areg'_{7}$

Error signals:

AccessViolation signalled in a P-process if the address in **Areg** is protected

Comments:

Secondary instruction

See chapter: 7

See also: *bsub devlb lb xbword*

ldc n	load constant

Code: Function 4

Description: Load constant into **Areg**.

Definition:

```
Areg' ← n

Breg' ← Areg
Creg' ← Breg
```

Error signals: *none*

Comments:
Primary instruction

See chapter: 7

See also: *adc mint*

ldchstatus load channel status

Code: 2C F3

Description: Get the status of an external channel pointed to by **Areg**. After execution, **Areg** contains the channel status. If the channel is in resource mode then **Breg** points to the resource data structure and **Creg** contains the identifier given to the channel, otherwise **Breg** contains the process descriptor of the process using the channel and **Creg** is undefined.

Definition:

```
if (Areg does not cause Unalign trap)
{
    if (Areg is virtual channel)
    {
        Areg' ← channel status

        if (channel is stopping)
            Breg' ← NotProcess.p
        else if (channel in resource mode)
        {
            Breg' ←  word[RC @ rc.Ptr]
            Creg' ←  word[RC @ rc.Id]
        }
        else
        {
            Breg' ← process descriptor of process using channel
            Creg' ← undefined
        }
    }
    else if (Areg is event channel)
    {
        Areg' ← channel status

        if (channel in resource mode)
        {
            Breg' ←  word[RC @ rc.Ptr]
            Creg' ←  word[RC @ rc.Id]
        }
        else
        {
            Breg' ← process descriptor of process using channel
            Creg' ← undefined
        }
    }
    else if (Areg is hard channel)
    {
        Areg' ← channel status

        Breg' ← process descriptor of process using channel
        Creg' ← undefined
    }
    else
        integerError
}
```

(continued)

Definition (continued):

```
where RC = ((Areg − MinEventChannel) + ExternalRCbase)
```

If the channel is a hard link then the bits in `Areg'` *are as follows:*

0	*set if error detected*
1–2	*undefined*
3	*set if parity error detected*
4	*set if disconnect error detected*
5–30	*undefined*
31	*0 (hard channel)*

If the channel is a virtual channel or an event channel then the bits in `Areg'` *are as follows:*

0	*channel mode: 0 = deactivated; 1 = activated*
1	*set if channel mode = stopping*
2	*undefined*
3	*set if in resource mode*
4	*set if packet or acknowledge pending*
5	*set if schedule pending*
6–30	*undefined*
31	*channel type: 0 = event; 1 = virtual*

Error signals:

PrivInstruction signalled if executed by a P-process
Unalign signalled if the address in **Areg** is not word aligned
IntegerError signalled if the address in **Areg** is not an external channel

Comments:

Instruction is privileged
Secondary instruction

See chapter: 12

ldcnt load message byte count

Code: 2C F0

Description: Load the number of bytes of data received by the previous *vin* instruction. This instruction must be executed after a *vin* and before a descheduling point, or other instruction which changes the workspace pointer, is executed.

Definition:

```
Areg′ ← word[Wptr @ pw.Length]

if (Areg′ = LengthError.p)
   IntegerError

Breg′ ← Areg
Creg′ ← Breg
```

Error signals:

PrivInstruction signalled if executed by a P-process
IntegerError signalled if the message was too long

Comments:

Instruction is privileged
Secondary instruction

See chapter: 8

See also: *vin vout*

ldconf	load from configuration register

Code: 2B FE

Description: Read configuration register addressed by **Areg** into **Areg**. This instruction is only intended for use during booting. It can be used by a running program but it will have a serious impact on interrupt latency.

Definition:

```
Areg' ← ConfigReg[Areg]
```

Error signals:

PrivInstruction signalled if executed by a P-process
IntegerError signalled if **Areg** is not a valid readable configuration address

Comments:

Instruction is privileged
Secondary instruction

See chapter: 12

See also: *stconf ldmemstartval*

lddevid load device identity

Code: 21 27 FC

Description: See *ldprodid*. *This instruction may be removed in future so* ldprodid *should be used instead.*

Definition:

```
Areg′ ← DeviceId

Breg′ ← Areg
Creg′ ← Breg
```

Error signals: *none*

Comments:
Secondary instruction

See also: *ldprodid*

ldflags — load error flags

Code: 2B F6

Description: Load the current error flags and trap enable bits into **Areg**. The other bits in **Areg** are set to zero.

Definition:

```
Areg' ← flag and trap enable bits from StatusReg

Breg' ← Areg
Creg' ← Breg
```

Error signals: *none*

Comments:
Secondary instruction

See chapter: 10

See also: *stflags*

ldiff	long diff

Code: 24 FF

Description: Subtract unsigned numbers with borrow in. Subtract **Areg** from **Breg** minus borrow in from **Creg,** producing difference in **Areg** and borrow out in **Breg**, without checking for overflow.

Definition:

```
if (diff_unsigned ≤ MostPosUnsigned)
{
   Areg'_unsigned ← diff_unsigned
   Breg' ← 0
}
else
{
   Areg'_unsigned ← diff_unsigned + 2^BitsPerWord
   Breg' ← 1
}

Creg' ← undefined

where diff_unsigned = Breg_unsigned − Areg_unsigned − Creg_0
        — the value of diff is calculated to unlimited precision
```

Error signals: *none*

Comments:
Secondary instruction

See chapter: 7

See also: *lsub*

ldiv | long divide

Code: 21 FA

Description: Divide the double length unsigned integer in **Breg** and **Creg** (most significant word in **Creg**) by an unsigned integer in **Areg**. The quotient is put into **Areg** and the remainder into **Breg**. Overflow occurs if either the quotient is not representable in a single word, or if a division by zero is attempted; the condition for overflow is equivalent to $\mathbf{Creg}_{unsigned} \geq \mathbf{Areg}_{unsigned}$.

Definition:

```
if (Creg_unsigned ≥ Areg_unsigned)
    IntegerOverflow
else
{
    Areg'_unsigned ← long_unsigned / Areg_unsigned
    Breg'_unsigned ← long_unsigned rem Areg_unsigned
}

Creg' ← undefined

where long_unsigned = (Creg_unsigned × 2^BitsPerWord) + Breg_unsigned
        — the value of long_unsigned is calculated to unlimited precision
```

Error signals:
IntegerOverflow can occur

Comments:
Secondary instruction

See chapter: 7

See also: *lmul*

ldl n	load local

Code: Function 7

Description: Load local variable at specified word offset in workspace, into **Areg**.

Definition:

```
Areg′ ← word[Wptr @ n]

Breg′ ← Areg
Creg′ ← Breg
```

Error signals:
AccessViolation signalled in a P-process if the address accessed is protected

Comments:
Primary instruction

See chapter: 7

See also: *stl*

ldlp n	load local pointer

Code: Function 1

Description: Load address of local variable at specified offset in workspace, into **Areg**.

Definition:

```
Areg′ ← Wptr @ n

Breg′ ← Areg
Creg′ ← Breg
```

Error signals: *none*

Comments:
Primary instruction

See chapter: 7

See also: *ldl*

ldmemstartval load value of *MemStart* address

Code: 27 FE

Description: Load the address of the first free memory location (as defined in the **MemStart** configuration register) into **Areg**.

Definition:
```
Areg' ← MemStart

Breg' ← Areg
Creg' ← Breg
```

Error signals:
PrivInstruction signalled if executed by a P-process

Comments:
Instruction is privileged
Secondary instruction

See chapter: 12

ldnl n	load non-local

Code: Function 3

Description: Load non-local variable at specified word offset from **Areg**, into **Areg**.

Definition:

```
Areg' ← word[Areg @ n]
```

Error signals:

Unalign signalled if the address in **Areg** is not word aligned
AccessViolation signalled in a P-process if the address accessed is protected

Comments:

Primary instruction

See chapter: 7

See also: *ldl ldnlp stnl*

ldnlp n	load non-local pointer

Code: Function 5

Description: Load address at specified word offset from address in **Areg**, into **Areg**.

Definition:

```
Areg′ ← Areg @ n
```

Error signals: *none*

Comments:
Primary instruction

See chapter: 7

See also: *ldlp ldnl wsub*

ldpi	load pointer to instruction

Code: 21 FB

Description: Load an address relative to the current instruction pointer into **Areg**. **Areg** contains a byte offset which is added to the address of the first byte following this instruction.

Definition:

`Areg'` ← *next instruction* + `Areg`

Error signals: *none*

Comments:

Secondary instruction

See chapter: 7

ldpri	load current priority

Code: 21 FE

Description: Load current process priority into **Areg**. This is zero for high priority and one for low priority.

Definition:
```
Areg′ ← Priority

Breg′ ← Areg
Creg′ ← Breg
```

Error signals: *none*

Comments:
Secondary instruction

See chapter: 8

ldprodid	load product identity

Code: 68 FC

Description: Load a value (the *device identity*) indicating the processor type into **Areg**.

Definition:

```
Areg′ ← DeviceId

Breg′ ← Areg
Creg′ ← Breg
```

Error signals: *none*

Comments:
Secondary instruction

ldresptr load resource queue pointer

Code: 62 F8

Description: Load a pointer to the next channel in a resource channel list, after the channel pointed to by **Areg**.

Definition:

```
if (Areg does not cause Unalign trap)
{
  if (Areg is event channel or Areg is internal channel or Areg is virtual input channel)
    Areg′ ← word[RC @ rc.Ptr]
  else
    IntegerError
}

where
  for an external channel
    RC = ((Areg − MinEventChannel) + ExternalRCbase)

  for an internal channel
    RC = Areg @ 1
```

Error signals:

PrivInstruction signalled if executed by a P-process
Unalign signalled if the address in **Areg** is not word aligned
IntegerError signalled if the address in **Areg** is not a virtual input or event channel or a legal internal channel address

Comments:

Instruction is privileged
Secondary instruction
The result of this instruction is undefined if:
- channel **Areg** is the last channel in the list
- the channel is *idle*
- the channel is not in resource mode.

See chapter: 12

See also: *erdsq irdsq stresptr*

ldshadow load shadow registers

Code: 60 FC

Description: Load some shadow registers (determined by **Areg** and **Breg**) from the block of store addressed by **Creg**. **FPAreg.sh**, **FPBreg.sh** and **FPCreg.sh** occupy 2 words each when stored. The floating point registers must have been stored in the same form as is given by the *stshadow* instruction.

Definition:

if (Priority = 1) **or** (Areg ≥ Breg) **or** (Areg < 0) **or** (Breg > 7)
  undefined effect
else
  load the shadow registers between the shadow markers specified in Areg *and* Breg *from consecutive increasing addresses in the order listed below into the block of memory starting at address* Creg

markers *registers*

0 –
  RegionReg0.sh, RegionReg1.sh, RegionReg2.sh, RegionReg3.sh *(4 words)*
1 –
  PstateReg.sh, WdescStubReg.sh *(2 words)*
2 –
  ThReg.sh *(1 word)*
3 –
  StatusReg.sh *(1 word)*
4 –
  WdescReg.sh, IptrReg.sh, Areg.sh, Breg.sh, Creg.sh, Ereg.sh, Xreg.sh, BMreg0.sh, BMreg1.sh, BMreg2.sh, EptrReg.sh *(11 words)*
5 –
  FPstatusReg.sh, FPAreg.sh, FPBreg.sh, FPCreg.sh *(7 words)*
6 –
  WlReg.sh, WuReg.sh *(2 words)*
7 –

FPAreg′ ← *undefined*
FPBreg′ ← *undefined*
FPCreg′ ← *undefined*
Areg′ ← *undefined*
Breg′ ← *undefined*
Creg′ ← *undefined*

Error signals:

PrivInstruction signalled if executed by a P-process
Unalign signalled if the address in **Creg** is not word aligned

Comments:

Instruction is privileged
Secondary instruction
The effect of the instruction is defined only when executed by a high-priority process

See chapter: 13

See also: *stshadow*

ldth	load trap handler

Code: 2C F2

Description: Load the contents of the trap handler register into **Areg**.

Definition:

```
Areg' ← ThReg

Breg' ← Areg
Creg' ← Breg
```

Error signals:
PrivInstruction signalled if executed by a P-process

Comments:
Instruction is privileged
Secondary instruction

See chapter: 10

ldtimer | load timer

Code: 22 F2

Description: Load value of current priority timer into **Areg**.

Definition:

```
Areg' ← ClockReg[Priority]

Breg' ← Areg
Creg' ← Breg
```

Error signals: *none*

Comments:
Secondary instruction

See chapter: 8

See also: *sttimer tin*

lend loop end

Code: 22 F1

Description: Adjust loop count and index, and do a conditional jump. Initially **Areg** contains the byte offset from the first byte following this instruction to the loop start and **Breg** contains a pointer to a loop end data structure, the first word of which is the loop index and the second is the loop count. The count is decremented and, if the result is greater than zero, the index is incremented and a jump to the start of the loop is taken. The offset to the start of the loop is given as a positive number that is subtracted from the instruction pointer.

Definition:

```
word'[Breg @ le.Count] ← word[Breg @ le.Count] − 1

if (word'[Breg @ le.Count] > 0)
{
  word'[Breg @ le.Index] ← word[Breg @ le.Index] + 1
  IptrReg' ← next instruction − Areg
}

FPAreg' ← undefined
FPBreg' ← undefined
FPCreg' ← undefined
Areg'   ← undefined
Breg'   ← undefined
Creg'   ← undefined
```

Error signals:

Unalign signalled if the address in **Breg** is not word aligned
AccessViolation signalled in a P-process if any address accessed is not writable

Comments:

Secondary instruction
Instruction is a descheduling point
Instruction is a timeslicing point

See chapter: 7

See also: *cj j*

lmul long multiply

Code: 23 F1

Description: Form the double length product of **Areg** and **Breg**, with **Creg** as carry in, treating the initial values as unsigned.

Definition:

$Areg'_{unsigned} \leftarrow prod_{unsigned}$ **rem** $2^{BitsPerWord}$
$Breg'_{unsigned} \leftarrow prod_{unsigned} / 2^{BitsPerWord}$

$Creg' \leftarrow$ *undefined*

where $prod_{unsigned} = (Breg_{unsigned} \times Areg_{unsigned}) + Creg_{unsigned}$
— *the value of* $prod_{unsigned}$ *is calculated to unlimited precision*

Error signals: *none*

Comments:
Secondary instruction

See chapter: 7

See also: *ldiv*

ls	load sixteen

Code: 2C FA

Description: Load the unsigned 16 bit object addressed by **Areg** into **Areg**.

Definition:

$\texttt{Areg}'_{0..15} \leftarrow \texttt{sixteen[Areg]}$

$\texttt{Areg}'_{16..BitsPerWord-1} \leftarrow 0$

Error signals:

Unalign signalled if the address in **Areg** is not 16 bit aligned
AccessViolation signalled in a P-process if the address in **Areg** is protected

Comments:

Secondary instruction

See chapter: 7

See also: *devls lsx ss ssub*

lshl	long shift left

Code: 23 F6

Description: Logical shift left the double word value in **Creg** and **Breg** (most significant word in **Creg**) by the number of places specified in **Areg**.

Definition:

$\text{Areg}' \leftarrow (\text{long} << \text{Areg}_{unsigned})\ \textbf{rem}\ 2^{\text{BitsPerWord}}$

$\text{Breg}' \leftarrow ((\text{long} << \text{Areg}_{unsigned})\ /\ 2^{\text{BitsPerWord}})\ \textbf{rem}\ 2^{\text{BitsPerWord}}$

$\text{Creg}' \leftarrow$ *undefined*

$\textbf{where}\ \text{long} = (\text{Creg}_{unsigned} \times 2^{\text{BitsPerWord}}) + \text{Breg}_{unsigned}$
— *the value of* `long` *is calculated to double word precision*

Error signals: *none*

Comments:
Secondary instruction

See chapter: 7

See also: *lshr norm*

lshr long shift right

Code: 23 F5

Description: Logical shift right the double word value in **Creg** and **Breg** (most significant word in **Creg**) by the number of places specified in **Areg**.

Definition:

$\texttt{Areg}' \leftarrow (\texttt{long} >> \texttt{Areg}_{unsigned})\ \textbf{rem}\ 2^{\texttt{BitsPerWord}}$

$\texttt{Breg}' \leftarrow ((\texttt{long} >> \texttt{Areg}_{unsigned})\ /\ 2^{\texttt{BitsPerWord}})\ \textbf{rem}\ 2^{\texttt{BitsPerWord}}$

$\texttt{Creg}' \leftarrow$ *undefined*

where $\texttt{long} = (\texttt{Creg}_{unsigned} \times 2^{\texttt{BitsPerWord}}) + \texttt{Breg}_{unsigned}$
— *the value of* `long` *is calculated to double word precision*

Error signals: *none*

Comments:
Secondary instruction

See chapter: 7

See also: *lshl*

lsub long subtract

Code: 23 F8

Description: Subtract with borrow in and check for overflow. The result of the operation, put into **Areg,** is **Breg** minus **Areg**, minus bit 0 of **Creg**.

Definition:
```
if (diff > MostPos)
{
  Areg′ ← diff − 2^BitsPerWord
  IntegerOverflow
}
else if (diff < MostNeg)
{
  Areg′ ← diff + 2^BitsPerWord
  IntegerOverflow
}
else
  Areg′ ← diff

Breg′ ← undefined
Creg′ ← undefined

where diff = (Breg − Areg) − Creg_0
        — the value of diff is calculated to unlimited precision
```

Error signals:
IntegerOverflow can be signalled

Comments:
Secondary instruction

See chapter: 7

See also: *ldiff*

lsum long sum

Code: 23 F7

Description: Add unsigned numbers with carry in and carry out. Add **Breg** to **Areg** (treated as unsigned numbers) plus carry in from **Creg,** producing the sum in **Areg** and carry out in **Breg**, without checking for overflow.

Definition:

```
if (sum_unsigned > MostPosUnsigned)
{
   Areg'_unsigned  ← sum_unsigned − 2^BitsPerWord
   Breg'           ← 1
}
else
{
   Areg'_unsigned  ← sum_unsigned
   Breg'           ← 0
}

Creg' ← undefined

where sum_unsigned = Areg_unsigned + Breg_unsigned + Creg_0
        — the value of sum is calculated to unlimited precision
```

Error signals: *none*

Comments:
Secondary instruction

See chapter: 7

See also: *ladd*

lsx	load sixteen and sign extend

Code: 2F F9

Description: Load the 16 bit object addressed by **Areg** into **Areg** and sign extend to a word.

Definition:

$\texttt{Areg}'_{0..15} \leftarrow \texttt{sixteen[Areg]}$

$\texttt{Areg}'_{16..BitsPerWord-1} \leftarrow \texttt{Areg}'_{15}$

Error signals:

Unalign signalled if the address in **Areg** is not 16 bit aligned

AccessViolation signalled in a P-process if the address in **Areg** is protected

Comments:

Secondary instruction

See chapter: 7

See also: *devls ls ss xsword*

mint minimum integer

Code: 24 F2

Description: Load the most negative integer into **Areg**.

Definition:

```
Areg' ← MostNeg

Breg' ← Areg
Creg' ← Breg
```

Error signals: *none*

Comments:
Secondary instruction

See chapter: 7

mkrc mark resource channel

Code: 62 FC

Description: Set the resource channel specified by **Areg** into resource mode. **Breg** is a pointer to a resource data structure (RDS) and **Creg** is the identifier to be given to the resource channel. If the channel is empty it will become an idle resource channel. If the channel is waiting it will be added to the list waiting on the RDS.

Definition:

```
if (Areg does not cause Unalign trap)
{
  if (Areg is virtual input channel or Areg is event channel or Areg is internal channel)
  {
    if (word[RC @ rc.Id] = NotProcess.p)
    {
      word'[RC @ rc.Id] ← Creg

      if (Areg is internal channel)
      {
        if (word[Areg] = NotProcess.p)                         — channel is empty
        {
          word'[RC @ rc.Ptr] ← Breg
          word'[Areg] ← ResChan.p                              — state = idle
        }
        else                                                   — client is waiting
          add channel Areg to list on RDS pointed to by Breg
      }
      else                                                     — Areg is an external channel
      {
        word'[RC @ rc.Ptr] ← Breg
        put channel into resource mode                         — state = queued
      }
    }
  }
  else
    IntegerError
}

Areg' ← undefined
Breg' ← undefined
Creg' ← undefined

where
  for an external channel
    RC = ((Areg − MinEventChannel) + ExternalRCbase)
  for an internal channel
    RC = Areg @ 1
```

Error signals:

PrivInstruction signalled if executed by a P-process
Unalign signalled if the address in **Areg** is not word aligned
IntegerError signalled if the address in **Areg** is not a virtual input or event channel or a legal internal channel address

Comments:

Instruction is privileged
Secondary instruction
The effect of the instruction is undefined if there is both a client waiting on the channel and a server waiting on the RDS

See chapter: 12

See also: *grant unmkrc*

move	move message

Code: 24 FA

Description: Copy **Areg** bytes to address **Breg** from address **Creg**. The copy is performed using the minimum number of word reads and writes.

Definition:

```
if (source and destination overlap)
  undefined effect

else for i = 0..(unsign(Areg) − 1)
  byte'[Breg + i] ← byte[Creg + i]

Areg' ← undefined
Breg' ← undefined
Creg' ← undefined
```

Error signals:
AccessViolation signalled in a P-process if any address accessed is protected

Comments:
Secondary instruction
Instruction is interruptible
The effect of the instruction is undefined if the source and destination overlap

See chapter: 7

See also: *devmove in move2dall out*

move2dall 2D block copy

Code: 25 FC

Description: Copy a 2D block of memory to another, non-overlapping, area using parameters set up by *move2dinit*. The copy is performed using the minimum number of word reads and writes. **Areg** is the number of bytes in each row, **Breg** is the address of the destination, and **Creg** is the address of the source.

Definition:

```
if (source and destination overlap)
  undefined effect

else for y = 0..(count − 1)
  for x = 0..(Areg_unsigned − 1)
    byte′[Breg + (y × dstStride) + x] ← byte[Creg + (y × srcStride) + x]

where
  count     = BMreg0_unsigned
  dstStride = BMreg1
  srcStride = BMreg2

Areg′ ← undefined
Breg′ ← undefined
Creg′ ← undefined

BMreg0′ ← undefined
BMreg1′ ← undefined
BMreg2′ ← undefined
```

Error signals:
AccessViolation signalled in a P-process if any address accessed is protected

Comments:
Secondary instruction
Instruction is interruptible

See chapter: 7

See also: *move2dinit*

move2dinit	initialize data for 2D block move

Code: 25 FB

Description: Set up the first three parameters for a 2D block move: **Areg** is the number of rows to copy, **Breg** is the width of destination array, and **Creg** is the width of source array. This must be executed before *each* 2D block move.

Definition:

```
BMreg0' ← Areg
BMreg1' ← Breg
BMreg2' ← Creg

Areg'   ← undefined
Breg'   ← undefined
Creg'   ← undefined
```

Error signals: *none*

Comments:
Secondary instruction

See chapter: 7

See also: *move2dall move2dnonzero move2dzero stmove2dinit*

move2dnonzero — 2D block copy non–zero bytes

Code: 25 FD

Description: Copy non-zero valued bytes from a 2D block of memory to another, non-overlapping, area using parameters set up by *move2dinit*. The copy is performed using the minimum number of word reads and writes. **Areg** is the number of bytes in each row, **Breg** is the address of the destination, and **Creg** is the address of the source.

Definition:

```
if (source and destination overlap)
   undefined effect

else for y = 0..(count − 1)
   for x = 0..(Areg_unsigned − 1)
      if (byte[Creg + (y × srcStride) + x] ≠ 0)
         byte'[Breg + (y × dstStride) + x] ← byte[Creg + (y × srcStride) + x]

where
   count     = BMreg0_unsigned
   dstStride = BMreg1
   srcStride = BMreg2

Areg' ← undefined
Breg' ← undefined
Creg' ← undefined

BMreg0' ← undefined
BMreg1' ← undefined
BMreg2' ← undefined
```

Error signals:
AccessViolation signalled in a P-process if any address accessed is protected

Comments:
Secondary instruction
Instruction is interruptible

See chapter: 7

See also: *move2dinit*

move2dzero — 2D block copy zero bytes

Code: 25 FE

Description: Copy zero valued bytes from a 2D block of memory to another, non-overlapping, area using parameters set up by *move2dinit*. The copy is performed using the minimum number of word reads and writes. **Areg** is the number of bytes in each row, **Breg** is the address of the destination, and **Creg** is the address of the source.

Definition:

```
if (source and destination overlap)
  undefined effect

else for y = 0..(count-1)
  for x = 0..(Areg_unsigned - 1)
    if (byte[Creg+(y×srcStride)+x] = 0)
      byte'[Breg+(y×dstStride)+x]←byte[Creg+(y×srcStride)+x]

where
  count     = BMreg0_unsigned
  dstStride = BMreg1
  srcStride = BMreg2

Areg' ← undefined
Breg' ← undefined
Creg' ← undefined

BMreg0' ← undefined
BMreg1' ← undefined
BMreg2' ← undefined
```

Error signals:
AccessViolation signalled in a P-process if any address accessed is protected

Comments:
Secondary instruction
Instruction is interruptible

See chapter: 7

See also: *move2dinit*

mul multiply

Code: 25 F3

Description: Multiply **Areg** by **Breg**, with checking for overflow.

Definition:

Areg′ ← Areg $\times_{checked}$ Breg

Breg′ ← Creg
Creg′ ← *undefined*

Error signals:
IntegerOverflow can be signalled by $\times_{checked}$

Comments:
Secondary instruction

See chapter: 7

See also: *prod*

nop	no operation

Code: 63 F0

Description: Perform no operation.

Definition:
no effect

Error signals: *none*

Comments:
Secondary instruction

norm normalize

Code: 21 F9

Description: Normalize the unsigned double length number stored in **Breg** and **Areg** (most significant word in **Breg**). The value is shifted left until the most significant bit is a one. The number of places shifted is returned in **Creg**.

Definition:

```
if (Breg_unsigned = 0) and (Areg_unsigned = 0)
  Creg' ← BitsPerWord
else
{
  Creg'            ← number of most significant zero bits in long_unsigned
  Areg'_unsigned   ← (long_unsigned << Creg') rem 2^BitsPerWord
  Breg'_unsigned   ← ((long_unsigned << Creg') / 2^BitsPerWord) rem 2^BitsPerWord
}
where long_unsigned = (Breg_unsigned × 2^BitsPerWord) + Areg_unsigned
        — the value of long_unsigned is calculated to double word precision
```

Error signals: *none*

Comments:

Secondary instruction

See chapter: 7

See also: *lshl lshr shl shr*

not not

Code: 23 F2

Description: Complement bits in **Areg**.

Definition:

```
Areg' ← ~ Areg
```

Error signals: *none*

Comments:
Secondary instruction

See chapter: 7

Code: 24 FB

Description: Bitwise or of **Areg** and **Breg**.

Definition:

```
Areg' ← Breg ∨ Areg

Breg' ← Creg
Creg' ← undefined
```

Error signals: *none*

Comments:
Secondary instruction

See chapter: 7

out output message

Code: FB

Description: Output a message (the corresponding input is performed by an *in* instruction, and must specify a message of the same length). **Areg** is the unsigned length, **Breg** is a pointer to the channel, and **Creg** is a pointer to the message. The process executing *out* will be descheduled if the channel is external or is not ready; it is rescheduled when the communication is complete. This instruction is also used to synchronize with an alternative or a resource.

Definition:

```
if (Breg does not cause Unalign trap)
{
  if (Breg is not a legal channel address)
    IntegerError
  else
    synchronize, and output Areg_unsigned bytes to channel Breg from address Creg
}

FPAreg' ← undefined
FPBreg' ← undefined
FPCreg' ← undefined
Areg'   ← undefined
Breg'   ← undefined
Creg'   ← undefined
```

Error signals:

PrivInstruction signalled if executed by a P-process
Unalign signalled if the address in **Breg** is not word aligned
IntegerError signalled if the address in **Breg** is not a legal channel address

Comments:

Instruction is privileged
Secondary instruction
Instruction is a descheduling point
Instruction is interruptible

See chapter: 8

See also: *altwt enbc enbg grant in mkrc outbyte outword vout*

outbyte output byte

Code: FE

Description: Output the least significant byte of **Areg** to the channel pointed to by **Breg** (the corresponding input is performed by an *in* instruction, and must specify a single byte message). The process executing *outbyte* will be descheduled if the channel is external or is not ready; it is rescheduled when the communication is complete. This instruction is also used to synchronize with an alternative or a resource.

Definition:
```
if (Breg does not cause Unalign trap)
{
  if (Breg is not a legal channel address)
    IntegerError
  else
    synchronize, and output least significant byte of Areg to channel Breg
}

word'[Wptr @ pw.Temp] ← undefined
FPAreg'               ← undefined
FPBreg'               ← undefined
FPCreg'               ← undefined
Areg'                 ← undefined
Breg'                 ← undefined
Creg'                 ← undefined
```

Error signals:
PrivInstruction signalled if executed by a P-process
Unalign signalled if the address in **Breg** is not word aligned
IntegerError signalled if the address in **Breg** is not a legal channel address

Comments:
Instruction is privileged
Secondary instruction
Instruction is a descheduling point
Instruction is interruptible
Uses the **pw.Temp** slot in the process workspace

See chapter: 8

See also: *altwt enbc enbg grant in mkrc out outword*

outword	output word

Code: FF

Description: Output the word in **Areg** to the channel pointed to by **Breg** (the corresponding input is performed by an *in* instruction, and must specify a four byte message). The process executing *outword* will be descheduled if the channel is external or is not ready; it is rescheduled when the communication is complete. This instruction is also used to synchronize with an alternative or a resource.

Definition:

```
if (Breg does not cause Unalign trap)
{
  if (Breg is not a legal channel address)
    IntegerError
  else
    synchronize, and output Areg to channel Breg
}

word'[Wptr @ pw.Temp] ← undefined
FPAreg'               ← undefined
FPBreg'               ← undefined
FPCreg'               ← undefined
Areg'                 ← undefined
Breg'                 ← undefined
Creg'                 ← undefined
```

Error signals:
PrivInstruction signalled if executed by a P-process
Unalign signalled if the address in **Breg** is not word aligned
IntegerError signalled if the address in **Breg** is not a legal channel address

Comments:
Instruction is privileged
Secondary instruction
Instruction is a descheduling point
Instruction is interruptible
Uses the **pw.Temp** slot in the process workspace

See chapter: 8

See also: *altwt enbc enbg grant in mkrc out outbyte*

pop pop processor stack

Code: 27 F9

Description: Pop top element of integer stack.

Definition:

```
Areg' ← Breg
Breg' ← Creg
Creg' ← undefined
```

Error signals: *none*

Comments:
Secondary instruction

See chapter: 7

See also: *dup rev*

prod	product

Code: F8

Description: Multiply **Areg** by **Breg** without checking for overflow.

Definition:

```
Areg′ ← Areg × Breg

Breg′ ← Creg
Creg′ ← undefined
```

Error signals: *none*

Comments:
Secondary instruction

See chapter: 7

See also: *mul*

readbfr read buffer pointer from VLCB

Code: 2B FD

Description: Get a pointer to the buffer (and the length of the packet in the buffer) of a virtual channel pointed to by **Areg**. After the instruction **Areg** holds a pointer to the channel buffer, **Breg** holds the length of the packet in the buffer and **Creg** indicates the state of the buffer: 0 (buffer is empty), 1 (last packet of a message is present), or 2 (any other packet is present).

Definition:

```
if (Areg is virtual input channel)
{
  Areg' ← address of channel buffer

  if (buffer is empty)
  {
    Breg' ← 0
    Creg' ← 0
  }
  else
  {
    Breg' ← length of packet in buffer

    if (buffer contains last packet of a message)
      Creg' ← 1
    else
      Creg' ← 2
  }
}
else
  IntegerError
```

Error signals:

PrivInstruction signalled if executed by a P-process
Unalign signalled if the address in **Areg** is not word aligned
IntegerError signalled if the address in **Areg** is not a virtual input channel

Comments:

Instruction is privileged
Secondary instruction

See chapter: 12

readhdr read virtual channel header

Code: 61 F4

Description: Copy the header from the virtual link one of whose channels is pointed to by **Areg**, to the address pointed to by **Creg**. **Breg** contains the length of the header.

Definition:

```
if ((Areg is virtual input channel or Areg is virtual output channel)
        and (Breg = header length) and header is not null)
  copy header from VLCB of virtual channel Areg to address pointed to by Creg
else
  IntegerError
```

Error signals:

PrivInstruction signalled if executed by a P-process
Unalign signalled if the address in **Areg** is not word aligned
IntegerError signalled if:
- the address in **Areg** is not a virtual channel
- header is not null
- **Breg** is not equal to the header length

Comments:

Instruction is privileged
Secondary instruction
Instruction is interruptible

See chapter: 12

See also: *insphdr sethdr writehdr*

rem remainder

Code: 21 FF

Description: Calculate the remainder when **Breg** is divided by **Areg**. The sign of the remainder is the same as the dividend. The remainder, r = x **rem** y, is defined by r = x − (y × (x / y)).

Definition:

```
if (Areg = 0)
{
  Areg ← undefined
  IntegerOverflow
}
else
  Areg′ ← Areg rem Breg

Breg′ ← Creg
Creg′ ← undefined
```

Error signals:
IntegerOverflow signalled when a remainder by zero is attempted

Comments:
Secondary instruction

See chapter: 7

See also: *div*

resetch reset channel

Code: 21 F2

Description: Reset the channel pointed to by **Areg**. Returns the channel to the empty state, without affecting whether the channel is activated or deactivated. If the channel is a byte-stream channel then the link hardware is reset and restarted. If the channel is a byte-stream channel that is *not* in byte-stream mode then **Areg** becomes undefined, otherwise **Areg** returns the process descriptor of the process waiting on the channel.

Definition:

```
if (Areg does not cause Unalign trap)
{
  if (Areg is internal channel)
  {
    if (process waiting on channel Areg)
      Areg' ← process descriptor of waiting process)
    else
      Areg' ← NotProcess.p

    word'[Areg] ← NotProcess.p
  }
  else if (Areg is virtual channel)
  {
    if (process waiting on channel Areg)
      Areg' ← process descriptor of waiting process)
    else
      Areg' ← NotProcess.p

    reset virtual channel
  }
  else if (Areg is byte-stream channel)
  {
    if (channel not in byte–stream mode)
      Areg' ← undefined
    else if (process waiting on channel Areg)
      Areg' ← process descriptor of waiting process)
    else
      Areg' ← NotProcess.p

    reset link hardware
  }
  else
    IntegerError
}
```

Error signals:
PrivInstruction signalled if executed by a P-process
Unalign signalled if the address in **Areg** is not word aligned
IntegerError signalled if the address in **Areg** is not a legal channel address

Comments:
Instruction is privileged
Secondary instruction

See chapter: 12

See also: *setchmode stopch*

restart restart

Code: 62 FE

Description: Restart execution of an interrupted L-process or null process. **Areg** is a pointer to a P-state data structure (PDS) which contains the state of the interrupted process (which will have been obtained using *stshadow*). The trap-handler of the restarted process will be the trap-handler of the process which executes *restart*. The control word of the PDS contains the state which will be loaded into the **StatusReg** when the restarted process begins executing. Note that information loaded from the control word includes the timeslice disabled bit and the interrupt state. These fields will contain the values which were present when the interrupt occurred. If the instruction causes a 'watchpoint' or 'single-step' trap, then a 'context' trap is taken after the instruction has performed all its writes to memory; the stack state delivered to the trap handler is the state before the instruction started.

Definition:

```
if (Areg does not cause Unalign trap)
{
  if (ThReg ≠ NotProcess.p)
    word'[ThReg @ th.Cntl] ← StatusReg          — process status/control bits only

  if (not single step or watchpoint trap)
  {
    Wptr'       ← word[Areg @ ps.sWptr]
    IptrReg'    ← word[Areg @ ps.sIptr]
    Areg'       ← word[Areg @ ps.sAreg]
    Breg'       ← word[Areg @ ps.sBreg]
    Creg'       ← word[Areg @ ps.sCreg]
    StatusReg'  ← word[Areg @ ps.Cntl]          — process status/control bits only

    if (restarted process was interrupted)
    {
      Ereg'     ← word[Areg @ ps.sEreg]
      EptrReg'  ← word[Areg @ ps.Eptr]
      Xreg'     ← word[Areg @ ps.sXreg]
    }

    if (StatusReg'sb.WtchPntEnbl = 1)
    {
      WlReg'    ← word[Areg @ ps.eWl]
      WuReg'    ← word[Areg @ ps.eWu]
    }

    enable interrupts
  }
}
```

Error signals:

PrivInstruction signalled if executed by a P-process
Unalign signalled if the address in **Areg** is not word aligned

Comments:

Instruction is privileged
Secondary instruction
The *ps.sWptr* slot of the PDS must be word aligned

See chapter: 13

ret	return

Code: 22 F0

Description: Return from subroutine and deallocate workspace.

Definition:

```
IptrReg' ← word[Wptr @ 0]
Wptr'    ← Wptr @ 4
```

Error signals:
AccessViolation signalled in a P-process if the address accessed is protected

Comments:
Secondary instruction

See chapter: 7

See also: *ajw call*

rev reverse

Code: F0

Description: Swap top two elements of integer stack.

Definition:

```
Areg′ ← Breg
Breg′ ← Areg
```

Error signals: *none*

Comments:
Secondary instruction

See chapter: 7

See also: *dup pop fprev*

runp run process

Code: 23 F9

Description: Schedule a (descheduled) process. The process descriptor of the process is in **Areg**; this identifies the process workspace and priority. The instruction pointer and trap handler are loaded from the processes workspace data structure.

Definition:

Put process `Areg` *onto back of appropriate scheduling list*

```
FPAreg′ ← undefined
FPBreg′ ← undefined
FPCreg′ ← undefined
Areg′   ← undefined
Breg′   ← undefined
Creg′   ← undefined
```

Error signals:

PrivInstruction signalled if executed by a P-process

Comments:

Instruction is privileged
Secondary instruction

See chapter: 8

See also: *endp startp stopp*

sb store byte

Code: 23 FB

Description: Store least significant byte of **Breg** into the byte of memory addressed by **Areg**.

Definition:

```
byte'[Areg] ← Breg0..7

Areg' ← Creg
Breg' ← undefined
Creg' ← undefined
```

Error signals:

AccessViolation signalled in a P-process if the address in **Areg** is not writable

Comments:

Secondary instruction

See chapter: 7

See also: *bsub devsb lb lbx ss stnl*

selth	select trap handler

Code: 60 F9

Description: Install a new trap handler (pointed to by **Areg**) for the current process. If the trap handler is in use, the current process is added to the trap handler list and descheduled. If the instruction causes a 'single-step' or 'watchpoint' trap, then a 'context' trap is taken after the status register has been saved, but before the new trap handler has been selected; the stack state delivered to the trap handler is the state before the instruction started.

Definition:

```
if (Areg does not cause Unalign trap)
{
  if (ThReg ≠ NotProcess.p)
    word'[ThReg @ th.Cntl] ← StatusReg                    — process status/control bits only

  if (not single step or watchpoint trap)
  {
    ThReg' ← Areg

    if (ThReg' = NotProcess.p)
      StatusReg' ← default control word

    else if ((word[ThReg' @ th.Cntl] ∧ sb.ThInUse) = 0)
    {
      StatusReg' ← word[ThReg' @ th.Cntl]                 — process status/control bits only

      if (word[ThReg' @ th.Cntl]sb.WtchPntEnbl = 1)
      {
        WlReg' ← word[ThReg' @ th.eWl]
        WuReg' ← word[ThReg' @ th.eWu]
      }
    }
    else
    {
      word'[Wptr @ pw.Iptr]          ← IptrReg
      word'[Wptr @ pw.TrapHandler]   ← ThReg'
      put process on trap handler list
    }
  }
}

FPAreg' ← undefined
FPBreg' ← undefined
FPCreg' ← undefined
Areg'   ← undefined
Breg'   ← undefined
Creg'   ← undefined
```

Error signals:

PrivInstruction signalled if executed by a P-process
Unalign signalled if the address in **Areg** is not word aligned

Comments:

Instruction is privileged
Secondary instruction
Instruction is a descheduling point

See chapter: 10

See also: *ldth tret*

setchmode set channel mode

Code: 61 F7

Description: Activate or deactivate a virtual or event channel, or start or reset a byte-stream channel. **Areg** points to an external channel. If **Breg** is *false* initially then the instruction deactivates a virtual or event channel, or resets a byte-stream channel. If **Breg** is *true* then the instruction activates a virtual or event channel, or starts a byte-stream channel.

Definition:

```
if (Areg does not cause Unalign trap)
{
  if (Areg is not external channel)
    IntegerError
  else if (Breg = false)
    deactivate channel or reset link
  else if (Breg = true)
  {
    if (Areg is virtual channel) and (header is null)
      IntegerError
    else
      activate channel or start link
  }
  else
    IntegerError

Areg' ← Creg
Breg' ← undefined
Creg' ← undefined
}
```

Error signals:

PrivInstruction signalled if executed by a P-process
Unalign signalled if the address in **Areg** is not word aligned
IntegerError signalled if:
- **Breg** is not 0 or 1
- the address in **Areg** is not an external channel
- a virtual channel with a null header is restarted

Comments:

Instruction is privileged
Secondary instruction

See chapter: 12

See also: *resetch stopch*

sethdr set virtual channel header

Code: 61 F8

Description: Set the physical link number and header type of the virtual link one of whose channels is pointed to by **Areg**. **Breg** holds the physical link number and **Creg** holds a word offset from **HdrAreaBase**. The link number of the channel is set to the initial value of **Breg** unless **Breg** is *NullHeader,* when the null header is set. The header offset of the channel is set to the initial value of **Creg** unless **Creg** is *NullOffset,* in which case the offset is unchanged.

Definition:
```
if (Areg does not cause Unalign trap)
{
  if not (Areg is virtual input channel or Areg is virtual output channel)
    IntegerError

  else if (MinLink ≤ Breg) and (Breg ≤ MaxLink)
  {
    if (0 ≤ Creg) and (Creg ≤ MaxHeaderOffset)
    {
      set header link number to Breg
      set header offset to Creg
    }
    else if (Creg = NullOffset)
      set header link number to Breg
    else
      IntegerError
  }
  else if (Breg = NullHeader)
  {
    if (input channel active or output channel active)
      IntegerError
    else
      set null header
  }
  else
    IntegerError
}

Areg' ← undefined
Breg' ← undefined
Creg' ← undefined
```

Error signals:
PrivInstruction signalled if executed by a P-process
Unalign signalled if the address in **Areg** is not word aligned
IntegerError signalled if:
the address in **Areg** is not a virtual channel
Breg or **Creg** contain illegal values

Comments:
Instruction is privileged
Secondary instruction

See chapter: 12

See also: *insphdr readhdr writehdr*

settimeslice — set timeslicing status

Code: 2B F0

Description: Enable or disable timeslicing of the current process, depending on the value of **Areg**, and set **Areg** to indicate whether timeslicing was enabled or disabled prior to execution of the instruction. If **Areg** is initially *false* timeslicing is disabled until either the process deschedules or timeslicing is enabled. If **Areg** is initially *true* timeslicing is enabled. This instruction is only meaningful when run at low priority.

Definition:

```
if (Areg = false)
   disable timeslicing
else if (Areg = true)
   enable timeslicing
else
   undefined effect

if timeslicing was previously enabled
   Areg' ← true
else
   Areg' ← false
```

Error signals:
PrivInstruction signalled if executed by a P-process

Comments:
Instruction is privileged
Secondary instruction

See chapter: 13

See also: *intdis intenb*

shl	shift left

Code: 24 F1

Description: Logical shift left **Breg** by **Areg** places, filling with zero bits. If the initial **Areg** is not between 0 and 31 then the result is zero.

Definition:

```
if (0 ≤ Areg ≤ 31)
  Areg' ← Breg << Areg
else
  Areg' ←  0

Breg'  ← Creg
Creg'  ← undefined
```

Error signals: *none*

Comments:
Secondary instruction

See chapter: 7

See also: *lshl lshr norm shr*

shr shift right

Code: 24 F0

Description: Logical shift right **Breg** by **Areg** places, filling with zero bits. If the initial **Areg** is not between 0 and 31 then the result is zero.

Definition:

```
if (0 ≤ Areg ≤ 31)
  Areg′ ← Breg >> Areg
else
  Areg′ ←  0

Breg′  ← Creg
Creg′  ← undefined
```

Error signals: *none*

Comments:
Secondary instruction

See chapter: 7

See also: *lshl lshr norm shl*

signal signal

Code: 60 F4

Description: Signal (or V) on the semaphore pointed to by **Areg**.

Definition:

```
if (Areg does not cause Unalign trap)
{
  if (word[Areg @ s.Front] = NotProcess.p)
    word'[Areg @ s.Count] ← word[Areg @ s.Count] +checked 1
  else
    remove process from front of semaphore list and put on scheduling list
}

FPAreg' ← undefined
FPBreg' ← undefined
FPCreg' ← undefined
Areg'   ← undefined
Breg'   ← undefined
Creg'   ← undefined
```

Error signals:
PrivInstruction signalled if executed by a P-process
Unalign signalled if the address in **Areg** is not word aligned
IntegerOverflow can be signalled by $+_{checked}$

Comments:
Instruction is privileged
Secondary instruction

See chapter: 8

See also: *wait*

ss store sixteen

Code: 2C F8

Description: Store bits 0..15 of **Breg** into the sixteen bits of memory addressed by **Areg**.

Definition:

sixteen'[Areg] ← $\text{Breg}_{0..15}$

Areg' ← Creg
Breg' ← *undefined*
Creg' ← *undefined*

Error signals:

Unalign signalled if the address in **Areg** is not 16 bit aligned
AccessViolation signalled in a P-process if the address in **Areg** is not writable

Comments:

Secondary instruction

See chapter: 7

See also: *devss ldlp ldnlp ls lsx sb stl stnl ssub*

ssub sixteen subscript

Code: 2C F1

Description: Generate the address of the element which is indexed by **Breg**, in an array of 16–bit objects pointed to by **Areg**.

Definition:

```
Areg' ← Areg + (2 × Breg)

Breg' ← Creg
Creg' ← undefined
```

Error signals: *none*

Comments:
Secondary instruction

See chapter: 7

See also: *bcnt bsub wcnt wsub wsubdb*

startp start process

Code: FD

Description: Create and schedule a process at the current priority. Initially **Areg** is a pointer to the workspace of the new process and **Breg** is the offset from the next instruction to the instruction pointer of the new process. The new process inherits the trap handler of the current process.

Definition:

```
if (Areg does not cause Unalign trap)
{
  word'[Areg @ pw.Iptr] ← next instruction + Breg

  word'[Areg @ pw.TrapHandler] ← ThReg

  put process Areg onto the back of the scheduling list for the current priority
}

FPAreg' ← undefined
FPBreg' ← undefined
FPCreg' ← undefined
Areg'   ← undefined
Breg'   ← undefined
Creg'   ← undefined
```

Error signals:
PrivInstruction signalled if executed by a P-process
Unalign signalled if the address in **Areg** is not word aligned

Comments:
Instruction is privileged
Secondary instruction

See chapter: 8

See also: *endp runp*

stconf	store to configuration register

Code: 2B FF

Description: Write **Breg** into the configuration register addressed by **Areg**. This instruction is only intended for use during booting. It can be used by a running program but it will have a serious impact on interrupt latency.

Definition:

```
ConfigReg'[Areg] ← Breg

Areg' ← Creg
Breg' ← undefined
Creg' ← undefined
```

Error signals:

PrivInstruction signalled if executed by a P-process
IntegerError signalled if **Areg** is not a valid writable configuration address or register is write-locked

Comments:

Instruction is privileged
Secondary instruction

See chapter: 12

See also: *ldconf*

stflags store error flags

Code: 2B F7

Description: Set the error flags and error trap enable bits to the values of the corresponding bits in **Areg**.

Definition:

```
StatusReg' ← flag and trap enable bits from Areg

Areg' ← Breg
Breg' ← Creg
Creg' ← undefined
```

Error signals: *none*

Comments:
Secondary instruction

See chapter: 10

See also: *ldflags*

stl n	store local

Code: Function D

Description: Store **Areg** into the local variable at the specified word offset in workspace.

Definition:

```
word'[Wptr @ n] ← Areg

Areg' ← Breg
Breg' ← Creg
Creg' ← undefined
```

Error signals:
AccessViolation signalled in a P-process if the address accessed is not writable

Comments:
Primary instruction

See chapter: 7

See also: *devsw ldl ldlp sb ss stnl*

stmove2dinit store move2dinit data

Code: 61 F0

Description: Save the state loaded into the processor by the *move2dinit* instruction to the move data structure pointed to by **Areg**.

Definition:

```
word'[Areg @ bmr.count] ← BMreg0
word'[Areg @ bmr.DeltaD] ← BMreg1
word'[Areg @ bmr.DeltaS] ← BMreg2

Areg' ← Breg
Breg' ← Creg
Creg' ← undefined
```

Error signals:

Unalign signalled if the address in **Areg** is not word aligned
AccessViolation signalled in a P-process if any address accessed is not writable

Comments:

Secondary instruction

See chapter: 13

See also: *move2dinit*

stnl n	store non-local

Code: Function E

Description: Store **Breg** into non-local variable at specified word offset from **Areg**.

Definition:

```
word'[Areg @ n] ← Breg

Areg' ← Creg
Breg' ← undefined
Creg' ← undefined
```

Error signals:

Unalign signalled if the address in **Areg** is not word aligned
AccessViolation signalled in a P-process if the address accessed is not writable

Comments:

Primary instruction

See chapter: 7

See also: *devsw ldlp ldnlp sb ss stl*

stopch stop virtual channel

Code: 61 FE

Description: Stop a virtual channel pointed to by **Areg**. If the channel is an input channel it will continue to acknowledge incoming data packets but will otherwise ignore them. The channel will remain in this state until the channel is reset. If it is an output channel it ensures no acknowledge is outstanding for a data packet sent on this channel before allowing the process executing this instruction to continue.

Definition:

```
if (Areg does not cause Unalign trap)
{
  if (Areg is virtual channel)
    stop channel
  else
    IntegerError
}

FPAreg' ← undefined
FPBreg' ← undefined
FPCreg' ← undefined
Areg'   ← undefined
Breg'   ← undefined
Creg'   ← undefined
```

Error signals:

PrivInstruction signalled if executed by a P-process
Unalign signalled if the address in **Areg** is not word aligned
IntegerError signalled if the address in **Areg** is not a virtual channel

Comments:

Instruction is privileged
Secondary instruction
Instruction is a descheduling point

See chapter: 12

See also: *resetch setchmode*

stopp — stop process

Code: 21 F5

Description: Terminate the current process, saving the current **IptrReg** and **ThReg** for later use. The **StatusReg** is also saved if the process has a non-null trap handler.

Definition:
start next process

```
word'[Wptr @ pw.Iptr] ← next instruction
word'[Wptr @ pw.TrapHandler] ← ThReg
if (ThReg ≠ NotProcess.p)
  word'[ThReg @ th.Cntl] ← StatusReg          — process status/control bits only

FPAreg' ← undefined
FPBreg' ← undefined
FPCreg' ← undefined
Areg'   ← undefined
Breg'   ← undefined
Creg'   ← undefined
```

Error signals:
PrivInstruction signalled if executed by a P-process

Comments:
Instruction is privileged
Secondary instruction
Instruction is a descheduling point

See chapter: 8

See also: *endp runp startp*

stresptr store resource queue pointer

Code: 62 F9

Description: Store **Breg** in the **rc.Ptr** slot of the resource channel pointed to by **Areg**. This instruction should only be used on a resource channel which is in a detached queue, i.e. after the queue has been detached by execution of *erdsq.*

Definition:

```
if (Areg does not cause Unalign trap)
{
  if (Areg is internal channel or Areg is virtual input channel or Areg is event channel)
  {
    word'[RCDS @ rc.Ptr] ← Breg

    Areg' ← Creg
    Breg' ← undefined
    Creg' ← undefined
  }
  else
    IntegerError
}

    where
      for an external channel
        RCDS = ((Areg − MinEventChannel) + ExternalRCBase)

      for an internal channel
        RCDS = Areg @ 1
```

Error signals:

PrivInstruction signalled if executed by a P-process
Unalign signalled if the address in **Areg** is not word aligned
IntegerError signalled if **Areg** is not a virtual input or event channel or a legal internal channel address

Comments:

Instruction is privileged
Secondary instruction

See chapter: 12

See also: *erdsq grant irdsq ldresptr mkrc unmkrc*

stshadow	store shadow registers

Code: 60 FD

Description: Store some shadow registers (as determined by **Areg** and **Breg**) into the block of store addressed by **Creg**. **FPAreg.sh**, **FPBreg.sh** and **FPCreg.sh** occupy 2 words each when stored. The floating point registers are stored in the same form as is used by the *fpldall* instruction.

Definition:

```
if (Priority = 1) or (Areg ≥ Breg) or (Areg < 0) or (Breg > 7)
  undefined effect
else
  store the shadow registers between the shadow markers specified in Areg and Breg in consecutive
  increasing addresses in the order listed below into the block of memory starting at address Creg

  markers   registers
   0 –
            RegionReg0.sh, RegionReg1.sh, RegionReg2.sh,
            RegionReg3.sh  (4 words)
   1 –
            PstateReg.sh, WdescStubReg.sh  (2 words)
   2 –
            ThReg.sh  (1 word)
   3 –
            StatusReg.sh  (1 word)
   4 –
            WdescReg.sh, IptrReg.sh, Areg.sh, Breg.sh, Creg.sh, Ereg.sh,
            Xreg.sh, BMreg0.sh, BMreg1.sh, BMreg2.sh, EptrReg.sh  (11 words)
   5 –
            FPstatusReg.sh, FPAreg.sh, FPBreg.sh, FPCreg.sh  (7 words)
   6 –
            WlReg.sh, WuReg.sh  (2 words)
   7 –

FPAreg' ← undefined
FPBreg' ← undefined
FPCreg' ← undefined
Areg'   ← undefined
Breg'   ← undefined
Creg'   ← undefined
```

Error signals:

PrivInstruction signalled if executed by a P-process
Unalign signalled if the address in **Creg** is not word aligned

Comments:

Instruction is privileged
Secondary instruction
The effect of the instruction is defined only when executed by a high-priority process

See chapter: 13

See also: *ldshadow*

sttimer	store timer

Code: 25 F4

Description: Initialize timers. Set the low and high priority clock registers to the value in **Areg** and start them ticking and scheduling ready processes.

Definition:

```
ClockReg'[0] ← Areg
ClockReg'[1] ← Areg
start timers

Areg' ← Breg
Breg' ← Creg
Creg' ← undefined
```

Error signals:
PrivInstruction signalled if executed by a P-process

Comments:
Instruction is privileged
Secondary instruction

See chapter: 13

sub	subtract

Code: FC

Description: Subtract **Areg** from **Breg**, with checking for overflow.

Definition:

```
Areg' ← Breg −checked Areg

Breg' ← Creg
Creg' ← undefined
```

Error signals:
IntegerOverflow can be signalled by $-_{checked}$

Comments:
Secondary instruction

See chapter: 7

See also: *diff add*

sum sum

Code: 25 F2

Description: Add **Areg** and **Breg**, without checking for overflow.

Definition:

```
Areg' ← Breg + Areg

Breg' ← Creg
Creg' ← undefined
```

Error signals: *none*

Comments:
Secondary instruction

See chapter: 7

See also: *add bsub*

swapbfr	swap buffer pointer in VLCB

Code: 61 F9

Description: Store address of the packet buffer pointed to by **Breg** into the VLCB of the virtual input channel pointed to by **Areg**. A pointer to the previous packet buffer is returned in **Areg**.

Definition:

```
if (Areg does not cause Unalign trap and Breg does not cause Unalign trap)
{
  if (Areg is virtual input channel)
  {
    set packet buffer for channel Areg to address in Breg
    Areg' ← address of previous packet buffer of channel Areg
  }
  else
    IntegerError

  Breg' ← Creg
  Creg' ← undefined
}
```

Error signals:

PrivInstruction signalled if executed by a P-process
Unalign signalled if the addresses in **Areg** and **Breg** are not both word aligned
IntegerError signalled if the address in **Areg** is not a virtual input channel

Comments:

Instruction is privileged
Secondary instruction

See chapter: 12

swapqueue	swap scheduler queue

Code: 60 F0

Description: Swap the scheduling list of priority indicated by **Areg**, where 0 indicates high priority and 1 indicates low priority. **Breg** and **Creg** are the front and back, respectively, of the list to be inserted. The old front and back pointers are returned in the **Areg** and **Breg**, respectively.

Definition:

```
Areg'            ← FptrReg[Areg]
Breg'            ← BptrReg[Areg]
FptrReg'[Areg] ← Breg
BptrReg'[Areg] ← Creg

Creg' ← undefined
```

Error signals:
PrivInstruction signalled if executed by a P-process

Comments:
Instruction is privileged
Secondary instruction
Breg and **Creg** must be word aligned

See chapter: 13

See also: *insertqueue swaptimer*

swaptimer swap timer queue

Code: 60 F1

Description: Swap the timer list of priority indicated by **Areg** and update the alarm register for the new list. An initial **Areg** of value 0 indicates high priority and 1 indicates low priority. **Breg** is the front pointer of the list to be inserted. The old front pointer is returned in the **Areg**.

Definition:

```
if (Breg ≠ NotProcess.p)
  TNextReg'[Areg] ← word[Breg @ pw.Time]

Areg'           ← TptrReg[Areg]
TptrReg'[Areg]  ← Breg

Breg' ← undefined
Creg' ← undefined
```

Error signals:

PrivInstruction signalled if executed by a P-process

Comments:

Instruction is privileged
Secondary instruction
Breg must be word aligned.

See chapter: 13

See also: *swapqueue*

syscall system call

Code: 60 F8

Description: Take a trap, indicating ‘system call’ as the reason.

Definition:
take a ‘syscall’ trap

Error signals: *none*

Comments:
Secondary instruction

See chapter: 10

See also: *goprot tret*

talt timer alt start

Code: 24 FE

Description: Start of timer alternative sequence. The **pw.State** location of the workspace is set to *Enabling.p*, and the **pw.TLink** location is set to *TimeNotSet.p*.

Definition:

enter alternative sequence

```
word'[Wptr @ pw.State] ← Enabling.p
word'[Wptr @ pw.TLink] ← TimeNotSet.p
```

Error signals:

PrivInstruction signalled if executed by a P-process

Comments:

Instruction is privileged
Secondary instruction

See chapter: 8

See also: *alt altend altwt disc disg diss dist enbc enbg enbs enbt taltwt*

taltwt timer alt wait

Code: 25 F1

Description: Wait until one of the enabled guards of a timer alternative has become ready and initialize **pw.Temp** for use during the disabling sequence. If the alternative has no ready guard but may become ready due to a timer, the process is placed onto the timer list.

Definition:

```
if (word[Wptr @ pw.State] = Ready.p)
  word'[Wptr @ pw.Time] ← ClockReg[Priority]
else if (word[Wptr @ pw.Tlink] = TimeNotSet.p)
{
  word'[Wptr @ pw.State] ← Waiting.p
  deschedule process and wait for one of the guards to become ready
}
else if (word[Wptr @ pw.Tlink] = TimeNotSet.p)
{
  if (ClockReg[Priority] after word[Wptr @ pw.Time]
  {
    word'[Wptr @ pw.State] ← Ready.p
    word'[Wptr @ pw.Time]  ← ClockReg[Priority]
  }
  else
  {
    word'[Wptr @ pw.Time] ← (word[Wptr @ pw.Time] + 1)
    insert this process into timer list with alarm time (word[Wptr @ pw.Time] + 1)
    if (no guards ready)
      {
        word'[Wptr @ pw.State] ← Waiting.p
        deschedule process and wait for one of the guards to become ready
      }
  }
}
else
  undefined effect

word'[Wptr @ pw.Temp] ← NoneSelected.o

FPAreg' ← undefined
FPBreg' ← undefined
FPCreg' ← undefined
Areg'   ← undefined
Breg'   ← undefined
Creg'   ← undefined
```

Error signals:
PrivInstruction signalled if executed by a P-process

Comments:
Instruction is privileged
Secondary instruction
Instruction is a descheduling point
Instruction is interruptible
Uses the **pw.Temp** and **pw.State** slots in the process workspace

See chapter: 8

See also: *alt altend altwt disc disg diss dist enbc enbg enbs enbt talt*

testpranal test processor analyzing

Code: 22 FA

Description: Load *true* into **Areg** if an error has occurred since the processor was last reset. This instruction is only intended for use during booting. It can be used by a running program but it will have a serious impact on interrupt latency.

Definition:

```
if error since last reset
  Areg' ← true
else
  Areg' ← false

Breg' ← Areg
Creg' ← Breg
```

Error signals:
PrivInstruction signalled if executed by a P-process

Comments:
Instruction is privileged
Secondary instruction

timeslice timeslice

Code: 60 F3

Description: For an L–process, put the current process on the back of the scheduling list. For a P–process, take a timeslice trap.

Definition:

put current process on back of list
start next process

```
FPAreg' ← undefined
FPBreg' ← undefined
FPCreg' ← undefined
Areg'   ← undefined
Breg'   ← undefined
Creg'   ← undefined
```

Error signals: *none*

Comments:

Secondary instruction
Instruction is a descheduling point
This instruction is unaffected by the current priority or by disabling timeslicing or interrupts

See chapter: 13

tin timer input

Code: 22 FB

Description: If **Areg** is after the value of the current priority clock, deschedule until current priority clock is after the time in **Areg**.

Definition:

```
if not (ClockReg[Priority] after Areg)
{
  word'[Wptr @ pw.State] ← Enabling.p
  word'[Wptr @ pw.Time]  ← (Areg + 1)
  insert this process into timer list with time of (Areg + 1) and start next process
}

FPAreg' ← undefined
FPBreg' ← undefined
FPCreg' ← undefined
Areg'   ← undefined
Breg'   ← undefined
Creg'   ← undefined
```

Error signals:
PrivInstruction signalled if executed by a P-process

Comments:
Instruction is privileged
Secondary instruction
Instruction is a descheduling point
Instruction is interruptible
Uses **pw.State** slot in the process workspace

See chapter: 8

See also: *enbt dist ldtimer talt taltwt*

tret	trap return

Code: 60 FB

Description: Return from trap handler and mark it as 'not in use'. If **Areg** is zero then restart the process which signalled the trap or start a process, with the current priority. If **Areg** is not zero then terminate the current process. The process executing this instruction must have a null trap handler and the **Wptr** must contain the address of a trap handler data structure (THDS).

Definition:

```
if (ThReg ≠ NotProcess.p)
  IntegerError
else
{
  if (word[Wptr @ th.Fptr] ≠ NotProcess.p)
  {
    move processes waiting for trap handler to front of current priority scheduling list
    word'[Wptr @ th.Fptr] ← NotProcess.p
  }

  if (Areg = 0)
  {
    Wptr'       ← word[Wptr @ th.sWptr]
    IptrReg'    ← word[Wptr @ th.sIptr]
    Areg'       ← word[Wptr @ th.sAreg]
    Breg'       ← word[Wptr @ th.sBreg]
    Creg'       ← word[Wptr @ th.sCreg]
    StatusReg'  ← word[Wptr @ th.Cntl]          — process status/control bits only
    ThReg'      ← Wptr

    if (StatusReg'sb.WtchPntEnbl = 1)
    {
      WlReg' ← word[Wptr @ th.eWl]
      WuReg' ← word[Wptr @ th.eWu]
    }
  }
  else
    start next process

  word[Wptr @ th.Cntl]sb.ThInUse ← 1
}
```

Error signals:

PrivInstruction signalled if executed by a P-process

Comments:

Instruction is privileged
Secondary instruction
The *th.sWptr* slot of the THDS must be word aligned

See chapter: 10

See also: *goprot ldth syscall*

unmkrc	unmark resource channel

Code: 62 FD

Description: Set the resource channel specified by **Areg** into normal mode. If the resource channel is idle it will become an empty channel. If the resource channel is queued it will become a waiting channel.

Definition:

```
if (Areg does not cause Unalign trap)
{
    if (Areg is virtual input channel or Areg is event channel or Areg is internal channel)
    {
        if (word[RCDS @ rc.Id] ≠ NotProcess.p)
        {
            word'[RCDS @ rc.Id] ← NotProcess.p

            if (Areg is internal channel)
            {
                if (word[Areg] = ResChan.p)
                    word'[Areg] ← NotProcess.p
            }
            else
                put channel into normal mode
        }
    }
    else
        IntegerError

    where
        for an external channel
            RCDS = ((Areg − MinEventChannel) + ExternalRCbase)

        for an internal channel
            RCDS = Areg @ 1

    Areg' ← Breg
    Breg' ← Creg
    Creg' ← undefined
}
```

Error signals:
PrivInstruction signalled if executed by a P-process
Unalign signalled if the address in **Areg** is not word aligned
IntegerError signalled if **Areg** is not a virtual input or event channel or a legal internal channel address

Comments:
Instruction is privileged
Secondary instruction

See chapter: 12

See also: *erdsq grant irdsq ldresptr mkrc stresptr*

vin variable-length input message

Code: 61 FC

Description: Input a message. The corresponding output is performed by a *vout*. **Areg** is the unsigned maximum message length in bytes, **Breg** is a pointer to the channel and **Creg** is a pointer to where the message is to be stored. The process executing *vin* will be descheduled if the channel is external or is not ready, and is rescheduled when the communication is complete.

Definition:

```
if (Breg does not cause Unalign trap)
{
  if (Areg = MostPosUnsigned)
    IntegerError
  else if (Breg is not a legal channel address)
    IntegerError
  else if (length of message is greater than Areg_unsigned)
  {
    word'[Wptr @ pw.Length] ← LengthError.p
    Areg bytes of store starting at address Creg ← undefined
    start next process
  }
  else
  {
    synchronize and input length of message bytes from channel Breg to address Creg
    word'[Wptr @ pw.Length] ← length of message
  }
}

word'[Wptr @ pw.Pointer] ← undefined
FPAreg' ← undefined
FPBreg' ← undefined
FPCreg' ← undefined
Areg'   ← undefined
Breg'   ← undefined
Creg'   ← undefined
```

Error signals:

PrivInstruction signalled if executed by a P-process
Unalign signalled if the address in **Breg** is not word aligned
IntegerError signalled if:
  Areg is equal to *MostPosUnsigned*
  the address in **Breg** is not a legal channel address

Comments:

Instruction is privileged
Secondary instruction
Instruction is a descheduling point
Instruction is interruptible

See chapter: 8

See also: *in ldcnt vout*

vout variable–length output message

Code: 61 FD

Description: Output a variable length message. The corresponding input is performed by a *vin* instruction. **Areg** is the unsigned length, **Breg** is a pointer to the channel, and **Creg** is a pointer to the message. The process executing *vout* will be descheduled if the channel is external or is not ready; it is rescheduled when the communication is complete. This instruction is also used to synchronize with an alternative or a resource.

Definition:

```
if (Breg does not cause Unalign trap)
{
  if (Breg is not a legal channel address)
    IntegerError
  else
    synchronize and output Areg_unsigned bytes to channel Breg from address Creg
}

word'[Wptr @ pw.Length]  ← undefined
word'[Wptr @ pw.Pointer] ← undefined

FPAreg' ← undefined
FPBreg' ← undefined
FPCreg' ← undefined
Areg'   ← undefined
Breg'   ← undefined
Creg'   ← undefined
```

Error signals:
PrivInstruction signalled if executed by a P-process
Unalign signalled if the address in **Breg** is not word aligned
IntegerError signalled if the address in **Breg** is not a legal channel address

Comments:
Instruction is privileged
Secondary instruction
Instruction is a descheduling point
Instruction is interruptible
Transfer is undefined if message length exceeds maximum allowed by *vin*

See chapter: 8

See also: *altwt enbc enbg grant mkrc out outbyte outword vin*

wait wait

Code: 60 F5

Description: Wait (or P) on the semaphore pointed to by **Areg**.

Definition:

```
if (Areg does not cause Unalign trap)
{
   if (word[Areg @ s.Count] = 0)
      put process on back of semaphore queue
      start next process
   else
      word'[Areg @ s.Count] ← word[Areg @ s.Count] −checked 1
}

FPAreg' ← undefined
FPBreg' ← undefined
FPCreg' ← undefined
Areg'   ← undefined
Breg'   ← undefined
Creg'   ← undefined
```

Error signals:
PrivInstruction signalled if executed by a P-process
Unalign signalled if address in **Areg** is not word aligned
IntegerOverflow can be signalled by $-_{checked}$

Comments:
Instruction is privileged
Secondary instruction
Instruction is a descheduling point

See chapter: 8

See also: *signal*

wcnt word count

Code: 23 FF

Description: Convert the byte offset in **Areg** to a word offset and a byte selector.

Definition:

```
Areg' ← (Areg ∧ WordSelectMask) / BytesPerWord
Breg' ← Areg ∧ ByteSelectMask

Creg' ← Breg
```

Error signals: *none*

Comments:
Secondary instruction

See chapter: 7

writehdr	write virtual channel header

Code: 61 F5

Description: Assign a packet header to the virtual link one of whose channels is pointed to by **Areg**. **Breg** is the unsigned length of the header and **Creg** is a pointer to the data to be written in the header.

Definition:

```
if (Areg does not cause Unalign trap)
{
  if (Areg is virtual input channel or Areg is virtual output channel)
    copy new header from address Creg to header area of virtual channel Areg
  else
    IntegerError
}

Areg'  ← undefined
Breg'  ← undefined
Creg'  ← undefined
```

Error signals:

PrivInstruction signalled if executed by a P-process
Unalign signalled if the address in **Areg** is not word aligned
IntegerError signalled if:
- the address in **Areg** is not a virtual channel;
- the header is too long to be a short header and no header offset has been set using *sethdr*;
- the unsigned length in **Breg** is zero or greater than the maximum header length

Comments:

Instruction is privileged
Secondary instruction
Instruction is interruptible

See chapter: 12

See also: *insphdr readhdr sethdr*

wsub — word subscript

Code: FA

Description: Generate the address of the element which is indexed by **Breg**, in the word array pointed to by **Areg**.

Definition:

```
Areg′ ← Areg + (BytesPerWord × Breg)

Breg′ ← Creg
Creg′ ← undefined
```

Error signals: *none*

Comments:
Secondary instruction

See chapter: 7

See also: *bsub ldlp ldnlp ssub wcnt wsubdb*

wsubdb form double word subscript

Code: 28 F1

Description: Generate the address of the element which is indexed by **Breg**, in the double word array pointed to by **Areg**.

Definition:

```
Areg' ← Areg @ (2 × Breg)

Breg' ← Creg
Creg' ← undefined
```

Error signals: *none*

Comments:
Secondary instruction

See chapter: 7

See also: *bsub ssub wcnt wsub*

xbword	sign extend byte to word

Code: 2B F8

Description: Sign-extend the value in the least significant byte of **Areg** into a signed integer.

Definition:

$Areg'_{0..7} \leftarrow Areg_{0..7}$
$Areg'_{8..BitsPerWord-1} \leftarrow Areg_7$

Error signals: *none*

Comments:
Secondary instruction

See chapter: 7

xdble	extend to double

Code: 21 FD

Description: Sign extend the integer in **Areg** into a double length signed integer.

Definition:

```
if (Areg ≥ 0)
   Breg′ ← 0
else
   Breg′ ← −1

Creg′ ← Breg
```

Error signals: *none*

Comments:
Secondary instruction

See chapter: 7

xor exclusive or

Code: 23 F3

Description: Bitwise exclusive or of **Areg** and **Breg**.

Definition:

```
Areg' ← Breg ⊙ Areg

Breg' ← Creg
Creg' ← undefined
```

Error signals: *none*

Comments:
Secondary instruction

See chapter: 7

xsword sign extend sixteen to word

Code: 2F F8

Description: Sign extend the value in the least significant 16 bits of **Areg** to a signed integer.

Definition:

$Areg'_{0..15} \leftarrow Areg_{0..15}$

$Areg'_{16..BitsPerWord-1} \leftarrow Areg_{15}$

Error signals: *none*

Comments:

Secondary instruction

See chapter: 7

xword extend to word

Code: 23 FA

Description: Sign extend an *N* bit signed number in **Breg** into a full word. To indicate the value of *N*, bit *N*–1 of **Areg** is set to 1 — all other bits must be 0.

Definition:

```
if (Areg is a not power of 2)
  Areg′ ← undefined
else if (Areg = MostNeg)                          — N is BitsPerWord
  Areg′ ← Breg
else if (Breg ≥ 0) and (Breg < Areg)              — Breg N bits and positive
  Areg′ ← Breg
else if (Breg ≥ Areg) and ((Breg >> 1) < Areg)    — Breg N bits and negative
  Areg′ ← Breg − (Areg << 1)
else                                              — Breg more than N bits
  Areg′ ← undefined

Breg′ ← Creg
Creg′ ← undefined
```

Error signals: *none*

Comments:

Secondary instruction

The result of the instruction is undefined if:

- **Areg** is not a power of 2
- **Breg** does not have its most significant (*BitsPerWord–N*) bits set to zero

See chapter: 7

B T9000 instruction set sorted by op-code

B.1 Primary functions

Function code	Mnemonic	Name
0	*j*	jump
1	*ldlp*	load local pointer
2	*pfix*	prefix
3	*ldnl*	load non-local
4	*ldc*	load constant
5	*ldnlp*	load non-local pointer
6	*nfix*	negative prefix
7	*ldl*	load local
8	*adc*	add constant
9	*call*	call
A	*cj*	conditional jump
B	*ajw*	adjust workspace
C	*eqc*	equals constant
D	*stl*	store local
E	*stnl*	store non-local
F	*opr*	operate

Table B.1 Instructions encoded as primary functions

B.2 Secondary functions

B.2.1 Instructions encoded without using *prefix*

Operation code	Memory code	Mnemonic	Name
00	F0	*rev*	reverse top of stack
01	F1	*lb*	load byte
02	F2	*bsub*	byte subscript
03	F3	*endp*	end process
04	F4	*diff*	difference
05	F5	*add*	add
06	F6	*gcall*	general call
07	F7	*in*	input message
08	F8	*prod*	product
09	F9	*gt*	greater than

Operation code	Memory code	Mnemonic	Name
0A	FA	*wsub*	word subscript
0B	FB	*out*	output message
0D	FD	*startp*	start process
0F	FF	*outword*	output word
0E	FE	*outbyte*	output byte
0C	FC	*sub*	subtract

Table B.2 Instructions encoded without using *prefix*

B.2.2 Instructions encoded using *prefix*

Operation code	Memory code	Mnemonic	Name
12	21F2	*resetch*	reset channel
13	21F3	*csub0*	check subscript from 0
15	21F5	*stopp*	stop process
16	21F6	*ladd*	long add
19	21F9	*norm*	normalize
1A	21FA	*ldiv*	long divide
1B	21FB	*ldpi*	load pointer to instruction
1D	21FD	*xdble*	extend to double
1E	21FE	*ldpri*	load current priority
1F	21FF	*rem*	remainder
20	22F0	*ret*	return
21	22F1	*lend*	loop end
22	22F2	*ldtimer*	load timer
2A	22FA	*testpranal*	test processor analyzing
2B	22FB	*tin*	timer input
2C	22FC	*div*	divide
2E	22FE	*dist*	disable timer
2F	22FF	*disc*	disable channel
30	23F0	*diss*	disable skip
31	23F1	*lmul*	long multiply
32	23F2	*not*	bitwise not
33	23F3	*xor*	exclusive or
34	23F4	*bcnt*	byte count
35	23F5	*lshr*	long shift right
36	23F6	*lshl*	long shift left
37	23F7	*lsum*	long sum
38	23F8	*lsub*	long subtract

Operation code	Memory code	Mnemonic	Name
39	23F9	*runp*	run process
3A	23FA	*xword*	sign extend to word
3B	23FB	*sb*	store byte
3C	23FC	*gajw*	general adjust workspace
3F	23FF	*wcnt*	word count
40	24F0	*shr*	shift right
41	24F1	*shl*	shift left
42	24F2	*mint*	minimum integer
43	24F3	*alt*	alt start
44	24F4	*altwt*	alt wait
45	24F5	*altend*	alt end
46	24F6	*and*	and
47	24F7	*enbt*	enable timer
48	24F8	*enbc*	enable channel
49	24F9	*enbs*	enable skip
4A	24FA	*move*	move message
4B	24FB	*or*	or
4C	24FC	*csngl*	check single
4D	24FD	*ccnt1*	check count from 1
4E	24FE	*talt*	timer alt start
4F	24FF	*ldiff*	long diff
51	25F1	*taltwt*	timer alt wait
52	25F2	*sum*	sum
53	25F3	*mul*	multiply
54	25F4	*sttimer*	store timer
56	25F6	*cword*	check word
5A	25FA	*dup*	duplicate top of stack
5B	25FB	*move2dinit*	initialize data for 2D block move
5C	25FC	*move2dall*	2D block copy
5D	25FD	*move2dnonzero*	2D block copy non–zero bytes
5E	25FE	*move2dzero*	2D block copy zero bytes
5F	25F5	*gtu*	unsigned greater than
72	27F2	*fmul*	fractional multiply
74	27F4	*crcword*	calculate CRC on word
75	27F5	*crcbyte*	calculate CRC on byte
76	27F6	*bitcnt*	count bits set in word
77	27F7	*bitrevword*	reverse bits in word
78	27F8	*bitrevnbits*	reverse bottom n bits in word
79	27F9	*pop*	pop processor stack

Operation code	Memory code	Mnemonic	Name
7E	27FE	*ldmemstartval*	load value of memstart address
81	28F1	*wsubdb*	form double word subscript
82	28F2	*fpldnldbi*	floating point load non-local indexed double
84	28F4	*fpstnldb*	floating point store non-local double
86	28F6	*fpldnlsni*	floating point load non-local indexed single
87	28F7	*fpadd*	floating point add
88	28F8	*fpstnlsn*	floating point store non-local single
89	28F9	*fpsub*	floating point subtract
8A	28FA	*fpldnldb*	floating point load non-local double
8B	28FB	*fpmul*	floating point multiply
8C	28FC	*fpdiv*	floating point divide
8D	28FD	*fprange*	floating point range reduce
8E	28FE	*fpldnlsn*	floating point load non-local single
91	29F1	*fpnan*	floating point NaN
92	29F2	*fpordered*	floating point orderability
93	29F3	*fpnotfinite*	floating point not finite
94	29F4	*fpgt*	floating point greater than
95	29F5	*fpeq*	floating point equality
96	29F6	*fpi32tor32*	`INT32` to `REAL32`
97	29F7	*fpge*	floating point greater than or equals
98	29F8	*fpi32tor64*	`INT32` to `REAL64`
9A	29FA	*fpb32tor64*	`BIT32` to `REAL64`
9B	29FB	*fplg*	floating point less than or greater than
9D	29FD	*fprtoi32*	`REAL` to `INT32`
9E	29FE	*fpstnli32*	floating point store non-local `INT32`
9F	29FF	*fpldzerosn*	load zero single
A0	2AF0	*fpldzerodb*	load zero double
A1	2AF1	*fpint*	round to floating integer
A3	2AF3	*fpdup*	floating point duplicate
A4	2AF4	*fprev*	floating point reverse
A6	2AF6	*fpldnladddb*	floating point load non-local and add double
A8	2AF8	*fpldnlmuldb*	floating point load non-local and multiply double
AA	2AFA	*fpldnladdsn*	floating point load non-local and add single
AC	2AFC	*fpldnlmulsn*	floating point load non-local and multiply single

Operation code	Memory code	Mnemonic	Name
B0	2BF0	*settimeslice*	set timeslicing status
B6	2BF6	*ldflags*	load error flags
B7	2BF7	*stflags*	store error flags
B8	2BF8	*xbword*	extend byte to word
B9	2BF9	*lbx*	load byte extended
BA	2BFA	*cb*	check byte
BB	2BFB	*cbu*	check byte unsigned
BC	2BFC	*insphdr*	inspect header
BD	2BFD	*readbfr*	read buffer pointer from VLCB
BE	2BFE	*ldconf*	load from configuration register
BF	2BFF	*stconf*	store to configuration register
C0	2CF0	*ldcnt*	load message byte count
C1	2CF1	*ssub*	16 bit word subscript
C2	2CF2	*ldth*	load trap handler
C3	2CF3	*ldchstatus*	load channel status
C4	2CF4	*intdis*	interrupt disable
C5	2CF5	*intenb*	interrupt enable
C7	2CF7	*cir*	check in range
C8	2CF8	*ss*	store 16 bit word
C9	2CF9	*chantype*	channel type
CA	2CFA	*ls*	load 16 bit word
CC	2CFC	*ciru*	check in range unsigned
CF	2CFF	*fprem*	floating point remainder
D0	2DF0	*fprn*	rounding mode to round nearest
D1	2DF1	*fpdivby2*	floating point divide by 2.0
D2	2DF2	*fpmulby2*	floating point multiply by 2.0
D3	2DF3	*fpsqrt*	floating point square root
D4	2DF4	*fprp*	rounding mode to round plus
D5	2DF5	*fprm*	rounding mode to round minus
D6	2DF6	*fprz*	rounding mode to round zero
D7	2DF7	*fpr32tor64*	`REAL32` to `REAL64`
D8	2DF8	*fpr64tor32*	`REAL64` to `REAL32`
D9	2DF9	*fpexpdec32*	floating point divide by 2^{32}
DA	2DFA	*fpexpinc32*	floating point multiply by 2^{32}
DB	2DFB	*fpabs*	floating point absolute
DE	2DFE	*fpchki32*	check in range of `INT32`
DF	2DFF	*fpchki64*	check in range of `INT64`
F0	2FF0	*devlb*	device load byte
F1	2FF1	*devsb*	device store byte

Operation code	Memory code	Mnemonic	Name
F2	2FF2	*devls*	device load sixteen
F3	2FF3	*devss*	device store sixteen
F4	2FF4	*devlw*	device load word
F5	2FF5	*devsw*	device store word
F8	2FF8	*xsword*	sign extend 16 bit word
F9	2FF9	*lsx*	load sixteen and sign extend
FA	2FFA	*cs*	check 16 bit word
FB	2FFB	*csu*	check 16 bit word unsigned
17C	2127FC	*lddevid*	load device identity

Table B.3 Instructions encoded using *prefix*

B.2.3 Instructions encoded using *negative prefix*

Operation code	Memory code	Mnemonic	Name
–01	60FF	*fpstall*	floating-point store all
–02	60FE	*fpldall*	floating-point load all
–03	60FD	*stshadow*	store shadow registers
–04	60FC	*ldshadow*	load shadow registers
–05	60FB	*tret*	trap return
–06	60FA	*goprot*	go protected
–07	60F9	*selth*	select trap handler
–08	60F8	*syscall*	system call
–0B	60F5	*wait*	wait
–0C	60F4	*signal*	signal
–0D	60F3	*timeslice*	timeslice
–0E	60F2	*insertqueue*	insert at front of scheduler queue
–0F	60F1	*swaptimer*	swap timer queue
–10	60F0	*swapqueue*	swap scheduler queue
–12	61FE	*stopch*	stop virtual channel
–13	61FD	*vout*	variable-length output message
–14	61FC	*vin*	variable-length input message
–17	61F9	*swapbfr*	swap buffer pointer in VLCB
–18	61F8	*sethdr*	set virtual channel header
–19	61F7	*setchmode*	set channel mode
–1A	61F6	*initvlcb*	initialize VLCB
–1B	61F5	*writehdr*	write virtual channel header
–1C	61F4	*readhdr*	read virtual channel header
–1D	61F3	*disg*	disable grant

Operation code	Memory code	Mnemonic	Name
–1E	61F2	*enbg*	enable grant
–1F	61F1	*grant*	grant resource
–20	61F0	*stmove2dinit*	store move2dinit data
–21	62FF	*causeerror*	cause error
–23	62FD	*unmkrc*	unmark resource channel
–24	62FC	*mkrc*	mark resource channel
–25	62FB	*irdsq*	insert at front of RDS queue
–26	62FA	*erdsq*	empty RDS queue
–27	62F9	*stresptr*	store resource queue pointer
–28	62F8	*ldresptr*	load resource queue pointer
–2C	62F4	*devmove*	device move
–2D	62F3	*icl*	invalidate cache line
–2E	62F2	*fdcl*	flush dirty cache line
–2F	62F1	*ica*	invalidate cache address
–30	62F0	*fdca*	flush dirty cache address
–40	63F0	*nop*	no operation
–84	62F0	*ldprodid*	load product identity

Table B.4 Instructions encoded using *negative prefix*

Instruction index

A

adc n, 22, 34, 229
add, 34, 230
ajw n, 22, 52, 231
alt, 87, 232
altend, 87, 233
altwt, 87, 234
and, 35, 235

B

bcnt, 36, 236
bitcnt, 63, 237
bitrevnbits, 63, 238
bitrevword, 63, 239
bsub, 37, 240

C

call n, 22, 52, 241
causeerror, 122, 242
cb, 49, 243
cbu, 49, 244
ccnt1, 58, 245
chantype, 185, 246
cir, 58, 247
ciru, 58, 248
cj n, 22, 41, 249
crcbyte, 63, 250
crcword, 63, 251
cs, 49, 252
csngl, 50, 253
csu, 49, 254
csub0, 58, 255
cword, 50, 256

D

devlb, 60, 257
devls, 60, 258
devlw, 60, 259
devmove, 60, 260
devsb, 60, 261
devss, 60, 262
devsw, 60, 263
diff, 34, 41, 264
disc, 87, 185, 265
disg, 87, 176, 267
diss, 87, 268
dist, 87, 269
div, 34, 270
dup, 28, 271

E

enbc, 87, 185, 272
enbg, 87, 176, 273
enbs, 87, 274
enbt, 87, 275
endp, 70, 276
eqc n, 22, 41, 277
erdsq, 176, 278

F

fdca, 213, 279
fdcl, 213, 280
fmul, 35, 281
fpabs, 132, 282
fpadd, 131, 283
fpadddbsn, 141, 147, 284
fpb32tor64, 141, 285
fpchki32, 140, 286
fpchki64, 140, 287
fpdiv, 131, 288
fpdivby2, 132, 289
fpdup, 128, 290
fpeq, 136, 291
fpexpdec32, 132, 292
fpexpinc32, 132, 293
fpge, 136, 294
fpgt, 136, 295
fpi32tor32, 141, 296
fpi32tor64, 141, 297
fpint, 140, 298
fpldall, 144, 196, 299
fpldnladddb, 132, 147, 300
fpldnladdsn, 132, 147, 301
fpldnldb, 128, 302
fpldnldbi, 128, 303
fpldnlmuldb, 132, 147, 304
fpldnlmulsn, 132, 147, 305
fpldnlsn, 128, 306
fpldnlsni, 128, 307
fpldzerodb, 129, 308
fpldzerosn, 129, 309
fplg, 136, 310
fpmul, 131, 311
fpmulby2, 132, 312
fpnan, 139, 313
fpnotfinite, 139, 314
fpordered, 136, 315
fpr32tor64, 140, 316
fpr64tor32, 140, 317
fprange, 133, 318
fprem, 131, 320
fprem, 133, 147
fprev, 128, 321
fprm, 131, 322
fprn, 131, 323
fprp, 131, 324
fprtoi32, 140, 147, 325
fprz, 131, 326
fpsqrt, 132, 327
fpstall, 144, 196, 328
fpstnldb, 129, 329
fpstnli32, 140, 330
fpstnlsn, 129, 331
fpsub, 131, 332

G

gajw, 52, 333
gcall, 52, 334
goprot, 105, 122, 201, 335
grant, 176, 337
gt, 41, 338
gtu, 41, 339

I

ica, 213, 340
icl, 213, 341
in, 76, 185, 342
initvlcb, 166, 188, 343
insertqueue, 204, 344
insphdr, 166, 188, 345
intdis, 202, 346
intenb, 202, 347
irdsq, 176, 348

J

j n, 22, 41, 349

L

ladd, 46, 350
lb, 28, 351
lbx, 28, 352

ldc n, 22, 28, 353
ldchstatus, 186, 354
ldcnt, 77, 356
ldconf, 159, 357
lddevid, 224, 358
ldflags, 122, 359
ldiff, 46, 360
ldiv, 47, 361
ldl n, 22, 28, 362
ldlp n, 22, 28, 363
ldmemstartval, 160, 364
ldnl n, 22, 36, 365
ldnlp n, 22, 36, 366
ldpi, 37, 367
ldpri, 70, 368
ldprodid, 224, 369
ldresptr, 176, 186, 370
ldshadow, 198, 371
ldth, 122, 372
ldtimer, 83, 204, 373
lend, 51, 374
lmul, 47, 375
ls, 28, 376
lshl, 48, 377
lshr, 48, 378
lsub, 46, 379
lsum, 46, 380
lsx, 28, 381

M

mint, 37, 382
mkrc, 176, 186, 383
move, 39, 385
move2dall, 61, 386
move2dinit, 61, 196, 387
move2dnonzero, 61, 388
move2dzero, 61, 389
mul, 34, 390

N

nfix, 21
nop, 391
norm, 49, 392
not, 35, 393

O

opr, 22
or, 35, 394
out, 76, 185, 395
outbyte, 79, 185, 396
outword, 79, 185, 397

P

pfix, 21
pop, 28, 398
prod, 34, 399

R

readbfr, 166, 188, 400
readhdr, 166, 188, 401
rem, 34, 402
resetch, 183, 185, 403
restart, 201, 404
ret, 52, 405
rev, 28, 406
runp, 70, 407

S

sb, 28, 408
selth, 122, 409
setchmode, 166, 186, 410
sethdr, 166, 188, 411
settimeslice, 68, 202, 412
shl, 35, 413
shr, 35, 414
signal, 86, 415
ss, 28, 416
ssub, 37, 417
startp, 70, 418
stconf, 159, 419
stflags, 122, 420
stl n, 22, 28, 421
stmove2dinit, 196, 422
stnl n, 22, 36, 423
stopch, 184, 188, 424
stopp, 70, 425
stresptr, 176, 186, 426
stshadow, 198, 427
sttimer, 68, 204, 428
sub, 34, 429
sum, 34, 430
swapbfr, 166, 188, 431
swapqueue, 204, 432
swaptimer, 204, 433
syscall, 105, 122, 434

T

talt, 87, 435
taltwt, 87, 436
testpranal, 437
timeslice, 68, 202, 438
tin, 83, 439
tret, 122, 440

U

unmkrc, 176, 186, 441

V

vin, 77, 185, 442
vout, 77, 185, 443

W

wait, 86, 444
wcnt, 36, 445
writehdr, 166, 188, 446
wsub, 37, 447
wsubdb, 37, 448

X

xbword, 49, 449
xdble, 50, 450
xor, 35, 451
xsword, 49, 452
xword, 50, 453

Index

Symbols

.., 218

', 219

{}, 227

@, 219

Numbers

2 dimensional block move, 61
  process state, 196

A

AccessViolation, 117, 216

activated, 157

active process, 66

active set, 6, 66

adc n, 22, 34, 229

add, 34, 230

address
  calculation, 219
  channel, 157
  logical, 98
  misalignment, 117
  physical, 98

address alignment, 12
  list of instructions, 13

address formation, 37

address translation, 98

address(variable), 29

addressing, 11, 36

AFTER, 84

after, 226

ajw n, 22, 52, 231

alarm registers, 18

alarm-time, 84

ALT, 91

alt, 87, 232

altend, 87, 233

alternative, 86
  degenerate, 92
  guards, 86
  instructions, 88
  outputting on internal channel, 90
  priority, 93
  replicated, 92
  sequence, 87
  synchronization, 75
  waiting, 90
  workspace, 88

altwt, 87, 234

AND, boolean, 43

and, 226

and, 35, 235

Areg, 16, 27

arithmetic
  boolean, 42
  checked integer, 34, 225
  fixed point, 35
  floating-point, 131, 225
    with NaN operands, 148
  fractional, 35
  integer, 34
  long integer, 46
  modulo, 34, 224
  signed integer, 34
  unchecked integer, 34, 224
  unsigned integer, 34

arrays, 36, 38
  assignment, 39
  floating-point, 129
  length, 39

assignment
  array, 39
  byte, 29
  multiple, 39
  single word, 29

B

back pointer registers, 18

bcnt, 36, 236

BITAND, 226

bitcnt, 63, 237

BITNOT, 226

BITOR, 226

bitrevnbits, 63, 238

bitrevword, 63, 239

BitsPerByte, 223

BitsPerWord, 223

bitwise logic instructions, 35

BITXOR, 226

block move, 39
  one dimensional, 39
  two dimensional, 61

block move registers, 17

block structured languages, 53

BMreg0..2, 16

boolean
  AND, 43
  arithmetic, 40, 42
  negation, 41
  operations, 43
  OR, 43

BptrReg0..1, 18, 66

breakpoint
  cause of trap, 113
  mechanism, 207

Breg, 16, 27

bsub, 37, 240

buffer (packet), 153, 166, 167, 168

byte addressing, 11

byte arrays, 39

byte assignment, 29

byte selector, 11

byte transfer, 79

byte-stream, mode, 155, 158, 165

byte-stream channels, 73
  implementation of, 155

ByteMode, 165

ByteSelectMask, 223

BytesPerLine, 223

BytesPerVLCB, 223

BytesPerWord, 223

C

cache, 211
  instructions, 213

main, 212
workspace cache, 212

call n, 22, 52, 241

calling a function, 57

calling techniques, 54

`CASE`, 44

causeerror, 122, 242
cause of trap, 113

cb, 49, 243

cbu, 49, 244

ccnt1, 58, 245

channel, implementation of
byte-stream, 155
event, 155
internal, 79
virtual, 152

channel address space, 157

channels, 7, 73, 151
activation modes, 157, 187
setting, 168, 187
compilation and configuration of, 151
external, 152
initialization of internal, 79
instructions according to usage, 185
mapping and configuration, 157
modes of operation, 157, 187
normal mode, 157
resetting, 182, 183
resource. *See* resource channels
resource mode, 157
states, 157, 187
timeout, 182
types of, 73, 186, 187

chantype, 185, 246

checked arithmetic, 34, 225

checking message lengths, 58

checking subscripts, 58

cir, 58, 247

ciru, 58, 248

cj n, 22, 41, 249

claim, 95

client-server model, 93
types of server, 178
unwanted clients, 181

ClockReg0..1, 18

clocks, 68, 83
clock registers, 18
clock tick, 68

Code, in instruction descriptions, 215

coding of instruction, 21, 215

command messages, 157

Comments, in instruction descriptions, 215, 217

comments, in instruction descriptions, 218

communication, 7, 73, 76
failure, 182
fixed-length, 76
on a guarded internal channel, 90
restarting process after failure, 184
variable-length, 77
zero-length, 78

comparison
floating-point, 136
IEEE anomalies, 139
IEEE implementation, 137
integer, 40
modulo, 84

compilation, channels, 151

concurrency, 65, 71

conditionals, 40, 42, 43

conditions, in instruction descriptions, 227

`ConfigReg`, 220

configuration of channels, 151, 157

configuration registers, 157, 220
loading and reading, 159

constants, 3
loading, 27, 33
machine constant definitions, 224
used in instruction descriptions, 223

context switch, 189
full, 198
partial, 189

context trap, 210

control bits, 18

control link, 8

control word
P-state, 103
trap-handler, 108

conventions, 3

conversion
object length, 49
type, 227

CPeek, 157

CPoke, 157

CRC evaluation, 63

crcbyte, 63, 250

crcword, 63, 251

Creg, 16, 27

cs, 49, 252

csngl, 50, 253

csu, 49, 254

csub0, 58, 255

current process, 66

cword, 50, 256

cyclic time, 84

D

data packet, structure, 153

data representation, 11

data structure, trap-handler, 107

data structures, 3
loop end, 51
P-state, 103
parallel process, 70
process workspace, 65
region descriptor, 104
resource, 96
resource channel, 96, 221
semaphore, 86

data-transfer, 39, 73
single byte, 79
single word, 79

deactivated, 157

Deactivated.p, 224

deadlock, 6

Definition, in instruction descriptions, 215, 216

deschedule, 6

descheduled process, 67

descheduling
  effect on FP stack, 143
  process state, 67, 189
  with single-step or watchpoint, 210

descheduling points, 67, 217
  list of, 67

Description, in instruction descriptions, 215

device access instructions, 59

device identity, 224

device-only access, 101

DeviceId, 224

devlb, 60, 257

devls, 60, 258

devlw, 60, 259

devmove, 60, 260

devsb, 60, 261

devss, 60, 262

devsw, 60, 263

diff, 34, 41, 264

disabling in alternative, 90, 91
  grant, 178

Disabling.p, 224

disc, 87, 185, 265

disg, 87, 176, 267

diss, 87, 268

dist, 87, 269

div, 34, 270

dup, 28, 271

dynamic allocation of workspace, 56

E

else, 227

empty, 157

enabling in alternative, 89, 91
  grant, 178

Enabling.p, 224

enbc, 87, 185, 272

enbg, 87, 176, 273

enbs, 87, 274

enbt, 87, 275

encoding of instruction, 21

encoding of instructions, 215

end-of-message token, 153

end-of-packet token, 153

endp, 70, 276

EOM token, 153

EOP token, 153

EptrReg, 16

eqc n, 22, 41, 277

erdsq, 176, 278

Ereg, 16

error
  cause of trap, 113
  flags, 117
  indication of, 119
  link, 154
  restarting process after, 195, 196
  signals, 116
  type, 120

error checking instructions, 57

error pointer
  L-process, 109
  P-process, 104

Error signals, in instruction descriptions, 215, 216

error signals, 8, 116, 216
  effect of, 118
  exceptions, 126

et.*ErrorType*, 120, 146

evaluation
  floating-point, 130
  function, 57
  integer, 30
  subscripts, 38

event channels, 73
  implementation of, 155
  resource, 173

exception handling, 145
  compound instructions, 147
  fprem, 147

execute permission, 101

executing process, 66

execution
  loading process state, 189
  of process, 6, 66

expression depth, 30

expression evaluation, 31
  using functions, 57

expressions
  floating-point, 130
  integer, 30

external channels, 152

ExternalRCbase, 160

F

false, 224
  representation of, 41
  signal if, 58

fdca, 213, 279

fdcl, 213, 280

fixed point arithmetic, 35

fixed-length communication, 76

flags, 18
  error and exception, 117

floating-point
  arrays, 129, 130
  class analysis, 139
  comparison, 136
  dyadic arithmetic, 131
  exception handling, 145
  exceptions, 126
    process state, 195
  expressions, 130
  formats, 125, 127
  instructions, 125
  load and operate, 132

manipulation of stack, 127
monadic arithmetic, 132
operations with NaN operands, 148
operators, 225
pointer, 128
register loading and storing, 128
remainder and range, 133
rounding mode, 131
sign bit manipulation, 133
square root, 133
stack, 17
stack registers, 127
state, 146
state and status word, 143
storing and loading state, 144, 196
type conversion, 139

flush dirty cache instructions, 213

fmul, 35, 281

fonts, use of, 3

footnotes, 4

forming address, 37

fpabs, 132, 282

fpadd, 131, 283

fpadddbsn, 141, 147, 284

FPAreg, 16, 127

fpb32tor64, 141, 285

FPBreg, 16, 127

fpchki32, 140, 286

fpchki64, 140, 287

FPCreg, 16, 127

fpdiv, 131, 288

fpdivby2, 132, 289

FPDivideByZero, 117, 126, 216

fpdup, 128, 290

fpeq, 136, 291

FPError, 117, 126, 216

fpexpdec32, 132, 292

fpexpinc32, 132, 293

fpge, 136, 294

fpgt, 136, 295

fpi32tor32, 141, 296

fpi32tor64, 141, 297

FPInexact, 117, 126, 216

fpint, 140, 298

`fpint()`, 227

FPInvalidOp, 117, 126, 216

fpldall, 144, 196, 299

fpldnladddb, 132, 147, 300

fpldnladdsn, 132, 147, 301

fpldnldb, 128, 302

fpldnldbi, 128, 303

fpldnlmuldb, 132, 147, 304

fpldnlmulsn, 132, 147, 305

fpldnlsn, 128, 306

fpldnlsni, 128, 307

fpldzerodb, 129, 308

fpldzerosn, 129, 309

fplg, 136, 310

fpmul, 131, 311

fpmulby2, 132, 312

fpnan, 139, 313

fpnotfinite, 139, 314

fpordered, 136, 315

FPOverflow, 117, 126, 216

fpr32tor64, 140, 316

fpr64tor32, 140, 317

fprange, 133, 318

fprem, 131, 320

fprem, 133, 147

fprev, 128, 321

fprm, 131, 322

fprn, 131, 323

fprp, 131, 324

fprtoi32, 140, 147, 325

fprz, 131, 326

fpsqrt, 132, 327

fpstall, 144, 196, 328

FPstatusReg, 16

fpstnldb, 129, 329

fpstnli32, 140, 330

fpstnlsn, 129, 331

fpsub, 131, 332

FptrReg0..1, 18, 66

FPUnderflow, 117, 126, 216

fractional arithmetic, 35

front pointer registers, 18

function call, 57

'Function' code, in instruction descriptions, 215

function evaluation, 57

functions, 56
in instruction descriptions, 227

future, 83

G

gajw, 52, 333

gcall, 52, 334

goprot, 105, 122, 201, 335

grant, 176, 337

graphics support, 61

grouper, 9

gt, 41, 338

gtu, 41, 339

H

handshake, event channel, 156

HdrAreaBase, 160

header, null, 169

header (packet), 153, 166, 167, 168
labelling, 164

header region, 160

HeaderLength, 165

I

ica, 213, 340

icl, 213, 341

identifier store address, 176

identity, of device, 224

idle, 157

IEEE, operators, 225

IEEE floating-point
- arithmetic, 125
  - implementation of, 125
- exceptions, 126
- implementation of comparisons, 137
- implementation of handler, 146
- operations, 125
- word format, 125

`IF`, 43

`if`, 227

ignorant server, 180

illegal instruction, 117

IllegalInstruction, 117

in, 76, 185, 342

initializing an internal channel, 79

initvlcb, 343

input, 76

insertqueue, 204, 344

insphdr, 166, 188, 345

instruction, illegal, 117

instruction component, 21

instruction data value, 21

instruction encoding, 21, 215

instruction pointer, 27
- of supervisor, 104
- trap-handler, 109

instruction protection, 97

INT to REAL conversion, 141

`int64()`, 227

intdis, 202, 346

integer
- comparison, 40
- expressions, 30
- length conversion, 49

integer length conversion, 12

integer overflow, 117

integer stack
- loading sequences, 32
- registers, 17, 27

IntegerError, 117, 216

IntegerOverflow, 117, 216

intenb, 202, 347

internal channels, 73
- implementation of, 80

internal registers, 17

interrupt point, 69

interruptible instructions, 217
- list of, 69

interruption, 68
- enabling and disabling, 202
- process state, 198
- restarting process after, 200

interrupts, using events as, 156

invalidate cache instructions, 213

IptrReg, 16, 27

irdsq, 176, 348

iteration, 51

J

j n, 22, 41, 349

jump table, 45

jumps, 40, 42

L

L-process, 69

ladd, 46, 350

lb, 28, 351

lbx, 28, 352

ldc n, 22, 28, 353

ldchstatus, 186, 354

ldcnt, 77, 356

ldconf, 159, 357

lddevid, 358
- values returned, 224

ldflags, 122, 359

ldiff, 46, 360

ldiv, 47, 361

ldl n, 22, 28, 362

ldlp n, 22, 28, 363

ldmemstartval, 160, 364

ldnl n, 22, 36, 365

ldnlp n, 22, 36, 366

ldpi, 37, 367

ldpri, 70, 368

ldprodid, 369
- values returned, 224

ldresptr, 176, 186, 370

ldshadow, 198, 371

ldth, 122, 372

ldtimer, 83, 204, 373

le.*LoopEndSlot*, 51, 220

lend, 51, 374

length of array, 39

LengthError.p, 224

library linkage, 54

link, 7

list
- scheduling, 66, 203
- timer, 85, 203

little-endian, 11

livelock, 6

lmul, 47, 375

loading
- additional state, 196
- current time, 84
- floating-point stack, 128
- operands, 32
- parameters, 53
- shadow state, 198

loading sequences, integer stack, 32

local, variables, 27, 65

LocalizeError, 165

logic
- bitwise instructions, 35
- shift instructions, 35

logical

- address, 98
- region, 99

long arithmetic, 46

long header, 154, 168

long shifts, 47

loop end data structure, 51

loops, 51

L-process, 7
- restarting after interruption, 201
- single-step, 208
- trap, 107, 111, 191
- watchpoint, 209

ls, 28, 376

lshl, 48, 377

lshr, 48, 378

lsub, 46, 379

lsum, 46, 380

lsx, 28, 381

M

main cache, 212

mapping functions, virtual link, 160

mapping of channel addresses, 162

MaxHeaderOffset, 223

MaxLink, 223

MaxPacketLength, 223

memory
- management, 97
- map, 159
- model, 59
- protection, 98
- regions, 99
- representation of, 219

memory semantics error, 121
- process state, 196

MemStart, 160

message lengths, checking, 58

MinEventChannel, 223

MinInvalidChannel, 160

MinLink, 223

mint, 37, 382

minus, 35

MinVirtualChannel, 223

misalignment, address, 117

miss, 211

mkrc, 176, 186, 383

modulo arithmetic, 34, 224

modulo comparison, 84

MostNeg, 223

MostPos, 223

MostPosUnsigned, 223

move, 39, 385

move2dall, 61, 386

move2dinit, 61, 196, 387

move2dnonzero, 61, 388

move2dzero, 61, 389

mul, 34, 390

multiple assignment, 39

multiple length arithmetic, 46

multiple length shifts, 47

N

N-bit object arithmetic, 51

negation, 35
- boolean, 41

next instruction, 218

nfix, 21

non-local variables, 36

NoneSelected.o, 224

nop, 391

norm, 49, 392

normal mode, channel, 157

normalizing, 49

not, 226

not, 35, 393

Not-a-Number
- implementation of, 148
- list of special quiet NaNs, 148
- representation of, 126

notation, 3

NotProcess.p, 224

null descriptor, 101

null header, 169

null process, the, 66

null trap-handler, 110
- trapping to, 121

NullHeader, 224

NullOffset, 224

O

object length conversion, 49

objects, 11, 218

occam
- as meta language, 3
- as source language, 3

omniscient server, 179

op-code, 215
- instructions sorted by, 455

operands
- in instruction descriptions, 215
- primary instructions, 22
- secondary instructions, 32

operators, 224
- in instruction descriptions, 226

opr, 22, 217

OR, boolean, 43

or, 226

or, 35, 394

out, 76, 185, 395

outbyte, 79, 185, 396

output, 76

outputting, to guarded internal channel, 90

outword, 79, 185, 397

overflow checking, integer, 34

P

P-process

restarting after interruption, 201
single-step, 207
trap, 193
watchpoint, 209

packet, 153
buffer, 153, 160, 166, 167, 168
structure of data, 153

PAR, 3, 71

parallel assignment, 39

parallel process data structure, 70

parallel processes, scheduling, 70

parameters to procedure, 53

partword arithmetic, 51

past, 83

pc.*RegionDescriptorSlot*, 105, 222

PDS, 103
creation of, 104

pfix, 21

physical address, 98

physical link, 152

pipeline, 9

pointer
floating-point, 128
trap-handler, 8, 107

pop, 28, 398

pop, 27

pp.*ParallelProcessSlot*, 70, 220

P-process, 7
trap, 97, 112

predecessor process, 72

prefixing, 21, 23

PRI ALT, 93

primary instructions, 22, 217

prime notation, in instruction descriptions, 219

Priority, 19

priority, 66, 68, 73, 218

privileged instructions, 97, 217

PrivInstruction, 117, 216

procedure call, 52

procedures, 52

procedures as parameters, 55

process, 5, 65
current, 66
descheduled, 67
descriptor, 19, 217
execution, 6, 66
initiation, 69, 73
interruption, 68, 200
L-process, 69
model, 5
the null, 66
P-process, 97
priority, 68, 218
restarting after communication failure, 182
restarting after error, 195, 196
termination, 69, 73

process state, 6, 15, 189, 217
2D block move, 196
additional, 196
descheduling, 189
execution, 189
floating-point, 196
in instruction descriptions, 216
L-process trap, 191
P-process trap, 193

process workspace data structure, 65

prod, 34, 399

product identity, 224

program notation, 3

protected mode, 7, 97

protection, 97
instructions used for, 105
the mechanism, 97
memory, 98
of instructions, 97
protection bit, 19

ps.*PstateSlot*, 103, 221

P-state
control word, 103
data structure, 103

PstateReg, 17, 102

push, 27

pw.Link, 66

pw.*ProcessWorkspaceSlot*, 65, 220

Q

Q(), 227

queue manipulation
resource, 177
scheduling list, 203
timer list, 203

queued, 157

queued process, 66

quiet NaN, 126
list of machine generated, 148

R

range checking, 58

range reduction, floating-point, 134

rc.*ResourceChannelSlot*, 96, 221

RDDS, 104

rds.*ResourceSlot*, 96, 221

read-only access, 101

readbfr, 166, 188, 400

readhdr, 166, 188, 401

reading the time, 84

Ready.p, 224

real arrays, 129, 130

REAL to INT conversion, 140

REAL to REAL conversion, 140

REAL32, 125

real32(), 227

REAL64, 125

real64(), 227

region descriptor, 101
data structure, 104

registers, 102
region descriptor registers, 17
RegionReg0..3, 17, 102
regions of memory, 99
register field, 19
registers, 3, 15
floating-point stack, 127
in instruction descriptions, 216
integer stack, 27
region descriptor, 102
state, 15
other, 17
rem, 226
$\mathbf{rem}_{IEEE}$, 226
rem, 34, 402
remainder
floating-point, 133
integer, 34
REPEAT .. UNTIL, 43
replicated alternative, 92
replicators, 51, 92
representing memory, in instruction descriptions, 219
ResChan.p, 224
resetch, 183, 185, 403
resetting a channel, 182, 183
resource
data structure, 96
instructions, 175
mechanism, 95
mode, 157
synchronization, 75
resource channel, 93, 173
data structure, 96, 221
placement, 173, 174
event, 173
identifier, 96
implementation of, 173
overview, 179
reverse channel, 174
usage of, 178
restart, 201, 404
restarting process after
communication failure, 182
error, 195, 196
interruption, 200
ret, 52, 405
return from procedure, 53
rev, 28, 406
reverse channel, 174
robust server, 181
rotation, 48
rounding mode, floating-point, 131
running process, 66
runp, 70, 407

S

s.*SemaphoreSlot*, 86, 221
sb, 28, 408
sb.*StatusOrControlBit*, 19
scheduled process, 66
scheduling, 66
lists, 66, 203
parallel processes, 70
scheduling list, 6
secondary instructions, 22, 217
See chapter, in instruction descriptions, 215
selth, 122, 409
semaphore, 85
data structure, 86
SEQ, 3
sequential operations, 27
setchmode, 166, 186, 410
initvlcb, 166, 188
sethdr, 166, 188, 411
settimeslice, 68, 202, 412
shadow, state, 17, 69
shadow registers, saving and reloading, 198
shared memory, 61
sharing a trap-handler, 110
shifts, long, 47
shl, 35, 413
short header, 154, 168
shr, 35, 414
sign bit manipulation, floating-point, 133
sign extension, 12
signal, 86, 415
signal if *false*, 58
signal if *true*, 58
signalling NaN, 126
signalling of errors, 116
signed integer arithmetic, 34
simple synchronization, 74
single-step
cause of trap, 113
mechanism, 207
process state, 196
trap enable bit, 19
size of workspace, 65
special pointer values, 11
sqroot(), 227
square root, 133
ss, 28, 416
ssub, 37, 417
stack
floating-point, 127
integer, 27
stack extension, 98
stack operations, 27
stack registers, 17
start next process, 218
startp, 70, 418
state
floating-point, 143, 146
L-process, 108
P-process, 103
shadow, 198
static chain, 53
status
bits, 18
floating-point status word, 143
StatusReg, 16, 18

stconf, 159, 419

stflags, 122, 420

stl n, 22, 28, 421

stmove2dinit, 196, 422

stnl n, 22, 36, 423

stopch, 184, 188, 424

stopp, 70, 425

stopping, 157

stopping a virtual channel, 182, 184

Stopping.p, 224

storing from floating-point stack, 129

stresptr, 176, 186, 426

structures, 36

stshadow, 198, 427

sttimer, 68, 204, 428

stub, 7

stub process, 55

sub, 34, 429

subscript evaluation, 38

subscripts, 36, 38, 58
  in instruction descriptions, 218

successor process, 70

sum, 34, 430

supervisor, 7, 97
  exiting from, 113
  instruction pointer, 104
  workspace, 104

swapbfr, 166, 188, 431

swapqueue, 204, 432

swaptimer, 204, 433

synchronization, 6, 7, 73

syscall, 105, 122, 434
  cause of trap, 113

T

t.*TrapReason*, 116

T805 compatibility, error handling, 121

table of constants, loading from, 33

talt, 87, 435

taltwt, 87, 436

termination, 5, 69, 73

terminology, 3

testpranal, 437

th.*TrapHandlerSlot*, 108, 222

THDS, 107
  creation of, 109

ThReg, 16

tick, 68

time, 83
  alarm-time, 84
  reading, 84
  starting, 205

TimeNotSet.p, 224

timer
  input, 84, 85
  list pointer registers, 18
  lists, 85, 203

TimeSet.p, 224

timeslice, 68, 202, 438

timesliced process, 6

timeslicing, 6, 42, 43, 51, 68
  cause of trap, 113
  effect on FP stack, 143
  enabling and disabling, 202
  forcing a timeslice, 202
  points, 68, 217
  process state, 194
  starting, 205
  timeslice disable bit, 19
  with single-step or watchpoint, 209

tin, 83, 439

TnextReg0..1, 18

token, 153

TptrReg0..1, 18

transferring data, 39

trap reason, 116

trap-handler
  changing trap-handler, 109
  control word, 108
  data structure, 107
  exiting from, 113
  instruction pointer, 109
  null, 110
  pointer, 8, 107
  queue, 109
  workspace, 109

traps, 7
  a comparison, 112
  causes, 113
  enable bits, 118
  exception handling, 145
  floating-point state, 144
  for multiple reasons, 114
  indication of cause, 114
  instructions, 122
  L-process, 107, 111, 191
  P-process, 97, 112, 193
  process state, 191
  to null trap-handler, 121

tret, 122, 440

true, 224
  representation of, 41
  signal if, 58

two dimensional block move, 61

type conversion, 139, 227

U

Unalign, 117, 216

unaligned address detection, 12

unary minus, 35

unchecked arithmetic, 34, 224

undefined, 218

undefined values, 3, 218

under protection, 7, 97

underflow, implementation of, 127

unmkrc, 176, 186, 441

unsigned, 219

unsigned arithmetic, 34

`unsigned()`, 227

unwanted client, 181

V

variable-length communication, 77

variables
  local, 27, 65
  non-local, 36

VCP, 153

VCPcommand, 165

VCPLink0..3HdrOffset, 161

VCPLink0..3MaxHeader, 165

VCPLink0..3MinHeader, 165

VCPLink0..3Mode, 165

vin, 77, 185, 442

virtual channel processor, 153

virtual channels, 73
  implementation of, 152
  stopping, 182, 184

virtual link, number, 160

virtual mode, 153, 165

VLCB, 154
  null header, 169
  setting up, 166

vout, 77, 185, 443

W

wait, 86, 444

waiting, 157

waiting in alternative, 90

Waiting.p, 224

watchpoint
  cause of trap, 113
  in a P-process, 104
  in an L-process, 109
  mechanism, 208
  process state, 196
  registers, 17
  trap enable and pending bit, 19

wcnt, 36, 445

`WdescReg`, 217

WdescReg, 16, 19

WdescStubReg, 17, 102

`WHILE`, 43

WlReg, 16

word address, 11

word arrays, 38

word assignment, 29

word normalization, 49

word rotation, 48

word transfer, 79

WordSelectMask, 223

workspace, 19, 27, 28
  address, 217
  adjustment, 52
  after *call*, 52
  cache, 212
  during alternative, 88
  dynamic allocation, 56, 98
  of supervisor, 104
  of trap-handler, 109
  pointer, 27
  process workspace data structure, 65
  size, 65

workspace 0, 29, 53, 65

`Wptr`, 217

Wptr, 19, 27

write permission, 101

write-back cache, 212

writehdr, 166, 188, 446

write-through cache, 212

wsub, 37, 447

wsubdb, 37, 448

WuReg, 16

X

xbword, 49, 449

xdble, 50, 450

xor, 35, 451

Xreg, 16

xsword, 49, 452

xword, 50, 453

Z

zero-length communication, 78

SALES OFFICES

EUROPE

DENMARK

2730 HERLEV
Herlev Torv, 4
Tel. (45-44) 94.85.33
Telex: 35411
Telefax: (45-44) 948694

FINLAND

LOHJA SF-08150
Katakatu, 26
Tel. (358-12) 155.11
Telefax. (358-12) 155.66

FRANCE

94253 GENTILLY Cedex
7 - avenue Gallieni - BP. 93
Tel.: (33-1) 47.40.75.75
Telex: 632570 STMHQ
Telefax: (33-1) 47.40.79.10

67000 STRASBOURG
20, Place des Halles
Tel. (33-88) 75.50.66
Telefax: (33-88) 22.29.32

GERMANY

8011 GRASBRUNN
Bretonischer Ring 4
Postfach 1122
Tel.: (49-89) 460060
Telefax: (49-89) 4605454
Teletex: 897107=STDISTR

1000 BERLIN 37
Clay Allee 323
Tel.: (49-30) 8017087-89
Telefax: (49-30) 8015552

6000 FRANKFURT
Gutleutstrasse 322
Tel. (49-69) 237492-3
Telefax: (49-69) 231957
Teletex: 6997689=STVBF

3000 HANNOVER 51
Rotenburger Strasse 28A
Tel. (49-511) 615960-3
Teletex: 5118418 CSFBEH
Telefax: (49-511) 6151243

8500 NÜRNBERG 20
Erlenstegenstrasse, 72
Tel.: (49-911) 59893-0
Telefax: (49-911) 5980701

7000 STUTTGART 31
Mittlerer Pfad 2-4
Tel. (49-711) 13968-0
Telefax: (49-711) 8661427

ITALY

20090 ASSAGO (MI)
V.le Milanofiori - Strada 4 - Palazzo A/4/A
Tel. (39-2) 89213.1 (10 linee)
Telex: 330131 - 330141 SGSAGR
Telefax: (39-2) 8250449

40033 CASALECCHIO DI RENO (BO)
Via R. Fucini, 12
Tel. (39-51) 593029
Telex: 512442
Telefax: (39-51) 591305

00161 ROMA
Via A. Torlonia, 15
Tel. (39-6) 8443341
Telex: 620653 SGSATE I
Telefax: (39-6) 8444474

NETHERLANDS

5652 AR EINDHOVEN
Meerenakkerweg 1
Tel.: (31-40) 550015
Telex: 51186
Telefax: (31-40) 528835

SPAIN

08021 BARCELONA
Calle Platon, 6 4th Floor, 5th Door
Tel. (34-3) 4143300-4143361
Telefax: (34-3) 2021461

28027 MADRID
Calle Albacete, 5
Tel. (34-1) 4051615
Telex: 46033 TCCEE
Telefax: (34-1) 4031134

SWEDEN

S-16421 KISTA
Borgarfjordsgatan, 13 - Box 1094
Tel.: (46-8) 7939220
Telex: 12078 THSWS
Telefax: (46-8) 7504950

SWITZERLAND

1218 GRAND-SACONNEX (GENEVA)
Chemin Francois-Lehmann, 18/A
Tel. (41-22) 7986462
Telex: 415493 STM CH
Telefax: (41-22) 7984869

UNITED KINGDOM and EIRE

MARLOW, BUCKS
Planar House, Parkway
Globe Park
Tel.: (44-628) 890800
Telex: 847458
Telefax: (44-628) 890391

AMERICAS

BRAZIL

05413 SÃO PAULO
R. Henrique Schaumann 286-CJ33
Tel. (55-11) 883-5455
Telex: (391)11-37988 "UMBR BR"
Telefax : (55-11) 282-2367

CANADA

NEPEAN ONTARIO
301 Moodie Drive
Suite 307
Tel. 613/829-9944

U.S.A.

NORTH & SOUTH AMERICAN
MARKETING HEADQUARTERS
1000 East Bell Road
Phoenix, AZ 85022
(1-602) 867-6100

SALES COVERAGE BY STATE

ALABAMA
Huntsville - (205) 533-5995

ARIZONA
Phoenix - (602) 867-6217

CALIFORNIA
Santa Ana - (714) 957-6018
San Jose - (408) 452-8585

COLORADO
Boulder (303) 449-9000

ILLINOIS
Schaumburg - (708) 517-1890

INDIANA
Kokomo - (317) 455-3500

MASSACHUSETTS
Lincoln - (617) 259-0300

MICHIGAN
Livonia - (313) 953-1700

NEW JERSEY
Voorhees - (609) 772-6222

NEW YORK
Poughkeepsie - (914) 454-8813

NORTH CAROLINA
Raleigh - (919) 787-6555

TEXAS
Carrollton - (214) 466-8844

FOR RF AND MICROWAVE
POWER TRANSISTORS CON-
TACT
THE FOLLOWING REGIONAL
OFFICE IN THE U.S.A.

PENNSYLVANIA
Montgomeryville - (215) 361-6400

ASIA / PACIFIC

AUSTRALIA

NSW 2220 HURTSVILLE
Suite 3, Level 7, Otis House
43 Bridge Street
Tel. (61-2) 5803811
Telefax: (61-2) 5806440

HONG KONG

WANCHAI
22nd Floor - Hopewell centre
183 Queen's Road East
Tel. (852) 8615788
Telex: 60955 ESGIES HX
Telefax: (852) 8656589

INDIA

NEW DELHI 110001
LiasonOffice
62, Upper Ground Floor
World Trade Centre
Barakhamba Lane
Tel. (91-11) 3715191
Telex: 031-66816 STMI IN
Telefax: (91-11) 3715192

MALAYSIA

PETALING JAYA, 47400
11C, Jalan SS21/60
Damansara Utama
Tel.: (03) 717 3976
Telefax: (03) 719 9512

PULAU PINANG 10400
4th Floor - Suite 4-03
Bangunan FOP-123D Jalan Anson
Tel. (04) 379735
Telefax (04) 379816

KOREA

SEOUL 121
8th floor Shinwon Building
823-14, Yuksam-Dong
Kang-Nam-Gu
Tel. (82-2) 553-0399
Telex: SGSKOR K29998
Telefax: (82-2) 552-1051

SINGAPORE

SINGAPORE 2056
28 Ang Mo Kio - Industrial Park 2
Tel. (65) 4821411
Telex: RS 55201 ESGIES
Telefax: (65) 4820240

TAIWAN

TAIPEI
12th Floor
325, Section 1 Tun Hua South Road
Tel. (886-2) 755-4111
Telex: 10310 ESGIE TW
Telefax: (886-2) 755-4008

JAPAN

TOKYO 108
Nisseki - Takanawa Bld. 4F
2-18-10 Takanawa
Minato-Ku
Tel. (81-3) 3280-4121
Telefax: (81-3) 3280-4131