Programming the NS32000

SMALL COMPUTER SERIES

Consulting Editors

Jan Wilmink
ORMAS bv, The Netherlands

Max Bramer
Thames Polytechnic

Programming the NS32000

Chris Martin
University of Sheffield

ADDISON-WESLEY PUBLISHING COMPANY
Wokingham, England • Reading, Massachusetts • Menlo Park, California
Don Mills, Ontario • Amsterdam • Bonn • Sydney • Singapore
Tokyo • Madrid • Bogota • Santiago • San Juan

Cover graphic by Laurence M. Gartel.
Photoset direct from the author's disks by Commercial Colour Press.
Printed in Great Britain by R. J. Acford.

British Library Cataloguing in Publication Data
Martin, Chris.
Programming the NS32000. – (Small computer series)
1. Microcomputers – Programming
2. Assembling (Electronic computers)
I. Title II. Series
005.2'6 QA76.6

ISBN 0-201-14663-0

Library of Congress Cataloging in Publication Data
Martin, Chris.
Programming the NS32000.

(Small computer series)
Includes index.
1. NS32000 series (Microprocessors) I. Title.
II. Title: Programming the NS thirty-two thousand.
III. Series.
QA76.8.N66M37 1987 005.265 86-8042
ISBN 0-201-14663-0

ABCDEF 8932109876

Preface

This book is for programmers. It is written for the reader who wants to write 32000 assembler programs and has access to a 32000 system and the usual reference manuals.

Despite previous experience, coding from a reference manual often involves searching back and forth from instruction definition to addressing modes to assembler directives, just to write a single line of code. This book sets out to bridge the gap between experience and applying it in 32000 assembler. To do this, it introduces both instructions and addressing modes together, starting with the basic modes and integer instructions and progressing to grander things – by the end of Chapter 3, it should be possible to write simple programs.

Chapters 3 to 8 include exercises to enable the reader to test his comprehension. They should be written and then run on a 32000 system to check that the code produces the right result – desk checking is not a substitute. To do this it is essential that your 32000 system has a machine code debugger. The exercises can not be done properly without one and writing assembler, relying on system calls to show how the program is working, is a short-cut to a padded cell. If no debugger is available, make sure your system supplier knows of and appreciates your distress.

The reader is expected to have a fair amount of computing experience. Typically, he may be an assembler programmer adding the 32000 series to his repertoire or a programmer competent in two or more high-level languages wishing to tackle 32000 assembler in order to speed up a procedure or because unhindered access to the system makes it more convenient.

Nothing very abstruse is needed: no explanation of bits, bytes or words is given. An understanding of arrays, the concept of stacks (the actual machine implementation is minutely described in Chapters 7 and 8), records as used in Pascal (or the C **struct**) and the idea of a heap as a source of blocks of storage provided by such system routines as *new*( ) in Pascal or **malloc**( ) and friends in C is assumed.

The book would also be useful to undergraduates with projects to develop on a 32000 system. Most of the computer terminology should be familiar and any missing background could be made up in lectures.

The first chapter is an introduction to and a survey of the 32000 series architecture. It presents the 32000 programming model, its general and special registers, the data types it can handle, the instruction classes and memory organization. You will then be able to compare and contrast the 32000 with other architectures you are familiar with.

Chapter 2 introduces the assemblers used by the 32000. There are two forms: those based on the National Semiconductor ASM16 assembler; and the Acorn ZASM running on the Cambridge second processor (for the BBC micro), the Cambridge workstation and the Master Scientific. As the differences lie mainly in the directives, the code throughout this book is in Acorn ZASM with notes on the changes required to convert to NatSemi format where necessary.

Chapter 3 includes a section on binary arithmetic for those who are not familiar with it and discusses the instructions for handling integers, including comparison and conditional branches. It also introduces the basic addressing modes and their assembler syntax.

In Chapter 4 boolean and logical instructions are discussed. In contrast to the requirements of some other languages, boolean for the 32000 series is not synonymous with logical as the instructions act only on the least significant bit and therefore allow TRUE to be represented by 1 and FALSE by 0. The shift instructions are included in this chapter as they are often used in conjunction with logical instructions.

The fifth chapter deals with the 32081 Floating Point Unit and its instructions which provide hardware floating point arithmetic to the IEEE standard with a few minor exceptions. The first part of the chapter introduces the IEEE standard, its single and double precision formats, rounding modes and special operands – infinities and NaNs. This is followed by the instruction definitions and ends with an example.

In Chapter 6 bits and bit fields are described in detail, together with the 32000 instructions for operating on them. These replace the sequences of ANDs, ORs and shifts usual on less advanced architectures.

Chapter 7 shows how the 32000 addressing modes provided for handling arrays, records and stacks can be used, ending with a section on the string move, search and comparison instructions.

Chapter 8 starts with a description of the branch and jump instructions not fully dealt with in the preceding chapters and moves on to calling procedures and the use of the stack for passing parameters and returning results. The very important topic of 32000 modules and module support is then dealt with, together with the assembler format of a module – the usual form of an assembler program.

Chapter 9 presents an actual 32000 operating system – Acorn's Panos – with its library routines and parameter passing conventions. The Panos scheme for handling exceptions and program errors is an excellent example of the high standard of design possible with the 32000's CPUs.

Exceptions, both interrupts and traps, and the programming needed to handle them are dealt with in Chapter 10, together with the 32000's support for operating systems through its recognition of supervisor and user states and the instructions used in writing operating system kernels. The last section of the chapter describes the main features of the Memory Management Unit.

There are three appendices.

The first is a 32000 instruction reference. In the main body of the text the instructions are described and illustrated in sufficient detail to make possible the writing of programs. Later on, however, questions on the allowable addressing modes of the operands or details of the instructions' actions will arise and are best dealt with in a reference summary. This appendix goes into detail on the different operand types and access classes and closes with a brief description of each instruction (in alphabetical order) showing its operands, the PSR and FSR flags which can be set by the instruction and the traps which may occur during its execution.

The second appendix lists the instructions grouped by function. It will be of use when you know what you want to do and want to find out which instruction will do it for you.

The third appendix is a collection of five small programs, all written for the Acorn assembler under Panos but which may easily be changed for another assembler and system. It is hoped that these will provide an initial model for your own programming and also some useful code.

ACKNOWLEDGMENTS

This book would not have been possible without access to a 32000 system and I would like to thank Tony Wilkinson, late of NatSemi, for arranging the loan of a board which allowed the book to go ahead.

I would also like to thank Mark Jenkin of Acorn Computers for including me in the field trials of the 32016 second processor. This provided an invaluable programming environment in which all the code for this book was developed and tested.

Dai Rees of National Semiconductor gave the draft manuscript a thorough and painstaking review and made a number of suggestions which I have been glad to incorporate. I would also like to thank the publisher's reviewers for the many mistakes and obscurities they pointed out – if I have managed to put them right, thank them; if I have not, blame me.

The publishers would also like to thank Acorn Computers Limited for permission to use material appearing in Chapter 9 describing Panos, the Acorn operating system; and National Semiconductor Limited for permission to use material appearing in Appendices A and B describing the NS32000 instruction mnemonics.

Contents

1 Introduction

This book is a complete reference to programming the National Semiconductor 32000 series of microprocessors, including the programming details needed to use the associated NS32081 Floating Point Unit. There is also a description of the main points of the NS32082 Memory Management Unit and the vectored interrupt processing mode made available by the NS32202 Interrupt Control Unit.

The different instruction groups are described and illustrated by presenting some of the smaller routines commonly found in utilities and library routines. These routines have been chosen to show the economy and style of programming the 32000. Sometimes only central sections of a routine are presented as the complete code would be too long for an example. All the routines have been run on a 32000 system and work correctly. They have been written to show how the programmer can make the best use of the instructions and addressing modes supported by this chip.

The members of the 32000 series available include the 32016, the 32008 and the 32032. The 32016 has a 16-bit path to memory, reading a word at a time, the 32008 (intended for smaller systems) reads a byte at a time and the 32032 (with a 32-bit memory path) reads a double word at a time; the wider the path to memory, the faster the CPU can get instructions and data and the faster programs will run. The ultimate aim of NatSemi is to produce the CPUs (and supporting chips) in CMOS with the lower power consumption which allows more transistors to be squeezed into the same area without melting the silicon.

Also in the series are the 32C016 (32016 in CMOS) and the 32332. The final conjunction of the CMOS and NMOS paths will take the form of the 32C532.

The 32032 has a speed advantage of some 50% over the 32016, and the 32332 is an enormous step up on this (250%), showing both NatSemi's commitment to the series and the soundness of the design in providing such a steady base for the leap forward. Earlier models have a 24-bit address but later models have 32 bits for the address, allowing an address space of 4 Gbytes (4096 Mbytes). With 16 Mbytes already available, it doesn't seem long ago that people stood up and cheered when given a measly 64K.

The support chips are not to be forgotten in the advance of the CPUs. The Memory Management chip (NS32082) is backed by the 32382 which supports a 4 Gbyte address space and a memory cache allowing high-speed access to instructions and data without having to wait for slow memory chips and the 32081 FPU is supplemented by the 32381 numerical processor.

The 32000 follows the Motorola 68000 in time and has a similar architecture supporting a direct addressing range of 16 Mbytes. The earlier 16-bit microprocessor chips from Intel and Zilog have a segmented architecture which is more difficult to program and more awkward to use.

The Motorola chip was the first to have 32-bit internal data paths and, as the price for being first, the architecture contains some pitfalls. It would appear that the architects of the NatSemi 32000 learned from this, as its instruction set is exceptionally symmetrical and it has designed-in rather than added-on support for virtual memory and interrupt dispatch tables.

The NS32000 instruction set is very advanced and a full appreciation of it and its addressing modes requires a knowledge of compiler code generation and the intricacies of multiprocessing and multitasking operating systems. However, the majority of assembler programmers writing within the ambit of an operating system will still find a rich store to delve into.

1.1 THE 32000 PROGRAMMING MODEL

In a high-level language (hereafter abbreviated, in line with convention, to HLL) data is kept in variables the length of which varies with the size of the object to be kept there. It is possible to have byte variables (usually containing characters), 16-bit word variables (short integers) and 32-bit double word integers. There may also be provision for short reals, 32 bits in length, and long or double precision reals which are 64 bits long. Arrays and structures form variables of greater lengths than these. Variables usually occupy one or more bytes in main memory though the C language does contain a 'register' specifier which causes the value to be held in a CPU register if possible.

CPU registers are similar in many respects to memory except that a register has a maximum length of data object it can hold. This length in bits is often used to characterize the CPU: for instance, the registers on the 32000 can contain data up to 4 bytes or 32 bits in length and it is therefore called a 32-bit microprocessor.

The main advantage of CPU registers is that data in them can be accessed very much faster than data in memory. On the 32016, adding two 32-bit numbers in registers takes four machine cycles while adding the same two integers stored in memory takes 32 cycles, a considerable increase.

Because of this speed advantage, the CPU also keeps many of its own variables in registers, called special or dedicated registers. Some of these registers may be altered by a program but, since the CPU's behaviour stands or falls (crashes) on their contents, any program altering them must have displayed a much higher degree of trustworthiness in operation than a mere user program. For this reason, the CPU operates in two modes: user mode and

supervisor mode. In user mode the instructions altering some of the special registers are not available but in supervisor mode there are no restrictions. This separation of the modes is also an aid to debugging, as a program fault which leaves the system intact gives the user a greater chance of finding its cause.

The CPU assumes that its general registers will only be used to contain integers though these may be bytes, words or double words. Floating point numbers will usually be handled in the registers on the FPU chip, unless of course it is absent, in which case routines must be written to simulate the floating point operations in the CPU registers. This will be expensive in both time and programming effort, fitting the FPU chip is much cheaper.

1.2 THE REGISTERS

The CPU has sixteen 32-bit registers in all, eight of which are general registers which may be used without restriction by the programmer, the remaining eight being the CPU's special registers. Some of these dedicated registers may be set or altered by the programmer but this will usually be done by the operating system rather than the user.

The eight general registers can be used to hold byte (8-bit), word (16-bit) or double word (32-bit) objects. These objects may be data, offsets or addresses – the registers are not specialized in any way. A register used for a byte operand holds it in the least significant 8 bits, a word operand is held in the least significant 16 bits. Moving a byte or a word operand into a register affects only the bits actually occupied, the high-order bits being unchanged. The general registers are referred to as R0 to R7.

The eight special registers are the program counter (PC), the two stack pointer registers SP0 and SP1, the frame pointer register (FP), the module register (MOD), the static base register (SB), the interrupt base register (INTBASE) and the processor status register (PSR).

PC, the program counter, contains the address of the first byte of the instruction being executed.

The stack pointer register SP0 is used by the operating system for its own private stack; SP1 is for the user stack, holding the program's temporary data space and procedure call information. The frame pointer register is used in conjunction with SP1 to allow a user procedure to refer to its own data on the stack.

The module register (16 bits long) points to the descriptor of the currently executing software module (in modula-2 terms), and the static base register points to this module's global variables.

INTBASE holds the address of the start of the current exception (interrupts and traps) dispatch table.

The processor status register (16 bits long) is divided into an upper and a lower byte. The lower byte is for user programming, containing bits indicating such things as the result of the last comparison; the upper byte is for system use as it shows whether the processor is in supervisor or user mode, whether interrupts are enabled and so on. The user byte of the PSR will be covered in

Chapter 3 while the supervisor byte, which is concerned with interrupts and user/supervisor mode will be covered in Chapter 10.

The program counter need not be described further as it is only altered by the processor itself while executing instructions, being incremented during normal instruction sequencing or replaced as a result of a jump, branch or subroutine entry or exit.

SP1, the user stack pointer, and FP, the frame pointer, will be discussed in Chapter 7 (stack addressing modes) and Chapter 8 (procedure calls). SP0, as the interrupt stack pointer, and INTBASE, the interrupt base register, will be dealt with in Chapter 10 (exception handling).

The module register will be discussed in Chapter 8.

There is one further register, the 4-bit configuration register, which has not been mentioned so far. It is intended to acquaint the processor with the presence of the optional support chips, the Interrupt Control Unit (ICU), the Floating Point Unit (FPU), the Memory Management Unit (MMU) and the Custom Slave Processor. The register's four bits are set and cleared by a single instruction (SETCFG) which is intended to be used only on system initialization after a reset. The four bits are called C, M, F and I. The I bit is set when the system includes an ICU, the M bit when it includes an MMU, the F bit if there is an FPU chip and the C bit for a Custom Slave Processor. If the I bit is set, interrupts will be vectored; otherwise they will be non-vectored (see Chapter 10 for the discussion of these terms). If the M bit is clear the memory management instructions will cause an 'undefined' trap to be executed. If the F bit is clear, the floating point instructions will be taken as undefined, and if the C bit is clear the same fate will befall the Custom instructions.

1.3 MEMORY ORGANIZATION

The earlier members of the 32000 series can directly address 16 Mbytes of memory; addresses can extend to 24 bits. Later members and support chips can use 32-bit addresses, extending the address space to 4 Gbytes (4 gigabytes – over 4000 Mbytes).

Memory can be handled as bytes, words (16 bits) or double words (32 bits), as shown in Fig. 1.1. A word is made up of 2 bytes in consecutive addresses. The least significant byte of the word is in the lower address (which need not be even) as in Fig. 1.2. A double word is made up of four contiguous bytes or two consecutive words. Again, the least significant byte of the double word is in the lowest address and this does not have to be a multiple of four (Fig. 1.3).

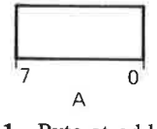

Fig. 1.1 Byte at address A.

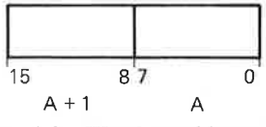

Fig. 1.2 Word at address A.

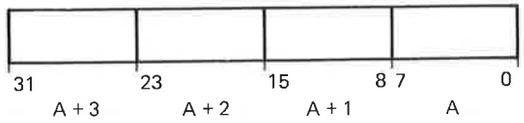

Fig. 1.3 Double word at address A.

Note that both bit numbers and addresses increase from right to left. To have the highest bit number to the left of the lowest is usual, but addressing is usually contrariwise and illogical. This scheme is used in all the National documentation and to use a different convention here would make it more difficult for those who were later to consult the manufacturer's documentation.

Memory is addressed as a sequence of bytes but the 32016 accesses it a word at a time on an even address and the 32032 addresses it a double word at a time with the first byte at an address which is a multiple of four. While the programmer is not restricted to even addresses for words or quad byte addresses for double words, starting on the wrong boundary will mean extra memory accesses. If you are counting the cycles it would be wise to make sure the data starts on the right address.

1.4 MEMORY MAPPING

Running large programs on a 32000 system does not need a lot of memory to be present if the Memory Management Unit is fitted. This works together with the CPU to translate the addresses referred to by the instructions (in this case called virtual addresses) into addresses corresponding to the memory which is physically present (physical addresses).

This means that a program too large to fit into memory can be broken down, leaving part in memory with the rest on disk. When the MMU is passed an address corresponding to a part of the program on disk, it can inform the CPU which arranges a swap. When this is complete the CPU can continue by restarting the instruction.

Of course, this tactic need not be confined to running large programs. If, on a workstation with a 32000 and MMU, you are writing a letter with a word processing program and you find you need to look up some figures in a file on the disk, you don't need to save the letter and read in a database program. You can simply ask for the database program, which will be read into memory, displacing part of the word processor, run the database to get the figures and then continue with the letter. All the movement of parts of the database, parts of the letter, database program and word processor will be taken care of by the system running on the 32000 working in conjunction with the MMU. Until very recently, this sort of system was available only on expensive machines with expensive operating systems. Now it can be done on a 32000 workstation.

It is clear that if all of a program in memory had to be written to disk

every time another program was to be run, there would be a considerable wait – an unnecessary one if the incoming program was much smaller than the outgoing one. To avoid this, the 32000 divides memory into pages of 512 bytes, the address space containing 32768 such pages, programs being divided into pages to be stored in memory. The pages need not be contiguous or even in sequence; the MMU's address translation registers take care of the correspondence by mapping each page in the program into a page of physical memory. Each 24-bit address is broken up into a 15-bit page number and a 9-bit offset within that page, the MMU replacing the virtual page number with the physical page number but leaving the offset unchanged. In this way only a few pages need to be transferred to and from disk, giving the system a short (and satisfying) response time.

This is not the limit of the MMU's capabilities. As a means of stopping programs destroying themselves when they mistakenly use a piece of data as an address, programs are divided into areas which may be read from but not written to (code and constants fall into this category), areas which may be both read from and written to (variable data space) and, for the benefit and continued survival of the operating system, areas which a user program may neither read from nor write to. Should a program misuse an area, the MMU in conjunction with the CPU can stop it in its tracks, allowing the programmer to catch the fault before it has pulled the ladder up after it.

This protection of pages is performed in the MMU by setting and clearing extra bits on each of the logical/physical address translation registers it contains. In fact, this is given an extra dimension as the bits (two of them) are given different meanings in supervisor mode to those in user mode. In user mode the program can be completely shut out of certain pages while being allowed read-only or read and write access to others. The supervisor (a trusted program) is given both read and write access to all the user's pages while being able to mark its own pages as read and write or read-only.

1.5 DATA TYPES

Previous microprocessors have all been able to handle data in bytes and some have had special instructions to handle 16-bit words: it is only very recently that microprocessors have appeared in the marketplace which are capable of handling 32-bit double words. The NS32000 has been designed so that all its general instructions have three varieties, one dealing with byte quantities, one with words and one with double words. These are not three separate instructions but an identical instruction performing a given action which the programmer can choose to be applied to byte, word or double word; only a 2-bit field in the basic instruction needs to be changed. (*Note*: Throughout the following text, these general instructions will be suffixed by the letter 'i'. Where the form of an instruction is required which acts on operands which are bytes, this is replaced by the letter 'b'; for word operands it is 'w'; and for double word operands it is 'd'.)

Another unusual feature of these 32-bit processors is that most instructions have two general operands; that is, they not only allow register-to-memory and memory-to-register operations but memory-to-memory and register-to-register as well. The interesting thing is that for a basic operation like move or add, it is the addressing mode of the operand which decides where the source operand is to come from and where the result is to go to. This of course is an enormous convenience to the harried compiler writer who no longer has to have a list of several instructions or even instruction sequences to perform the same high-level language operation under all circumstances. Now one instruction (or sequence, the millenium has not yet arrived) can be used in all cases by choosing the appropriate addressing modes for its operands.

It is also unusual for both signed and unsigned integers to be catered for, and in the same instruction: for instance, the compare instructions provide a result which can be treated either as the result of an unsigned comparison or as a signed one, the decision being taken by the choice of the conditional branch instruction following the comparison.

If the NS32081 Floating Point Unit is available, the CPU can handle both 32-bit and 64-bit floating point quantities as easily as integers. This it does by handing the operation and the operands over to the FPU. This (and the Memory Management Unit, the Interrupt Control Unit and a Custom Slave Processor) work very closely with the CPU which fetches operands and stores results for the slave chip – unless, of course, they are in the chip's registers – with the slave performing the operation while the CPU can do something else. Because of this close relationship when, later on, integration technology advances to the point that they can all fit on to the same piece of silicon as the CPU, no change in the program will be needed; it will simply run faster.

As well as integers and floating point numbers, the NS32000 can set and clear single bits and operate on fields of bits (up to 32 bits in length). The addressing modes available make it possible for the programmer to start the field at any bit using a bit offset – the number of bits from the base address. The single bit instructions also have two special interlocked versions which read the bit and then either set it or clear it as a single indivisible instruction. In a system which contains more than one processor, the same byte (word, double word) of memory could be written to by a second processor just after the first had read it; if the first processor then makes a decision based on what it has read (which the second processor has just changed) it could lead to confusion and a hung system. These interlocked instructions guarantee that a second processor will not be allowed a byte until the first one has completed its read and write cycle.

There are also instructions to add and subtract numbers in packed decimal form. I am not going to say much about these – not that I dislike COBOL (how could I? I'm paid by a COBOL program) – but because packed numbers do not feature in programs as frequently as the other data types.

1.6 OPERATORS

Integers are treated equally by the instructions which apply to them, the identical instruction operating on bytes, words or double words. The primary instructions are the moves which, as pointed out before, can be memory-to-memory, register-to-register or a combination of these. The moves include variations which can convert a smaller quantity to a larger one (byte to word or double word, word to double word) with either sign or zero extension, thus keeping the integer as signed or unsigned.

There are also the usual addition and subtraction instructions both with and without the carry bit in the sum, to make multiple precision arithmetic possible without distress.

The multiplication and division operators in the instruction set have carefully chosen varieties to cover all needs. There is a single length multiplication, a word by a word giving a word result, and, in addition, a double length multiplication in which a word by a word gives a double word result and double word by double word gives a quad word (64-bit) result. The NS32000 is the only microprocessor to do this. Divisions are even more generously provided. There are both single length divisions (in which a word divided by a word gives a word result) and a double length division in which, for instance, a quad word dividend and a double word divisor give a quad word result – the upper half of the quad word is the quotient and the lower half the remainder. Using the double length division the programmer can decide either to round or to truncate the result. Where microprocessors provide a single length division, it is usually the CPU that decides whether the result will be rounded or truncated. Each of these two types of division has a corresponding remainder instruction (one is called `MOD` as it corresponds to the mathematical modulus function).

It is clear that programmers have been given a considerable say in the design of this chip.

Floating point operations are performed by the NS32081 Floating Point Unit, though in a system without it they would be performed by software alerted by a specific 'undefined' trap. The floating point equivalents of the integer arithmetic operations are all implemented (the symmetrical architecture) with add, subtract, multiply and divide as well as compare, negate and absolute value. A most unusual, and sadly needed, feature of the FPU is the provision of illegal floating point quantities. These are values which, when encountered by the FPU, cause a trap to be executed. By filling all otherwise uninitialized reals with these values, a program attempting to use a real it has not set can be caught in the act – leading to the publication of fewer erroneous results.

The move instructions for floating point values have, as well as a move from one general address to another, a conversion from long floating point to short floating point and short to long and from any integer type to either floating point type. In addition, there are conversions from either floating point type to any integer type in three flavours: converting to the nearest integer (rounding), to the integer nearest to zero (truncating) or to the largest integer less than or equal to its value (the floor function). No tricky subroutines to write here – buy the floating point chip instead, it must be cheaper!

For logical operations the chip has a special NOT operator which operates on the standard values for TRUE and FALSE in HLLs (1 and 0 respectively) to convert them into the opposite value. There is also a set condition operator which puts 1 (TRUE) in its destination if the condition code matches and 0 (FALSE) if it doesn't: the instruction `SGEi` sets its destination to TRUE if the result of the last comparison was greater than or equal, otherwise it is set to FALSE.

1.7 HLL DATA STRUCTURES

Languages like Pascal and C provide the programmer with the means to define data structures which correspond more closely to the objects being mapped than simple integers, reals, booleans and characters.

The array is an ordered sequence of data, all the elements of the array being of the same type, a particular element being accessed by subscripting the array name with one or more indices.

If the data to be represented is a mixture of types as would be found, for instance, in a personnel record having both character fields (names and addresses) and numerical fields (age, length of time employed and so on), the C **struct** or Pascal **record** is used instead, permitting data of as many different types as required to be grouped together and increasing human readability.

Another data structure less visible than these, but underlying the working of all modern languages, is the stack. It is most widely used to provide a clean procedure calling method but is also used in the evaluation of arithmetic expressions in interpretive implementations of languages such as BASIC. Data is pushed on to or popped off the stack. An item is popped off it by reading it from the address held in a dedicated register (the stack pointer) and then incrementing the register by the length of the item. Pushing a data item is the reverse of this: the stack pointer is decremented by the length of the data which is then written to the stack pointer address. This description implies the usual implementation of a stack (which is also the NS32000 implementation) which starts in the highest memory address and builds towards lower addresses; in some implementations, though, the stack starts at the first byte past the end of the program and builds upwards.

These are the data structures commonly found in HLLs and the designers of the NS32000 have taken some trouble to provide addressing modes and instructions which make them easy to deal with.

Array elements are accessed in older architectures by performing a series of multiplications and additions (one of each for each dimension) and putting the result into an index register as an offset to the address of the first element of the array. If the elements of the array are not bytes (the basic addressing step) the result must be multiplied by the length of each element.

The NS32000 reduces array indexing to two basic steps. The instruction INDEXi performs the calculations necessary for the indexing step for one dimension, accumulating the result in a register. When this instruction has been repeated for each array dimension, the scaled index addressing mode uses the accumulated index to access the array element no matter whether the element size is byte, word, double word or quad word – the index is scaled according to element size. There is yet a further instruction available, CHECKi, which performs a range check on the index for each dimension and removes the addressing bias from it at the same time.

Addressing the components of a record is done with the base and displacement addressing mode: the base register is set to the address of the start of the record and the fixed offset to the item required is then put in the displacement.

Very often an item of data occupies a byte or word when its range of values take up only a small part of the range of values available. In Pascal it is possible to pack items in a record so that they occupy only the minimum number of bits needed to cover their range. For instance, an integer in the range 0 to 7 can have all its values represented in three bits (one for the sign). The item may then span byte or word boundaries and needs a logical AND and a shift to extract it: to insert it requires a shift, an AND and an OR. In addition to the code generated, an extensive table of masks may need to be kept in memory where every such instruction sequence can get at it easily.

With the coming of the NS32000 all this is gone. To extract a sequence of bits (up to 32 bits in length) no matter where they start you simply use EXTi. If the offset of the field is constant you can economize by using the EXTSi instruction instead. To insert a bit field you have the complementary instructions INSi (for variable offset) and INSSi (for fixed offset). Compiler writing is going to become too easy!

To use a stack as a procedure calling mechanism the CPU has two instructions: enter and exit (clearly named, too). The enter command sets up a link to the preceding stack frame, makes space in the new stack frame for the called procedure's local variables and optionally saves some or all of the general registers. Within the procedure both the parameters with which the procedure was called and the local variables can be accessed by fixed displacements from the frame pointer register. The exit instruction reverses this process, restoring the registers (if they were saved), resetting the stack pointer to where it was before the enter instruction was executed so that the appropriate return instruction can continue execution of the calling program.

As well as this way of using a stack, there is also the top of stack (TOS) addressing mode. In this the access class of the instruction's operand is used to determine whether the stack is pushed or popped. If the operand is a source operand (read access class) the stack will be popped if TOS is used; if it is a destination operand (write) it will be pushed. This is very useful in putting parameters on the stack. For each parameter, a TOS mode MOVi before enter will put either values or addresses on the stack where they can be easily accessed by the called procedure.

Strings are widely used in high-level languages, most often as strings of bytes representing characters to be input or output. In the NS32000 architecture, strings are not limited to bytes but may be words or double words as well. There are instructions to move a string from one place to another, compare two strings and search a string for a given value. The instructions can run through a string either forwards or backwards and may also be under the control of an until/while condition, the instruction terminating either when a particular value is encountered or when a value different from the given one is encountered. If the string elements are bytes there is also a translation option; the bytes can be translated by using them as an index into a table before they are moved, compared or tested.

1.8 JUMPS AND CONDITIONAL JUMPS

The branches and conditional branches use an addressing mode which gives the effective address in terms of a displacement from the program counter. This displacement comes in three sizes: byte size with a range of −64 to 63, word size with a range of −8192 to 8191 and a double word displacement with a range of -2^{29} to $2^{29}-1$ to cater for the extension of the addressing space to 32 bits, (though at present assemblers will probably not allow displacements outside $-2^{24}+1$ to $2^{24}-1$). There are also jump instructions which take a general mode operand: they can be used to make indirect jumps.

The branches available in the CPU's instruction set include a conditional branch in which the condition is encoded in the basic instruction, allowing the compiler writer to use the same instruction sequence to encode boolean expressions by simply taking the appropriate code from a table to perform the test required. The conditions also include an unconditional condition (making it a plain jump), so the same basic instruction can be used throughout the section of code being generated.

Both Pascal and C have an indexed jump, called **case** in Pascal and **switch** in C, which is usually coded as a jump into a table of addresses. The CPU has a case branch to perform this jump based in the value in a register; to make sure the jump is to a legitimate address, the CHECKi instruction (mentioned in connection with array indices) can be used before the CASEi jump.

Encoding **for** loops usually requires three instructions; increment the loop index, compare it with the limit and then branch on the result. The 32000 has one instruction, ACBi, which performs all three actions and, as the increment is set into the instruction as a 4-bit signed integer, it can be any number from −8 to 7 rather than just 1 and −1.

The procedure call instructions have already been mentioned, they are for entering code in another module. There is also a jsr (jump to subroutine) and a bsr (branch to subroutine) which are used to call routines within the current module. jsr takes a general addressing mode operand while bsr uses a displacement from the program counter.

2 The 32000 assemblers

2.1 INTRODUCTION

This chapter starts with the assembler directives used to define constants and allocate space for variables of different types beginning with a brief survey of the different data types the assembler recognizes. The second part of the chapter covers the forms of expressions the assembler accepts as operands to the directives and instructions.

2.2 ASSEMBLER DATA TYPES

The data types handled by the 32000 series can be divided into two classes: those, like integers, for which it provides instructions to perform the usual operations on the data type; and those, like arrays, for which it provides a basic set of operations required in the manipulation of the data.

The assembler data types are therefore integers, bytes, words and double words and short and long reals – all else is the work of man.

2.3 THE ASSEMBLER

The species *32000 Assembler* has (at present) two subspecies, *ASM16* (originating in North America from National Semiconductor) and *ZASM* (the British variety, coming from Acorn). As this book is being written in Britain, the subspecies ZASM is easier to observe, but note will be taken of the differences between ZASM and ASM16: the assembler syntax of the local subspecies will be used in examples but, where necessary, a note showing the differences will be added. In the following sections, the assembler directives used in allocating storage areas and constants of the data type being discussed will be given.

2.4 THE ASSEMBLER LINE

An assembler operates on lines like FORTRAN (the first high-level assembler):

each line is syntactically independent of all the other lines. A line has the following general format

{label} {mnemonic {operands }} {;comments}

where

- **label** is optional (though mandatory on certain directives) and must start in the first character position of a line. The NatSemi assembler expects every label to be followed by a colon (:) – as compensation, labels need not start in column 1. If the label is followed by two colons, it is considered public and can be referred to by code in another module; if the label is followed by a colon and a hyphen, it is taken as referring to data or a procedure in another module which is defined there. This method of defining external labels is not very satisfactory and the use of the 'import' and 'export' directives makes for a more readable (and therefore less confusing) program. These directives and a complete description of modules are to be found in Chapter 8.
- **mnemonic**, the directive or instruction code, is followed by operands (if needed). Operands are separated by commas.
- **comments** (preceded by the mandatory semicolon) are optional. They are, however, very important as they give clues, which may prove to be invaluable, as to your intent when you wrote the instruction.

Each field of the source line must be separated from the next by a tab or one or more spaces.

In the rest of this chapter, only assembler directives will be mentioned, in particular, those which define constants and allocate memory. Most directives may be labelled and followed by comments. Each of the following directives may have one or more operands, separated by commas.

In NatSemi's ASM16, directives are distinguished from instruction codes by a prefixed '.' but the Acorn assembler doesn't follow this lead – it does, however, make it easy to distinguish the two varieties.

2.5 INTEGERS

The 32000 series will operate directly on integers of three different sizes and in two different modes. The three sizes are bytes (8 bits in length), words (16 bits) and double words (32 bits). There are also instructions which perform operations (division, multiplication) on quad words (64 bits). The differences between the sizes concern the programmer only in the range of numbers which can be used with a particular size, and the amount of memory they occupy. Apart from this, operating on one or other of the sizes makes little difference to the code; just one letter in the opcode has to be changed. This may seem obvious enough but is an enormous advantage – an advantage not yet evident in some mainframe architectures I could mention! Instead of having to

remember different rules for each integer size you have to remember only one way to add, subtract, compare, multiply or divide, which applies across the board. Even the difference between the two modes has been simplified by thoughtful design – writing a comparison between two unsigned integers needs only a slight change to the condition tested afterwards.

In structure, a signed integer has one bit reserved for use as a sign indicator. By tradition this bit is the most significant one in the integer and it indicates a negative number if it is set to 1 and a positive number (or zero) if it is 0.

A signed byte has the structure shown in Fig. 2.1. A signed word is two bytes long but only the most significant byte contains a sign bit (Fig. 2.2). A signed double word contains 4 bytes with again only the most significant carrying the sign bit (Fig. 2.3).

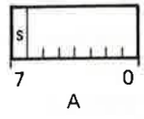

Fig. 2.1 A signed byte.

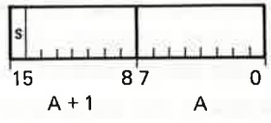

Fig. 2.2 A signed word.

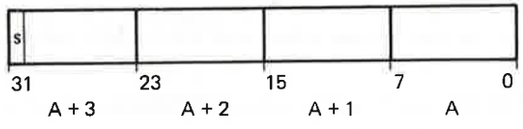

Fig. 2.3 A signed double word.

Unsigned integers have the same structure except that the sign bit becomes another value bit, allowing the integer to take twice as many positive values for the loss of the negative ones a signed integer can take. The ranges of signed and unsigned integers of the three different sizes are shown in Table 2.1.

The Acorn assembler directives to define integer constants are:

```
dcb   operand {, operand ...}
```

and to allocate space in bytes:

```
allocb   operand
```

Table 2.1 Ranges of signed and unsigned integers.

	Signed	*Unsigned*
byte	–128 to 127	0 to 255
word	–32768 to 32767	0 to 65535
double word	–2147483648 to 2147483647	0 to 4294967295

where 'operand' is the number of bytes to allocate.

A `dcb` operand can be an expression which has a value in the range –128 to 255: you chose the appropriate range depending on whether you consider it an unsigned or a signed byte. The operand may also be a string which may have as many characters as will fit on a single line.

The NatSemi assembler has two directives to define byte constants:

```
.byte    { [rep] } operand ...
```

and

```
.sbyte   { [rep] } operand ...
```

The first is used to define unsigned and the second to define signed constants and the operands must lie within the appropriate ranges. In all the NatSemi directives, an operand may be preceded by a repetition factor in brackets: `[5]3` would be equivalent to the sequence `3,3,3,3,3` if written out in full.

To define one or more words, use

```
dcw   operand ...
```

A word operand can be an expression in the range –32768 to 65535 or a two-character string. NatSemi again has two directives, one for signed constants and one for unsigned. Signed constants are defined by

```
.sword   { [rep] } operand ...
```

and unsigned ones by

```
.word    { [rep] } operand ...
```

To define double word integers in the Acorn assembler, use

```
dcd   operand ...
```

A double word operand may be an expression in the range –2147483648 to 2147483647 (signed or unsigned), or a string of up to four characters. The equivalent ASM16 directives are

```
.sdouble { [rep] } operand ...
```

and

```
.double  { [rep] } operand ...
```

though the highest unsigned constant cannot exceed the highest signed value.

For the moment, expressions will be taken to be simply integer or string constants and will be explained more fully in the next section.

Integer constants may be written in binary, octal, decimal or hexadecimal, the default being decimal. Decimal numbers are written without a base. In the other cases the format is an optional sign followed by the base followed by one or more digits, the number depending on the size of the integer being defined. If the sign is omitted it is regarded as positive; otherwise it can be either '+' or '−'. The base code is #b for binary, #o for octal and #x for hexadecimal.

Some examples of integer constants are

```
#b11100111
#o347
231
#xe7
```

Because hexadecimal constants are so frequently used, the Acorn assembler allows them to be written in a shortened form: :e7 instead of #xe7 shown above.

The NatSemi assembler uses b' instead of #b for binary, o' or q' for octal and h' or x' for hexadecimal. It also allows a d' prefix for decimal constants. The above constants in ASM16 format would be

```
b'11100111
o'347
231 or d'231
x'e7 or h'e7
```

Strings are equally simple to define. They consist of one or more ASCII printing characters enclosed in apostrophes ('). The Acorn assembler has a counted string form: enclosing the characters of the string in quotes (") causes a count byte to be put in front of the string which can be used in printing or moving it. There is also a string escape character (*) which can be used to insert non-printing characters like line feed (*0a) and carriage return (*0d). The presence of this escape character means that to insert the character (*) into a string you must double it (**); an apostrophe or quotes must also be preceded by an asterisk (or doubled) if it appears in a string delimited by the same character. Some examples of strings are:

```
'This string does not start with a count byte.'
"This string is counted."
'Non-printing characters: *0a*0d'
"A counted string with embedded *""
'Embedded '' here.'
```

In the NatSemi assembler, a string may be delimited by either apostrophes (') or quotes ("): it makes no difference to the contents. If a string delimiter appears in the string, it must be doubled.

Examples of the Acorn directives are:

```
dcb   -128          ;signed byte
dcb   240           ;unsigned byte
dcb   '*',:8c,:0d   ;unsigned bytes
dcw   32000         ;unsigned word
dcw   -512          ;signed word
dcd   -10000000     ;signed double word
dcd   4000000000    ;unsigned double word
dcb   'A string with CR-LF',:0a,:0d
```

Examples of the NatSemi directives are:

```
.byte    3
.sbyte   -3
.byte    3,'A',h'0A',h'0D'
.word    [2]h'0A0D
.sword   -64
.double  h'FFFF0000,h'0000FFFF
.sdouble 'abcd'
.byte    'A string with CR-LF',h'0A,h'0D
```

2.6 REALS

Single precision reals take 4 bytes and double precision ones take 8 bytes. A real number can be divided into two parts, the exponent and the fraction. This will be familiar to anybody who has programmed in a high-level language where the number 1000000 is commonly written 1e6 (1E6 for FORTRAN freaks). Here the exponent denotes a power of ten. In real life, though, a power of two is used (a Very Large Business has used 16 but it is not as satisfactory) and the number 256 would be represented as 1b8 (1 times 2 to the power 8).

Single precision reals have 8 bits devoted to the exponent and 23 bits for the fraction, the most significant bit of the 4-byte quantity being used for the sign bit. The format is shown in Fig. 2.4. The format of a double precision real (8 bytes long) is shown in Fig. 2.5.

All floating point numbers are held in normalized form; that is, the most significant non-zero bit of the fraction is shifted left or right (while altering the exponent appropriately) until it is just to the left of the binary point, in the

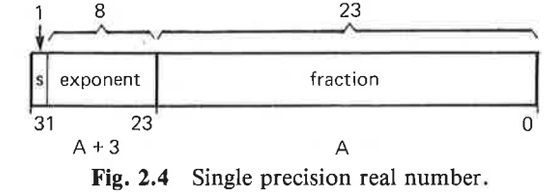

Fig. 2.4 Single precision real number.

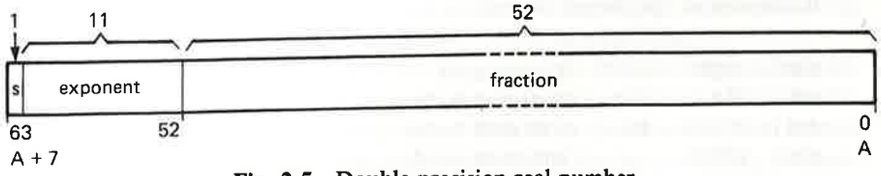

Fig. 2.5 Double precision real number.

units position. Since the first bit of the fraction is always a 1 it makes sense to omit it, allowing the fraction to be 1 bit longer for the price of a little extra complication.

The value in the exponent field of a single precision real ranges from 1 to 254, representing the exponent values −126 to 127. The exponent is biased by 127 to make it positive so that no sign bit is needed; when the value in the exponent field is 127 the true exponent is zero. The remaining two values are reserved for special floating point values. One is zero, which is represented as an all-zero exponent and fraction. (Zero is not included in the range of the ordinary format of floating point numbers as the ones nearest to zero are 1.18e−38 and −1.18e−38, leaving a gap.) The other special values are the group of numbers with 255 as their exponent. These are used to denote positive and negative infinity if the fraction is zero or Not-a-Number (NaN) otherwise. NaNs can be used to initialize floating point variables before a program starts; they cause a reserved operand exception on any attempt to use them, and bring to light subtle errors caused by using variables before they have been set.

For a double precision real, the value of the exponent field ranges from 1 to 2046, representing the exponents −1022 to 1023; again the exponent is biased, this time by 1023. The zero value is associated with double precision 0.0 and the double precision infinity has exponent field 2047.

The range of values spanned by single precision reals is from $2^{127} \times (2 - 2^{-23})$ to 2^{-126} in positive values and from -2^{-126} to $-2^{127} \times (2 - 2^{23})$ in negative values. The largest number is approximately 3.4×10^{38} and the smallest approximately 1.17×10^{-38}.

Double precision reals go from $2^{1023} \times (2 - 2^{-52})$ to 2^{-1022} and from -2^{-1022} to $-2^{1023} \times (2 - 2^{-52})$. In decimal the largest is approximately 1.8×10^{308} and the smallest 2.2×10^{-308}.

There are in fact two forms of real zero. Both have the fraction and exponent fields zero but the sign bit can be either 0 or 1 – both varieties will be taken to be zero when a floating point compare is used.

Short real constants are defined in Acorn's assembler by the directive `dcf`, long reals by `dcl`; the real number itself is written in the familiar FORTRAN way. It may contain either an 'E' (or 'e') or a decimal point or both, or it may take the form of an integer constant. In addition, the digits may have a sign before the first one and the 'E' may be followed by a signed or unsigned number – the exponent. Examples of real constants are:


```
.123        1.23        123.        -12.3        +0.123        123
123e        -1.23e2     123e-2      3e6
123e+5      123.0e-9
```

Real constants defined in the assembler are:

```
dcf   0.301029957   ;a short real
dcl   2.3025850929  ;a long real
```

The NatSemi assembler uses `.float` and `.long` instead of Acorn's `dcf` and `dcl`. The constant must also have either a decimal point or end with an exponent (just 'e' will do) and must start with a digit. The constant may be signed but integer constants are not acceptable. Some examples of legal NatSemi real constants are:

```
0.123       1.23        123.        -0.123       +123.0
+123e+0     12.3e1      -12.3e-5
```

Some example directives are:

```
.float    +12.3e+12
.float    [2]0.01
.long     2.3025850929
```

2.7 ASSEMBLER EXPRESSIONS

The full range of possible expressions is, in fact, astonishing. If it included real numbers it would be possible to write programs in it!

This is an area in which the usage of the NatSemi and Acorn assemblers differ sharply and it will be necessary to deal with them separately to make sure that each assembler is adequately covered. However, before embarking on a tour of the two assemblers, a description of their common approach to symbols can be factored out.

The rules governing symbol names differ, but the way they are used is the same. A symbol is a name for a value, which can be a constant, an address in memory or a synonym for a register. It is clear therefore that a symbol takes not only a value but also a type. A symbol that has a constant value is called an absolute symbol, a symbol referring to an address is called relative, and a symbol representing a register has to be defined in a special way so that its type is clear.

The different types of symbols are given values in different ways. An absolute symbol is given a value in a similar way to variables in a high-level language: the symbol appears on the left-hand side of a 'becomes' (the symbol appears in the label position of an 'equates' directive) and the value given it is the value of the expression on the right-hand side (the operand).

There are two kinds of relative symbol; one which refers to a name which is defined in another, separately assembled, part of the program, and one which refers to a location in the current assembly. The first is called, reasonably enough, an external symbol and the second type takes a value which is an offset to a particular base register. The value is defined by both the offset and the base register. A label on an instruction will have the offset shown in the assembler listing as the 'address' of the instruction and the base register will be the program counter. The base register for a label on a data generation or storage allocation directive will (usually) be the static base register with the offset as shown in the listing.

There are several symbols which cannot be used as variable names as these are reserved for the names of the general registers, the floating point registers and some others:

```
R0 R1 R2 R3 R4 R5 R6 R7
F0 F1 F2 F3 F4 F5 F6 F7
FP SP SB PC
TOS       EXTERNAL
UPSR PSR  INTBASE  MOD
MSR  PTB0 PTB1 EIA PF0 PF1 BPR0 BPR1 BCNT SC
```

The first three lines contain the names of the general and floating point registers and the CPU base registers. `TOS` is the symbol for top-of-stack addressing and `EXTERNAL` for external addressing. The general registers have been covered in detail but, of the others, only `PC` has been explained; the floating point registers come in the next chapter and the `FP`, `SP` and `SB` base registers will be found in Chapter 8 on procedures. The symbols on the last two lines are the names of the special CPU registers and the registers in the Memory Management Unit – both of which are covered in Chapter 10.

2.7.1 Acorn constants

The form of constants used in the Acorn assembler is reviewed in Table 2.2, where <digits> is to be interpreted as 0 to 9 for decimal, 0 to 9 and A (or a) to F (or f) for hexadecimal, 0 to 7 for octal and 0 or 1 for binary. The octal designator is letter 'O' or letter 'o'. All constants are unsigned (any minus sign preceding the first digit is taken as an arithmetic operator) and are taken as 32-bit quantities: the largest decimal number is 2147483647. In hexadecimal this is `:7fffffff`, in octal `#o17777777777` and in binary a sequence of thirty-one 1s.

Table 2.2 Integer constants.

decimal	<digits>
hexadecimal	#X<hex digits> or #x<digits> :<hex digits>
octal	#O<oct digits> or #o<digits>
binary	#B<digits> or #b<digits>

Finally, short strings of up to four characters in the uncounted form (enclosed in apostrophes) or up to three characters in the counted form (enclosed in quotes) can be used as constants in expressions. The equivalence between strings and numbers should be noted carefully as, though logical, it is not immediately obvious. Strings are, as you would expect, laid down in memory with the left-most character (or the count byte) first. Numbers, however, are laid down in memory *least significant byte first*. Thus the string `'A'` (uncounted) is equivalent to `:41` but the string `'AB'` is equivalent to `:4241` as the number `:4241` will enter memory least significant byte first; that is, as `:41`, `:42`. With counted strings the effect is still more bizarre: `"abc"` is equivalent to the number `:63626103` as `"abc"` is laid down in memory as `:03`, `:61`, `:62`, `:63`. If strings are to enter into expressions, you are well advised to do some little tests first – and, since the result may not be what the reader of your program expects, each occurrence should be carefully commented.

2.7.2 Acorn symbols

In the Acorn assembler, symbols may be made up from upper- and lower-case letters, digits and underscores(_) but must begin with a letter or an underscore.

Symbols may be of any length from 1 to 255 characters (the longest acceptable assembler line) of which 63 are significant. Symbols with the same first 63 characters will be considered identical: external symbols (names of procedures, entry points and common blocks), however, are kept to 31 characters and, with this, Acorn can be considered to have finally dragged linkers out of the 8-bit past (external symbols were once limited to as little as six or eight characters, even on some mainframes).

A few acceptable symbols are:

```
pi          Pi          PI
Label9      a9left      11090
InterfaceFacility
BlockWrite              SetBinaryTime
_MainGlobals            read_input
```

The Acorn assembler distinguishes between cases in symbols by default, so the first three symbols will be taken as different. If for some reason,this is not desirable, using `OPT +C` on the command line will cause upper- and lower-case letters to be considered the same and the three symbols above will be taken as identical.

Absolute symbols are given values by either the directive `equ`, which creates the symbol, giving it the value of the expression in the operand position, or by the directive `set`, which has the same effect as `equ` except that, if the symbol has already been used, it will simply be given a new value. The

`equ` directive expects to create the symbol and will fail with an error message if the symbol already exists.

It can sometimes be useful to define a symbol as an alias for a register name. If a procedure in a high-level language is being translated and it is possible to use registers for some of the local variables rather than memory, this can be made clear by equating the variable name to the register name. The name can then be used in instructions as a register but the name makes it easier to see what is going on. The directive to do this is `equr` (equate register.)

Relative symbols, other than external ones, are defined by being used to label an instruction or a data or storage directive. Again, this procedure creates them and it is an error for them to have appeared before. There is one special relative symbol which may appear in expressions and has a variable value, the symbol `$`. It always has the value of the current location counter – in an instruction it is the address of the first byte of the instruction and in a data or storage area it is the value of the location counter at the start of the line in which it appears.

External symbols are defined by appearing in the operand field of an `import` or `importc` directive, making symbols defined outside the module available to it, or of an `export` or `exportc` directive, making symbols defined within a module available outside it. If the symbol is a procedure entry point, `importc` or `exportc` must be used; if it denotes data, `import` or `export` must be used. These four directives are dealt with in Chapter 8 (Procedures and modules).

2.7.3 Arithmetic

Assembler arithmetic is performed with 32-bit numbers, ignoring any overflow; there are six arithmetic operators (Table 2.3).

For '–' to be taken as a unary minus, it must be the first character of the expression or appear immediately after a left parenthesis.

The modulus operator gives the remainder when its left operand is divided by the right one and uses the `REMi` instruction: both `31 % 10` and `31 % (-10)` give the result `1` and both `-31 % 10` and `-31 % (-10)` give `-1` since the remainder is calculated as

dividend – (dividend QUO divisor) × divisor

Table 2.3 Acorn arithmetic operators.

-	unary minus
*	multiply
%	modulus
/	divide
-	subtract
+	add

where the division rounds down to zero and the remainder always has the sign of the dividend; so

$$-31-(-31/10)\times 10$$

gives

$$-31-(-3)\times 10 = -1$$

2.7.4 Acorn logical operators

The assembler has conditional NOT, conditional AND and conditional OR operators as well as the usual AND, OR, Exclusive OR and complement operators (Table 2.4).

Table 2.4 Logical operators.

~	bitwise complement
&	bitwise AND
|	bitwise OR
^	bitwise Exclusive OR
!	conditional NOT
&&	conditional AND
||	conditional OR

AND, OR and Exclusive OR treat their operands as 32-bit words and perform the appropriate logical operation pairwise on the bits. Complement is a unary operator and shares the same symbol as Exclusive OR: the symbol is taken to be complementary whenever it appears as the first character of an expression or immediately after a left parenthesis (like unary minus). In any other position in an expression, it is taken as Exclusive OR.

The conditional operators `!`, `&&` and `||` have the same effect as `~`, `&` and `|` respectively. They differ only in their precedence. So, if

```
a   equ   1
b   equ   2
c   equ   -9
```

then the expression

```
! a<b
```

would evaluate to zero (equivalent to FALSE) as the complement of `a<b` (TRUE with the value –1 or `:ffffffff`) but

```
~ a<b
```

would be evaluated as

```
(~ a) < b
```

which will be TRUE (−1) since ~ a is −2. These operations should be used when combining the logical results of relational expressions.

Some other examples of the use of the logical operators are:

```
prec1   equ   ~ 2              ;value is :ff ff ff fd
prec2   equ   ~ 2<< 3          ;value is :ff ff ff e8
cond1   equ   3 > 2 && 3 = 2 ;false (0)
cond2   equ   3 > 2 || 3 = 2 ;true (-1)
```

The expression ~ 2 complements a double word of thirty-one 0s with a 1 bit in bit 1, making thirty-one 1s with a 0 bit in bit 1. In the assembler printout, the value :ff ff ff fd will be printed as :fd ff ff ff as it prints the bytes in the order they are in memory. The least significant byte in binary is #b11111101, so when shifted left 3 bits this becomes #b11101000 which is :e8.

2.7.5 Acorn shift operators

There are two shift operators, a left shift and a right shift. The left shift is represented as << and the right shift as >> and there must be no gap between the characters. The shifts are logical shifts: they feed in zeros into the vacated bit positions at either end. This can give unexpected results when shifting negative numbers and, if you intend doing this, it is worth writing a small piece of assembler to check that the result is what you expect. In action, the left-hand operand is shifted by the number of bit positions given by the right-hand operand; negative shift counts perform the opposite shift. An example of their use is:

```
:1234 >> 2   →   :48d
:1234 << 2   →   :48d0
```

2.7.6 Relational expressions

All six of the possible relations (Table 2.5) are available and always perform signed comparisons. If the relation is true all 1s (:ffffffff) are returned, if it is false all 0s are returned.

Table 2.5 Relational operators.

>	greater than
>=	greater than or equals
=	equals
<>	not equals
<=	less than or equals
<	less than

2.7.7 Precedence and parentheses

As in high-level languages, there is a precedence ranking between operators which determines how an expression is to be evaluated in the absence of parentheses (Table 2.6). Thus the expression

```
1 & 3 << 2 | 5 ~ 7
```

will be evaluated as

```
((((1) & (3<<2)) | 5) ~ 7) → 2
```

Table 2.6 Acorn operator precedence.

(highest)
unary minus, complement
left shift and right shift
AND, OR and Exclusive OR
multiply, divide, modulus
add, subtract
relational operators
conditional NOT
conditional AND and OR
(lowest)

2.7.8 Absolute, relative and external expressions

If the symbols in an expression are all absolute, any of the operators may be used. If, however, one of the symbols is relative or external then it may only be used in conjunction with a plus or minus sign followed by an absolute expression. For a relative symbol only an absolute, positive or negative offset can be applied to it – there is no obvious sense in shifting or multiplying a relative symbol. Finally, the only operator allowed between two relative symbols (which may not be external) is a minus, which will give the gap between them as an absolute value. The two symbols must also be relative to the same base register.

2.7.9 NatSemi constants

The syntax of constants used by the NatSemi assembler is given in Table 2.7,

Table 2.7 Integer constants.

decimal	D' <digits> or <digits>
hexadecimal	X' <digits> or H' <digits>
octal	O' <digits> or Q' <digits>
binary	B' <digits>

where <digits> is one or more digits from 0 to 9 for decimal, from 0 to 9 and A to F for hexadecimal, from 0 to 7 for octal, and a sequence of 0s and 1s for binary. The permitted range of an integer constant is determined by the context. A constant intended for a byte must lie in the range −128 to 255, for a word it must lie in the range −32768 to 65535, and for a double word in the range −2147483648 to 2147483647. No integer may lie outside the double word range. All integers may be signed; both a preceding plus sign and a minus sign is acceptable – an unsigned integer is assumed to have a prefixed plus sign.

Floating point constants may not take the form of an integer; they must contain either a decimal point or an exponent (or both). The decimal point may *not* be the first character but may be the last or may be in the middle. An exponent is the character 'E' followed by a decimal number of between one and three digits. This number may be signed and will be assumed positive if not. The assembler allows both long and short floating point values to be used but stores them both internally as long.

Further details on floating point numbers will be found in Chapter 5.

String constants are sequences of ASCII characters delimited by either apostrophes (') or quotes ("): there is no difference between strings delimited by apostrophes and those delimited by quotes. The maximum length of a string is determined by the context in which it is used: in a byte context it may consist of a single character only, in a word context it can contain two characters and in a double word context, four. When used as the only operand of a `.byte` directive, it can contain as many characters as will fit on the line after the directive – a line may contain up to 132 characters.

2.7.10 NatSemi symbols

In the assembler, symbols are used to refer to a constant, an address or a register, so symbols have both a value and a type associated with them. A number of symbols are reserved and may not be used. They include the register names and the instruction mnemonics, the directive names, and the logical, arithmetic and shift operator names.

A symbol may contain upper- and lower-case letters, digits, underscore (_) and period (.). The first character of a symbol may not be a digit. A symbol's length is limited only by the length of a source line, but two symbols with the first eight characters the same will be taken as identical. The assembler does not distinguish between upper- and lower-case letters.

There are many more reserved symbols in this assembler than in the Acorn one. They are

- all the instruction mnemonics,
- all the directive names,
- all the CPU, MMU and FPU register names,
- the names of the operators, and
- `EXT` and `TOS`.

Symbols fall into three classes: constant, relative and register synonyms. Constant symbols may have an integer, a floating point or a string value: the

Table 2.8 NatSemi operators.

+	unary plus
-	unary minus
COM	unary complement
NOT	conditional NOT
*	multiply
/	divide
MOD	modulus
AND	bitwise AND
OR	bitwise OR
XOR	bitwise Exclusive OR
SHL	shift left
SHR	shift right
+	add
-	subtract

limit on the length of a string symbol is set by the ultimate context the string will be used in.

The value of a relative symbol is an offset to one of the four base registers, PC, SB, SP or FP. PC and SB relative symbols may be defined by using them as labels on an instruction (PC relative) or a data or storage directive (SB relative). It is also possible to define them directly by giving the offset (as an absolute expression) together with the base register or by another relative symbol combined with an absolute expression.

External symbols are defined by appearing in the operand field of an `.import` or `.importp` directive, making symbols defined outside the module available within it, or an `.export` or `.exportp` directive, making symbols defined in a module available outside it. If the symbol is a procedure entry point, `.importp` or `.exportp` must be used; if it denotes data, `.import` or `.export` must be used. These directives are discussed at greater length in Chapter 8 (Procedures and modules).

The operators provided by the NatSemi assembler are shown in Table 2.8. The unary plus has no effect at all. The unary minus is only recognized when it is the first character of an expression or when it occurs immediately after a left parenthesis. The shift operators both feed 0s into the vacated places.

The operators are applied to the symbols and constants in an expression with a precedence given in Table 2.9.

Table 2.9 NatSemi operator precedence.

(highest)
unary plus, unary minus, unary complement, conditional NOT
multiply, divide, modulus, AND, shift left, shift right
add, subtract, OR, Exclusive OR
(lowest)

Expressions using only constants and absolute symbols may contain any of the operators. A relative and an absolute term may be combined by multiplication, division, MOD, AND, SHR, SHL, addition, subtraction, OR and XOR. Of these only addition and subtraction would be used in practice as the results of the other operators would be highly dependent on the value of the offset of the relative symbol. This is easily changed by inserting or removing a line of assembler, with what could be disastrous results.

An external term may only be combined with an absolute expression using either addition or subtraction; when using subtraction, the absolute expression must be subtracted from the external symbol.

Two relative symbols may only be subtracted from each other, the result will then be absolute.

3 Integer arithmetic

3.1 BINARY ARITHMETIC

This first section on the details of binary arithmetic is intended for readers not fully conversant with carry and overflow and how they arise in binary integer arithmetic. It may be skipped if it has nothing new for you.

The addition table for binary digits is very simple. Figure 3.1 shows addition when there is no carry from the right: when both bits are 1 the result is 0 with a carry, represented by 10.

	0	1
0	0	1
1	1	10

Fig. 3.1 Binary addition with no carry.

	0	1
0	1	10
1	10	11

Fig. 3.2 Binary addition with carry.

Figure 3.2 shows the results of adding two bits together when there is a carry from the right. Here the result of adding 0 and 0 is 1 as the addition is 0+0+carry and the addition of 1 and 1 is 1 with a carry.

As an example, the sum of `#b101100111` and `#b10001` is

```
101100111   359
    10001    17
---------------
101111000   376
```

The addition of the two right-most bits (taken as the least significant ones) gives 0 with a carry; the following two 1 bits each give 0 with a carry which eventually winds up as 1 in the fourth bit; the lower number is extended to the left by zeros and the sum of each of these is clear.

In the above example it has been tacitly assumed that there is no limit to the number of bits to the left, but as memory is divided into bytes, words or double words, addition in terms of the 32000 is not quite so simple. To save

space, examples will be in terms of bytes though the principles can be extended in an obvious way to words and double words.

Take for example the addition of the two bytes #b00111000 and #b11001111:

```
 00111000    56
 11001111   207
-----------
100000111   263
```

The result contains 9 bits, the least significant 8 bits of which will find room in the byte result with the most significant 1 out in the cold. The excess bit is, in fact, preserved in the PSR and is called the carry bit. After an addition, it is set to 0 if the addition does not result in a carry or a 1 if it does. The advantage of this can be seen if you consider the above 8-bit sum as two separate 4-bit sums: the lower 4 bits give rise to a carry which is added into the lowest bit of the upper 4 bits. This process can be used to extend the length of sums which the 32000 can handle to multiple double words when the carry is included in the additions of the more significant parts of the sum.

The example shown above is considered to be an addition of two unsigned bytes as the sign bit has been taken to be a value bit. In the integer formats given in the previous chapter, this sign bit has been introduced simply as an indicator (0 means positive, 1 means negative), but there is more to it than that. The sign bit is part of a convention for dividing the unsigned values that a memory unit can take into two equal groups, one taken as representing negative numbers and the other representing positive numbers. The convention used by the 32000 (and almost all other computers) is called the two's complement representation, as the negative numbers are considered to have been subtracted from the power of two just too large to fit into the unit. For example, the largest power of two just too large for a byte is 2^8 – it requires 9 bits to represent it. To get the representation of −1 in two's complement, 1 is subtracted from 2^8:

```
100000000
        1
-----------
 11111111
```

Using conventional arithmetic, 1 from 0 won't go, so borrow 1. From the addition table, 1 from 10 leaves 1. The borrow must be collected in the next column, giving 1 from 0 again. This continues until the eighth bit is subtracted from the eighth and ninth bit of the minuend giving 1 and completing the transaction.

Similar examples show that the negative numbers are represented as:

```
 11111111    -1
 11111110    -2
    :         :
 10000001   -127
 10000000   -128
```


The remaining unsigned numbers form the positive two's complement numbers, taking the same values as they have when considered as unsigned

```
 01111111    127
 01111110    126
    :         :
 00000001      1
 00000000      0
```

This does give a rather peculiar end-to-end effect, with the lowest negative number coming immediately after the highest positive one, but this arrangement is defensible on two grounds: firstly, it is more convenient electronically and, secondly, it goes unnoticed.

The same applies to words and double words with an appropriate increase in the number of bits considered.

Addition is performed in exactly the same way as for unsigned quantities (thus the electronic convenience of two's complement) with the sign bit being added in the same way as the other bits, giving a result correct in both sign and magnitude. For example,

```
 00110100   +52
 11111010    -6
-----------
100101110   +46
```

The carry is ignored in signed arithmetic; as you can see, the result is positive (0 sign bit) and is the two's complement representation of +46. It also works the other way round:

```
 11001100   -52
 00000110    +6
-----------
 11010010   -46
```

In this case there is no carry but, as it is being ignored anyway, this is of no consequence.

Although very convenient, there is the problem of overflow. For example,

```
 10110011   -77
 10100000   -96
-----------
101010011   +83
```

The correct result is −173 which is smaller than the lowest signed byte value, −128. As you can see, the addition of the two bit 7s has resulted in a zero with a carry and there has been no compensating carry into bit 7 to keep the sign the same. If one of the bit 6s is changed to a 1 you get

```
 10110011    -77
 11100000    -32
-----------
110010011    -109
```

which is perfectly all right. You can make similar examples from positive numbers each with a 1 in bit 6 or each with a 1 in bit 5 and one with a 1 in bit 6. I have a Casio fx-570 to do this binary arithmetic on, otherwise (being a poor arithmetician) I couldn't have guaranteed the above results.

Strictly speaking, overflow occurs when the carry from the addition of the bit 6s (talking bytes) differs from the carry resulting from the addition of the bit 7s.

3.2 ADDITION

The NS32000 has nine add instructions, in three generic groups. The first is `add`, plain and simple, and in its generic form it is written

```
ADDi src, dest
```

The `src` integer is added to `dest` and the result stored in `dest`. (Of course, `addb` is required for a byte operand, `addw` for a word operand and `addd` for a double word operand.)

The two operands the add instruction takes are described as the source (`src`) and destination (`dest`) operands and from Appendix A you can see that either may take any one of the forms of a general operand. The whole might and panoply of the general operand will not be arrayed here for fear of the effect that future shock might have on you: to start with, only the register, immediate and memory space forms will be used.

To demonstrate the register form, here is an addition of two words in registers:

```
addw   R1, R0
```

The word in bits 0 to 15 of general register 1 will be added to the word in register 0 and the result will overwrite the destination word.

Note that whenever a byte or a word in a register is changed, only bits 0 to 7 (for a byte) or bits 0 to 15 (for a word) are altered; the higher order bits remain unchanged. This is important to realize at the outset, as there are two operand modes used for indexing in which all 32 bits of the general purpose register are used and it is fatally easy to use word or byte operations to set or

alter the register contents forgetting about the high-order bits which may not be all zeros!

To demonstrate the immediate form let us suppose that the byte in register R7 contains a number in the range 10 to 15 decimal and that it is to be turned into an ASCII code from 'A' to 'F' – a stage in a binary-to-hexadecimal conversion. If the number is 10, it must become 'A' so the expression `'A' - 10` must be added to it.

```
addb   ='A'-10, R7
```

is the instruction required. The `src` expression is a constant and is actually incorporated into the instruction itself, being then called an immediate operand. This usage has the advantage of speed as its value is known as soon as the instruction is read; no further memory access is needed. The space taken up in the instruction by an immediate operand is determined by the size of integer demanded: here both `src` and `dest` operands are bytes and the constant will be stored as a byte, but if a double word was required (`addd`) the constant would take up 4 bytes in the instruction. Note that `dest` may not be an immediate operand as it would be overwritten and this is not usually (meant to be) the fate of constants. The assembler will mark this as an error.

To distinguish an immediate operand, the Acorn assembler requires an `=` before the expression. The NatSemi assembler will take any expression with an absolute value (that is, not relative to an address) as immediate: the above example for the NatSemi assembler would be

```
addb    'A'-10, R7
```

simply omitting the Acorn `=`.

The final operand form (for the moment) is Memory Space, which has four forms. The first is used to read parameters and to read and write to local variables in procedures – it will be dealt with in Chapter 8. The second form is used to read and write to the stack and will also be processed in Chapter 8. The third form is used in accessing variables and blocks of storage in the static data area which acts as a local 'global' area for all the procedures in a module. While the details of this must wait again for Chapter 8 it will be tacitly assumed that variables, arrays and records have been allocated storage in this area. The last Memory Space operand form is addressing relative to the program counter. This form of address can be used for variables and arrays, and will be used if no static area is declared, but is more usually associated with labels as the object of jump and conditional jump instructions and will be dealt with later in this chapter. The actual operand form is concealed from the user as the assembler generates it automatically on being given a name as an operand, be it label or variable. Assuming, then, that a double word variable has been declared earlier in the program and has been initialized:

```
addd   =10, sum
```

will add the immediate value 10 decimal to the double word `sum`. As it is a double word to double word addition, 10 will be kept as part of the instruction in 4 bytes even though one is sufficient to hold it.

If the value to be added had been in the range −8 to 7, it would have been possible to use one of the `ADDqi` instructions instead. Like the `ADDi` instructions, these exist in the three operand sizes and add the 'quick' operand to the destination: the quick operand is kept as a 4-bit value in the instruction allowing an economical addition of a small integer. The quick range has been selected so that an address in a register may be decremented by up to the size of a quad word or incremented by up to double word size plus a bit for luck. If we take it that `R5` has the address of a block of double words in it, then to get the address of the next double word we need

```
addqd   4, R5
```

The `4` will be sign-extended up to a double word before being added to the register. A word `add` might have sufficed but be careful when adding negative numbers to a register used as an index – any bits between 16 and 31 will not take part in a word addition and you may have an exasperating circular addressing problem to unpick.

The last of the three types of add instruction is one in which the carry bit is included in the addition. Its mnemonic is `ADDci` and the effect of

```
addcd   a, b
```

is that `b` becomes the sum `a+b+c`, where `c` is the value of the carry bit in the PSR: 1 if it is set, 0 if reset. This is only useful when doing multiple precision adds, when the objects to be added together consist of two or more double words. Assuming that the numbers consist of three double words each, the following code will perform a multiple precision addition

```
addd   a, b
addcd  a+4, b+4
addcd  a+8, b+8
```

The first `addd` adds the least significant double words, `a` and `b`, leaving the result in `b` and any carry out of the sum in PSR C. The second `addcd` adds the middle double words together with any carry and the last one does the same. The result, if carry is not set (unsigned) or F is not set (signed), is the multiple precision sum. In multiple precision numbers, only the most significant unit has the sign bit; all the other bits are taken as ordinary value bits.

The effect on the flags in the user byte of the PSR of an addition is confined to the C and F flags: the C flag is left at 1 if a carry occurred and 0 otherwise; the F flag is set to 1 if an overflow occurred, otherwise it is set to 0. The negative (N), zero (Z) and low (L) flags are not changed by either addition or subtraction. Only comparison operations are empowered to alter them.

This is a considerable step away from 8-bit and 16-bit microprocessors where these flags are changed not only by addition and subtraction but by load instructions as well. When there is only one register (or one arithmetic register) it seems reasonable to alter the machine state flags according to the value of this register, but with eight integer and eight floating point registers this practice has little to recommend it.

3.3 MOVEMENT AND CONVERSION

It is fine to add, but first there must be something to add. Up to this point, it has just been assumed that the registers and memory variables have the desired contents. This can be guaranteed for variables with a `dcb`, `dcw` or `dcd`, but registers are born with indeterminate contents and must be set dynamically. This introduces the move instructions, four generic groups of them, two similar to the `add` groups, `MOVi` and `MOVQi`, and two capable of converting a smaller unit to a larger one, byte to word for instance.

`MOVi` is easily defined. It moves the source operand to the destination operand and has the standard byte, word and double word varieties.

```
MOVi   src, dest
```

moves the contents of `src` to overwrite the contents of `dest`. The `src` operand can also be in immediate mode.

```
movb   =-55, byte
```

sets `byte` to −55. As with the `add` group, if the value is in the range −8 to 7, `MOVQi` can be used instead: to set register 3 to zero,

```
movqd   0, R3
```

will serve. It may not be necessary to set all 32 bits (the quick 0 will be extended to a double word 0) but make sure that `R3` is not used as an index.

The two other move groups are used to extend a shorter unit to a longer one. There are two ways of doing this: either the shorter unit is taken to be unsigned and the higher bits are all zeroed, or it is taken as signed in which case the higher bits are all set to the sign bit of the shorter unit. The generic name for the unsigned extension is `MOVZii`, and the possible varieties are `movzbw` (byte zero extended to word), `movzbd` (byte extended to double word) and `movzwd` (word to double word). Earlier on it was pointed out that the immediate operand `=10` would be kept in 4 bytes as it was a double word operand. Here is a way to save a little space at the price of a couple of machine cycles.

```
movzbd   =10, R0
```

will set `R0` to (double word) 10 while keeping the value in a byte.

Signed extensions are very similar; the generic opcode is `MOVXii` with the same variations as `MOVZii`.

```
movxbd   =-10, R0
```

will set all 32 bits of `R0` to −10.

3.4 SUBTRACTION

Subtraction is the addition of the minuend and the two's complement of the subtrahend. The PSR C bit is set if there is a borrow condition which occurs when there is *no* carry out of the top bit; otherwise the C bit is cleared:

```
  01100110   +102
- 00000110     +6
```

becomes

```
  01100110   +102
+ 11111010     -6
 101100000   +96
```

In this case the result can be contained in 8 bits and the carry out of bit 7 means (for a subtraction) that no 'borrow' has occurred and the PSR C bit will be cleared.

In the following example the minuend and the subtrahend differ in sign and, with the values chosen, the result is too large for 8 bits:

```
  01100110   +102
- 11100010    -30
```

becomes

```
  01100110   +102
+ 00011110    +30
  10000100   -124
```

The lack of a carry out of bit 7 indicates that the result is too large for 8 bits and the C bit will be set. The F bit will also be set as the carry into bit 7 (a 1) differs from the carry out of bit 7 (a 0). Comparing this with the first subtraction you will note that the carry into bit 7 and the carry out are the same (a 1).

The subtraction instructions parallel the addition instructions except that

there is no quick version. There is the form which ignores carry, `SUBi`, and the form which includes it, `SUBCi`. For instance,

```
subw   i, j
```

subtracts the contents of the word called `i` from the contents of the word called `j`, leaving the result in `j`.

Multiple precision subtraction is like addition.

```
subd   m, n
subcd  m+4, n+4
subcd  m+8, n+8
```

performs a multiple precision subtraction of the three double words called `m` from the three called `n` with the result being left in `n`.

Two further instructions in the same line of business are

```
NEGi    src, dest       ;negate an integer
```

and

```
ABSi    src, dest       ;get the absolute value
```

These are both simple instructions, but nonetheless welcome. The code required to perform these functions in their absence is a nuisance to write, and being able to use a single instruction instead of two or three (with conditional jumps which can be troublesome) is a boon to the compiler writer. However, they also have their little ways.

There is always one number that `ABSi` must fail on. You will have noticed, in the discussion above on two's complement, that there was no counterpart to the most negative number for a given unit. For a byte this number is −128 and the highest positive number that will fit into a byte is +127 (255 is, of course, only available when the byte is considered unsigned). For a word it is −32768 and for a double word −2147483648.

If `ABSi` is asked to get the absolute value of one of these numbers it will set the F flag and the number will be transferred to the `dest` operand unchanged.

`NEGi` calculates the two's complement of the `src` operand by subtracting it from 0 and placing the result into the `dest` operand. This is in all respects a normal subtraction and the C bit in the PSR will be set if there is a borrow – as there will be for every number except zero. Like `ABSi`, the largest negative number for a given `i` cannot be negated and, if this is attempted, `NEGi` will set the F flag and transfer the `src` operand to the `dest` operand untouched.

3.5 COMPARISON AND CONDITIONAL BRANCHES

Comparison can be viewed as a subtraction of the first operand from the second keeping only the sign of the difference and whether it is zero or not.

This information about the difference is kept in the N, Z and L bits of the PSR: the sign is kept in the N bit. If the difference is negative, the N bit is set; otherwise it is clear. The Z bit is set if the difference is zero and it is clear if the difference is non-zero. The L bit is quite interesting; it records the result of the comparison when the two operands are regarded as unsigned numbers. It is set if the first operand is greater than the second one. In considering binary addition, it was shown that the same process produced the result which could be thought of as signed or unsigned depending on the operands. Here it is brought into sharper focus. The same comparison instruction gives rise to two results, one to be used if the operands were considered unsigned, the other if they were signed.

All three bits are set by each comparison. If you consider the quantities signed, the result of the comparison will be sought in the values of the Z and N bits; if the quantities were unsigned, the bits tested will be Z and L.

Normally, this level of detail is unnecessary as the testing will be done by a 'branch on condition' or a 'save condition' instruction. For comparing signed numbers (`CMPi A, B`) the mnemonics are:

`bgt`	Branch if A > B
`bge`	Branch if A ⩾ B
`beq`	Branch if A = B
`bne`	Branch if A < > B
`ble`	Branch if A ⩽ B
`blt`	Branch if A < B

For comparing unsigned numbers (`CMPi A,B`) the mnemonics are:

`bhi`	Branch if A > B
`bls`	Branch if A ⩽ B
`beq`	Branch if A = B
`bne`	Branch if A < > B
`bhs`	Branch if A ⩾ B
`blo`	Branch if A < B

Here `hi` stands for 'HIgher', `ls` for 'Lower or the Same', `hs` for 'Higher or the Same' and `lo` for 'LOwer', distinguishing them from the signed conditions. `beq` and `bne` are used for both signed and unsigned numbers as the same instruction serves equally well for either. It is important to keep the above tables in mind as, if you go into the details of what `bgt` tests, you will find that it branches on the N bit being set; that is, if `A` is subtracted from `B` and you get a negative result it shows that `A` is greater than `B`. This is just an explanation of what might appear puzzling and perhaps incorrect on close examination but the choice of the condition name has been made to correspond to the result of the comparison of `A` and `B` rather than the sign of the difference.

There are also branch instructions for testing for carry and overflow from the integer arithmetic instructions – the C and F flags. These are:

`bcs`	Branch if carry is set
`bcc`	Branch if carry is clear
`bfs`	Branch if flag is set
`bfc`	Branch if flag is clear

and an unconditional branch

`br`	Branch unconditionally

These can be used to call attention to an unexpected carry or overflow and the unconditional branch will be used whenever a piece of code must be skipped. For instance, when translating an `IF ... THEN ... ELSE ... FI` construction, the last instruction in the `THEN` clause would be an unconditional branch to the code following the `FI`.

The operands of these branch instructions will almost always be labels as the length of 32000 instructions is not simple to calculate. The labels will be translated by the assembler into Memory Space mode using a displacement from the PC register. This displacement can go up to some 16 million bytes positive or negative and the familiar 'offset out of range' errors of the older 8- and 16-bit processors fade into the awkward past.

The compare instructions are in two groups, integer comparisons (`CMPi`) and quick comparisons (`CMPQi`).

```
cmpqd   0, unsg
blo     nz
```

will compare the contents of `unsg` with zero and branch to `nz` if `unsg` is not zero. The use of unsigned comparisons with `CMPQi` must be carefully thought out. It is possible to write

```
cmpqd   0, unsg
bhi     nz
```

unthinkingly expecting it to jump to `nz` if `unsg` is not zero. However, a moment's thought will show that 0 can never be higher than `unsg` and that therefore the branch will never be taken – the quick values other than 0 don't seem to cause the same confusion. In most cases it will be useless to compare the negative quick values with an unsigned number (in word units, –1 corresponds to the unsigned 65535) unless you are being devious – not a good idea in assembler.

3.6 MULTIPLICATION

There are two forms of multiplication, one chiefly for use by compiler writers and the other by assembler programmers. The first form (`MULi`) multiplies two integers putting the product in the destination integer. The multiplication is

single length as the product is assumed to be no larger than the integer size and, in fact, any high-order bits will be truncated. As an example

```
mulw   =5, R0
```

will multiply the contents of bits 0 to 15 of R0 by 5, leaving bits 16 to 31 unchanged. If the result of the multiplication exceeds 65535 (unsigned) or the range −32768 to 32767 (signed) then *without warning* the excess bits will be lost, leaving the true result modulo -2^{16}. This suits compiler writers as there is no high-order part of the product to check or make space for. (They have ways of checking range before the multiplication and can avoid potential overflows.) It can also be extremely useful for writing linear congruential random number generators which are based on expressions like

```
r = a * b MOD m
```

Often m can be chosen to suit the CPU word length and the truncation above means that no MOD operation is needed.

Extended multiplication (MEIi) can sometimes be more useful than MULi as it develops the full double length product of the integers. The operands are interpreted as unsigned – only MULi gives a signed multiplication. The source operand for MEIi (the first one) is taken to be integer length while the destination is double integer length. Using meib, the source is one byte in length but the destination is two bytes long; with meiw the source is one word long while the destination is a double word; and for meid the source is a double word and the destination two double words or a quad word.

Before the multiplication, the second operand must be in the least significant half of the destination. If this is in memory, the least significant integer is the one with the lower address.

```
src    dcb    55
dest   dcb     7
       dcb     0

       ...

       meib   src, dest
```

After the multiplication, src will still be 55 but dest will be 129 and dest+1 will be 1. The result is 385 which is 256 + 129.

If the destination is to be a general register, *two* registers will always be used whether the integer is a double word or not. The registers will be taken as an even–odd pair with the even register name being used as the operand; the pair is taken as the even register and the next consecutive odd one. The second integer operand is put into the appropriate part of the even register and the result will have the low-order integer (of the resulting integer pair) in the even

register and the high-order integer in the following odd one. For comparison, the same multiplication will be done but with dest being the even–odd register pair R4–R5.

```
movqb  7, R4      ;multiplier
meib   =55, R4    ;multiplicand
```

The result will be in two bytes, the low-order one in bits 0 to 7 of R4 and the high-order one in bits 0 to 7 of R5. None of the other bits in either register will have been changed.

This multiplication is, of course, perfect for multiple precision work. Using our friends a and b from the addcd example earlier their product can be calculated by

```
A   dcd   1, 2          ;the digits A0 and A1
B   dcd   4, 5          ;the digits B0 and B1
C   dcd   0, 0, 0, 0 ;the digits C0, C1, C2 and C3
    ...

    movd   B, R0        ;*** multiply B0 by A0
    meid   A, R0        ;result in R0-R1
    movd   R0, C        ;result to C0-C1
    movd   R1, C+4

    movd   B, R0        ;*** multiply B0 by A1
    meid   A+4, R0
    addd   R0, C+4      ;add result to C1-C2
    addcd  R1, C+8

    movd   B+4, R0      ;*** multiply B1 by A0
    meid   A, R0
    addd   R0, C+4      ;add result to C1-C2
    addcd  R1, C+8

    movd   B+4, R0      ;*** multiply B1 by A1
    meid   A+4, R0
    addd   R0, C+8      ;add result to C2-C3
    addcd  R1, C+12
```

This algorithm is a specialized form of Algorithm M (Multiplication of non-negative integers) on page 253 of the second volume of Donald Knuth's *The Art of Computer Programming*. To save you going and fetching your copy (you haven't got a copy!?) the algorithm works on the same principle of paper and pencil multiplication of decimal numbers. Starting with the least significant digit of the multiplier, multiply it with each digit (starting again with the least significant) of the multiplicand adding the product digits to the partially formed sum. The only difference is that these digits, instead of going

from 0 to 9 or 0 to 15, go from 0 to 4 294 967 295! After the multiplication the values of C to C+3 will be 4, 3, 1 and 1 respectively.

3.7 DIVISION, MODULUS AND REMAINDER

The 32000 is unusually generous in its provision of division operations. There are two single length divisions and a double length one corresponding to MEIi above, and, as well as this, there are two remainder operations, one for each of the single length divisions.

Both the single length divisions (called DIVi and QUOi) take byte, word or double word integers and provide a single length integer result. Division does not always give an exact integer result and the difference between the divisions lies in the steps they take to deal with any fraction.

QUOi behaves rather as one has come to expect a single length division to behave; it simply throws the fraction away. DIVi, however, rounds the quotient down to the next lower *or more negative* integer. So, for positive numbers, both divisions will give the same result. For instance, dividing 111 by 10, DIVi will round the exact result (11.1) down to 11 and QUOi will return the same result by throwing away the fraction. It is with negative results, obtained when the signs of dividend and divisor differ, that the quotients delivered by the two divisions can part company.

In dividing −97 by 20 using DIVi, the exact quotient −4.85 will be rounded down to −5, the nearest integer less than or equal to −4.85. The same division performed using QUOi will give −4 as the result. This is the nearest integer less than or equal to *the absolute value* of −4.85. You will probably not find the quotient from positive numbers surprising but you should think carefully when negative numbers are involved.

Each single length division has a corresponding remainder function which is calculated by the same algorithm: MODi corresponds to DIVi and REMi to QUOi. In both cases the remainder is calculated from

dest − ((dest DIV src) TIMES src)

where dest DIV src is calculated in the same way as DIVi or QUOi and TIMES is a conventional multiplication. The remainder always takes the sign of the divisor (the src operand).

To take the same examples as before, the remainder from 111 DIV 10 will be 1 whether using MODi or QUOi but the remainder from -97 DIV 20 will be 3 (−97−(−5×20)) using MODi and −17 (−97−(−4×20)) from REMi. Some high-level languages like Pascal use DIVi and MODi to implement their DIV and MOD operators; others, like FORTRAN, use QUOi and REMi instead. When using QUOi and REMi be very careful when negative numbers are expected.

Double length division, DEIi, takes its operands to be unsigned like MEIi but, while MEIi starts with a single length destination operand and ends with a double length one, DEIi starts with a double length dividend (destination) and

ends with two single length results – a quotient and a remainder. MEIi and DEIi also use registers in a similar way. The double integer destination operand must be set to a double length value before the DEIi instruction: after the division the lower integer of the destination contains the remainder and the higher integer the quotient. As with MEIi, if the destination is in memory, the lower integer has the lower memory address; if the destination is to be a register, an even–odd pair must be used with the even register containing the lower integer and the odd register the higher. After the division, the even register will be left with the remainder and the odd register with the quotient. As an example of its use, here is a binary-to-decimal conversion

```
dec   allocb  10      ;number of decimal digits
      bin     dcd     1234

              ...

      movd    bin, R2   ;lower integer
      movzbd  =10, R4   ;index to DEC
loop  movqd   0, R3     ;zero extended
      deid    =10, R2   ;get BIN MOD 10
      addb    ='0', R2  ;make an ASCII digit
      movb    R2, dec-1[R4:b]
      movd    R3, R2    ;old quotient to new dividend
      acbd    -1, R4, loop
```

This example contains an instance of a new instruction and a new addressing mode.

The instruction is ACBi (Add, Compare and Branch). It is intended for loop control and has three operands; the first is a quick integer (−8 to 7), the second a register and the third a label. It adds the quick integer to the register (the i determines what length of the register takes part in the addition) and then, if the result is not zero, it branches to the label. It is treated more fully in Chapter 8.

The new addressing mode is scaled indexing, which comes in two parts: the first part is a general operand except that it may not be an immediate operand or contain a scaled index itself, and the second is the scaled index. In this case the general operand is a variable name with an offset (dec-1) and the scaled index is a byte index on the value of R4 ([R4:b]). It need not be a byte index (word, double word and quad word scaling is provided for), and the effect is to multiply the value in the register by 1, 2, 4 or 8 before using it. This allows the same value in the register to access sequences of any of these four integer (or floating point) types. It is vital to realize that the register is used as a *signed, 32-bit* value and you must make sure that all its 32 bits are correct. Scaled indexing is dealt with in more detail in Chapter 7.

In this simple case, the scaled index operand dec-1[R4:b] causes the value of R4, considered as a signed, 32-bit integer, to be added to the address dec-1. R4 starts with the value 10 and on the final pass through the loop it has the

value 1; the destination operand of the movb therefore goes through the values dec+9 to dec and the successive digits will be put into dec in reverse order as required, since the first digit is the least significant one of the equivalent decimal number.

The rest of this short program is quite simple. The double length dividend, made up from the value in bin (to be turned into decimal), is made into a quad word integer by extending it with a double word zero. Signed values would be dealt with by saving the sign and converting the absolute value (cf. ABSi).

The number is then divided by 10 and the remainder (in R2) is the next digit in the conversion. To make it into an ASCII character it just needs the value of the character 0 adding to it as the ASCII characters 0 to 9 are an ascending sequence. The final act is to transfer the quotient from R3 to R2 to make the next dividend, then extend it with zeros, and so on. The loop ends when ten digits have been converted.

EXERCISES

3.1 Given the following definitions:

```
a     dcd   1000
b     dcd    768
c     dcd    512
e     allocd   1
```

a) What happens when

```
movd    c, e
```

is executed?

b) Assuming that e has been set to 4 zero bytes, what value does it have after

```
movw    c, e
```

c) If e is again double (4-byte) zero, what value does it have after

```
movb    b, e
```

d) What value does e have after

```
movd    a, e
addd    b, e
```

e) What value does e have after

```
movqd   0, e
movw    a, e
mulw    c, e
```

Check your answer with a program.

f) What value does e have after

```
movqd   0, e
movd    b, e
divd    c, e
```


g) If R0 contains 1000 (as a double word) and R1 contains 800 (also as a double word), what do they contain after

```
deib    =250, R0
```

Check your result again against a small program.

3.2 The text gives an example of a multiple precision (MP) multiplication, using meid with the 'digits' of the MP number being represented by double words. The product of an MP number and a single precision (SP) number is simpler – as in long multiplication by hand, each digit of the MP number is multiplied by the SP digit with the carry from the preceding multiplication being added in. For instance:

$$\begin{array}{r} 768 \\ \times 9 \\ \hline \end{array}$$

Starting with the least significant digit, 9 × 8 is 72, put down 2 and carry 7, and so on. Using bytes to represent the digits, write *and test* a program using meib to multiply the 5-byte MP number (0, 1, 1, 1, 1) by 10 – the answer should, of course, be (0, 10, 10, 10, 10). To check that the carry is correct, multiply this result by the SP digit 100 – the product should be (3, 235, 235, 235, 232). Note that the sign must be handled separately as MEIi performs unsigned multiplication.

3.3 Division of an MP number by an SP number can be done using DEIi and the rules of long division. To start the process, a zero digit together with the first digit of the dividend is divided by the SP digit: the quotient gives the first digit of the result and the remainder (as the high-order digit) together with the next digit of the dividend (as the low-order digit) form the next two digits to be divided. The final remainder can be ignored – if the precision is considered sufficient – or the result can be rounded to even, as described in Chapter 5, Section 5.1.2.

3.4 A year is a leap year if it is divisible by 4 and *not* divisible by 100 or if it is divisible by 400. Thus, 1984 was a leap year but 1900 was not; the year 2000, however, will be. Write a program which takes the number in a word called Year and sets a byte called LeapYear to 1 if the Year is a leap year and 0 if it is not. Test the program with a selection of years to make sure that it returns the correct result in all cases.

3.5 The day of the week (DoW) for the 1st of January of a given year is calculated by:

$$\text{DoW} = (1 - d) \text{ MOD } 7$$

where

$$d = 7 - (\text{Year} + \text{Year}/4 - \text{Century} + \text{Century}/4 - 1 - \text{Leap}) \text{ MOD } 7$$

Century is 19 for the years from 1900 to 1999 and the results 1–7 correspond to the days Sunday–Saturday. Write and test a program to calculate the day of the week corresponding to the 1st of January for the year in the word called Year, using the leap year code from Exercise 3.4. Check your results against some calendars.

4 Boolean, shift and logical operations

4.1 LOGICAL OPERATIONS

The AND function is performed by

```
ANDi   src, dest
```

Each bit in dest corresponding to a 0 bit in src is set to 0, the bits corresponding to 1s in src remaining unchanged. The result is put into dest. As usual, there are three forms for byte, word and double word operands. The AND operation is generally used to mask off a part of memory unit. For instance, to get at bits 6 to 10 of a word you could use

```
D       dcw    #b1000110100111011

        movw   =#b11111000000, R0
        andw   D, R0
```

and the result, in R0, will be #b10100000000. As it is not much use stuck in the middle of a word, the next step would be to shift it 6 bit positions right, to bring its least significant bit (bit 6) into bit 0 of R0. Of course, this can be done more efficiently with the Extract and Insert instructions (which will be discussed in Chapter 6). However, ANDi does have a number of other uses. As a simple example, it can determine the modulus of numbers with respect to a power of two, to see whether a number is odd

```
        ...
        movw   number, R0
        andw   =1, R0
        cmpqw  0,R0
        beq    even
;number is odd
        ...
```

and, more adventurously, to see whether a number is divisible by 16


```
        ...
        movb   quorum, R5
        andb   =#b1111, R5
        cmpqb  0, R5
        beq    div16
;number is not divisible by 16
```

ANDi isolates the bottom 4 bits (which will be zero if the number is divisible by 16) so that CMPQi can test them against zero, if they are all zero then the number is divisible otherwise it is not.

Finally, it can be used to see if each of a selection is set or whether one or more are set. For instance, suppose you are scanning a market research database in which each entry has a selection of important attributes coded as bits in a word and you are looking for men aged 40–45 who are houseowners and read the *Boy's Own Paper*. Let's assume that the bits assigned to these attributes are: men – bit 0 set; age 40–45 – bit 5 set; houseowners – bit 10 set; and readers of the *Boy's Own Paper* – bit 15 set.

The word you are looking for, then, has the value :8421 (the other bits are set to 0 here but their actual value doesn't matter). So, to pick out the required entries, given that the word from the entry is in R0,

```
andw   =:8421, R0  ;select the bits
cmpw   =:8421, R0  ;check that each is 1
beq    found       ;jump if successful
```

The andw isolates the bits in question, setting all the others to zero, then cmpw tests the bits to see if each is set to 1 and, if they all are, the branch will be taken. If, on the other hand, entries with at least one of these attributes (rather than all) were being sought, the result after andw could be tested against zero; if it was non-zero, then at least one of the bits was set.

While AND is used to extract bits by setting unwanted bits to zero, OR is used to set specific bits to 1. The 32000 instruction is

```
ORi   src, dest
```

Each bit in dest corresponding to a 1 bit in src is set to 1 with the other bits remaining unchanged; the result is put into dest.

Page 43 in the previous chapter gives a piece of code to do a binary-to-decimal conversion and, as part of this, the remainder after division by 10 was converted into an ASCII digit character. The operation required was to convert the remainder (0 to 9) into the ASCII code for the digits 0 to 9, which is 48 to 57. The means chosen was to add 48. The usual way of doing this is to use an OR instruction as, in hexadecimal, the digits 0 to 9 have the values :30 to :39 and the instruction to use is

```
...

deid    =10,  R2    ;get BIN MOD 10
orb     ='0',  R2    ;make an ASCII digit

...
```

The deid has been put in to show where in the previous piece of code the change should be made. The immediate operand ='0' has the value :30 and has been deliberately chosen instead of the alternatives =48 or =:30 as it makes the purpose of the instruction clearer – always a good thing in assembler programming. It is possible to use addb but this implies that the byte is being used as an integer rather than a set of eight bits, making the code harder to understand. In another example, it is used to make a letter lower-case. In this example, it is assumed that the byte variable char contains the next byte to be made lower-case – if it is a letter between 'A' and 'Z'.

```
...
cmpb    char, ='A'
blt     NotALetter
cmpb    char, ='Z'
bgt     NotCaps
orb     =#b100000, char    ;now lower-case
...
```

Note that the second operand of CMPi can be immediate as it is not written to. The first four instructions check to see if the value of char lies between the ASCII codes for 'A' and 'Z' – if it doesn't, the conditional branches cause the lower-case function to be skipped. The instruction which does this relies on the fact that the ASCII code for 'A' is #b01000001 and the code for 'a' is #b01100001, the difference being #b00100000 – and this difference is the same for all the letters. Inserting this bit with an ORi will therefore convert any upper-case letter into a lower-case one.

It might be instructive to see the above coding changed to upper-case letters; the alterations are quite small. The tests must now check for letters in the range 'a' to 'z' and the orb becomes andb with the immediate bit string complemented:

```
...
cmpb    char, ='a'
blt     NotLower
cmpb    char, ='z'
bgt     NotALetter
orb     =#b11011111, char    ;now upper-case
...
```

The XOR function is a very interesting and slightly mysterious function, though this is not obvious from its truth table. It can, for instance, be used to

change one value into another and back again. Take the ASCII codes for the letters 'A' and 'Z' which are :41 and :5a respectively; if you XOR these together (again the Casio fx-570 comes into its own) you get :1b. Now, you will find that 'A' xor :1b is 'Z' (:41 xor :1b = :5a) and that 'Z' xor :1b is 'A'. This is a very useful way of toggling a variable between two values while using only an XOR with a constant operand – the XOR of the two values. Using this as an example,

```
AZ      dcb    'A'

        ...

        xorb   ='A' ^ 'Z' ; now toggled
```

The complement function simply inverts all the bits in the src operand and places the result in the dest operand:

```
COMi    src, dest
```

For instance,

```
hi      dcb    #b11110000

        ...

        comb   hi, R7
```

will end up with R7 containing #b00001111 in bits 0 to 7 – the other bits will be unaffected. Note that COMi inverts *all* the bits in the src operand; to invert only part of an integer, you should use the XORi instruction. If only bits 2 to 5 of hi were to be inverted this would be done by XORing hi with the mask #b111100 which has 1s in bits 2 to 5:

```
hi      dcb #b11110000

        ...

        movb   hi, R7    ;get src
        xorb   =#b111100, R7
```

which will leave #b11001100 in R7. The inversion takes place because every 1 in src which corresponds to a 1 in the mask will become 0 and every 0 in src corresponding to a 1 in the mask will become 1. The bit positions corresponding to 0s in the mask will remain as they are – mysterious? Definitely!

The last instruction in this group simply clears bits. This can be done by using ANDi with a mask of 1s in every position except those to be cleared, but this instruction is more direct. The Bit Clear instruction

```
BICi   src, dest
```

clears the bits in the dest operand which correspond to 1 bits in the src operand. For instance, to clear bits 2 to 5 in a byte operand

```
cl      dcb   #10110111

        ...

        bicb  =#b111100, cl
```

leaves the byte at cl as #b10000011.

4.2 BOOLEAN OPERATIONS

The logical operations described above operate on all the bits in the dest operand and some languages choose to recognize only the numbers 0 and 1 as truly boolean. Pascal insists that TRUE be greater than FALSE as well, which can be arranged with these values.

Using the numbers 0 and 1, AND, OR and XOR all deliver only 0 or 1 when applied to operands with either of these values. Only complement will not work and, for this reason, a boolean complement instruction called NOTi is provided

```
NOTi   src, dest
```

It inverts the least significant bit of the src operand, placing the result into dest; NOT 1 gives 0 (COM 1 would be −2) and NOT 0 gives 1 (COM 0 is −1).

The other boolean operation (or rather group of operations) is

```
Scondi   dest
```

which expands into

```
SEQi   dest   ;EQual
SNEi   dest   ;Not Equal
SCSi   dest   ;Carry Set
SCCi   dest   ;Carry Clear
SHIi   dest   ;HIgher
SLSi   dest   ;Lower or the Same
SGTi   dest   ;Greater Than
SLEi   dest   ;Less than or Equal
SFSi   dest   ;Flag Set
SFCi   dest   ;Flag Clear
SLOi   dest   ;LOwer
SHSi   dest   ;Higher or the Same
SLTi   dest   ;Less Than
SGEi   dest   ;Greater than or Equal
```


This converts the condition code which is set after a comparison instruction (EQ, NE, HI, LS, GT, LE, LO, HS, LT, GE) or an arithmetic operation (CS, CC, FS, FC) into a boolean value. If the condition is true, dest will be set to the integer 1; otherwise it will be set to 0. As for all the other integer instructions, there are byte, word or double word forms.

This instruction will be mainly used by compiler writers in the code to evaluate logical expressions. For instance, the code to evaluate

```
L = 5 .LE. I .AND. I .LT. 10
```

(I knew you'd recognize the language) could be

```
...
cmpqd  5, I
slew   L
cmpd   I, =10
sltw   R0
andw   R0, L
```

The first comparison sets the condition code and slew then uses it to set L true if 5 ⩽ I or false if not. The second comparison together with sltw puts the (boolean) result of I < 10 into R0. Finally, L is set to the desired result by ANDing R0 with it, the result being placed in L. The quaint variation in the sizes of logical and integer variables will be remembered with affection ...

4.3 SHIFT OPERATIONS

The high-level language user will not have come across shifts (unless enlightened and a C user). Shift instructions move a binary bit pattern up or down a word. The word is considered to have its most significant bit (bit 15) on the user's left and its least significant bit (bit 0) on the right; thus a right shift is equivalent to dividing the word (unsigned) by 2 and a left shift multiplies it by 2. To demonstrate: taking a word as the memory unit being shifted, let it have the value 160 which is #b10100000. If this is divided by 2 it becomes 80 which has the bit pattern #b01010000 – the same pattern but moved one bit to the right. While shifts can be used instead of division or multiplication when the divisor or the multiplier is a power of two, they can only give correct results on unsigned integers.

All the shift instructions on the 32000 have the same form. They have two operands, the first of which is the count of the number of binary positions the pattern is to be shifted and the second is the address of the pattern or the name of the general register containing it. Both operands are general operands but only the count can be an immediate operand and the count is always a byte integer, the i on the instruction only affecting the size of the destination integer. The second operand acts both as the source and the destination; it contains the pattern in its initial state and the pattern after shifting is placed

back there. The count is signed: if it is positive the shift is to the left, towards the most significant bit position, and if it is negative it is towards the right, the least significant position. A zero count is permitted and does nothing. The count has strict limits on it: if the destination is a byte integer it must be in the range −7 to 7; if a word is being shifted the count must be in the range −15 to 15; and for a double word the range is −31 to 31. If the count is outside this range, the result is undefined.

There are three different types of shift instruction, differing in the way they fill the emptied bit positions left when the original pattern is shifted. The first type is the Arithmetic shift, so called because, when the pattern is shifted to the right, the most significant bit positions are filled with the sign bit. This means that it can be used to divide even signed numbers by 2 (behaving like DIVi) as long as the number is not −1; shifting −1 to the right gives −1 again. In shifting patterns to the left the least significant bit positions are filled with zeros. This cannot be used as a multiply by 2 for signed numbers without a lot of pain, as the first zero shifted into the sign bit changes the sign of the number. Note that an arithmetic left shift behaves *exactly* like a logical left shift.

In either direction, the bits shifted down past the least significant position or up past the most significant position are lost.

This shift instruction is

```
ASHi   count, dest
```

Several examples of its use are

```
a       dcb   #b00110101
b       dcb   #b10001100
c       dcw   -334    ;#b1111111010110010
        ...

        ashb  = 1, a    ;leaves #b01101010
        ashb  =-2, b    ;leaves #b11100011
        ashw  =-8, c    ;leaves #b1111111111111110
```

The last example divides −334 by 256 and comes up with the answer −2. This is the same as the result obtained by dividing by 2 eight times in succession, rounding each inexact result to the more negative integer like DIVi – be careful.

The second type of shift instruction is the logical shift. This simply moves the bit pattern to the left or the right, losing 1s off one end and filling with zeros at the other. Its mnemonic is

```
LSHi   count, dest
```

Using the same examples as for ASHi to show up the differences between them,


```
a       dcb   #b00110101    ;53
b       dcb   #b10001100    ;140
c       dcw   -334          ;#b1111111010110010
        ...

        lshb  = 1, a        ;leaves #b01101010 = 106
        lshb  =-2, b        ;leaves #b00100011 = 35
        lshw  =-8, c        ;leaves #b0000000011111110
```

The result of the first example is the same as it was for arithmetic left shift – both ASHi and LSHi bring zeros into the least significant bit on a left shift. In the second example, the sign bit is set and while ASHi copies it into the vacated bits at the high end as it shifts, LSHi simply moves it right and brings in zeros. This is true of the third example as well.

Considering the integers as unsigned, you will see that the first shift multiplies a by 2 and the second divides b by 4. The third starts with an unsigned value of 65202 (word two's complement 65536 – 334) and ends up with 254 which is the truncated division of 65202 by 256 (2^8).

The last of the three shift instructions is ROTi, standing for rotate: it rotates the integer bit pattern to the left or the right with bits falling off one end being inserted into the resulting empty bit position at the other. This is used to present a portion of the pattern at either the top or the bottom end of the integer without destroying it. If you need to count the number of 1 bits in an integer without destroying the integer, you could write

```
        movd    int, R7         ;get the integer
        movqd   0, R6           ;bit count
        movb    =32, R5         ;no. of bits to count
count   cmpqd   0, R7           ;is the top bit set?
        blt     rotate          ;no, rotate left
        beq     finish          ;no more bits
        addqd   1, R6           ;yes, count one bit
rotate  rotd    =1, R7          ;move the next bit up
        acbb    -1, R5, count   ;continue loop
finish
```

The variables needed by the code (the count of the number of 1 bits, the number of bits in the integer to count and the bit pattern itself) are kept in registers but, if this is not convenient, they could be kept in memory instead, at the cost of slower execution. R7 contains the bit pattern to be tested. This is taken to be a double word though it could, of course, be a byte or a word instead. R6 contains the count of the number of 1 bits found and is set initially to zero. R5 contains the number of bits in the integer, 32 for a double word.

The cmpqd instruction together with the blt is used to test if the top bit is set. If it is, the register will appear to contain a negative integer and the blt will fail: note that the test here is whether 0 is less than R7 and that this will be true only if R7 is positive. If R7 is negative the addqd instruction will add one to the

bit count in R6, counting off the 1 bit now in the most significant bit position of the integer. In either case, the bit pattern in R7 will be rotated to the left, bringing the bit in position 30 into the sign bit position while the erstwhile sign bit goes into the vacated bit 0.

The loop will continue with the value in R5 being decremented by 1 before control is passed to the `cmpqd` instruction again; the loop will end when, after decrementing, the value in R5 is zero.

EXERCISES

4.1 In the following, use one of the instructions described so far to perform the required function to a **double word**. In each case the `src` operand will be a constant and can be written as an immediate operand in an appropriate radix. Only the bits mentioned are to be changed: all the other bits in the double word must remain unaffected.

a) Clear bits 7–9 of the double word.
b) Set bits 16–20 to 1s.
c) Invert the values of bits 20–25.
d) Invert only the 1 bits in the double word.
e) Clear bits 7, 15, 23 and 31.

4.2 Using one or more logical instructions, perform the following functions on a **word**:

a) Set bits 5–10 to the value `#b101101`.
b) Part (a) would normally be coded using two instructions, the first preparing bits 5–10 for the second instruction which sets them to the required values. Do Part (a) again with a different *first* instruction.
c) Do Part (a) a third time, this time with a different *second* instruction.

This sequence has been prepared assuming the conventional solution to Part (a) – an unconventional solution to (a) may mean that *both* instructions have to be changed for (b) and make (c), as written, impossible. There are three sets of instructions which may be used to perform this function and the exercise requires all three to be found – the sequence (a), (b), (c) is in the nature of a hint.

4.3 a) Using a shift and a logical instruction, put the contents of the (much abused) bits 5–10 of Exercise 4.2 into bits 0–5 of register 4 with bits 6–31 being all zeros.

b) Repeat Part (a) with bits 8–15.

4.4 Using logical instructions and a shift, put bits 0–3 of register 0 into bits 8–11 of a word – do not assume anything about the state of the other bits in the register and do not change any of the other bits in the word.

4.5 Using shifts and logical instructions (and an add), add 1 to the value in bits 8–11 of a word ignoring any carry out of the most significant bit. Do not alter any of the other bits in the word.

4.6 Rewrite Exercise 3.4 to use boolean operations to set the value of `LeapYear`.

5 The floating point unit

5.1 THE IEEE STANDARD

In 1977 the Intel standard for floating point arithmetic was published by John Palmer after consultation with Professor W. Kahan of the University of California at Berkeley. At the end of that year, the IEEE formed working group 754 to draft a standard for binary floating point arithmetic and, at Professor Kahan's suggestion, it adopted an expanded form of the Intel standard to work on.

Now, Draft 10.0 (dated 2 December 1982) is still awaiting official adoption by the IEEE!

In those years many people worked on the standard and several prepared alternative drafts but the draft resulting from the extraordinarily detailed and unremitting work of Professor Kahan's graduate student, Jerome T. Coonen, was eventually put forward as the result of the group's deliberations. Nearly a hundred people are cited in the draft standard as having contributed to it, coming from big brand-name companies like Hewlett-Packard, Apple Computer, Intel, DEC, Motorola, NatSemi (of course) and even IBM. It seems likely, however, and just, that this milestone in computer architecture will be linked to the names Kahan and Coonen.

The standard is a fascinating document and delving deeper into the papers published explaining its implementation and details leads one to the belief that all computers must be like this some day. At present, moving mathematical software (like the NAG library) from one machine to another is fraught with numerical danger – you are lucky enough if it works without detailed modification, don't even think of getting the same answers!

In the future, there will be no trouble at all in moving software from one IEEE standard machine to another – even the answers will be identical.

5.1.1 Floating point arithmetic

To understand the FPU properly – and if you don't know what's going on you're really only toying with it – you need some acquaintance with the operations which must be performed between accepting two floating point operands and returning a result.

To do this while avoiding large numbers, I shall use a toy floating point representation with a 4-bit exponent and a 6-bit fraction. This will be laid out like an IEEE standard system, except for the format, both because the standard system is the best there is and to make it easier to transfer ideas from the model to real life.

When doing arithmetic by hand, the floating point operands will be written down as

```
1.ffffff be
```

with an `f` representing each bit of the fraction, `b` (by analogy with the FORTRAN E) meaning '2 to the power of' and `e` being the number's signed power of two. The fraction, when it has the implicit `1.` in front of it, is called the significand.

The actual bit layout (compare the floating point formats in Chapter 2) is shown in Fig. 5.1. The sign bit `s` is the most significant bit but it is not in a two's complement format like the integers. Changing a number from positive to negative is done by changing the sign bit from 0 to 1; no other change is made. This is known as sign and magnitude format. The exponent is not signed but takes the values from 0 to 15, with 0 being the lowest exponent value and 15 the highest. With this format, two floating point numbers (with the same sign) can be compared like integers as, with the same exponent, the fractions increase from 0 to 63 and different exponents compare as unsigned numbers. To convert from this form of exponent to the true, signed form a bias value is subtracted: in the standard this is chosen as the exponent value `011...1` which in this model system is `0111` or 7. Thus the true exponent corresponding to `0111` is 0 and that corresponding to `0001` is −6.

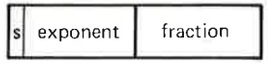

Fig. 5.1 Model floating point format.

Floating point significands are always *normalized*; that is, if the significand of any result does not lie between 1 and 2 (may equal 1, must be less than 2), it is multiplied or divided by 2, while altering the exponent appropriately, until it does. A normalized significand must therefore always start with a 1 bit. If so, why keep this 1 in the format? If it is understood to be there but is not, there is room for another bit at the end of the fraction – and so it is: the significand, without the leading 1, is called the fraction and is marked `frc` in Fig. 5.1.

Before any arithmetic operation can start, the exponent and fraction of the operands are split and put into separate internal registers; the fraction is then given the leading 1 bit and becomes the significand.

Floating point (hereinafter shortened to FP) addition and subtraction are pretty well the same operation, except that, for subtraction, the sign of the

source operand is changed.

The first example shows 7 (1.110000 b2) being subtracted from 64 (1.000000 b6):

```
 1.000000 b6
-1.110000 b2
```

Before adding, the binary points must be aligned, which is done by shifting the fraction with the smaller exponent to the right until the exponents are equal:

```
 1.000000 b6
-0.000111 b6
 0.111001 b6
```

The result must now be normalized giving

```
1.110010 b5
```

When multiplying, the significands are simply multiplied as they stand and the result exponent is obtained by adding the (true) operand exponents: multiplying 15 (1.111000 b3) by 3 (1.100000 b1):

```
 1.111000 b3
 1.100000 b1
10.1101000000000 b4
```

In this case normalization means a shift to the right with the exponent being increased from 4 to 5 giving

```
1.011010 b5
```

when the trailing zeros have been removed to reduce the fraction to the 6 bits allowed.

Addition (or subtraction) can result in a shift of one bit to the right (as above) or up to six bits to the left; multiplication will only ever require one right shift and not always even that.

Division is performed in a similar manner to multiplication except, of course, that the fractions are divided and the divisor's exponent is subtracted from the dividend's. The result will be greater than 1/2 and less than 2 so the fraction will need at most one left shift to normalize it.

There is a problem with floating point arithmetic caused by the fraction being a window into the true result showing only the top 6 bits. If two nearly equal numbers are subtracted, the result will be left with a number of leading zero bits reducing its accuracy. For instance, if 5.125 (1.010010 b3) is subtracted from 5.75 (1.011100 b3) you get


```
 1.011100 b3
-1.010010 b3
 0.001010 b3
```

and you can see that the original 7 significant bits have shrunk to only 4, as the original trailing bits of the exact representation of each operand have been lost and normalization will insert trailing zeros.

A similar effect comes from subtracting a small number from a large one: subtracting 0.078125 (1.010000 b-6, 5/64) from 1 (1.000000 b0) we get

```
 1.000000 b0
-1.010000 b-6
```

Aligning the binary points gives

```
 1.00000000 b6
-0.00000101 b6
 0.11111011 b6
```

and the result has one more bit than can be fitted into the format. This is the basis of a simple test to find out how many *effective* bits the fraction has. A number, initialized to 1, is repeatedly divided by 2 and subtracted from 1 until the result no longer differs from 1; that is, the single bit in the minuend has dropped below the last significant bit of the FP representation of 1. In our model system, the last subtractions would be

```
 1.000000 b0  (1.0)
-1.000000 b-6 (1/64)
```

After alignment, we have:

```
 1.000000 b0
-0.000001 b0
 0.111111 b0
```

This is followed by

```
 1.000000 b0  (1.0)
-1.000000 b-7 (1/128)
```

which after being aligned gives

```
 1.000000 0 b0
-0.000000 1 b0
 0.111111 1 b0
```

If this last subtraction returned the value 1, the effective length of the fraction would be 6; most FP systems, however, have a guard bit beyond the last true fraction bit and this subtraction could still leave a result less than one (depending on the rounding mode in use). A guard bit appears only in the FP registers; it disappears (by rounding) when the number is moved from the register into memory.

The IEEE standard implementation guide suggests *three* bits past the last official fraction bit: a guard bit, a rounding bit and a 'sticky' bit which shows whether the last true result had any 1s after the guard and rounding bits.

5.1.2 Rounding

If 15.5 (`1.111100 b3`) and 2.5 (`1.010000 b1`) are multiplied together we get

```
1.111100 b3
1.010000 b1
-----------
10.011011 000000 b4
```

This is the exact result of the multiplication – two places before the binary point and twelve after. Now, if this is normalized, the bit in the sixth binary place is moved into the seventh and is in imminent danger of getting lost. To get the best possible representation of the exact result in the given FP format (that is, the number in the FP format which is 'nearest' to the exact number), the significand is rounded to six binary places – and there are several ways of rounding. The IEEE default rounding mode is 'round-to-even' and, to illustrate this, the two FP numbers next to the exact result are

```
1.000101  b4 (lower)
1.0011011 b4 (exact)
1.000110  b4 (higher)
```

The rule for round-to-even says that, of the two numbers next to the exact result, the one with a least significant *zero* bit is to be chosen: this is `1.000110 b4` (19.5 decimal). The exact result differs from this by only half a bit in the least significant fraction place. As the result is not exact, the standard requires that an 'inexact result' flag be set, accessible to the programmer.

This is one of four rounding methods that the standard says must be available and the choice of this one as the default method is underlined by Richard Karpinski's story in 'Paranoia: a Floating Point Benchmark' from the February 1985 *Byte* (McGraw-Hill). It appears that the Vancouver Stock Exchange decided to provide a stock index. This started with a nominal value of 1000.00 and was up-dated after each transaction by recalculating it to 5 decimal places and then truncating the last two. After 22 months the index stood at around 520 though stock prices had been going up! There were about 2800 transactions per working day and all those lost fractions had produced a value that was badly out.

It may be thought that simple rounding would have been enough but Richard Karpinski points out in his article that if the digits 01 to 49 had been rounded down and those from 50 to 99 rounded up there would still be a consistent 1% error: 49 times in a random sample the value is rounded down but it is rounded up in 50. The answer to this is to round down from 01 to 49, round up from 51 to 99 and when a 50 comes up, round it so that the resulting last digit is even – beautifully simple, beautifully exact.

This is the standard's recommended default rounding method. The three other methods are rounding towards zero and rounding towards positive or negative infinity.

Round-to-zero is usually known as *truncation* as the excess bits are simply snipped off, the remaining bits being unchanged. If 4.5 were being rounded to an integer in this mode it would become 4 and −4.5 would become −4, the positive numbers going down towards zero and the negative coming up.

Rounding to either of the infinities means that, of the two FP numbers next to the exact result, the lower would always be chosen when rounding towards negative infinity and the higher when rounding to positive infinity. Rounding 4.5 towards positive infinity would give 5, rounding −4.5 would give −4: rounding towards negative infinity the results would be 4 and −5.

5.1.3 Special operands

The IEEE standard reserves the lowest and highest exponent values for special operands. The lowest (0), together with a zero fraction, is used to represent zero; with a non-zero fraction, it represents the denormalized numbers. The highest exponent, when all the exponent bits are 1, is used for positive and negative infinity (zero fraction) and the class 'Not-a-Number' (non-zero fraction).

Zero does not, at first sight, seem a suitable bedfellow for curiosities like infinities, denormalized numbers and NaNs, but it is not included in the FP set with the legal exponents and fractions. The smallest (least magnitude) FP number in our model is `1.000001 b-6` which has `000001` as a fraction and `01` as a (biassed) exponent. This number is $1.015625*2^{-6}$ or 0.01586914 – a far cry from zero. Even in a real FP system the smallest numbers are still, given the accuracy, noticeably different from zero. Even given that zero is represented by a zero exponent and zero fraction, the gap is still an embarrassment as the difference between the smallest and next smallest FP numbers is 1/64th of it, and sensitive routines can be upset when confronted by large steps. This consideration led to the introduction of *denormalized* numbers, represented by a zero exponent and a non-zero fraction. They are taken (in our model) to have the value

```
0.ffffff b-6
```

Any of the bits given as `f` may be zero, even the leading bits – they cannot be normalized as the exponent cannot be decreased. If they are considered as

an extension of the FP numbers below the heretofore smallest, the gap is reduced from about 1/64 to 1/4096, the same on either side of the smallest number. This allows a very small number to dip into the denormalized numbers at one point in a calculation and re-emerge later on with no more than a loss of precision – the jump from 1/64 to 0 represents a complete loss of precision!

The 32081 FPU does not handle denormalized numbers and so does not support the standard's gradual underflow: if they are used, they cause a 'reserved operand' exception and a number dropping below the smallest FP number will be set to zero by default.

5.1.4 Overflow

At the other end of the scale stands overflow. In the model, the multiplication of 45 (1.011010 b5) by 6 (1.100000 b2) gives

```
 1.011010 b5
 1.100000 b2
-----------
10.000011 100000 b7
```

which when normalized and rounded gives

```
1.000000 b8
```

However, the highest legal exponent is 7 and therefore this number is too large to be represented and constitutes a case of overflow. This is an exception in everybody's book and how exceptions are handled is up to the system designer.

In the past, operations such as divide by zero have been treated as an overflow, but in a standard conforming system they not only cause a distinct exception but the two infinities have been provided to give the system a suitable response, the appropriate one being chosen by the sign of the dividend.

There are other operations like taking the square root of a negative number, subtracting two infinities of the same sign, multiplying zero by infinity and dividing zero by zero for which there is no suitable result but the NaN. These show that the operation does not have a number as a result. Two classes of NaNs are described by the standard: signalling and non-signalling. Signalling NaNs cause an exception when used; non-signalling (or quiet) NaNs propagate through operations.

In the 32081 FPU both NaNs and infinities are reserved operands and will always cause an exception when an attempt is made to use them for arithmetic. They can be moved around without trouble though. The NaNs can still be useful; however, if uninitialized FP variables and arrays are set to a NaN, any attempt to use them will cause the program to fail and allow the programmer to correct it. Since any non-zero fraction can be used, it can be set to a value

which will show the variable it originated from (in case it has been moved around a bit) – a further aid to debugging.

5.1.5 Standard FP operations

As well as the usual four arithmetic operations, an FP system which conforms to the standard will have a remainder operation defined by

$$r = x \text{ REM } y$$

with

$$r = x - y*N$$

where N is the nearest integer to x/y. At first sight this seems a peculiar operation to choose for a standard, but an operation which provides r accurate to the fraction length in use is vital to the accuracy of the sin and cos routines. These must reduce their argument, perhaps many times larger than π, to the range

$$-\pi \leqslant f \leqslant \pi$$

using the transformation

$$f = x \text{ REM } \pi$$

where x is the argument.

A further unusual hardware operation required by the standard is square root: this is a minor variation of division and is used almost as frequently.

There must also be conversions from integer to floating point and rounding conversions of floating point to integer, the choice of rounding mode to be the same as for the arithmetic operations.

Finally, binary-to-decimal conversions must be provided, and the standard lays down that the conversion should be accurate enough for a decimal string to be converted to floating point and back again without change.

The 32081 has the arithmetic operations and the integer-to-FP, FP-to-integer conversions but does not have remainder, square root or binary-to-decimal conversions. Both the remainder operation and binary-to-decimal conversion need the extended formats, which add a minimum of an additional 8 bits to the single length fraction and 11 bits to the double length one, to be performed accurately, and the FPU does not support them.

5.2 ARITHMETIC INSTRUCTIONS

There are seven floating point arithmetic instructions: the four arithmetic operations together with negate, absolute and comparison. All of them come

in a version for single precision operands and for double precision and they all take two general class operands.

There are the obvious differences between these instructions and the integer ones. The register names are those of the floating point registers (F0, ..., F7 for short reals and the names of the even registers, F0, F2, F4 and F6 for long reals) and immediate values are either 4-byte single or 8-byte double values depending on the instruction. In other words, they have the same status and standing as any of the CPU instructions; no incantations, divining of entrails or casting of runes needed.

The four arithmetic instructions are

```
ADDf   src, dest
SUBf   src, dest
MULf   src, dest
DIVf   src, dest
```

The f at the end of the instruction is replaced by F if a single precision operation is required and by L for double precision. Of the two operands, the first is the source operand and the second the destination; in each case the result of the operation is placed in the destination. SUBf subtracts the source operand from the destination and DIVf divides the destination operand by the source.

The absolute value and negate instructions are

```
ABSf   src, dest
NEGf   src, dest
```

The absolute value and negate operate only on the sign bit of the source operand: with ABSf the destination operand receives the source operand with a zero sign bit (a positive value) and NEGf copies the source operand into the destination with an inverted sign bit.

The comparison instruction is

```
CMPf   src1, src2
```

It compares the first operand with the second, setting the CPU's PSR Z (zero) bit if they are equal or if they are zeros with opposite signs, and setting the N (negative) bit if the first operand is greater than the second. The L bit, used in integer comparisons to indicate the result if the integers are considered unsigned, is always cleared so that bhi will never branch, bls always will, and blo will have the same effect as bne and bhs as beq.

The other conditional branch instructions will behave in the same way as with an integer comparison.

5.3 MOVEMENT AND CONVERSION

There are seven instructions for moving and converting floating point values and integers and they fall into three groups. There is an instruction which just moves a floating point value from one place to another; the second group moves floating point values from one place to another while converting a short real to a long real or the other way around; and finally, there is a group which converts from integer to real and vice versa.

The plain movement of reals, preserving their length, is done by

```
MOVf   src, dest
```

where, as with the arithmetic instructions, the f is replaced by either F for single precision or L for double precision.

Lengthening or shortening reals is done by

```
MOVFL   src, dest
MOVLF   src, dest
```

where the tag FL indicates conversion from short (F) to long (L) and LF the reverse.

Converting from integer to real is done by

```
MOVif   src, dest
```

where the i may be either b for byte, w for word or d for double word and f is replaced by F or L depending whether the real is to be short or long. The first operand contains the integer and, if it is a register, it must of course be one of the general registers R0, ..., R7; the second operand is the destination of the converted integer and again, if a register, must be an appropriate floating point register – F0 to F7 for short, an even one for long.

The final three instructions in this set tackle the task of converting reals to integers and, as is clear from Pascal at least, there are several ways of skinning this particular cat. The instructions are:

```
ROUNDfi   src, dest
TRUNCfi   src, dest
FLOORfi   src, dest
```

ROUNDfi (f and i are replaced in the same way as in the MOVif instruction) rounds the real to the nearest integer – if the integers above and below are equally close, the even integer is chosen. TRUNCfi truncates the real, returning the integer closer to zero; that is, simply removing the fraction. FLOORfi converts the real to the largest integer *less than or equal* to it. ROUNDfi is the standard's real-to-integer conversion in round-to-even mode, TRUNCfi is the conversion in round-to-zero mode and FLOORfi is the conversion in round-to-

negative infinity mode; there is no conversion in round-to-positive infinity mode.

To show how they work they will be applied in turn to the numbers −2.3, −2.7, −4.5, −5.5, 2.3, 2.7, 4.5 and 5.5 (see Table 5.1). As can be seen, the effects of ROUND are foreseen fairly easily, as are those of FLOOR and TRUNC for positive values – they have the same effect. They differ, however, for negative numbers and do not provide immediately obvious results; if they are to be used, careful thought should be given to the result to forestall unwanted surprises.

Table 5.1 Converting reals to integers.

Source real	*Destination integer*		
	ROUND	TRUNC	FLOOR
−2.3	−2	−2	−3
−2.7	−3	−2	−3
−4.5	−4	−4	−5
−5.5	−6	−5	−6
2.3	2	2	2
2.7	3	2	2
4.5	4	4	4
5.5	6	5	5

5.4 AN EXAMPLE

The following example illustrates the use of the floating point instructions in a real piece of code. The code comes from *Numerical Methods, Software, and Analysis* by John Rice (McGraw-Hill Book Company, 1983) and is part of a program in FORTRAN he gives to show the effects of round-off in computation by calculating π in five different ways. This is the second method in which 6*ARCSIN(0.5) is calculated to give an approximation to π. The FORTRAN code is

```
      DATA PI / 3.14159 26535 89793 23846 /
C
C     2. TAYLOR'S SERIES FOR ARCSIN(0.5) TIMES 6.
C     TERM = NEXT TERM IN SERIES FOR X = 0.5
      SUM = 3.0
      TERM = SUM*0.25/6
      DO 20 K=2, 30
         SUM = SUM+TERM
         TERM = TERM*(2*K-1)**2*0.25/(2*K*(2*K+1))
         ERR  = SUM - PI
20    CONTINUE
```

In the assembler program, pi is defined as a double precision constant (all the floating point arithmetic is done in double precision) and the variables are

all kept in registers instead of memory. sum is in F0, the double length floating point register takes over F1 as well, term is in F2 and F4 is used to hold the floating point forms of the integer factors before they are incorporated in term. The code for this program can be improved but is written so as to parallel the FORTRAN code as an illustration. The first part of the program defines pi and calculates the initial values of sum and term:

```
pi      dcl     3.14159265358979324
        movl    =3.0, F0        ;initial sum
        movl    F0, F2          ;term = sum ...
        mull    =0.25, F2       ;... *0.25 ...
        divl    =6.0, F2        ;/ 6
```

In the next section of the program, the loop count is set up in R0. The loop runs from 2 to 30 and therefore is executed (30−2+1) times; the initial value of K (=2) is set up in R1 and the existing value of term is added into sum:

```
        movb    =30-2+1, R0     ;the loop count
        movqw   2, R1           ;initial K
loop    addl    F2, F0          ;sum = sum+term
```

Now the new value of term is calculated starting with (2K-1)**2 which is calculated in integer and then converted into double precision floating point. The result so far is then multiplied by 0.25 to get the numerator:

```
        movw    R1, R2          ;form 2K-1 ...
        addw    R2, R2          ;... in R2
        addqw   -1, R2
;square (2K-1) in integer
        mulw    R2, R2
;convert (2K-1)**2 to double precision
        movwl   R2, F4
;term = term * (2K-1)**2
        mull    F4, F2
;term = term * 0.25
        mull    =0.25, F2
```

Now the denominator is calculated with 2K being formed in R2 and, from this, 2K+1 in R3; they are multiplied, converted to double precision and term divided by the result:

```
        movw    R1, R2          ;K to R2
        addw    R2, R2          ;2K in R2 ...
```

```
            movw        R2, R3          ;... and in R3
            addqw       1, R3           ;2K+1 in R3
            mulw        R2, R3          ;2K(2K+1) in R3
            movwl       R3, F4          ;converted in F4
;term = term / (2K(2K-1))
            divl        F4, F2
```

Though the last instruction looks wrong (as if dividing F4 by F2) it is correct: the source operand is the divisor and the destination the dividend. All that remains now is to calculate the difference, increment K and decrement and check the loop count:

```
            movl        F0, F4          ;sum into F4
;Err = Sum - Pi
            subl        pi, F4
            addqw       1, R1           ;K = K+1
;continue the loop if the decremented
;loop count is not zero
            acbb        -1, R0, loop
```

No call or code to print the floating point values has been included as this will depend on the operating system or assembler library available.

5.5 THE FPU STATUS REGISTER

It may seem, at first glance, that the round-to-even mode makes the others superfluous: however, the modes rounding to the infinities can be used to implement interval arithmetic or, at the very least, a sensitive calculation could be made twice – once in round-to-positive infinity and then in round-to-negative infinity. The two results would then be the upper and lower bounds on the true result and would give very useful information on the error involved in the calculation.

The rounding modes are changed by setting the FPU's status register, the FSR, and this can only be done with the instructions

```
lfsr        src             ;put src into FSR
sfsr        det             ;put FSR into dest
```

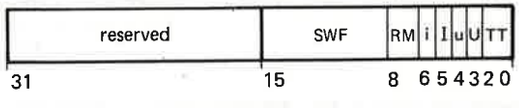

Fig. 5.2 The FSR.

The FSR is a double word with the fields shown in Fig. 5.2. The SWF field is not used by the FPU and is reserved for use by NatSemi software; the rounding mode field (RM) takes the following values

- 00 round-to-even (default)
- 01 round towards zero
- 10 round towards positive infinity
- 11 round towards negative infinity

The inexact result flag ('i', bit 6) is one of the standard's sticky bits. It is initialized to zero (as are all the other fields) by power on or a hardware reset and will be set to 1 when a result has too long a fraction after normalization and the bit left over contains a 1 bit; that is, the exact result of an operation lies in between two FP numbers. Once it is set, it remains set until reset by the programmer (using `sfsr` and `lfsr`) or a hardware reset – the sticky part.

The underflow flag ('u', bit 4) is another sticky bit: it is set when a result has a (biassed) exponent less than 1. If the underflow trap is not enabled the result will be set to zero and execution will continue.

The function of the other bits is concerned with traps and is dealt with in Chapter 10.

To set the rounding mode, you need the code

```
sfsr        R0                  ;get the FSR
inssb       =%10, R0, 7, 2      ;set rm to positive infinity
lfsr        R0                  ;return FSR
```

The `sfsr` and `lfsr` instructions copy the FSR into R0 and copy R0 into the FSR respectively. The `inssb` instruction (discussed in the next chapter) inserts the bit field 10 into bits 7 and 8 of R0. Similar code can be used to set new values in any of the fields.

To examine any of the FSR flags, say the underflow flag, the code needed is:

```
sfsr        R0                  ;get FSR
cbitb       =4, R0
lfsr        R0                  ;return FSR
bfs         uflow               ;branch if u/f
```

The `cbitb` instruction copies bit 4 (the underflow flag) into the PSR's F flag and then clears the bit for the next time. The `bfs` instruction branches to the code to deal with the underflow if bit 4 of the FSR was set.

EXERCISES

5.1 Both the first volume of Knuth's *The Art of Computer Programming* (Appendix B) and *The Handbook of Mathematical Functions* edited by

M. Abramowitz and A.I. Stegun (Washington, DC: U.S. Govt. Printing Office, 1964), have tables of mathematical constants in octal to high precision. Using the information given in this chapter on double precision floating point format, convert the values of `e` and `pi` given in octal below to 64-bit floating point numbers:

```
e  = 2.55760 52130 50535 51246
pi = 3.11037 55242 10264 30215
```

Check your results by putting the resulting 64-bit hexadecimal numbers into double precision reals (using a compliant language) and printing them in decimal.

5.2 A crude way of converting binary floating point numbers into their decimal representation is to convert them to a suitable range (say 10.0 to 0.000001) by multiplying or dividing them by an appropriate power of ten and then repeatedly multiplying them by ten, removing the leading digit and printing it. Write a program to do this assuming that the numbers are less than 10.0 and positive. The first digit is extracted by using `TRUNCfi` on the real and printing the integer; then, subtracting the integer, multiplying by ten and printing the next digit can be repeated to give the number of decimal places required.

5.3 Single precision floating point should give an accuracy of about 7 decimal digits. Using the program in Exercise 5.2 to print the digits, examine the effect on the result of the 4 rounding modes. These will only affect the multiplication by ten as the extraction of the leading digit is forced to be in round-to-zero mode.

5.4 A crude way of converting the decimal representation of a real number into binary floating point is to take each digit of the decimal, multiply it by an appropriate power of ten and accumulate it. Write a program to convert the decimal representations of `e` and `pi` below into double precision floating point and compare it with the results of Exercise 5.1. Why do they differ, which is the more accurate representation and what is the source of the error?

```
e  = 2.71828 18284 59045 23536
pi = 3.14159 26535 89793 23846
```

6 Bits and bit fields

6.1 INTRODUCTION

Modern high-level languages like Pascal and C allow the programmer to put information into spaces which are smaller than the basic machine storage unit or which cross unit boundaries. While this practice is, in general, open to question unless a machine-dependent object is being represented or space is to be conserved at the expense of time, it is of enormous help in handling bit-mapped graphic images where objects, and even text, must be placed at any position on the screen and it must be possible to examine any bit or sequence of bits.

On older (obsolete?) architectures, getting the value of a bit (in the general case) involved bringing the byte or word containing the bit into a register, shifting the bit to the least significant position and then performing an AND instruction to leave the bit in an otherwise empty register.

Thus, three instructions were usually needed to extract a bit and in addition, for a bit array, the compiler writer had to prepare code to calculate the address of the byte containing the indexed bit and the shift required.

For bit fields the position was similar but, as well as the address of the byte, the number of bytes covering the field had to be calculated too. Very often the position or length of the field had to be restricted otherwise the instruction sequence to extract it would be too complicated.

This was the situation when the day of the 32000 series dawned. In these CPUs both bits and bit fields are addressed by a simple base and offset and all the work is done in one instruction. The base can be thought of as the base address of an array and the offset as a count (either positive or negative) of the number of bits from the least significant bit of the byte addressed by base. What could be easier?

The base can be either the address of a byte in memory or the name of a register. If it is a register, there are two obvious restrictions: as the offset is taken from the least significant bit, a negative offset has no meaning and the maximum offset is 31.

Only the eight general registers can be the object of a bit or field instruction, though there are instructions to set and clear bits in the PSR which will be dealt with in Chapter 10. When the base is a memory address the sky is

truly the limit in the case of the offset. It may range from –2 147 483 648 to 2 147 483 647 *bits* corresponding to –268 435 456 to 268 435 455 *bytes*, at which the Instruction Set Reference Manual laconically observes 'this is considerably greater than the memory space currently implemented'.

As you would expect, the byte containing the bit described by base and offset is:

```
EA(base)+(offset DIV 8)
```

where the function `EA()` returns the effective address of **base** after all indexing and indirection is complete: though perfectly clear, a picture or two will not go amiss. In the first example, **base** is the name of a register, `R4`, and the offset is 19 – the bit pointed at is bit 19 of `R4` (Fig. 6.1).

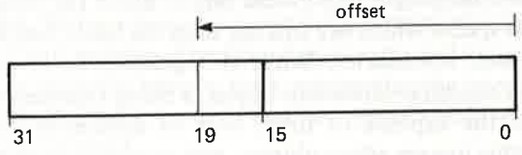

Fig. 6.1 A bit in a register.

For a register, the offset number is the same as the number of the bit pointed at. With bits in memory, things are a little more complicated. Before using this form of bit addressing it is wise to draw a picture or do a little calculation as one bit looks very much like another and it may not become obvious that you have got hold of the wrong one until it is too late. Figure 6.2 shows an example of a positive offset to a byte in memory; the byte address has been selected as `:1000` to ease the strain of the mental arithmetic involved and the offset is 25.

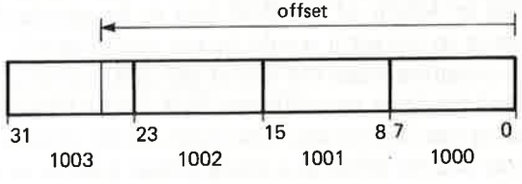

Fig. 6.2 A bit in memory (offset>0).

The offset is measured from bit 0 (the least significant) of the byte at address `:1000`. `25 DIV 8` gives 3 with 1 over so the offset is 3 bytes and one further bit. Adding 3 to `:1000` gives the address of the byte with the designated bit, `:1003`; the offset from bit 0 of this byte is 1, making bit 1 of the byte at `:1003`.

To appreciate the difference between positive and negative offsets, now take the offset –25 from the same byte at `:1000` (Fig. 6.3).

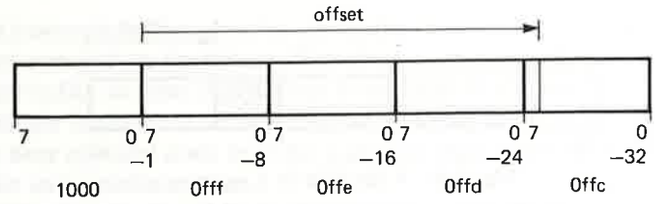

Fig. 6.3 A bit in memory (offset<0).

The starting point is still bit 0 of `:1000` and the bit 0s of the bytes preceding `:1000` are at offsets of –8, –16 and so on. The offset –25 breaks into –24 and –1 giving the bit addressed in the *fourth* byte before `:1000` and the top bit, bit 7.

Bit fields are designated by a bit address, as above, together with a length; the offset in the bit address may be negative but the length may not, as it is always taken in the direction of increasing bit number and increasing address. This, though eminently logical, does make bit fields addressed with a negative offset a mite disconcerting.

Like bits, bit fields may be applied to (general) registers as well as memory and, when applied to registers, there is the obvious restriction (in addition to the restrictions on bit addressing in registers) that the bits in the field must not fall outside the register; in other words, the sum of the offset and the length must not exceed 32.

When the bit field is in memory, the bit addressing is as generous as before but the field length may not exceed 32 bits. There is a further restriction on fields of more than 25 bits that they may not span more than three byte boundaries; that is, the field must be contained within four bytes. This restriction is due to the CPU accessing memory in double words. Figure 6.4 may help here – from which one may surmise that (offset `MOD 8`) + length must not exceed 32.

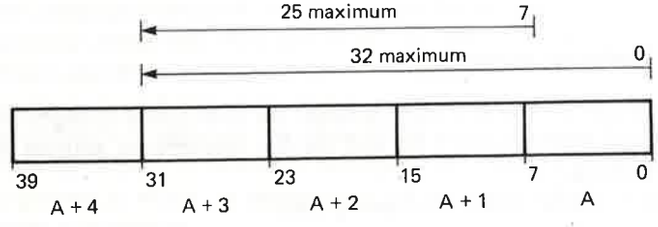

Fig. 6.4 The limits of a bit field in memory.

Some examples of bit field definitions in pictures may prove helpful when you are trying it out for yourself: first, a register (Fig. 6.5). The offset is 7 and the length is 5; the bit field described consists of bits 7 to 11 inclusive of the register.

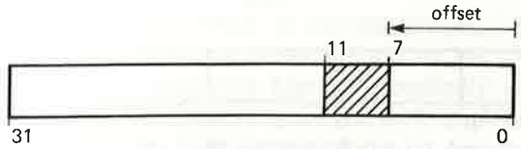

Fig. 6.5 A bit field in a register.

Now consider a field in memory with a positive offset. In Fig. 6.6, the offset is 19 from byte A which is bit 3 of the second byte up from A. The length of the field is 10 and therefore extends from bit 3 of A+2 to bit 4 of A+3 inclusive. When the offset is negative you find that the bit field extends back over the offset; if the offset is −19 with the same length you get Fig. 6.7. Here the 10 bits start at bit 5 of A-3 and go up to bit 6 of A-2; note that measuring from bit 0 of A, bit 0 of A-1 is offset −8, bit 0 of A-2 is offset −16 and so on.

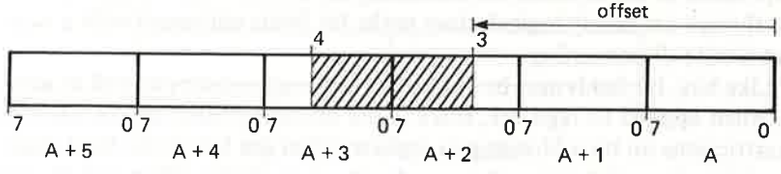

Fig. 6.6 A bit field in memory (offset> 0).

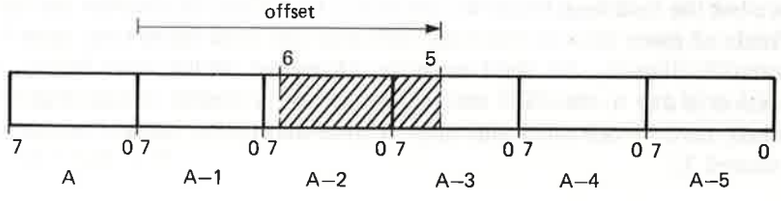

Fig. 6.7 A bit field in memory (offset< 0).

6.2 BIT INSTRUCTIONS

The bit instructions fall into three groups: the first group changes or tests the bit, the second searches for a bit and the last converts an address and offset into a bit address.

The instructions in the first group are:

```
TBITi     offset, base          test bit
SBITi     offset, base          set bit
SBITIi    offset, base          set bit interlocked
CBITi     offset, base          clear bit
CBITIi    offset, base          clear bit interlocked
IBITi     offset, base          invert bit
```


In all the instructions the i is to be replaced by b, w or d which gives the size of the offset operand: as the object affected by the instruction is a bit, the use of the integer suffix in these instructions is not used to give the object size. The offset integer is always taken to be signed and may be an immediate operand.

The base operand must be either a general register name or an address; it cannot be an immediate value.

Another peculiarity of these instructions is that there is only one object which takes both the parts played by source and destination operands in the instructions; that is, the set, clear and invert instructions change the operand. This will be familiar ground for readers coming from less sophisticated architectures, but almost all the instructions described so far have had a source operand which is read but not written to and a destination operand which is read and written.

All the instructions copy the bit given by base and offset into the F bit in the PSR where it can be tested by a bfs (branch if F set) or bfc (branch if F clear) instruction. The addressed bit is then modified as described in the list above; except, of course, for TBITi which leaves it unchanged. To illustrate this command, take the form

```
tbitb   R1, array
```

where array is a boolean array in its true form – one bit to an element. By making the operand integer a byte the offset may range from −128 to 127 and, avoiding unnecessary complication, the address array would be either the first or the last byte of the array. Assuming it is the first byte and R1 contains the array index (0 to 127), any of the boolean elements in the array may be examined.

It is often necessary to expand the boolean to word size in order to use it in a logical expression. This can be done by using the Scondi instruction described in Chapter 4: SFSi can be used to set a byte, word or double word to 1 if F is set and 0 otherwise, or, if the inverted value of the boolean is needed SFCi will set the integer operand to 0 if F is set and to 1 if it is clear.

The set, clear and invert bit instructions can be dealt with together as they differ only in the effect they have on the bit. The SBITi instruction sets the bit addressed to 1, the CBITi instruction sets it to 0 and the IBITi instruction leaves it the opposite to how it found it – 0 if it was a 1, 1 if it was 0.

These instructions also test the bit as TBITi does, by copying it into the F bit; this can be useful in ordinary programs but finds its greatest use in operating systems in which, as instructions are not interrupted, it can be used to implement semaphores.

The set bit and clear bit instructions have a second form which will be more fully discussed in Chapter 10 on operating system support. In this form they perform the same function as SBITi and CBITi but, while executing, they set the Interlock pin on the CPU. If there are two or more CPUs accessing the same memory, any other CPU will be stopped while this operation takes place, preventing the other CPU changing the bit after it has been read.

As an illustration of the use of the test, set and clear bit instructions here is a form of the Sieve of Eratosthenes which you will find in the answer to Exercise 8 of Section 4.5.4 in the second volume of Donald Knuth's *The Art of Computer Programming*.

The Sieve has achieved prominence recently due to two articles in *Byte*, the first in September 1981, and the second in January 1983. The first proposed using a particular implementation of the Sieve algorithm as a benchmark both for different machines and, on the same machine, different languages. The second article has the same algorithm written in nine languages from Forth to COBOL, taking in FORTRAN, C, Pascal and BASIC on the way.

The purpose of the Sieve is to find all the primes up to a given number N. In its simplest form this is done by writing out the integers from 2 to N and then striking out all the multiples of 2, then all the multiples of 3, and so on. At the end of this process the remaining numbers will be the primes.

This procedure can be simplified by removing all the even numbers to start with so that (in programming terms) the length of the array of numbers is halved; now the array element $X[j]$ represents the number $2j+1$ and, as all that is needed is to show whether a number has been struck out or not, X can be an array of bits.

The description of the algorithm is in four steps; the number M which appears in the description is the number of elements in the array X.

1. Set $X[k]$ to 1 for $k=1...M$. Set $j=1$, $k=1$, $p=3$ and $q=4$.
2. If $X[j]=0$, go to (4), otherwise output p (the next prime) and set $k=q$.
3. If $k \leqslant M$ set $X[k]=0$, $k=k+p$ and repeat this step.
4. Set $j=j+1$, $p=p+2$, $q=q+2p-2$. If $j \leqslant M$ go to (2).

The simplest way to set all X to 1 is to make sure it is contained in an integral number of double words and then set them all to -1, but that would be cheating and it also wouldn't show off the set bit instruction. The code is:

```
N       equ     16384           ;find all primes up to N
M       equ     N/2             ;number of elements in X
X       allocd  M>>5            ;32 bits in a double word

        movw    =M-1, R0        ;k running from M-1 to 1
setX    sbitw   R0, X           ;set X[k]
        acbw    -1, R0, setX    ;loop
        sbitw   =0,X            ;missed X[0], so do it

        movqd   0, R1           ;j = 0
        movqd   0, R2           ;k = 0
        movqd   3, R3           ;p = 3 (the second prime)
        movqd   3, R4           ;q = 3

testX   tbitw   R1, X           ;X[j]=0?
        bfc     loop            ;go on to next bit
```



```
;X[j] is set and so p is the next prime.
        movd    R4, R2          ;k = q
clearX  cmpd    R2, =M          ;k < M?
        bge     loop
        cbitw   R2, X           ;X[k] = 0
        addd    R3, R2          ;k = k+p
        br      clearX          ;repeat the loop

loop    addqd   1, R1           ;j = j+1
        addqd   2, R3           ;p = p+2
        addd    R3, R4          ;q = q+2p-2
        addd    R3, R4
        addqd   -2, R4
        cmpd    R1, =M          ;j < M?
        blt     testX
```

The code departs slightly from the statement of the algorithm by making `j`, `k` and `q` one less than the algorithm values. For `j` and `k` this allows their value to be used directly as the offset as this starts from 0 and, in the code, the index of `X` runs implicitly from 0 to `M-1` rather than the algorithm's 1 to `M`. Since `k` is set from `q` in step 2, `q` also had to be reduced by 1.

The second group of bit instructions mentioned consists of only one instruction:

```
FFSi base, offset   find first set bit
```

The meaning of `base` in this instruction is different from that in the test, set, clear and invert instructions. In those instructions it was an address to which the offset was added to point to a bit; in this instruction `base` is the address of the operand to be searched. To point up this difference, the `base` and `offset` operands are reversed with `base` coming first, in the others `offset` was first and `base` second. Another difference is that the offset length is fixed at byte and the `i` suffix applies to the base operand which is to be searched. The offset is an *unsigned* number and the range of values which it may take is limited by the number of bits in the base operand: for `ffsb` it must be in the range 0 to 7, for `ffsw` in the range 0 to 15 and for `ffsd` 0 to 31. `FFSi` searches for the first 1 bit in the base operand, starting at the bit given by the offset and continuing until it finds a 1 bit or reaches the last bit in the base object. If a 1 bit is found, the offset is changed to the offset at which the bit was encountered and, to show that it has been successful, the F bit in the PSR is cleared. If no 1 bit is encountered before the end of the base operand, the offset is set to zero and the F flag is set to 1 to distinguish the case where the least significant bit of the base operand is 1.

Note that if a further scan from the last point reached is required then the offset must be incremented otherwise the same bit will be discovered.

As an example, here is a piece of code to sum the number of 1 bits in a double word:

```
dw      dcd     :12345678       ;unlucky for some!

        movqb   0, R0           ;bit count starts at 0
        movqb   0, R1           ;offset starts at 0
bits    ffsd    dw, R1
        bfs     fin             ;no more bits
        addqb   1, R0           ;one more bit
        addqb   1, R1           ;offset advanced to next bit
        cmpb    =32, R1         ;last bit already checked?
        bgt     bits            ;no, R1 < 32
fin                             ;bit count in R0
```

The double word `dw` is presented as a dummy. In a real program it would probably be passed in a register or as a parameter. The register `R0` holds the count of the number of 1 bits found and `R1` is the current offset. The `FFSi` instruction searches for the next bit and if none is found the `bfs` instruction (branch on F set) takes execution out of the loop. This can happen immediately, in which case the count is zero, or later when the count will be left as it was on the last time through the loop. If the flag (F) is clear, a bit has been found, the count is incremented, the offset is incremented to pass over the bit just found and its value tested to see if it has now passed the end of the double word.

The third group also has one instruction, `cvtp` – convert to bit pointer. So far, bits have been addressed by the combination of a base address and an offset. (This you may call a relative bit address; if the base address is zero then the bit offset becomes an absolute bit address, giving the number of bits from the least significant bit in the whole memory space.) As the base is zero it can be ignored and you are left with a 32-bit quantity as the absolute bit address. This is the result of `cvtp`, the base address (the number of bytes from the start of memory) is converted into an absolute bit address by multiplying it by 8 and adding the offset.

This can be a considerable advantage in languages like Pascal and C which keep their pointers in double words: it allows them to introduce bit pointers in the same format as their present pointers.

In this instruction there are three operands:

```
cvtp    offset, base, dest
```

none of which may be immediate. `offset` *must* be a register; `base` and `dest` may be registers. As the `base` operand must be an address in memory, the use of register mode for it is taken to mean that the address is contained in the register.

6.3 BIT FIELD INSTRUCTIONS

There are two bit field instructions which each exist in a short and a long form; the instructions perform the load and store operations for bit fields. The bit field load instruction is called `extract` and the store instruction is called `insert`. Their mnemonics are:

EXTi	Extract Field
EXTSi	Extract Field Short
INSi	Insert Field
INSSi	Insert Field Short

Dealing with the short forms first, the instruction format is:

```
EXTSi   base, dest, offset, length
INSSi   src, base, offset, length
```

`base` is a general operand in both instructions but has the same restrictions as the base operands for the bit instructions. In effect, it must be an address so a register name is assumed to contain the address to be used and immediate values are not allowed.

`src` and `dest` are fully general operands except for the machine-wide restriction that operands which are written to (like `dest`) cannot be immediate; thou shalt not overwrite a constant. The integer suffix `i` applies to the `src` and `dest` operands only; these may be bytes, words or double words.

In the short instructions, `offset` and `length` are both coded into a single byte. `offset` gets 3 bits and `length` 5; the offset can therefore be any bit within the byte addressed by `base`. `length` is the usual 1 to 32 bits with the extra restriction on lengths over 25 bits.

As an example of the use of the short instructions here is a binary-to-hexadecimal conversion:

```
        hextbl dcb      '0123456789abcdef'
        binary dcd      :89abcdef
        string allocb   8

        movqd  4, R0                        ;four bytes to convert ...
        movqd  -8, R1                       ;... into 8 bytes
loop    extsd  binary-1[R0:b], R2, 4, 4     ;get hi nybble
        movb   hextbl[R2:b], string+8[R1:b] ;enter into string
        addqd  1, R1                        ;point to next byte in string
        extsd  binary-1[R0:b], R2, 0, 4     ;get lo nybble
        movb   hextbl[R2:b], string+8[R1:b] ;enter into string
        addqd  1, R1                        ;point to next byte in string
        acbd   -1, R0, loop                 ;go back for more
```

In this example the nybbles of a binary number `binary` are being converted into

hexadecimal bytes and put into `string`. As you may notice (among other things), the converted bytes are put into the string in the reverse order to the nybbles read out of the number; this is because the binary number is stored in memory as shown in Fig. 6.8, taking up 4 bytes. Each byte is divided into its component nybbles to show how the hexadecimal digits are ordered in memory.

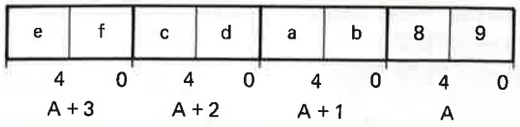

Fig. 6.8 A binary double word in memory.

After the code has run the string has to be as shown in Fig. 6.9, with the divisions now representing the bytes of the string. As you can see, the nybbles of each byte of `binary` have to be converted in the opposite order to the order of the bytes in memory: there must therefore be two extract commands in the loop.

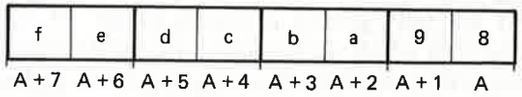

Fig. 6.9 A packed decimal double word in memory.

The addressing mode used to access the bytes of the binary number and the string bytes is called *scaled indexing* and will be dealt with in the next chapter as it is closely connected with arrays. At the moment it would be a help if you could just accept that the operand

```
binary-1[R0:b]
```

accesses the byte at `binary-1+R0`. `R0` starts at 4 and diminishes to 1 (the last time through the loop), so as `R0` goes from 4 down to 1 the byte addressed goes from `binary+3` down to `binary`.

Similarly,

```
string+8[R1:b]
```

addresses the byte at `string+8+R1`: as `R1` varies from −8 to −1 (the last time through the loop) the byte addressed varies from `string+0` up to `string+7`. Those of you familiar with arrays in BASIC or C will not be shaken by the idea of an array which starts at 0 as both these arrays, `binary` and `string`, do.

The use of an index running from 1 to 4 to access array elements in `binary[0]` to `binary[3]` may be a bit more unexpected, but is due to the

indexing arrangements depending on the instruction `ACBi`. This terminates a loop when the index is zero. All through this book you will find loop indices running from $-n$ up to -1 and from n down to 1 with the indexed operands being biassed to compensate.

In the above example, to get the actual byte of `binary` accessed to go from `binary+3` to `binary+0` while the index ran from 4 down to 1, the operand in the indexed instruction was `binary-1`. In the opposite direction `string+8` was used in combination with an index running from −8 to −1, accessing the bytes `string+0` up to `string+7`.

The extract instructions convert the extracted field to the integer size given by `i` (here `d`) by filling the rest of `dest` with zeros. In this program it is very important that the 4-bit nybble be converted to a 32-bit integer, as the scaled indexing mode always uses all 32 bits of the register. If the upper bits of `R2` were not cleared, the indexed operand could be accessing a byte in memory far away from the desired one.

If the extracted field is longer than the destination integer, the high-order bits of the field are discarded.

The short insert instruction can be illustrated by the reverse, converting a string of bytes representing a hexadecimal number into binary. The code layout is very similar to the previous one.

```
        binary  allocd  1               ;resulting binary double word
        string  dcb     '89abcdef'

                ...

        movqd   4, R0                   ;index to binary
        movqd   0, R1                   ;index to string
loop    movb    string[R1:b], R2        ;get string byte
        cmpb    R2, ='a'                ;byte ≥ 'a'?
        blt     nyb1                    ;no, make a digit
        subb    ='a'-:0a, R2            ;convert into :0a..:0f
nyb1    inssb   R2, binary-1[R0:b], 4, 4;into packed
        addqd   1, R1                   ;point to next byte in string
        movb    string[R1:b], R2        ;get string byte
        cmpb    R2, ='a'                ;byte ≥= 'a'?
        blt     nyb2                    ;no, make a digit
        subb    ='a'-:0a, R2            ;convert into :0a..:0f
nyb2    inssb   R2, binary-1[R0:b], 0, 4;into packed
        addqd   1, R1                   ;point to next byte in string
        acbd    -1, R0, loop            ;go back for more
```

The integer size `i` suffix on the `INSSi` instruction applies to the `src` operand, in this case a byte. In inserting the `src` field into the destination bit field (described by `base`, `offset` and `length`), the `src` operand is right justified in the field; that is, bit 0 of the `src` operand goes into the least significant bit of the field and, if the length of the `src` integer is greater than the bit field, only the

lowest `length` bits of the `src` integer are used. If the `src` integer is smaller than the length of the bit field, the upper part of the field will be filled with zero bits. In this case only the least significant 4 bits of the `src` byte will be put into the field, the remaining high-order bits being discarded.

The long forms of the extract and insert instructions differ from their short cousins in that the offset has a full register to itself instead of 3 bits, though `length` stays the same.

Effectively the difference is that, in the short instructions you must address down to the byte containing the first bit of the field, and in the long form of the instructions the full range of the offset may be used to address any byte in memory. Very probably, the long form will be used by compiler writers who can arrange more easily for the compiler to calculate the bit offset from a fixed base address than to calculate the address to the nearest byte. On the other hand, hand-programmers will probably use the short version as the address calculations will not be too complex and the offset will usually be fixed. To alert the programmer to the fact that a different form of the instruction is being used, the operands are in a different order:

```
EXTi        offset, base, dest, length
INSi        offset, src, base, length
```

`offset` must be in one of the general registers, `base` has the same restrictions as in the short instructions, as do `src` and `dest`, and `length` is again a constant. The hexadecimal-to-binary code is not suitable as an example of the long instructions, as these instructions draw their strength from the use of the offset register to proceed in increments of the bit field length through the object in memory. However, packed decimal numbers are ordered by increasing bit number and a double word packed decimal looks like Fig. 6.10.

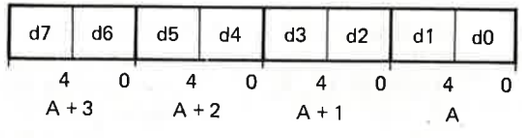

Fig. 6.10 A double word packed decimal number.

The subdivisions are nybbles with the first bit of the field, within each byte, marked in. Now it is possible to convert from packed decimal to decimal with the offset increasing from 0 to 28 in steps of 4. The desired string result is in exactly the same format, with the subdivisions marking bytes rather than nybbles. The code is:

```
packed    dcd     :12345678
string    allocb  8
```



```
          movqd   0, R0               ;initial offset
          movqd   -8, R1              ;string index
loop      extb    R0, packed, R2, 4   ;get next digit
          orb     ='0', R2            ;convert into '0'..'9'
          movb    R2, string+8[R1:b]  ;enter into string
          addqd   4, R0               ;point to next nybble in packed
          acbd    1, R1, loop         ;next byte in string
```

As you can see, this code is considerably simplified by being able to use the offset register, so having only one extract instruction. The use of a register for the offset has made no difference, in this case, to the number of registers used by the code; the nybble is again extended, by high-order zeros, to byte length.

To illustrate the insert instruction, the obvious choice is conversion of a string of bytes into a packed decimal:

```
packed    allocd  1                       ;8 digit packed decimal
string    dcb  '12345678'

          movqd     0, R0                 ;initial offset
          movqd     -8, R1                ;string index
loop      movb      string+8[R1:b], R2    ;get next string byte
          insb      R0, R2, packed, 4     ;put decimal into packed
          addqd     4, R0          ; point to next nybble in packed
          acbd      1, R1, loop           ;next byte in string
```

There is little new here; the `INSi` instruction, like its `INSSi` cousin, right justifies the `src` operand in the bit field, discarding any high-order bits if the `src` integer is longer than the bit field or zero filling the high-order bits in the bit field if the `src` integer is shorter. Here the `src` integer is a byte but only the low 4 bits are used; these will be put into the nybble in `packed` with the 4 high-order bits of `src` being discarded.

EXERCISES

A disk filing system keeps track of the allocation of sectors on the disk by means of a bit map. The disk has 1536 sectors of which the first 32 are used for the catalogue – the bit map must therefore contain 1504 bits or 47 double words. Sector numbering starts at zero and the first free sector is number 32. A sector is allocated when the corresponding bit in the map is 1 and is unallocated when it is 0.

6.1 Write and test a small program to leave the number of the first unallocated sector (if there is one) in R0 or zero if they are all in use.

6.2 Write and test another program which, given the number of an unallocated sector, marks it allocated.

6.3 To delete a file, the system simply marks the sectors it uses as unallocated again. The numbers of the sectors used by the file are held in one or more

'extents' of 16 words, the end of the list being marked by a zero sector number. Assuming that the file does not have more than 16 sectors, write a program to read the sector numbers from the extent and set the corresponding bit in the bit map to zero. Note that the description above implies that the program ends either when it encounters a zero sector number or has dealt with all 16 sectors.

6.4 Tests to see if a character is in a particular character class are widely used in programming: for instance, to find the end of an identifier the check may be whether the character is a letter or a digit. This can be very conveniently implemented as a test on a bit in a set of 128 or 256 bits depending on whether 7-bit or 8-bit ASCII is in use. Using one of the instructions from this chapter and an appropriately initialized bit string (128 bits, 4 double words), write the code to check whether a byte in register 0 is a letter or a digit, setting the F flag if it is either.

6.5 Try Exercise 4.2 using an instruction from this chapter rather than shifts and logical instructions.

6.6 Try Exercise 4.5 using an instruction from this chapter rather than shifts and logical instructions.

6.7 A packed array of integers in the range 0–31 is represented in memory as a sequence of 5-bit elements. Write a program which, given an index in the range 0–127 in register 1 and a value in the range 0–31 in register 0, inserts the value into the appropriate element of the array: the index corresponding to the first element of the array is 0.

6.8 Write a program which, for the same array as in Exercise 6.7, searches it for the first element with the value 15.

7 Arrays, records, stacks and strings

7.1 ARRAYS

There are two instructions for dealing with arrays and a number of instructions for string handling but, in the main, the high-level language data structures in the chapter title are handled by special addressing modes.

One of these addressing modes, the scaled index mode, appeared briefly in the last chapter. When a compiler has to translate an array reference like

```
int rand[12];

   ...

rand[i] = srand();
```

it needs to know three things: the value of the index; the value of the index for the first element of the array (here, in C, arrays always start at 0 but in Pascal or FORTRAN 77 they can start at any number, even negative); and the length of each element.

Later in this chapter the instructions for checking and calculating general array indices will be described but, for the moment, only the method of getting the address of the correct element in this simple case will be examined. The compiler has the address of the start of the array (the base address) and the index of the first element is 0, so to get the address of the first element adding the index value to the base address is enough. The index of the second element is 1 and, to get the address of the second element this must be multiplied by the length of the element – if this is a 32-bit integer, the appropriate multiplier is 4. In general, therefore, the address of the *i*th element is `base+4*i`.

Now the compiler is likely to have the number `i` in a register and it is inconvenient to have to shift it or multiply it as this may need another register. However, for the common element sizes, the 32000 series provides automatic scaling as part of an addressing mode. To illustrate it:

```
rand    allocd   12  ;the array declaration

        ...
```

```
        movd    i, R0           ;get the index
        movd    rand[R0:d], R1  ;element into R1
```

The scaled index mode is here applied to the array base address, though it can be a more complex addressing mode, and the value in R0 is scaled by 4 (4 bytes in a double word) before it is added to rand.

The general format of this addressing mode is

```
mode[Rn:i]
```

with i being replaced by b, w, d or q (in this case and in this case only, q is added to the letters which i may be replaced by; it stands for quadword, 8 bytes) and Rn is scaled by 1 for b, 2 for w, 4 for d and 8 for q. Mode may be any of the other addressing modes except immediate and another scaled index – you can't address multidimensional arrays with one cluster of scaled index modes.

The effect of

```
movd   rand[R0:d], R1
```

is to take the double word value in R0, multiply it by 4 (in a private place, the register remains unchanged) before adding to rand (Table 7.1).

Table 7.1 Effect of scaled index.

R0 *value*	*Effective address*
0	rand+0
1	rand+4
2	rand+8
...	
n	rand+4*n*

Note well: scaled index mode *always* treats the index register as a *double word*. It is usually best to use double word operations on such registers even though (if you are using small unsigned numbers as the index) word or perhaps byte operations are sufficient. A later change to the program (using a negative index or increasing the range of the index) may cause a working program to misfunction with subsequent loss of hair, sleep, etc.

You may sometimes have a collision of interests between the register, seen as an index, and the same register seen as a loop counter. The loop instruction in the 32000 series, ACBi, stops the loop when the register reaches zero (from either direction) while an index must usually be allowed to take the value zero: the solution is to apply a negative offset to the addressing mode before the scaled index.


```
for (i=0; i<8; i++)
    rand[i] = srand();
```

would be translated as

```
rand    allocd      8

        movd        =8, R0          ;loop counter and index
loop    cxp         srand           ;assume result in R1
        movd        R1, rand-4[R0:d]
        acbd        -1, R0, loop
```

The instruction cxp is a function call and its full explanation is to be found in the next chapter.

This formulation ensures that R1 is moved to rand-4+4 the last time round the loop which is as desired. You will notice that the elements of rand are filled in reverse order, starting with the last one. If this is undesirable for some reason you can code the translation:

```
rand    allocd      8

        movqd       =-8, R0         ;loop counter and index
loop    cxp         srand           ;assume result in R1
        movd        R1, rand+8*4[R0:d]
        acbd        1, R0, loop
```

In this example, when R0 has the value −8 the first time round the loop, the value in R1 is moved to the address rand+8*4-4*8, which is the first element of rand, and the last time round the loop it is rand+8*4-1*4 which is rand+7*4, the last element of rand. As an additional bonus, you can use movqd which is a shorter instruction than movd since the immediate value adds 4 bytes to the instruction length.

This is a special case, where the index starts at zero and where there is only one dimension. In the general case there can be several dimensions and, in Pascal and FORTRAN 77, the upper and lower bounds of arrays may be any numbers, positive or negative, as long as the lower bound does not exceed the upper one. The beauty of the 32000 series for compiler writers is the provision of two instructions which together do all the calculations needed to translate indices into an offset to the base array.

To explain these instructions, a little excursion into the underlying details of arrays may be welcome. Take as a first step the FORTRAN array:

```
DIMENSION A(5,7)
```

This in memory looks like:

		I				
		1	2	3	4	5
	1	0	1	2	3	4
	2	5	6	7	8	9
	3	10	11	12	13	14
J	4	15	16	17	18	19
	5	20	21	22	23	24
	6	25	26	27	28	29
	7	30	31	32	33	34

In FORTRAN the first index moves fastest as one goes sequentially through memory. The numbers inside the square give the offset (for the I of the column and the J of the row) of the byte `A(I,J)` from the first byte allocated to the matrix: the offsets are calculated by subtracting 1 from I to get the offset along the row and then adding 5 times `(J-1)` to it.

Generalizing this, the calculation required, given

```
DIMENSION B(M,N)
```

to get `B(I,J)` is

```
B+(J-1)*M+(I-1)
```

where `B` is the base address of the array.

Each of the indices I and J have been zero adjusted by subtracting the index value of the first element in that row from them: the second index, after zero adjusting, is multiplied by the length of a row of the matrix. In Pascal, this example would be

b: **array** [1..*N*, 1..*M*] **of** *real*;

as in this language arrays have their last index moving fastest as you go through memory. The indexed element chosen above would be `b[j,i]`.

In the general case, where the upper and lower bounds can be any integers, the index calculation is slightly more complicated and, to make it clear, it is helpful to introduce adjusted indices and dimension lengths. Adjusted indices (called zero adjusted above) are the indices with the lower bound of the dimension subtracted from them to make them start from zero (and positive). This transforms them from something 'user-friendly' into an offset – a real world object. To distinguish the adjusted form from the original

index, the index letter will be given an apostrophe, `i'`, and the length of the dimension corresponding to the index `i` (the difference of the upper and lower bounds) will be called `Di` – the actual dimension length is, of course `Di+1`.

With these conventions, the index calculation for the (Pascal) array

a: **array** [1..*u*, *m*..*v*, *n*..*w*] **of** *real*;

for the element `a[i,j,k]` with `Dj=v-m`, `Dk=w-n` (`Di`, the length of the first dimension, is not used) is

```
a + (i'*(Dj+1)+j')*(Dk+1)+k'
```

If another dimension was added to `a` making it

a: **array** [1..*u*, *m*..*v*, *n*..*w*, *p*..*q*] **of** *real*;

and the element indexed was `a[i,j,k,r]` the calculation would be extended to

```
a + ((i'*(Dj+1)+j')*(Dk+1)+k')*(Dr+1)+r'
```

(with `Dr` being `q-p`) and the way to extend it further is clear.

Now the 32000 instruction

```
CHECKi   dest, bounds, src
```

both checks that the index, say `i`, lies within the bounds `l` and `u` and subtracts the lower bound from it leaving the result in `dest`; this is the first operation required in the index calculation.

The instruction

```
INDEXi   accum, length, index
```

performs the calculation

```
accum*(length+1) + index
```

which, by comparing it with the index formula, is the remaining part.

The `CHECKi` instruction requires the bounds operand to point to two integers (size given by `i` in the usual way) *with the upper bound given first.* This is easy to get wrong, as the natural way of thinking of them is lower–upper. The `src` operand is the index to be checked and its size is given by `i`; the `dest` operand must be a register and is always a 32-bit value.

If the index does not lie in the given bounds, the F flag in the PSR is set and can be tested by the conditional branches `bfs` or `bfc`; otherwise it is cleared.

Both the index and length operands of INDEXi may be registers or in memory and their size is governed by i though they are zero extended to 32 bits before use; accum must be a register and is always 32 bits.

The index operand is expected to be zero adjusted; the dest operand of CHECKi suits it perfectly. The length operand must be the difference of the upper and lower bounds of the dimension, one less than the number of elements.

The way the instructions are combined can be shown by considering the calculation of the offset from the base of the array declared as

markov: **array** [4..13, −11..−5,6..7] **of** *real*;

and indexed by

markov [*i*, *j*, *k*] := [*F2*];

where the unPascal right-hand side is used as shorthand for an expression which is waiting in the FPU register F2.

First the bounds need to be set up for the three dimensions together with the length. As noted above, the actual value of length asked for is (upper bound − lower bound), another example of 32000 service. The upper bound is first, followed by the lower bound and the length:

```
b1      dcb     13, 4           ;upper bound then lower
b2      dcb     -5, -11, 6
b3      dcb     7, 6, 1
```

The length of the first dimension of the array is never needed, so has been omitted. Bytes have been chosen as the integer size for this example as the dimensions fall into the signed byte range. Now, just to make a complete example, here are the declarations of the three indices:

```
i       dcb     6               ;range 4..13
j       dcb     -10             ;range -11..-5
k       dcb     7               ;range 6..7
```

The index calculations take place in three stages: a single CHECKi for the first index (in FORTRAN this is the last index), which is the one moving slowest as you go through memory; two CHECKi/INDEXi pairs, one for each of the remaining dimensions; and an instruction with a scaled index operand which uses the calculated offset to access the array element.

```
checkb  R0, b1, i               ;check & adjust i
checkb  R1, b2, j               ;check & adjust j
```



```
indexb  R0, b2+2, R1            ;(i'*(Dj+1)+j')
checkb  R1, b3, k               ;check & adjust k
indexb  R0, b3+2, R1            ;()*(Dk+1)+k'-finished
movf    F2, markov[R0:d]
```

Very neat – very clear.

7.2 RECORDS AND STRUCTURES

In C they are called structures; in Pascal they are called records and they even have something in common with that venerable old FORTRAN institution, the COMMON area. In terms of assembly language (that means in real life) all these high-level language constructions, however new, exciting and state-of-the-art they may seem, consist of a base address and a series of fixed offsets from that base. For instance, the C structure

```
struct dir_entry {
    char name[8];
    char type[4];
    short extent, record, map[16];
} disc0;
```

consists of an area of 48 bytes with the first byte called **disc0**; the bytes of **name** have offsets 0 to 7 from **disc0**, the bytes of **type** have offsets from 8 to 11, **extent** has an offset of 12, **record** of 14, and the 16 short integers (taken to be words here) of **map** have offsets 16, 18, ...,47. The NatSemi assembler has the best directives for describing structures; **disc0** would be:

```
        .dsect
name    .blkb   8
type    .blkb   4
extent  .blkw   1
record  .blkw   1
map     .blkw   16
        .endseg
```

The Acorn assembler does not have equivalent directives to .dsect and .endseg; instead you would have to write:

```
;dir_entry offsets
name    equ     0               ;offsets 0..7
type    equ     8               ;offsets 8..11
extent  equ     12              ;offset 12
record  equ     14              ;offset 14
map     equ     16              ;offset 16..47
;entry length 48 bytes.
```

which has the same effect but doesn't allow you to, say, add another byte to `type` and have all the other offsets alter automatically (and correctly).

The advantage of these formulations is that if you have an array `directory` declared as

struct dir_entry directory[31]

then you can easily access the different items in each entry of `directory` by

```
addr    directory, R1 ;address to R1
...
movw    extent(R1), this_extent
...
movw    record(R1), this_record
...
;adjust R1 to point to the next entry
addd    =48, R1
```

To get `extent` into `this_extent` and `record` into `this_record` (`addr` puts the address of the first operand into the second operand, see Section 7.3 below) and to access the words in `map` in sequence:

```
        movd    =-16, R2
mapl    movw    map+2*16(R1)[R2:w], R0
;... do something with it ...
        acbd    1, R2, mapl     ;continue until finished
```

Accessing `COMMON` is similar except that the common area is external to the piece of code using it and though the principle of a base address and offset still holds, the base address is classed as external and is handled differently. This is discussed in the next chapter.

7.3 POINTERS

Another high-level language feature which is simple or even obvious in assembler is the pointer. Many and strange are the ways used by the writers of weighty tomes labelled {*A guide to*} /{*First Course in*}, {*Pascal*} /{*C*} (perm 2 from 4) to explain pointers where the assembler programmer accepts them as

those perfectly natural components of the real world, addresses, and their use as indirect addressing. An example has already been given in the section above, where the address of the first byte of an area has been put into a register and the parts of that area are accessed by using the register with a constant offset. Assembler programmers graduating from 8-bit CPUs (hi there, welcome to the future) will be used to indirect addressing as it was often the only way to get variable addressing.

The simplest form of indirect address has already been used throughout this book:

```
movd    rand, R1
```

Here, it is the contents of `rand` which is put into `R1` (and you would be very surprised if it wasn't), not the address of `rand`. The address is used to find the bytes in memory to put into `R1`; it is a pointer to the first of these bytes. Of course, the bytes themselves could be another address and to use this second address to get to the final bytes you would write:

```
movd    rand, R1    ;get the address
movd    0(R1), R2   ;get the contents
```

The first instruction moves the 4 bytes starting at the address `rand` into `R1`, then the second instruction uses these contents as an address to get the final sequence of 4 bytes into `R2`. Many books on high-level languages take pages to explain this.

It can also be written as one instruction:

```
movd    0(rand), R1
```

provided `rand` is addressed relative to the frame pointer, the stack pointer or the static base register: see Section 8.2 on memory relative addressing in the next chapter.

The `addr` instruction also has a part to play in this; its format is:

```
addr    src, dest
```

This instruction places the address of the `src` operand into `dest`. `dest` is always a double word and `src` can be any addressing mode other than immediate. If `src` is a register, the register is assumed to contain an address and the 32-bit contents of the register is transferred to `dest`.

The register relative addressing mode has almost exactly the same effect as the register mode. The instruction

```
addr    0(R2),dest
```

transfers the contents of R2 to dest exactly as

```
addr    R2, dest
```

does. However, register relative mode has a displacement which is added to the contents of the register to make the address, so

```
addr    5(R2), dest
```

will transfer the contents of R2 incremented by 5 to dest.

It is wise to avoid using register mode for the src operand in this instruction as it can be more than a little confusing to a reader, even yourself, after the passage of a little time.

The addr instruction can be seen as directly implementing the & operator in C. This returns the address of the object it is applied to and is used in initializing pointers. For instance, the piece of C code

```
char *cp, str[32];

cp = &str[5];
```

would be translated into

```
cp      dcd     0     ;pointer to char
str     allocb  32    ;string
        addr    str+5,cp
```

If, later in the program, you had

```
cp = &str[i];
```

this would be translated as

```
movd    i, R3          ;get i
addr    str[R3:b],cp ;store pointer
```

7.4 STACKS AND STACK ADDRESSING

The 32000 makes provision for two stacks, one for the user and one for the supervisor. The supervisor, which will be discussed in more depth in Chapter 10, is a service program with one essential property – whatever the user program gets up to, the supervisor will not crash. It is to help in this that the supervisor gets its own stack, which the user cannot access. Both these stacks are manipulated by the same addressing modes so, while all the talk in this

section will be about the user stack, the same methods can be used to deal with the supervisor stack – if you're so privileged.

The stack is managed by the stack pointer (SP) register; there are two, SP0 and SP1. SP0 is used when the User bit in the PSR is set to 0, showing that the CPU is running in supervisor mode. SP1 is used when the bit is 1 and it is in User mode. SP1 can be initialized by an LPRi instruction and its value can be obtained by SPRi. Normally, neither of these instructions will be needed as the operating system you are running under will initialize the stack pointer to what it considers a suitable value. The actual value of the stack pointer is only of interest if you want to make sure it isn't encroaching on your code or data – something that will again usually be done by library routines acting for the supervisor. Nevertheless, both these instructions and their effect on the system will be dealt with in Chapter 10.

The stack is used by procedure jump and entry instructions (covered in the next chapter) and it is also available to act as temporary memory during a calculation.

The stack grows downward in memory. Its first location is usually in high memory and creeps down towards the locations where your code is loaded (Fig. 7.1).

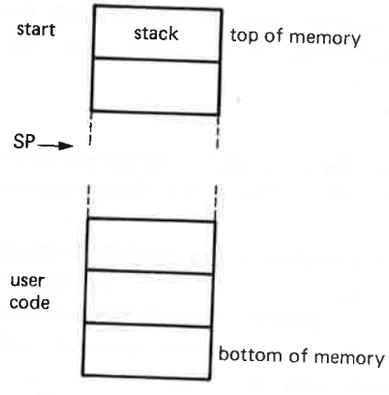

Fig. 7.1 Typical 32000 memory layout.

Negotiations with the stack are performed with the top-of-stack (TOS) addressing mode, the use of which could not be easier. To 'push' an item on to the stack you use

```
movd   src, TOS
```

where the movd could be replaced by either of the other two integer moves or either of the floating point moves. What happens is that the stack pointer is decremented by the length of the item deposited on the stack and then the object is moved there.

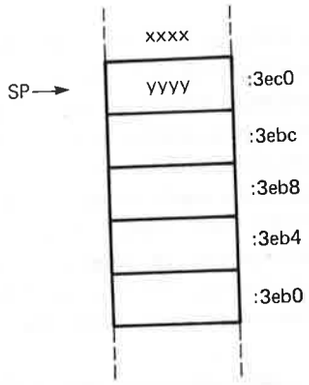

Fig. 7.2 The stack before a double word push.

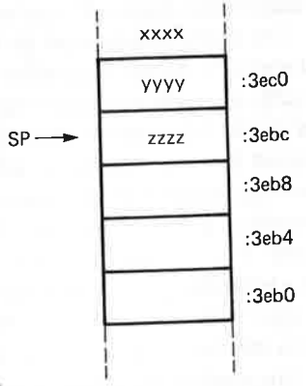

Fig. 7.3 The stack after a double word push.

If we assume that before the double word move the stack looks like Fig. 7.2, then, after `src` has been pushed on to the stack, it will look like Fig. 7.3, with `zzzz` the newly pushed quantity. Of course, if the integer is only a byte, the stack pointer will decrease by 1; if a word, it will decrease by 2; if a double precision floating point value, it will decrease by 8.

Popping items off the stack is as simple. The TOS addressing mode is used as the `src` operand and

```
movw   TOS, dest
```

will move a word off the stack into `dest` and increment the stack pointer by 2. It is entirely the responsibility of the user to keep track of the sizes of integers or reals on the stack.

Using the stack with arithmetic instructions is as easy, though there are differences between the use of TOS mode as `src` and as `dest`. Used as a `src` operand, the stack is simply popped, the item being taken off the stack and the stack pointer incremented. Used as a `dest` operand, however, no pushing or popping will be done as the value of the operand is read from the stack, used in the arithmetic operation and the result written back on top of the original value.

Even if both operands are in TOS mode there is still no fuss. First the `src` operand performs as described above, leaving the stack pointer pointing to the previous entry and then the `dest` operand reads this entry and writes the result back on top of it.

To find out exactly what happens in a particular instruction you must consider the *access classes* of the operands. The `src` operand above has a 'read' access class and for this class the stack is popped. The `dest` operand has a 'read – modify – write' access class and for this class no alteration to the stack pointer is made, the value being read, modified as the instruction sees fit and then written back to the same position it left no more than a couple of clocks since.

The `MOVi dest` operand has a 'write' access class and this, in combination with the TOS addressing mode, causes the value addressed by the `src` operand to be pushed on to the stack.

The access classes for each operand of each instruction are given in Appendix A. The sizes of the operands are also given in this appendix and show how much space on the stack is used.

Some instructions either do not have an `i` or `f` indicator associated with them or, if they do, also have an operand with a fixed length. `addr` always writes a double word to its `dest` operand, the count operand for shifts is always a byte and `movxbd` reads a byte and writes a double word.

There is, however, one instruction where you do have to be careful and that is the `MEIi` instruction – multiply extended integer. This instruction (see Chapter 3) multiplies two integers of size `i` together and puts the full double length result back in `dest`. If `dest` is on the stack, then before the instruction is initiated the space for the double result must have been allocated. It does *not* pop an `i` integer and then push a `2i` one.

7.5 BLOCKS AND STRINGS

Strings are sequences of integers (bytes, words or double words) with a length limited only by available memory. The instructions provided to handle them allow the general integer sequences to be moved, compared or scanned. These instructions have translating variants, but only for byte strings, using a 256-byte table to translate each byte of the string before it is operated on. Strings of bytes may thus be easily translated from one code to another with a move instruction, compared using a different collation sequence or scanned with a coded table to find the next one of a group of delimiters. However, while these instructions perform operations which on a more primitive CPU would require a complex loop with several instructions, there is a price to be paid. At least two and, for some forms, as many as five registers need to be set up before an instruction. For this reason simpler *block* instructions are provided in which the size of the strings is limited to 16 bytes (8 words or 4 double words) and no registers, other than any used by the operands, have to be set up. The block instructions also exist in only two varieties, move and compare, the move prototype being

```
MOVMi   block1, block2, length
```

where `block1` is the address of the first byte of the first block and `block2` of the second. The `length` operand has a range of 1 to 16 if the blocks are bytes (`i` is `b`), 1 to 8 for word blocks and 1 to 4 for double word blocks. As the blocks are moved in double word chunks, if `block2` overlaps `block1`, `block2` will be left with indeterminate contents. If you want to fill a section of memory with the same value or the same sequence of values, you will have to use a string move instead.

The compare instructions have a similar form

```
CMPMi   block1, block2, length
```

with the same interpretations of the operands and the same restrictions on the range of the `length` operand. The blocks are compared integer by integer in the same way as the `CMPi` instruction and the comparison continues until two unequal integers are encountered or the end of the block is reached. If all the integers are equal, the Z bit in the PSR will be set, otherwise it will be cleared; if two unequal integers are found, the N bit will be set for the result when the two integers are considered signed and the L bit will be set to show the unsigned result. The conditional branches used after this comparison are exactly the same ones as would be used after a `CMPi`: `bgt` will branch if the first unequal integer in `block1` is (signed) greater than the corresponding one in `block2` and so on for the other branches.

The string instructions may use some or all of registers `R0` to `R4`. All three types, move, compare and scan, use `R0` and `R1`; move and compare use `R2` as well; `R3` is used for the translating form of the instruction; and `R4` is needed if either the 'until match' or the 'while match' options are used.

`R0` contains the number of elements in the string and `R1` the address of the first element of the first string (source string for a move); scan (called 'skip string' `SKPSi`) operates on only one string but both move and compare expect the address of the first byte of the second string (destination string for move) in `R2`. All three registers `R0`, `R1` and `R2` are held to contain 32-bit quantities – make sure the length in `R0` fills the register.

All three registers are altered by the instructions: `R0` is left containing the number of string elements not used when the operation is completed; `R1` and `R2` are left with the address of the next string element to be worked over had the instruction not ended.

The three instruction types are

```
MOVSi
CMPSi
SKPSi
```

and take no operands (they are all in registers) but may have options.

The move instruction, without options, will simply move elements from the first string to the second until `R0` is zero; `R1` and `R2` will point to the byte following the end of each string.

The compare instruction continues until it finds two unequal string elements and then stops; it clears the Z bit (indicating that they are unequal) in the PSR and sets the other bits to show the result of the final comparison taken as between two unsigned integers and as between two signed ones: `R0` contains the number of elements not compared and `R1` and `R2` point, respectively, to the first unequal elements in strings 1 and 2. If all the elements compare equal, the

instruction stops when `R0` is zero and sets the Z bit: `R1` and `R2` point past the end of their respective strings.

The scan instruction (`SKPSi`) is only intended to be used with either the 'while match' or the 'until match' option – it does nothing useful otherwise. These options are selected by putting `[w]` for 'while' or `[u]` for 'until' in the instruction's operand field.

The compare and move instructions may also have either of these options selected

```
MOVSi     [w]          ;while option
CMPSi     [u]          ;until option
```

in which case their operation is modified by the presence or absence of a match between the current element in string 1 and the integer in `R4`. If the option is 'while', the operation continues only as long as the current element from string 1 is the same as the integer in `R4`. If the option is 'until', the operation stops as soon as a match is found without completing the compare or move.

Note that the NatSemi assembler does not require the brackets (`[]`) around the option letters – this is the Acorn format.

As the string operation can end either because all the elements have been looked at or because of a while or until match, the F bit is used to give a single quick way of distinguishing the cases: if the string ends on a while or until condition, the F bit is set; otherwise it is clear.

There is also a 'backwards' option which may be used either alone or together with until/while. Instead of going from the start of the string to the end, it starts at the end going towards the beginning. A variation of this may appear in an advanced version of the chip in which proceedings start simultaneously at both ends and work towards the middle.

However, dealing with the here and now, the difference when using the backwards option is that `R1` (and `R2` unless the operation is scan) must initially point to the last element in the string rather than the first, otherwise everything behaves in the same way as for the default forwards movement. To choose backwards movement, the letter `b` must appear (in brackets for the Acorn assembler) in the operand field; if it is combined with the while or the until option then the `b` must be separated from the `w` or `u` by a comma (both assemblers)

```
MOVSi     [b]          ;backwards
CMPSi     [b,w]        ;backwards and while match
SKPSi     [b,u]        ;backwards and until match
```

To illustrate the versatility of these instructions, they will be used to code two of the C standard string functions.

In C, strings are terminated by a zero byte (a null string will contain just this byte) and all C strings have a fixed maximum length which, for the purposes of this illustration, will be assumed to be given as the constant `MAXL`.

The function **index** searches the string s for the first occurrence of the character c: if the character is found the function returns its address; if the character is not found an address of 0 is returned which C interprets as the special pointer value NULL.

```
;first find terminator to get length to search
        movd    =MAXL, R0       ;max string length
        addr    s, R1           ;first byte
        movqb   0, R4           ;terminator
        skpsb   [u]             ;seek zero byte
        subd    =MAXL, R0       ;get actual string length
        negd    R0              ;no. of bytes to scan
;terminator found: search for c
        addr    s, R1           ;first byte
        movb    c, R4           ;byte to match
        skpsb   [u]             ;scan until
;what's happened?
        bfs     fin             ;found, pointer in R1
        movqd   0, R1           ;not found, null pointer
fin
```

There is a bit more than just letting SKPSi rip since, as it is looking for one character, it could easily slip past the string terminator without noticing (and perhaps scan through the rest of memory looking for it), so first the actual string length up to the terminator must be established by doing exactly the same scan but this time for the zero byte, limited by the declared length of the string. In a proper version of the routine there would be arrangements for bringing parameters in and passing results out but, in the absence of an established convention, this is left to the imagination.

The procedure **strcmp** is rather easier as the until option can be used at the same time as the comparison to stop it at the end of the shorter string; the procedure compares two strings (here called s1 and s2) and returns an integer (here assumed to be a 32-bit integer) with the value 0 if the strings are equal, −1 if s1 is less than s2 or 1 if s1 is greater than s2. The strings are considered equal if they are identical, the same length and the same sequence of bytes; s1 is considered less than s2 if, at the first differing byte, the byte in s1 is (unsigned) less than the corresponding byte in s2.

```
        movd    =MAXL, R0       ;max length of s1
        addr    s1, R1
        addr    s2, R2
        movqb   0, R4           ;terminator
        cmpsb   [u]
        bfs     ends            ;end of string
        shid    R0              ;1 if s1>s2 else 0
        addd    R0, R0          ;2 if s1>s2 else 0
        addqd   -1, R0          ;1 if s1>s2 else -1
        br      fin
ends    cmpqb   0, 0(R2)        ;equal if string 2 also zero
        sned    R0              ;0 if both ended else 1
        negd    R0, R0          ;0 if both ended else -1
fin
```


The number of characters to be searched can be set to MAXL since the cmpsb will stop either at the first differing byte or at the string terminator (match on the byte in R4).

If it ends on finding the terminator (which will be s1's terminator), the strings are equal if the byte in s2, pointed at by R2, is also a terminator; otherwise s1 must be shorter than s2 and is therefore less than s2.

If the cmpsb ends on a differing byte, the shid instruction, together with the two following instructions, will leave 1 in R0 if the byte in s1 was greater than that in s2 and −1 otherwise. Note that this includes the case where s2 is shorter than s1 as the byte in s1 (which cannot be zero) will compare high to s2's terminator.

EXERCISES

7.1 Exercise 6.7 accessed an array under the assumption that the index had already been checked and was in the correct range. Using instructions from this chapter, add checking code to that program for the index. If the index (instead of running from 0 to 127) ran from −64 to 63, what changes would have to be made to the checking code?

7.2 In Exercise 6.3, a program was written to delete a file by deallocating all its sectors: as presented there, a file could only have at most 16 sectors. The DFS is enhanced (by popular request) to allow more than 16 sectors to a file by adding another word to the sector list which points to a continuation of the list with another extent of identical format. The pointer on the last (or only) list is zero. Extend the program from Exercise 6.3 to delete files with sector lists in the new format. You may assume that the sector lists are in memory and that you are given the address of the pointer word for the first one in register 1: the pointers are all unsigned offsets to this address. The layout of each sector list is:

```
next    allocw  1       ;zero or offset
list    allocw  16
```

7.3 The business end of an integer desk calculator program contains six routines:

1. Push the double word in register 0 on to the stack.
2. Pop the double word at the top of the stack into register 0.
3. Add the top two double words on the stack together, leaving the result on the stack.
4. Subtract as in routine 3 above.
5. Multiply as in routine 3 above.
6. Divide as in routine 3 above.

The division routine should have a check for an attempted division by zero, leaving the maximum negative double word integer on the stack as an error return. The various routines are selected by a 32-bit value in the range 1–6 in register 1, the call to routine 1 assuming the presence of the value to be pushed in register 0.

Write a program to these specifications.

To test the program you've written, write a small loop to exercise it, taking the values as a sequence of double words with a count and result. For instance, to multiply 3 by 6 and check the answer, use

```
test    dcd    7, 1, 3, 1, 6, 5, 2, 18
```

where the first value is the count of double words (not including itself) followed by the four commands and the result (as the final double word). The push command 1 above is followed immediately by the value to be pushed.

7.4 Write and test a piece of code which returns the address of the first non-blank character in a buffer. The buffer is declared as:

```
buffer    allocb   1    ;no. of bytes
          allocb   80   ;buffer contents
```

The first byte is the count of bytes in the buffer and is in the range 0–80, the first byte in the buffer being found at `buffer+1`. The flag F should be set if a non-blank byte is encountered, otherwise it should be clear. If the buffer is empty F should be clear on exit.

7.5 In Exercise 6.4 a bit test was used to see whether a character was in a certain character class or not. This is often only part of the job as it is used in a loop to collect the identifier byte by byte. A more efficient means of doing this is available in the `skpst` instruction: with a translation table set up to contain 1s in the bytes corresponding to letters and digits and zeros elsewhere, it can be used in one form to skip over bytes which are neither letters or digits and then, in another form, to skip over letters and digits until it comes to the first byte after the identifier. With the address of the first byte of the identifier and the first byte which follows it, a `movsb` can be used to transfer it from the buffer to a place of its own (say `ident`, 32 bytes long).

Using the declaration of `buffer` in Exercise 7.4 above, write and test a piece of code to extract an identifier from it. What changes would be needed to cope with

a) an empty buffer,

b) a buffer which does not contain an identifier?

What changes would be needed if the code was to be used to extract not only the first but the following identifiers from the same buffer?

7.6 As the string instructions operate not only on bytes but on words and double words too, they can be used as efficient table lookups as well as the

more usual string searches. Assuming that register 4 contains a three letter abbreviation of a month name with the fourth byte zero, using a suitably initialized table and a `skpsd` instruction write a piece of code to convert the month name into a month number (1–12) setting F if it is successful and clearing it if not.

8 Jumps, procedures and modules

8.1 JUMPS

Jumps are the real-life counterpart of the much maligned GOTO in high-level languages (at least those languages which haven't dropped it in shame) and are essential to the well-being of assembler programs. Some jumps have already been mentioned in connection with comparison in Chapter 3 – these were the conditional branches together with the unconditional branch `br`.

Another branch instruction, `ACBi`, has been used from time to time in loops with a brief explanation. Now is the time for a full description.

The general format of the instruction is

```
ACBi    inc, index, dest
```

where `inc` is a quick integer – a small constant in the range −8 to 7, `index` is a general operand addressing an integer of length given by `i` in the mnemonic, and `dest` is a label in the program (it could be given as an offset from the instruction itself but this is unsafe programming).

In operation, `inc` is sign extended to the integer length and added to `index` with the sum replacing the previous value. If the sum is not zero, the program will be resumed at `dest`; otherwise the instruction following the `ACBi` will be executed.

It is used for controlling loops and, since assembler loops tend to have index registers addressing arrays, a subterfuge is needed to get at the first element (offset 0) as `ACBi` ends the loop on a zero index. This subterfuge will generally take two forms, the simplest being to add an offset to the array address to compensate for the index stopping short of zero; for instance,

```
string    allocb      80
byte      allocb      1

          movzbd      =80, R1
loop      movb        string-1[R1:b], byte

          ... do something ...

          acbd        -1, R1, loop
```


In this fragment of code, the bytes are taken from `string` in reverse order. If this is not convenient you can write

```
string    allocb      80
byte      allocb      1

          movxbd      =-80, R1
loop      movb        string+80[R1:b], byte

          ... do something ...

          acbd        1, R1, loop
```

Note how the `movzbd` has had to change to `movxbd` to sign extend the byte −80 to double word. `R1` must be a full double word integer because it is used in the scaled index operand of `movb`.

The compensating offset is calculated so that the value of the index plus the offset (which can, of course, be negative) is zero at the appropriate point in the loop. In the first loop, this point is the last time through when `R1` is 1 and 1+(−1) is zero as desired; in the second loop the calculation is done for the first time through when `R1` is −80.

Things get a little more complicated when word, double or quad word indexing is used – an example of double word indexing should be enough:

```
int       allocd      25
I         allocd      1

          movzbd      =25, R1
loop      movd        int-4[R1:d], I

          ... do something ...

          acbd        -1, R1, loop
```

which goes through `int` starting at the end and

```
int       allocd      25
I         allocd      1

          movxbd      =-25, R1
loop      movd        int+25*4[R1:d], I

          ... do something ...

          acbd        1, R1, loop
```

going forwards through `int`.

In each of the last two loops, R1 is multiplied by 4 before use so, in the last time through the first loop, R1 will be 1 which will be converted to 4 before being added to the address int-4. In the first time through the second loop, R1 will be -25 scaling to -25*4 which when added to int+25*4 gives int+0 as required.

In choosing the mnemonics for the different jump instructions, NatSemi chose the name 'branch' for those that took their destination as a distance relative to the current address in the program counter (the first byte of the instruction) and gave the name 'jump' to those instructions that used a more general address.

The branch instructions add their operand value (a constant which can be either positive or negative) to the address in the PC which causes execution to continue at the new address. Normally the operand will be a label as the length of 32000 instructions is not at all easy to calculate, but it is possible (though foolhardy in the extreme) to use the PC symbol, $, to give the offset directly

```
br   $+3
nop
nop
nop
```

In this artificial example the br instruction takes two bytes, one for br and one for the operand; the expression $+3 adds 3 to the PC, starting execution again at the second nop. If you were to replace the first nop by a simple, no-nonsense, plain, common or garden old movb you could find the second nop moved away from the br by anything from 2 to a theoretical 18 bytes, depending on the addressing mode of the two operands: you could easily arrange for the branch to land in the middle of an instruction – a fate so easily avoided by using a label instead.

The branch instructions will be more generally used in assembler programs as jumps are most useful in getting exotic effects. There are two instructions with the title 'jump'. One is jump, which causes execution to continue at the address given by its operand and, as this is a general operand, the address may be in a register or on the stack or it may be reached indirectly through an address in one of these places. One example of the use of this instruction is in implementing an alternative return from a FORTRAN subroutine where the address is passed as a parameter. It would also be used to jump into a ROM where the code is at a fixed physical address, though it is more likely that the second jump, jsr jump to subroutine (see next section), would be used instead.

There is also a multi-way branch which acts like the computed GOTO in FORTRAN; it is called

```
CASEi   src
```

The operand is an integer (with its size given in the usual manner by i), which is added to the address in the program counter (the address of the first byte of

the instruction) to give the point at which execution is to continue. As in the case of the computed GOTO, the instruction has a table of address offsets, one of which is picked by the operand which will usually be in scaled index mode. An example will help:

```
one     ;a label
        ...
two     ;another label
        ...
three   ;a third label
        ...
        movd    i, R3           ;set jump index
case1   casew   addr1[R3:w]
addr1   dcw     one-case1
        dcw     two-case1
        dcw     three-case1
```

The labels one, two and three are to be understood as being scattered through the text preceding case; immediately before case there is an instruction which sets R3 to one of the values 0, 1 or 2, depending on whether the branch is to be to one, two or three. The scaled index mode operand of case multiplies R3 by 2 before adding the result to addr to get the entry in the table; the word entry is then extended to 32 bits internally before being added to the PC.

The case instruction has a label on it so that the offsets in the table give the distance from the first byte of case to the label; as execution will never reach the byte following case (unless by accident – a fatal one), the table can follow immediately. Note that all the offsets will be negative and that they will be sign extended to 32 bits before being added to the PC. The offsets can be bytes if the distances are short enough, or double words; it just depends on the way i is replaced. If you are using the scaled index mode, though, the scaling must agree with the integer size.

8.2 PROCEDURES

There are two types of procedures, internal and external: in this section only internal procedures will be dealt with as external procedures are closely tied up with modules and belong in the next section.

Procedures are familiar from high-level languages where they may be called subroutines or functions instead. A great deal is made of the difference between procedures and functions, but from an assembler point of view they are the same thing: a piece of code entered with a special instruction, arguments passed in by one means or another, results passed out perhaps and then another special instruction to continue with the calling program. The difference between a procedure here and another there is not whether results are made available to the calling program but how arguments and parameters are passed.

This is not the place to catalogue the methods used in antiquity, but the means provided by the 32000 series for the task: the stack. The stack is a perfect method for keeping track of procedures. It provides an apparently infinite expanse of memory so there is no need to calculate a size for the largest parameter list, and it is, again apparently, capable of holding the information for any number of routines calling routines calling routines ...

There are two instructions to call internal procedures and they are distinguished in the same manner as `jump` and `br`; `jsr` takes a general operand as its address to jump to and `bsr` takes an offset from the program counter. `bsr` would therefore be used to call procedures directly by name, whereas `jsr` would be used if the procedure was being called indirectly. In C terms, `jsr` would be used to call a procedure via a pointer or even a pointer to a pointer to a function returning... Apart from this they both behave in the same way, they jump to the address given by the operand while pushing the address of the next instruction after `bsr/jsr` on to the stack. Internal procedures, whichever means is used to call them, end with the same return instruction, `ret`; this removes the return address from the stack and puts it into the PC, returning execution to the instruction following the call.

This is just the bare mechanics of call and return. There is also the matter of passing parameters to the called routine and returning results. This, by 32000 convention, uses the stack and `MOVi` and `MOVf` with TOS addressing mode for the destination to put them there. As this takes place before the call and the return address is pushed on to the stack at the call, they are to be found at a positive displacement from the stack pointer and what is called memory space addressing mode must be used to fetch them. For illustration, assume that a procedure needs three parameters passed to it; one double word integer, an address and a single precision real. The calling sequence would go a little like this:

```
movd    int, TOS      ;int to stack
addr    struct, TOS   ;addr to stack
movf    x, TOS        ;real to stack
bsr     proc          ;now call routine
```

On entry into `proc` (if the value of the stack pointer had been `:0f010` before the `movd`) the stack would look like Fig. 8.1. The integer double word occupies the bytes `:0f00f` to `:0f00c`, the address the bytes `:0f00b` to `:0f008`, the real `:0f007` to `:0f004` and finally the return address is in `:0f003` to `:0f000`.

In all these cases the least significant byte of the object is, as usual, in the lowest addressed byte of the four it occupies and, when it is pushed on to the stack, the stack pointer is decremented to point to this byte.

Before the first stack instruction in the example above, the stack pointer had the value `:0f010` and was pointing at the least significant byte of the object that had last been pushed on to it; pushing the `int` caused the stack pointer to be decremented by 4 and the `int` to be written to that address. After the call, SP has the value `:0f000` in it.

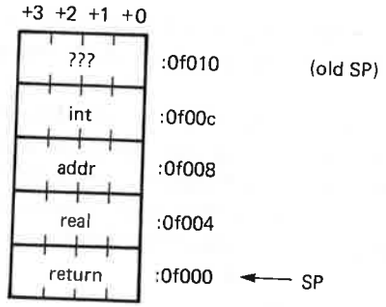

Fig. 8.1 Procedure arguments on the stack on entry.

The procedure would, quite reasonably in my view, like to get at its arguments and here there is a difference in the way the Acorn assembler does it from the way the NatSemi assembler allows. The NatSemi assembler allows symbols to be given the addresses of objects on the stack and thus symbols could be defined as:

```
int       .equ      x'0c(SP)      ;SP + offset x'0c
struct    .equ      x'08(SP)      ;SP + offset 8
real      .equ      x'04(SP)      ;SP + offset 4
```

so you could use them in instructions like

```
movd    int, R0
```

and

```
movf    real, F0
```

The Acorn assembler will allow only absolute, SB-relative or PC-relative symbols to be defined and, if you wanted to use symbols to make it easier to alter the program, you would have to write

```
int       equ       :0c           ;offset from SP understood
struct    equ       :08
real      equ       :04
```

with a corresponding change in the instructions

```
movd      int(SP), R0
```

and

```
movf      real(SP), F0
```

The addressing mode allowing us to write `disp(SP)` is called the memory space mode and can be used with the PC as well as the stack pointer. There are

two other dedicated registers it can also be used with, the static base register and the frame pointer register. Using these two registers will be dealt with shortly.

The address parameter called `struct` is intended to be the address of a structure (or record) and there is a truly amazing addressing mode which allows the programmer to get at the fields of the structure without the necessity of putting its address into a register first: it is called *memory relative*. For the purposes of illustration, let us assume that the `struct` whose address is the second parameter has the following fields:

```
a       dcl     1.2
i       dcw     3
d       dcd     4
s       dcb     'A string'
```

The long real `a` is the first object in the structure and has offset 0, the word `i` has an offset of 8, the double word `d` has an offset of `:0a` and the first byte of the string has an offset from the beginning of the structure of `:0e`. Using the memory relative addressing mode in the procedure you can write

```
movd    :0a(8(SP)), R3
```

to move the contents of `d` in the structure pointed at by the second parameter to the register `R3` – in one addressing operation. Disentangling it you get `8(SP)` as the address of the pointer to the structure and `:0a(8(SP))` takes the pointer, adds the displacement `:0a` to it and ends up with the address of the first byte of the double word `d`; this could be done with older architectures but it required setting indirect bits all over the place, which removed a bit from the addressing range and required whoever sent the parameter to be sure and set its indirect bit – a fruitful source of error. Here it is all done in the instruction wanting to access the object and much more under control.

Now that a way of getting at parameters is available, one of the next concerns is the use of registers. With so many registers, it soon becomes a habitual part of the 32000 programming style to keep things in registers as far as possible. When this is extended to procedures, an element of caution enters on to the scene as one of the registers used by the procedure may contain something the calling procedure would rather not have spoilt. This problem is solved by two further instructions, one to save registers on the stack and the other to restore them. There is some controversy about the best way to do this. One faction holds that to save all the registers a procedure uses on entry to the procedure is wasteful as some of the registers may not be in use by the calling procedure and therefore need not be saved. In this line of thought each procedure should, just before calling another procedure, save any registers it feels necessary. The other opinion thinks that the saving is slight and the additional convenience (to the caller) of being confident that none of the registers will be altered by the called procedure is more important. NatSemi

have not taken sides in this argument, they have simply provided two separate sets of register saving instructions, one for use by a calling procedure and the other for use on entry to a procedure – the operating system can plump for one or other convention.

The instructions just to save and restore registers are, as you have come to expect, sensibly named. Another company not only chose mnemonics which nobody but a memory freak could remember but also copyrighted them! Excellent idea – somebody else might have decided that they were some kind of industry standard instead of choosing the more obvious, and memorable, ones. However, the 32000 instructions are called

```
save    reglist
```

and

```
restore reglist
```

`reglist` is a list of one or more general register names, separated by commas and enclosed by brackets; to save registers `R1`, `R3` and `R5` you would write

```
save    [R1, R3, R5]
```

and to restore them later

```
restore [R1, R3, R5]
```

They are saved on the stack and, if the stack pointer before the save is `:1fd00`, after the `save` instruction the stack looks like Fig. 8.2 and the stack pointer is left with the value `:1fcf4`.

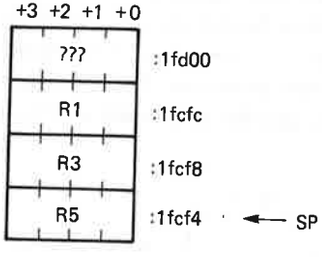

Fig. 8.2 The stack after `save [R1, R3, R5]`.

The registers are saved starting with the lowest numbered one; that is, after the save, the stack pointer contains the address of the most significant byte of the double word containing the highest numbered register.

The other pair of register savers goes rather further than this as it includes some of the other things procedures can usefully do on entry. If an error

occurs while a procedure is executing, it is very helpful to be able to print out the chain of procedure calls and each procedure's variables from the point of error. To do this, the part of the stack used by a particular procedure (a stack frame) needs to contain the address of the start of the stack space used by the procedure that called it – and so on up to the main program. Each part of the stack used by a procedure is called a stack frame and the NS32000 architecture provides a frame pointer which is set up by the procedure entry instruction (and cleared by the procedure exit) to offer just such a service. These instructions are called

```
enter    reglist, constant
```

and

```
exit     reglist
```

with `reglist` given as for `save` and `restore`. (The purpose of `constant` will be unveiled later.)

If we assume that the stack pointer has the value `:1d080` and the frame pointer has the value `:1d100` then, after executing the instruction

```
enter    [R2, R4], 0
```

the stack will have the appearance of Fig. 8.3. The old value of the frame pointer has been pushed on to the stack and the frame pointer now points to this stack entry; the two registers `R2` and `R4` have also been pushed on to the stack (in order from `R0` to `R7`) and the stack pointer is left pointing to the last entry. As it would be cruel to deny the procedure use of the stack while it is performing its task, it will need to save a pointer to the parameters or perhaps move them from the stack into memory. However, since the frame pointer remains in the same position (as far as the procedure is concerned) until the `exit` instruction, it can be used to access the parameters instead, leaving the stack pointer free for other purposes. To show how this can be done, the parameter setting and call and the procedure entry code will be combined as:

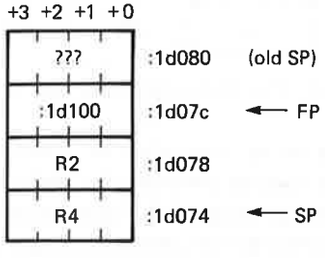

Fig. 8.3 The stack after `enter [R2, R4], 0`.


```
         movd     int, TOS       ;int to stack
         addr     struct, TOS    ;addr to stack
         movf     x, TOS         ;real to stack
         bsr      proc           ;now call routine
         ...
proc     enter    [R2, R4], 0
         ...
```

which gives the stack picture shown in Fig. 8.4 after the `enter` instruction has been executed.

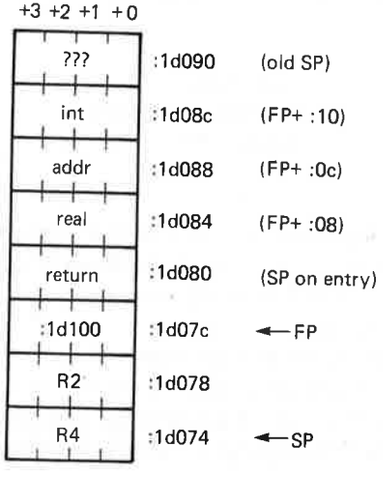

Fig. 8.4 The stack with procedure arguments after `enter`.

Now the stack can be used as desired while the parameters can be accessed from any part of the procedure code by using an FP-relative (memory space addressing mode) operand. The two forms of addressing mode given above, which used the stack pointer, can also be used with the frame pointer. The address parameter, in the example on accessing parts of the structure it was supposed to point to, was used to move its double word field into `R3` with the following instruction and memory relative addressing mode:

```
movd     :0a(8(SP)),R3
```

Assuming the same layout for the structure but the offsets shown for the frame pointer above, the operand can be rewritten as:

```
movd     :0a(:0c(FP)), R3
```

The last topic in internal procedures arises from automatic variables, also known as local variables. They are allocated space in the current stack frame and disappear when the procedure exits (or, rather, when the `exit` command is executed) – this being the purpose of the mysterious constant on the `enter` instruction. The full description of `enter` is: first, it pushes the contents of the frame pointer on to the stack and then copies the resulting value of the stack pointer into the frame pointer; second, it subtracts the value of the constant (second) operand of the `enter` instruction from the stack pointer, thus making a space of (constant) bytes in the stack frame; and third, it pushes the contents of the registers given in the `reglist` operand on to the stack after the space. To illustrate this, we can assume that the procedure must make a copy of the structure addressed by the second parameter in the example above. The procedure will now need 22 bytes of local storage to hold it. Just taking the procedure entry code this time, with the alteration to allocate local storage, we get:

```
proc    enter    [R2, R4], 22
        movmw    0(12(FP)),-22(FP), 7
        movmw    14(12(FP)),-8(FP), 4
                 ...
```

The instruction `movmw` has been used here on the assumption that a 32016 is in use. Transferring bytes when the bus width allows words to be moved doesn't make much sense; if a 32032 is being used instead, then as far as possible double words should be used.

After execution of the `enter` instruction and the two following move block commands the stack would look like Fig. 8.5. The stack picture has been structured to show the fields of the structure as separate elements on the stack; the fields can now (and throughout the code) be referred to as, for example,

```
movl    -16(FP), F2 ; long real
movd    -0c(FP), R4 ; dbl word
```

The `exit` instruction reverses the actions taken by `enter`. First, it restores the registers saved by `enter`. It assumes that it has the correct `reglist`, popping double words off the stack and putting them into registers, starting with the highest numbered in the list. Second, it restores SP to the value it had before `enter` by copying the current contents of the frame pointer into it. Lastly, it restores the frame pointer by copying the value saved in the stack back into it.

There is one further item which has been kept, appropriately, till last; the return instruction. This has the format

```
ret     constant
```

and takes the return address off the stack, putting it into PC, causing execution to continue immediately after the `jsr/bsr` call. There are some

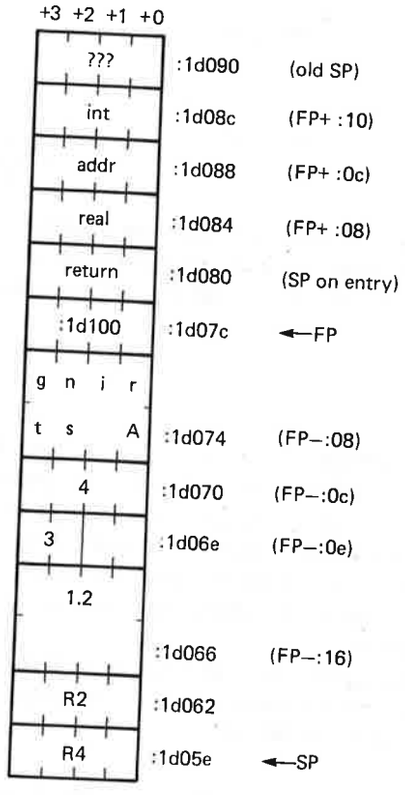

Fig. 8.5 The stack with procedure arguments after a block move.

obvious provisos. `ret` assumes at its execution that the stack pointer *is* pointing to the return address placed on the stack by the call, and care must be taken that everything put on to the stack while the procedure is executing is taken off again before it ends. This also applies to the `exit` instruction as it assumes, like `ret`, that the stack pointer has the same value it had immediately after the `enter` instruction. The constant given as the operand for `ret` is used to remove parameters from the stack. It is in rather a peculiar position as the parameters are put on to the stack before the call and removed by `ret` at the end of the procedure; however, providing care is taken with the call (using a macro instead of writing the code each time) this should cause no problems. Using the procedure example given earlier, with its three parameters taking up 12 bytes on the stack, the appropriate form of `ret` instruction to use would be:

```
movd    int, TOS        ;int to stack
addr    struct, TOS     ;addr to stack
movf    x, TOS          ;real to stack
bsr     proc            ;now call routine

        ...
```

```
proc    enter    [R2, R4], 0

                 ...

        exit     [R2, R4]
        ret      12              ;remove parameters from stack
```

There is, however, a case in which the routine can't tell how many parameters to remove from the stack, like **printf** in C or *write* in Pascal, which are called with a variable number of parameters. Even here, the 32000 has the answer. There is an instruction

```
ADJSPi   src
```

which can be used following the call to remove `src` bytes from the stack; and it is at the call that the number of bytes occupied by parameters is known. The operand is taken to be a signed integer and is sign extended to 32 bits before being used. The stack pointer is adjusted by subtracting this operand from it; if the integer is positive, the stack is lengthened and if negative, the stack is shortened. For the purpose of ridding the stack of surplus parameters, the integer must be negative.

The size of this integer is given, as usual, by `i`. If the procedure used above did not take exactly three parameters, the procedure top and tail and the calling sequence could be modified to:

```
        movd     int, TOS        ;int to stack
        addr     struct, TOS     ;addr to stack
        movf     x, TOS          ;real to stack
        bsr      proc            ;now call routine
        adjspb   =-12            ;remove 12 bytes

                 ...

proc    enter    [R2, R4], 0

                 ...

        exit     [R2, R4]
        ret      0               ;leave stack unchanged
```

The NatSemi assembler has a nifty set of procedure directives which makes it much easier to write parameter passing procedures, as the careful calculation of parameter addresses is not needed. In outline, a procedure is written in the format:

```
;Start procedure block.
<label>  .proc
```



```
;Parameter block.
{label}  .blki

         ...

{label}  .blki
;Returned values block.
         .returns        ;returned values block
{label}  .blki

         ...

         .blki
;Local variable block.
         .var            [reglist]
{label}  .blki

         ...

{label}  .blki
;Procedure body.
         .begin

         ... instructions ...

         .endproc
```

The procedure starts with the `.proc` directive labelled with the procedure name and ends with `.endproc`; immediately following `.proc` comes a description (in allocation directives) of each parameter, each of which may be labelled, in the order of the statements pushing them on the stack before the call.

If the procedure returns one or more values, or a multi-word value, these are described by another set of directives following the `.returns` directive. Returned values are overlaid on the stack space taken up by the parameters and, if there is more returned value space than parameter space, the parameter space must be extended appropriately. The local variables are described in a block starting with `.var` which has an optional `reglist` operand giving the registers to be saved on entry. All the variables can be labelled or not as needed.

Finally, the instructions forming the procedure body are bracketed between `.begin` and `.endproc`; this last directive generates an `exit` instruction with the same `reglist` as that on the `.var` (thus avoiding unfortunate clerical errors), followed by a `ret` (for local procedures) or `rxp` (for exported procedures).

As an example of the use of these directives, here is an exported procedure box which calculates the volume of a box given the length, width and height.

```
box::     .proc
length:   .blkd       ;local name for 1st parameter
width:    .blkw       ;second parameter
depth:    .blkw       ;third parameter

;return value block-takes up exactly the same
;space as the parameter block
          .returns
volume:   .blkd       ;return the volume
howbox:   .blkd       ;success/fail indicator
          .var        [R0]

;local variable block
temp1:    .blkd
temp2:    .blkd

;start the procedure code
          .begin
;sign extend width and depth to double word
          movxwd      width, temp1
          movxwd      depth, temp2

;use R0 as an accumulator
          movd        length, R0
          muld        temp1, R0
          muld        temp2, R0

;set the return value
          movd        R0, volume
;initialize the condition to success
          movd        'Good', howbox

;set the indicator to 'fail' if the volume
;is zero or negative
          cmpqd       0, R0
          blt         fin           ;0 < R0, ok
fail:     .equ        'Bad!'
          movd        fail, howbox
fin:
          .endproc
```

Note how easy it is to think that the `blt` should jump to an instruction setting `fail` into `howbox`, until careful analysis shows that the branch is taken when `volume` is greater than 0!

8.3 MODULES

In its support of modules, the 32000 series has made another giant leap forward. Modules embody the modern concept of information hiding, allowing data structures to be manipulated solely by procedures bred to the purpose with no other access permitted. This means that as a list can only be written to or read from by carefully tested procedures, the list can be kept uncorrupted and not only can other external code be prevented from making unexpected changes to it but also any use of the list can be guaranteed to have no unexplained side-effects.

The ideal, in this type of programming, is to have a program split into several modules which affect one another only in clearly defined and documented ways, this being enforced by allowing strictly controlled modes of access which are easier to test and have their correct action verified. The key words, together with `module`, are `import` and `export`: using the directive `export`, a module allows knowledge of only those objects (procedure entry points or data) it sees fit to go to the outside world. Any other procedures or data are unreachable from the outside and only the code within the module can use them. The `import` directive operates to bring knowledge of objects exported by other modules into a module, again preventing indiscriminate access, channelling it through a group of directives at the beginning of the module and clearly visible.

A module will usually be a collection of like-minded procedures which will need some shared variables and some variables which keep their value between calls, either condition making it impossible for them to be on the stack. As the constant addresses on entry to any procedure in the module are the offsets to the program counter and the static base register, these variables may be allocated relative to either register; however, only constants or shared code should be relative to PC so that the module is 'ROMable' (tethered to a fixed physical address as when in ROM) and to allow protection by memory management from overwriting by wild code. True variables should be allocated relative to the static base register.

This setting up of registers on entry to a module is all done by hardware support. It can be done by software, but software must be tested, can contain bugs, takes up code space and is slower and, finally, can differ from one operating system to another. Here, with the aid of a simple table constructed by the linker, is full module support – the same for all seasons.

To be accessible, every module in the system must have an entry in this table (Fig. 8.6), each entry containing four 32-bit pointers.

The MOD register always points to the first byte of the entry for the module at present being executed. This entry is one of the 4096 entries the module table can contain: as MOD is a 16-bit register, the table must be complete in the first 64 Kbytes of memory and, as each entry is 16 bytes long, there can only be 4096 entries. With memory mapping (see Chapter 10) this becomes 4096 entries for each user; it should suffice.

The first pointer in the entry is the address of the start of an area of

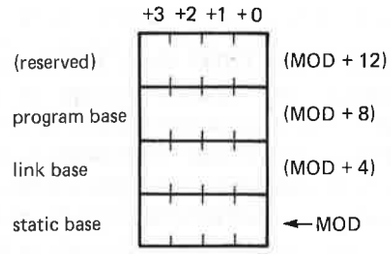

Fig. 8.6 Module table entry format.

memory which the procedures in the module use for shared and static variables; on entry to any of the procedures of the module, the static base register is given this value. This static data area is similar to a FORTRAN common area in that, although all the procedures have access to it, it doesn't have to be an area within the module – it can be anywhere in memory. The linker will usually collect static areas together (as it would common areas) and allocate them after all the modules.

The next pointer is called the link base and points to the start of a table containing the addresses of all the names imported or exported by the module. There is a difference in the entries for data and for procedures; a data item is represented by an address but a procedure is represented by two 16-bit items, the address of the entry in the module table for the module which contains it and an offset giving its entry point relative to the module's program base. Like the module's static area, the link table may be anywhere in memory.

The third pointer is the procedure base. It contains the address of the first byte of code in the module. All the procedures in the module will have positive offsets to this address and there is an obvious, minor, restriction that the entry point of the last procedure exported be within 64 Kbytes of the first.

The last pointer position is reserved.

The source format of a module for the Acorn assembler is:

```
        module      <name>
        import      <variable list>
        importc     <procedure list>
        export      <variable list>
        exportc     <procedure list>

        ...

        areadef     static, [write, data], <align>
;make the area a static base area
        defsb       static
;start of static data area
        area        static
        ...

;static data definitions

        ...

        area                                ;code area by default
;code for internal and external procedures
proc1   ...

        end
```

For the NatSemi assembler it is:

```
        .module     <name>
        .import     <variable list>
        .importp    <procedure list>
        .export     <variable list>
        .exportp    <procedure list>

        ...

        .static
;start of static data area
;static data definitions

        ...

        .endseg                         ;end of static data
        .program                        ;start of code
;code for internal and external procedures
proc1   ...

        .endseg                         ;end of code segment
        .end
```

The differences are small. The NatSemi directives are all prefaced by a '.' and the static data segment and the program segment start differently and must end with `.endseg` rather than being implicitly ended by the start of the next segment.

The Acorn segments are delimited by `area` directives which may be followed by a symbol. This symbol must have been defined by an `areadef`

directive which also allocates attributes to the symbol and (the last parameter) an alignment for the area. To define a static base segment, the symbol must appear as the operand of a `defsb` directive. If `area` is not followed by a symbol, it refers to the default area which is for code and is byte aligned.

A procedure which is to be called from another module differs in two ways from an internal procedure. It must be called by a `cxp` instruction and must end with an `rxp` rather than `ret` and, as well as the return address, has the contents of the MOD register pushed on to the stack. The destination is no longer an address (as for `jsr`) or an offset to the PC (as for `bsr`) but a descriptor in the local link table. The `cxp` instruction is

```
cxp    index
```

where `index` is the number of the entry in the table.

The `cxp` instruction has to do two things; find the address of the procedure called and set up the registers to suit the called module while saving enough information to get back to the calling one again. First, it saves the information needed to return to the calling module. The MOD register and the address of the next instruction after the `cxp` are saved on the stack; although the MOD register needs only 2 bytes of the stack, an extra (but reserved) 2 bytes is taken up to make a double word entry. Next, once the old MOD register has been saved, the module table address is copied from the link table entry to the MOD register and the first double word in the entry addresses used to set the SB register. Finally, the sum of the offset (in the link table entry) and the second double word of the module table entry (the procedure base) is copied into the program counter and execution of the called procedure begins.

It is important to note the presence of the MOD register on the stack as it changes the offsets of the parameters from FP. If the previous procedure example were refurbished to become an external call, it would become:

```
            importc     extp

            ...

            movd        int, TOS        ;int to stack
            addr        struct, TOS     ;addr to stack
            movf        x, TOS          ;real to stack
            cxp         extp            ;now call external proc

                        ...

<somewhere in another module ...>

extp        enter       [R2, R4], 0

                        ...

            rxp         12              ;remove parameters from stack
```


and the stack layout after the `enter` instruction would be as shown in Fig. 8.7. As can be seen, the parameter offsets are increased by 4.

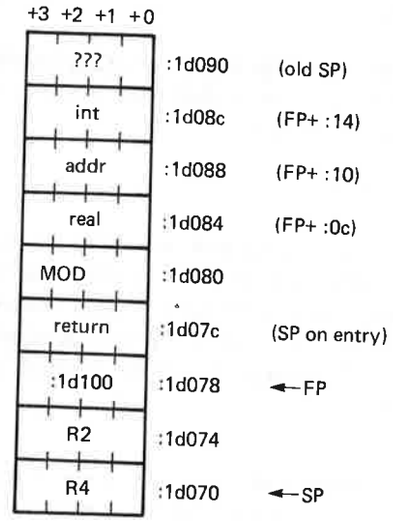

Fig. 8.7 The stack with procedure arguments after an external call and `enter`.

The `rxp` instruction clearly has more to do than its cousin `ret`: as well as fetching the return address from the stack and putting it into the program counter, it must also reset the registers to the calling module values. The first part is easy: it takes the MOD table entry address from the stack and puts it into the MOD register. Now it has the old entry, all that is needed is to get the old contents of the SB register from the module entry and it's done. There is of course the little matter of adjusting the stack to remove the space taken up by any parameters there may have been, and this is done in the same way as for `ret`. If desirable, `ADJSPi` can be used instead, as noted earlier.

There is one more external item which may be accessed, external data. A module may allow certain of its variables to be public – a random number generator may make its seed public, allowing other modules to reset it, or a heap manipulation module might have a public variable containing the current address of the bottom of the heap.

This type of item is accessed by an addressing mode, external mode, which acts in some ways like memory relative. Suppose there is a data area defined as

```
            export      afar
            areadef     far, [write, data], byte
            area        far
afar        allocb      8               ;offset+0
bfar        dcw         9               ;offset+8
cfar        dcl         3.14            ;offset+10
```

and you want to get the present value of `bfar`. You must write

```
import    afar
movw      afar+8, R4
```

If the imported variable `afar` is the second external variable to be declared, at run-time, its address will be in the second entry of the module's link table and the operand will be translated into

```
movw      ext(1)+8, R4
```

with `ext(1)` referring to the second link entry and `8` being the offset between the address in the link table (that of `afar`) and the first byte of `bfar`. Like the memory relative modes, you not only get a pointer to an item but you have an offset to add to it by which *any* item in the referenced block – no matter how big the offset (is -2^{30} to 2^{30} enough?) – can be accessed in one instruction.

EXERCISES

8.1 Change the code for the desk calculator of Exercise 6.3 to use the `CASEi` instruction when deciding on the bit to be executed given the instruction number. Note that as the instruction codes run from 1 to 6, a `CHECKi` instruction will be needed as well.

8.2 How could the desk calculator code in Exercise 8.1 above be changed to make it an internal procedure called by `bsr`?

8.3 Write a module to return a random integer in register 0. It should have two entry points: one, `RandomSeed`, takes a double word integer parameter passed on the stack to set up the double word `seed` in the module's static base area; and the other, `RandomHarvest`, returns the random integer. If you don't have your own favourite formula, you can find a wide selection of good (and bad) ones on page 102 of Knuth's *Art of Computer Programming,* Volume 2.

8.4 Change the code to delete files in Exercise 7.2 to make it an internal procedure with one parameter, the address of the first extent, passed on the stack. Save and restore all registers used by the routine.

8.5 Write and test a module containing a procedure called `frexp` which is passed two parameters on the stack, a 64-bit floating point number and the address of a double word. The routine is to put the exponent of the floating point number into the double word as an integer in the range −1022 to 1023 – which requires the conversion of the raw exponent – and returns the floating point number with an exponent bringing its value into the range 1.0 to 0.5 in floating point register 0.

9 Panos – the Acorn 32016 operating system

9.1 INTRODUCTION

To the assembler programmer an operating system appears first as a library and a convention on procedure calling sequences and then as a command language and a set of utilities. In this chapter the first view of an operating system will be taken, with the next chapter taking a behind the scenes view, showing how a special set of 32000 instructions makes the task of writing an operating system lighter.

As each operating system will have its own ideas on libraries and calling sequences, a particular system, Panos, Acorn's operating system for the Cambridge second processor for the BBC micro and the Cambridge workstation, has been chosen as an example. Both machines use an NS32016, the second processor running at 6 MHz and the workstation at 8 MHz.

9.2 THE PANOS LIBRARY

The purpose of an operating system is to do all the detailed work of dealing with peripherals, handling exceptions, allocating memory and managing a real-time clock while allowing its user to access the hardware without the special expertise of the operating system writer. This is done by means of a library which may, in fact, consist of a number of entry points into the operating system itself. The library imposes the concept of files and directories on to the hardware disk storage and offers procedures for creating files, opening them, reading or writing them and deleting them. It also allows the real-time clock to be set or read, and provides the user with ways to field errors and deal with them or, by default, print appropriate messages. It would allow a user program to allocate and dispose of blocks of memory; in particular to allocate the necessary space for a program to run, releasing it when the program notified the system that it had finished.

This then is the operating system library; in the particular case of Panos the library has 13 sections:

1. error handling
2. argument decoding

3. data conversion
4. memory allocation
5. input/output
6. file support
7. loader
8. random numbers
9. time and date
10. condition handling
11. event handling
12. global variables
13. program control

The condition handler is very important and Section 9.3 deals exclusively with it.

9.2.1 Error handling

Many of the routines in the library make provision for an error number to be returned. Several have two entry points, one returning an error number and the other using the condition handler to notify errors. The error number is 32 bits long and has four fields: an 'info' field showing whether error information is available; an 'interface' and a 'detecting' facility field showing which facility was called by the user (interface) and which facility actually detected the error (the facilities are the sections of the library given in the table above); and an error code field. There are three procedures which are used to interpret the error code. The most usual one is `GetErrorMessage`, which returns three strings with the name of the interface facility, the name of the detecting facility and the text of the appropriate error message. Of the other two procedures, one is used to save the error information so that a later procedure can change the interface facility name, and the other returns additional information when there is any.

By providing these procedures, Acorn hope that software writers will be encouraged to use them rather than write their own, so that users get a standard response from the system when something goes wrong.

9.2.2 Argument decoding

The systems software for the second processor, compilers and utilities, will need to access parameters from the command line. Many operating systems provide a means of passing command lines to the programs they invoke but often this, though simple to use, is primitive and results in a confusion of different parameter conventions. Here, Acorn provide a discipline encompassing both compilers and utilities. The parameters expected are described by a prototype string in which each parameter is given a keyword and can be further specified as having one or more arguments of prescribed types. For instance, a compiler command line might be described as


```
Source/A/E-asm List/S Help/S
```

The `A` after the `Source` keyword means that it must have at least one argument and the `E-asm` allows a filename to have the `-asm` appended to it if it does not already have an extension. The `S` after `List` and `Help` shows that they are state keywords. If they are present the compiler can take action but they don't have parameters. If the `List` keyword is given, the compiler would write its output to a file with the name `source-lis` and if the `Help` keyword was present some brief information about the expected command line could be displayed on the screen.

A copy utility could have the key string

```
File/A/? To/K[vdu:]
```

which requires the keyword `File` (which need not be present if the arguments are presented in order) to have at least one argument, the `?` allowing it to have an indefinite number. The `K` after `To` shows that the keyword must be present if there is an argument and `[vdu:]` is to be taken as the default if no keyword and argument is given. This would allow

```
copy Roff4.c Roff41.c Roff4.h
```

to be used to display the three files on the screen one after the other.

There is a procedure to initiate the decoding of the command string and further procedures to read string, state, boolean or integer arguments. For keywords with variable numbers of arguments, the procedure `GetNumberOfValues` will return the number of arguments actually present.

9.2.3 Data conversion

The Panos system has a convention for number representations and, to go with argument decoding, it provides a number of procedures to convert from a string representation to binary integer and the other way around. To give the widest range of numbers, the base (unless decimal) must precede the number and be separated from the first digit by an underscore (_): for instance, the hexadecimal number represented in the assembler convention as `:9a` is written in the Panos convention `16_9a`. These numbers can have a sign; −15 in octal would be represented by `-8_17`. In these procedures, the string representation is always as shown above, with the base (in the range 2 to 36 inclusive) always in decimal and the number using the digits 0 to 9 up to decimal base and A to Z (or a to z, case is not significant) for the 'digits' 10 to 35. There is a Panos entity called a 'Cardinal' catering for unsigned numbers which range from 0 to 4 294 967 295. A signed number (an integer) must be in the range −2 147 483 648 to 2 147 483 647.

It is also possible to convert booleans (0 or 1) to and from strings (FALSE and TRUE).

9.2.4 Memory allocation

There are several procedures in this section which a program may call to be allocated a block of memory from the heap. It is possible to 'tag' allocated blocks of memory so that all blocks with the same tag can be deallocated at once. Otherwise, single allocated blocks are deallocated singly: the allocator joins adjacent deallocated blocks so as to make it more likely that new allocations can be satisfied without extending the boundaries of the heap.

Allocated blocks may also be split and the heap boundary can be set to another value to enable memory to be used differently. As the stack grows downwards towards the top of the heap there is a possibility that they may meet; a procedure to read the present end of heap address allows the programmer to check for this.

The first 64 Kbytes of memory are reserved for use by the hardware supported module table and this area is also handled as a heap by Panos so that the loader can make requests for space in it before loading another module. For the curious, a procedure called `GetStoreInformation` returns the start of the heap, its size and the amount of free space for both the heap and the module heap.

9.2.5 Input/output

The basic unit for files is the byte and all files connected to a program have their contents moved in or out as a stream of bytes. Several files may be open at one time, the actual number depends on the version of Panos.

As Panos uses a BBC board to perform all its input/output it uses the BBC filing systems, the DFS, the NFS (Econet) and the new ADFS which can either work with double density floppy disks or a Winchester. The particular filing system in use is set by a global variable and the format of the file names must then comply with the conventions of the system in use.

Input from the keyboard and output to the screen is also provided for in two modes; raw mode and filtered mode. In raw mode all characters are sent to the screen or returned from the keyboard. In filtered mode (on output) only printing characters together with clear screen, newline and carriage return are passed to the screen, all others being removed, and on input from the keyboard characters are buffered until a newline or carriage return is found and erase (backspace) and kill (ctrl-U) are honoured. The RS423 input and output lines from the BBC board can be used for input or output and output to the Centronics printer port is supported as well.

There are four special devices which can be attached to any of the actual input/output facilities – `Input:`, `Output:`, `Control:` and `Error:`. These devices are set by default to the BBC keyboard and screen but procedures are available to connect them to files, the printer or the RS423 lines instead.

The actual procedures for transferring bytes come in two forms; one set which inputs from or outputs to the appropriate special device and the other which has the stream as a parameter and can thus be used to read from or write

to any open stream. The input/output procedures are at a fairly low level, imposing no structure on the stream of bytes read or written. There are procedures to read or write a single byte and to read or write a block of bytes. There are two procedures which allow random accessing of files; one returns the value of the file pointer and the other sets it to a given value. There are also procedures to show how many bytes are left on a stream, whether it is actually connected to a file or to the keyboard or RS423.

9.2.6 File support

This is a miscellaneous bunch which, while essential, doesn't belong anywhere else. Panos files are date stamped (using the real-time clock) and procedures are provided to read a file's date and to set it to a given value. There is also a `Touch` procedure which simply alters a file's date stamp to the current date and time.

There is a procedure to delete a file by name and two procedures handling file names, one to rename a file and the other to convert the Panos file name (base with extension) to that appropriate to the filing system in use: the Panos file name `Dump-asm` would be changed to `a.Dump` if the DFS was the underlying filing system. Finally there is a procedure to set a global string called the working directory; file names without a directory prefix will use this string to make a full path name.

There are also procedures to handle directories, load and save files and deal with file names containing wild characters: ? matching any single character, * matching zero or more characters and ... matching zero or more names in a path name.

9.2.7 Dynamic loading

Panos supports dynamic loading of procedures. When a user program calls a procedure which is not present in memory, Panos fields the call and sets the loader to finding and loading the missing module. Procedure and external data names may be 'declared' to the loader and will thereafter be available to any program.

9.2.8 Random numbers

Many programming languages provide for a random number function and, to centralize this function rather than have each language implementation invent a different function for its own library, Panos includes it in among the other more traditional operating system routines. The random number is a 32-bit unsigned integer and the section includes a procedure to initialize the generator's seed.

9.2.9 Time and date

Panos supports a real-time clock held as a 64-bit integer representing the number of centiseconds since midnight on 1 January 1900 – just a few years after Hermann Hollerith introduced the US Census to punched cards. A quick calculation with the trusty fx-570 shows that this clock should be good for another 5 or 6 million centuries. There are procedures to set the clock and read it and to convert from the 64-bit format into a textual format or a standard format. The textual format is

```
31 Mar 86 12:53:08
```

and the standard format is

```
1986-03-31 12:53:08.98
```

which adds the centiseconds to the time as a two-place decimal. There are also procedures to return just the time or just the date or both time and date together in either of the string formats. It should not be necessary for anybody to have to write their own time and date routine unless they want the day of the week as well.

9.2.10 Event handling

The events handled are those occurring on the BBC board, such as buffer full or empty, keyboard interrupt, interval timer interrupt and so on. The event handler is a routine with prescribed parameters, written by the user and called by Panos when the event occurs.

The library contains procedures to declare an event handler (as you can have different event handlers for different events) and remove it from consideration, to enable and disable events and to return the status of an event.

Events are only signalled when the system is outside event handlers; there is no nesting of event interrupts. This forces events to be queued as Panos has no control over the amount of time the user spends in a handler. There is therefore no guarantee that more events of the same type have not already occurred by the time an event is dealt with; in particular, the interval timer and TV sync pulse events may be subject to a variable delay before the handler sees them.

9.2.11 Global variables

Panos supports a number of string variables, some of which it relies on for its correct operation (the current filing system, for instance) and others mainly for the user, like the result code of the last program run. These are all kept in a central table and are made available to all programs run under Panos by means of the `GetGlobalString` procedure in this section of the library. Programs and

utilities can also set new strings into the table by using `SetGlobalString`, or alter the values of existing ones – though a group of globals beginning `SYS$` may only be read as it is the system's responsibility to change them. The `DeleteGlobalString` procedure removes strings from the table and `GetGlobalStringName`, given an index, returns the indexed name in the global string table.

A useful set of strings, set by Panos from the initialization file, are those determining the mapping of extensions on to filing system names. These all begin `File$-ext`, where `ext` is the Panos file type extension, and have values of the form <prefix> – <suffix>. When a file is passed to Panos with an extension matching `ext` it is transformed (by `PhysicalFileName` from file support) into <prefix> file <suffix>, painlessly preprocessing the file names to fit the prejudices of the filing system.

9.2.12 Program control

The procedures in this section form a very wide ranging set of functions for running programs from within an executing program. As Panos performs memory allocation and the loader is an integral part of the system rather than another utility, it is possible for a program to load and run another program without being itself terminated in the process. The environment passed on to the new program is defined and will be the same as the parent's initially, though the child program can change things as necessary (but closing open files won't work) with the environment being restored on exit from the child.

The child need not be a program but can be simply a procedure in a file. The Call procedure causes it to be loaded from the file and passed an argument string. There is an Obey procedure which will start the command interpreter reading commands from a file; this file may be passed an argument string. Invoke will execute a Panos command or utility from the set of known commands.

9.3 CONDITION HANDLING

This section of the library is primarily intended for compiler writers to use as a general error handling system – both for internal compiler errors and for system errors. It is, however, so versatile and easy to use that all stand alone programs should include it, using an assembler module if the language does not allow it to be used directly.

Each module can have its own condition handler which must be made known to Panos by using the 'handler' directive provided by ZASM. The handler will be called when a hardware exception occurs while the module is executing or within a system routine called by the module. There is also a means of signalling an exception by calling one of the procedures in this group which allows the programmer to use the handler to field software errors as well. Centralizing error handling makes life much easier.

The handler you write will be called with the following parameters

```
CallType             : Cardinal
AdditionalParameter  : Integer
CurrentEnvironment   : Environment
ExceptionEnvironment : Environment
```

and it returns an integer result. `Cardinal` is an unsigned 32-bit integer and `Integer` is a signed 32-bit integer; `Environment` is a structure with the layout shown in Fig. 9.1. The record consists of 22 double words and the offset of the first byte of each double word is given. The user program status register is only 8 bits long and will be in the byte `:10` offset, and the module register which is 16 bits long will be in bytes `:14` and `:15`.

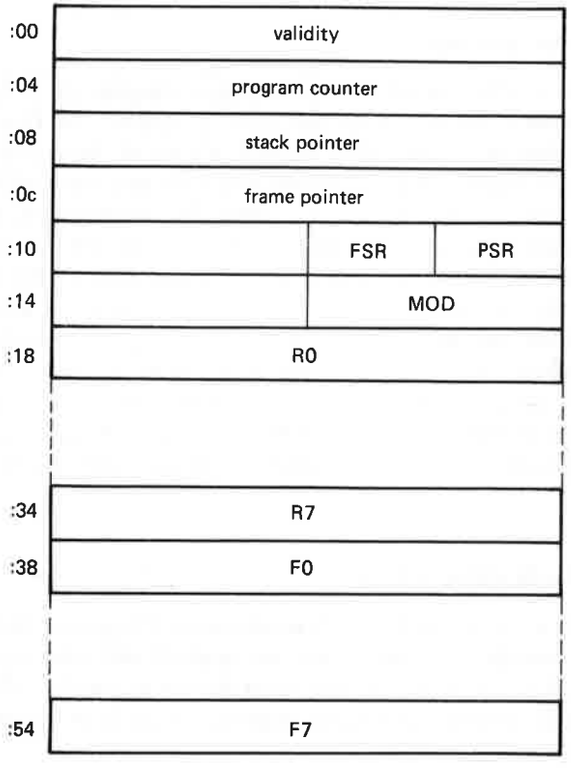

Fig. 9.1 Panos environment record format.

The first double word is used to show which of the following double words have valid entries by setting bits 0 to 20, the remaining bits being reserved. The bits are set to show that the entry is valid, unset if it is not; the correspondence is:

0 Program counter
1 Stack pointer
2 Frame pointer
3 User PSR
4 Module Register
5 R0
...
12 R7
13 F0
...
20 F7
21 FSR

The parameter `CallType` takes the following values:

0 Initialize module
1 Stop – program finishing
2 Exception
3 Exception passed on
4 Diagnose
5 Diagnose passed on
6 Describe frame
7 Describe module data
8 Unwind

The remaining values are reserved, at present unused.

The descriptions of the various calls is the reverse of the other library descriptions as here it gives the parameters with which the user's handler routine is called from Panos and describes the result that the handler must return. So while the heart leaps when it sees calls such as `Diagnose` (produce a description of the fault on the output stream), `DescribeFrame` (produce textual information on the current output stream describing the current procedural frame) or `DescribeModuleData` (describe the static data of the current environment on the output stream), these are all functions the user has to write to help in discovering how the current module can have gone wrong!

The calls with `CallType 0` and `1` are to allow the module to initialize itself (`CallType 0`) and, before the program using the module halts, to clean up as necessary (`CallType 1`). With these two calls no environment is necessary and none is passed; the `AdditionalParameter` is zero for initialize and contains the return code passed to `Stop` for the termination call.

The calls for `Exception` and `ExceptionPassedOn` have the full set of parameters. For both these calls `AdditionalParameter` contains the error code; this is negative if the exception arises from an error trapped by Panos and is positive if it arises from a signal. `ExceptionPassedOn` is used to indicate that the first module handler called couldn't deal with the exception for some reason. It could be used by a module deep in the program's run-time system to pass the

fact of an exception back up the hierarchy until it comes to a module the user might recognize.

These two calls have a counterpart in `Diagnose` and `DiagnosePassedOn` except that for these two a description of the fault must be written on the current output stream.

The calls `DescribeFrame` and `DescribeModuleData` are intended for compilers with source level debugging systems. They would be called when the user requested a trace back from the point of exception or when moving from module to module checking the values of the variables to find the cause of the error. In both these calls there is no additional parameter or exception environment record but a record for the current environment is given.

The `Unwind` call is made when a handler returns a zero result when called for an exception. The handler has to set the current environment to the state it would have when the procedure exited normally. When this has been done, the system can call the next handler in line with a passed on exception.

Of these nine calls only five (initialize, stop, exception, exception passed on and unwind) are used by Panos. The others (diagnose, diagnose passed on, describe frame and describe module data) are calls provided for the system running under Panos to make. The procedure `CallHandler` has to be provided with the parameters given in the description. Generally speaking, using these calls in assembler modules would require a mound of code writing and they are much more suited for compilers which can crank out code to describe data and print traceback info without difficulty.

The last procedure in this section is `Signal`, which forces a call to the handler of the current module. It has two parameters. One shows the cause of the signal which may be either one of the Panos errors or a special code which the handlers have been trained to respond to. The second parameter is the address of a buffer which can be used to pass further information to the handler if needed.

9.4 CALLING SEQUENCES

In the last chapter on procedures, a method of passing parameters to a procedure was shown but nothing was said about procedures (all right then, functions) returning results. The usual way to do this is to leave the result in a register which restricts the result type to integer, pointer (address) or real. This restriction is common in most high-level languages where the way to describe a procedure returning more than one result or a result longer than an integer or a real doesn't seem to have been invented yet.

This argument doesn't apply to assembler as it has not yet vanished into the academic grasp, and ingenuity and a feel for programming still has a place. Acorn have defined a set of standards for passing parameters and returning results which allow any kind of object to be passed to or returned from a procedure, and as these calling sequences are used in the Panos library procedures they should be described for that reason if not only for their elegance.

It should be noted that the calling conventions do not ask the called procedure to save any registers it may use, this being left to the calling procedure, and any general registers you want to use again after the call should be saved before the calling sequence starts and restored afterwards; if you are returning a result in `R0` or `R1`, of course, the results should be moved elsewhere before these registers are restored.

9.4.1 Parameters

Parameters are passed on the stack as shown in the previous chapter but the different kinds of argument are described in different ways. There are four categories of arguments: integers, reals, strings and arrays. Of these, integers and reals may be passed as values while the other two types must be passed as pointers. Values are passed as 32-bit and 64-bit quantities only. If a byte is passed it must be zero extended (if unsigned) or sign extended to 32 bits before being put on to the stack. The 64-bit size is used to pass long reals and items needing more than 32 bits. References or addresses can be used for arguments which are passed as values though addresses must be used if the argument is to be changed.

An integer passed as a value is pushed on to the stack as a 32-bit quantity if it fits; otherwise it must be put there as two contiguous double words, the more significant double word first followed by the less significant one. For example, the Panos library procedure `XCloseStream` takes one `Cardinal` parameter, the number of the stream to be closed:

```
        importc   XCloseStream

        ...

InStrm  allocd  1   ;stream number

        ...

        movd   InStrm, TOS
        cxp    XCloseStream
```

Pointers are passed as 32-bit unsigned values using the `addr` or `movd` instruction with the top of stack addressing mode. As an example, the procedure `XDeallocate` (which doesn't return an error code – `Deallocate` does) takes one parameter which is the pointer to the block of memory to deallocate. It is called by

```
        importc   XDeallocate

        ...
```

```
block     allocd   1    ;block pointer

          ...

          movd   block, TOS
          cxp    XDeallocate
```

Real values are passed by a MOVF for short reals and MOVL for long ones, again with the TOS addressing mode.

Strings are more interesting as they have two items pushed on the stack for each string: a length and a pointer. The length is passed as a 32-bit unsigned number and the pointer is the address of the first byte of the string. The procedure Touch takes a single string parameter, the name of the file to be stamped with the current time. It can be called by:

```
          importc   Touch

          ...

NowFile   dcb    "dfs::0.a.DateChg"

          ...

          movzbd   NowFile, TOS          ;length
          addr     NowFile+1, TOS        ;pointer
          cxp      Touch
```

The counted string form using quotes is chosen to make the first byte of the string the length. It would be troublesome to calculate; this length is a byte integer so it must be zero extended to double word before being pushed on to the stack. The first character of the string is the first byte after NowFile which is NowFile+1.

Structures and arrays are represented by the address of the first byte and there must be a prior agreement between the calling procedure and the called procedure on the size of the structure. This will normally be part of the called procedure's description.

The order in which parameters are pushed on to the stack is the reverse of that expected. In the Acorn convention, the first parameter after the procedure name is the *last* to be pushed on to the stack and thus has a fixed offset (12) from the frame pointer. The procedure XBlockWrite writes a group of bytes to the current output stream and takes two parameters, the number of bytes to write and the address of the first of them. Its description is

```
XBlockWrite(CARDINAL: Blength; ADDRESS: Buffer);
```

and its calling sequence is


```
          importc   XBlockWrite

          ...

Message   dcb    "Sorting and Searching"

          ...

          addr    Message+1, TOS    ;first byte
          movzbd  Message, TOS      ;number to write
          cxp     XBlockWrite
```

9.4.2 Results

If only one result is returned and it is an integer, real or pointer, then it is passed in the appropriate register; R0 for a 32-bit integer or address, R0 and R1 for integers larger than 32 bits with R0 having the less significant half and R1 the more significant, F0 for a short real and F0 and F1 for a long real. As an example, the procedure XBlockRead reads a number of bytes from the current input stream. It takes two parameters like XBlockWrite and returns the number of bytes read. Its description is

```
XBlockRead( CARDINAL: Blength; ADDRESS: Buffer );
```

and its calling sequence is

```
          importc   XBlockWrite

          ...

BufLen    allocd   1
Buffer    allocb   256

          ...

          addr   Buffer, TOS    ;address
          movd   =256, TOS
          cxp    XBlockRead
          movd   R0, BufLen     ;number read
```

If there is more than one result or the result is a string, information on where the result is to be put is pushed on to the stack after the parameter information. If more than one result is returned, the order of the result information on the stack is last to first like the parameters.

For a single string result, two items must be pushed on to the stack; first the maximum length of the string (the amount of storage in bytes allocated to it) and second the address of the first byte of the string. On return from the procedure, R0 will contain the actual number of characters in the returned

string. This will never exceed the maximum length of the string and, if the string to be returned is too large, the procedure may signal an error. The procedure IntegerToString has the description

```
IntegerToString(INTEGER: Number; CARDINAL: TheBase);
                STRING: ResultString;
```

and it can be called by

```
        importc   IntegerToString

        ...

Number  dcd   12345
Result  allocb   10   ;result string
RLgth   allocd   1    ;actual result length

        ...

        movd   =16, TOS   ;the base
        movd   Number, TOS
        movd   =10, TOS    ;max string length
        addr   Result, TOS
        cxp    IntegerToString
        movd   R0, RLgth   ;result length
```

Procedures returning two or more scalar results (integers, reals or pointers) return the first result as for a single result. The other scalar results must have a pointer to their resting place pushed on to the stack. The procedure Allocate which allocates a block of memory to the requesting program has the description:

```
Allocate (INTEGER: Size);
          INTEGER: ResultCode; ADDRESS: BlockPointer;
```

If the size is negative then up to |size| bytes will be allocated. The result code, if positive, gives the number of bytes actually allocated. It can be called by the sequence:

```
        importc   Allocate

        ...

Block   allocd   1   ;pointer to block

        ...

        movd   =-1024, TOS   ;size
        addr   Block, TOS    ;returned pointer
        cxp    Allocate
        movd   R0, BlkLgth   ;size allocated
```


If the first result (when there is more than one) is a string, it is returned as for a single string result. Any other string results must have three items pushed on to the stack for each string. The first must be the address of a 32-bit integer in which the actual length of the string will be returned, the second a 32-bit integer containing the maximum size of the string, and the third the address of the first byte of the string. The Date procedure has no parameters and returns a result code which, if positive, shows the operation has succeeded and a string containing the date in the format '1 Apr 86'. The string will have eight or nine characters depending on the number of digits in the day. It is described as

```
Date(); INTEGER: ResultCode;  STRING: TheDate;
```

and is called by the sequence

```
        importc   Date

        ...

TheDate allocb   10   ;even numbers are beautiful
DateLen allocd   1    ;TheDate length

        ...

        addr   DateLen, TOS   ;string actual length
        movd   =10, TOS       ;max string length
        addr   TheDate, TOS   ;string address
        cxp    Date
;the result code is in R0,
;and TheDate and DateLen have been set
```

10 Operating systems support

10.1 INTRODUCTION

Most users of the 32000 CPUs will have an operating system in residence, provided by the supplier of the system. This chapter is for those who want to work up a system from scratch, or are porting a system from another CPU to the 32000.

The purpose of an operating system (or, more strictly, the operating system kernel) is to schedule processes, to deal with input and output, handle errors and allocate memory between user programs, utilities and its own routines.

Input/output is usually done by means of interrupts, where a peripheral (or rather, the chip the actual peripheral is connected to) attracts the CPU's attention by signalling a CPU pin. The CPU is designed so that an automatic sequence of events takes place whenever a signal on this pin is sensed: stopping the program being executed, saving enough information to restart it and then passing control to another routine (part of the operating system) to service the interrupt, moving a byte from the peripheral to memory or from memory to the peripheral, or even just acknowledging that the interrupt has been sensed. In a well designed architecture much the same sequence of events will take place when an instruction turns up an error condition; information will be saved showing not only what error was found but the address of the instruction in which it occurred. This makes it possible to acquaint the user with an exact error message and enough information about its position to work out how it occurred (in a high-level language program) and so correct it.

Proper architectures will also distinguish between a user program and the operating system, both in the instructions the user program may execute and in the areas of memory it may read or write.

In the 32000 series this distinction is made by one bit in the PSR: the U bit. When this is set, the CPU is said to be in user mode and certain instructions, and some forms of other instructions, cannot be used. This effectively reserves these instructions for the supervisor program preventing any user program changing sensitive system registers or, by mistake, overwriting the supervisor or its private memory. The stack plays a large part in 32000 programming and, consequently, there is a private stack for the

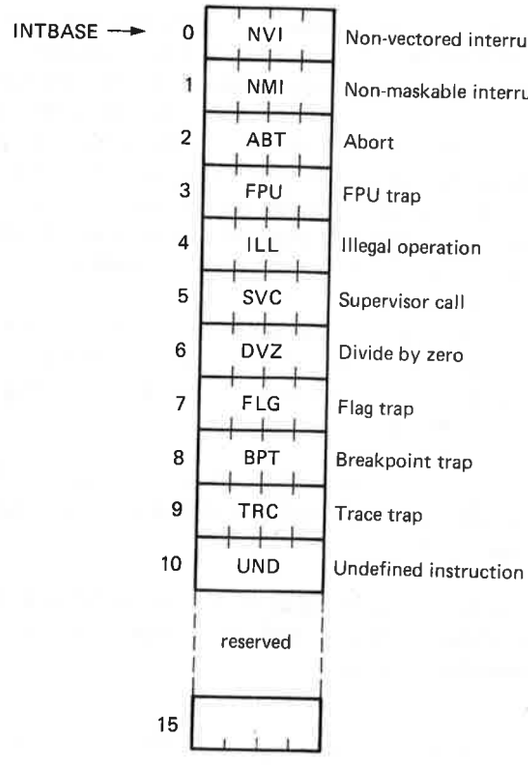

Fig. 10.1 Interrupt dispatch table.

supervisor and its interrupt routines to use. This is managed by having two stack pointers, SP0 and SP1, chosen by the state of the S bit in the PSR. If this bit is on, SP1 is used in any stack operation; if it is off, SP0 is used. It is usual for the supervisor to have SP0, leaving SP1 for the user. When changing from user to supervisor state, as the U bit is changed from 1 to 0 so is the S bit.

10.2 EXCEPTIONS

In the 32000 vernacular, both interrupts and errors are called exceptions, and may be divided into two parts – external and internal. Interrupts are external exceptions as they are caused by external events – a peripheral signalling that it has a character ready for the CPU or that it is ready to receive another character or that it has completed a transfer into memory. Internal exceptions, which include errors, are known as traps, and arise either from fault conditions sensed during the execution of an instruction or from the use of special instructions to voluntarily relinquish control to the supervisor.

All exceptions are dealt with in much the same way by the 32000 CPUs. The first step is an adjustment of the program counter, the PSR or the stack pointer depending on the type of the exception and whether a string instruction

(the only type of instruction which can be interrupted) is in progress. Next the PSR is saved on the stack and then a vector number is read and used to extract an external procedure descriptor (the same as in the link table) from the interrupt dispatch table and an external procedure call made with it.

The address of the interrupt dispatch table is kept in the INTBASE register which allows the supervisor to change to a different set of service routines with no more than a single instruction to alter the address in INTBASE, and the reservation of a block of addresses in low memory whether they are used or not is no longer necessary. The dispatch table may therefore be anywhere in memory.

The dispatch table has the form shown in Fig. 10.1. Each slot is 32 bits wide and at an increasing offset from INTBASE. All the exceptions except for the first two, the non-vectored interrupt and the non-maskable interrupt, are traps. These occur for the following reasons.

ABT: instruction abort trap. An exception has been found by the MMU; the MMU's status register must be looked at to find the cause of the abort (see the later section on the MMU).

FPU: FPU exception. The FPU status register must be looked at to find the cause. The FPU status register is laid out as shown in Fig. 10.2, with TT showing the reason for the trap:

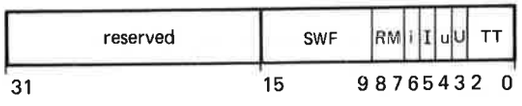

Fig. 10.2 FPU status register.

1. underflow (causes a trap only if bit 3 is set) – this trap is taken if a floating point result falls between the smallest (magnitude) normalized number and zero;
2. overflow – the result is too large to be represented as a floating point number of the precision requested;
3. divide by zero – this trap occurs when a *non-zero* value is to be divided by zero;
4. illegal instruction – an instruction, in the right format to be a floating point one, is not recognized; a possible opcode but not implemented;
5. invalid operation – there are two reasons for this trap, a reserved operand (positive or negative infinity, a denormalized number or Not-a-Number) has been used in a floating point operation (note that a `MOVf` instruction with a reserved operand does not cause a trap) or *both* operands of a `DIVf` instruction are zero;

6. inexact result (causes a trap only if bit 5 is set) – this occurs when the result of an operation, which may be either a floating point or an integer, must be rounded to fit the result format. This trap is taken only if no other error has occurred.

Bit 3 (U) enables the underflow trap (TT = 1) and bit 5 (I) enables the inexact result trap (TT = 6). Bit 4 (u) is set whenever an underflow occurs whether bit 3 is set or not and remains set until the user resets it; similarly, bit 6 is set by the first inexact result, independently of the state of the trap enable bit, and must also be reset by the user.

The FSR register can be set or copied by

```
LFSR   src    ;load FSR from src
SFSR   dest   ;store FSR into dest
```

where the FSR is loaded or stored as a 32-bit object. To set or clear single bits, it can be brought into a general register and the bit operations used on it before it is returned. Multiple bits can be set or cleared with the logical instructions. The `EXTSi` instruction can be used to extract the TT field from the copy.

ILL: illegal operation. An instruction has attempted to access a privileged register or one of the privileged instructions has been encountered when in user mode. The privileged registers are the PSR (but only if the top byte is accessed), the INTBASE register, the CFG register and all the MMU registers; these registers are accessed by:

```
LPRi   src    ;copies src to the register
SPRi   dest   ;copies the register to dest
```

The `i` in these instructions determines how much of the register is affected; in user mode, both `lprb` and `sprb` may be used with PSR as only the lower byte of the PSR, the user PSR, is affected – it is the `lprw` and `sprw` forms which are privileged. Either of these instructions applied to the INTBASE or the CFG register is privileged. The next section on the supervisor has a fuller description of the use of these instructions.

The MMU registers are accessed by a similar pair of instructions:

```
LMR    mmureg, src    ;copy src to MMU reg
SMR    mmureg, dest   ;copy MMU reg to dest
```

These are both privileged; the MMU and its registers are discussed in a later section.

The privileged instructions are:

```
LPRi   procreg, src    ;if procreg is ...
SPRi   procreg, dest   ;... INTBASE or PSR
```

```
bicpsrw   src          ;W form, ...
bispsrw   src          ;... B not privileged
setcfg    cfglist
RETT    constant
RETI
lmr    mmureg, src
smr    mmureg, dest
MOVSUi    src, dest
MOVUSi    src, dest
rdval    loc
wrval    loc
```

All these instructions are discussed in this chapter.

SVC: a supervisor call instruction was executed; this is used as a way into the kernel, to use a kernel routine. It will usually have one or more parameters which are either passed in registers or follow the `svc` code byte in memory to be picked up by the kernel using the return address passed to it as part of the trap sequence.

DVZ: a divide by zero trap caused by the `src` operand of one of the instructions `DEIi`, `DIVi`, `MODi`, `QUOi` or `REMi` being zero.

FLG: a `flag` instruction found the F bit set to 1; this is used deliberately as a way of causing a trap on integer overflow.

BPT: a `bpt` instruction was executed; the program is under the control of a debugger and a breakpoint was requested by the user on the instruction just about to be executed. Note that this doesn't imply the presence of a debugging system, just that, if there was one, this is the way it would set breakpoints.

TRC: an instruction was executed while the trace bit (bit 1) in the PSR was set. This can also be used by a debugging system to step through instructions singly, displaying the results of each instruction after it is completed. To do this requires a trace trap service routine and code to print the registers, decode the PSR and so on. In the old days, each instruction had to be examined to see if it was a branch instruction and, if it was, to change the branch target to somewhere in the monitor so as not to lose control. The trace bit and trap make this obsolete overnight; whatever the instruction, control passes to the trace trap when it has finished.

UND: an undefined instruction has been found. This instruction code has either not been implemented or it is an instruction which must be executed by the FPU or MMU and the appropriate bit in the CFG register has not been set.

The non-maskable interrupt (NMI) is taken when the CPU's NMI pin is signalled. This interrupt is used for very high priority signals such as might be caused by a power failure. There can be a few milliseconds before the CPU's supply voltage falls to the point where it fails to work properly and, if the system is so designed, vital information can be sent to non-volatile memory.

The 32000 series supports up to 256 levels of interrupt, so that interrupts can be given priorities and those outside influences which require speedier handling can be given a higher priority interrupt than the others. These additional levels of interrupt require one or more NS32202 Interrupt Control Units (ICUs) to be connected between the interrupt signals from the outside and the CPU. If the system has no ICU then one maskable interrupt is available reacting to signals on the CPU's INT pin. The legend 'maskable' refers to the control exercised by the I bit (bit 11) in the PSR: if this bit is 1, then the CPU will recognize signals on the INT pin; if it is 0 these will be ignored.

The different types of exception are acted on by the CPU in the following priority order:

- traps, except the trace trap
- non-maskable interrupt
- maskable interrupt
- trace trap

Since traps other than the trace trap do not occur together there is no need for a priority within the traps other than that shown above by giving the lowest priority to the trace trap.

The actions performed by the CPU on recognizing an exception differ slightly between interrupts and traps. Interrupts take place either when an instruction has completed (note, instructions are *not* interrupted) or, for string instructions only, when a cycle of the instruction is complete. As string instructions can take almost any desired length of time (they may be any practical desired length), it was thought unfair to keep everybody waiting and they were given fixed points in their cycle where interrupts could be recognized. In each of the three string instruction types, the last act in the cycle of actions on a particular element is to decrement `R0`, the count of the number of elements remaining. It is just after this point that interrupts may be recognized. If an interrupt is pending, the P bit in the PSR is cleared (stopping a trace trap taking place immediately on returning from the interrupt) and the return address (the address at which the interrupted code will restart) is set to the first byte of the string instruction.

After the little point of string/non-string instruction has been dealt with, and bits U and S cleared in the PSR thus changing to supervisor mode and interrupt stack, interrupts are masked (bit I cleared) so that the interrupt routine will not itself be interrupted and a copy of the PSR is put on to the (interrupt) stack as a 16-bit value. The MOD register is pushed after the PSR, also as a 16-bit value, the return address (the address of the next instruction unless a string instruction was interrupted) put on the stack and the external procedure descriptor read from the dispatch table. The correct entry in the dispatch table to look at is given by a 'vector' byte, which has the value 0 for a non-vectored interrupt and 1 for a non-maskable interrupt. If an ICU is present, the vector to use is read from `:fffe00` (just 512 bytes from the magic

16 Mbyte end of memory), some provision being made by the hardware to have an appropriate byte available there. The correct entry in the dispatch table is given by:

```
INTBASE + vector*4
```

The action after this is that of a `cxp` instruction setting up the registers for a new module: the module part of the descriptor is copied into the MOD register, the SB register is set from the entry in the module table and the PC is set from the sum of the descriptor offset and the module entry program base.

The action for traps is similar but some details differ. Traps other than the trace trap set the return address to the beginning of the instruction just completed, so an earnest enquirer can find the instruction in which the exception occurred: the trace trap must not do this, of course, otherwise it would never get anywhere. The vectors used to find the entry in the dispatch table are those marked against the entries in the diagram above.

Again, since they are errors, traps other than the trace trap restore the user stack pointer to its value before the trapped instruction started. The CPU's design goes to some trouble to make sure that the trace trap (which is handled after interrupts and all the other traps) is correctly dealt with. As the trace bit can be set from a user program (not much use unless there is provision for the dispatch table descriptor to be made to point to the user's procedure), the user would be a bit confused if all the other exceptions appeared too.

The stack, on entry to an exception routine looks like Fig. 10.3. The PSR is unchanged from the value before the exception and the return address, for traps, will be that of the instruction causing the trap, with the exception of the trace trap when it is the next instruction.

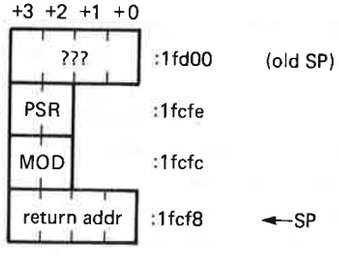

Fig. 10.3 Stack layout on entry to an exception routine.

Exception routines must end with a `RETT` instruction, the `RETI` instruction being used only for maskable interrupts when the system includes an ICU. `RETI` sends a special acknowledgement to the ICU which it needs to keep its house in order. The format of the `RETT` instruction is

```
RETT    constant
```


with `constant` being 0 for interrupt routines. If `constant` is non-zero, it will increase the stack pointer by its value allowing traps to pass parameters to the service routine on the stack in a similar way to the use of `RET` in procedures. The stack pointer affected will be the one selected by the PSR *after* the PSR on the interrupt stack has been restored. This facility would really only be useful for an SVC trap where, to be of any use at all, some indication of the purpose of the call must be made available to the kernel.

10.3 SUPERVISOR

This section will not go into the implementation of a kernel or supervisor (unfortunately space forbids), but will be content to describe the tools for building one provided by the 32000 CPUs.

The first task of a supervisor is to initialize the system – in the case of the 32000: initialize the dedicated registers, start the clock (no supervisor can work properly without a clock) and start the command interpreter. It is then in a state where the user can start work.

Initialization starts on receipt of a reset signal on the CPU's RST pin. This can be arranged to occur automatically when power is switched on, and on a number of systems there is also a Reset button for use in dire emergency. The 32201 Timer Control Unit, which no self-respecting 32000 system would be without, has got the necessary circuitry in it to provide satisfactory reset signals to the CPU and other components.

When the CPU recognizes a reset signal, it clears the PC, PSR and FSR registers which (PC) causes the first instruction to be sought at address `:000000`, puts the CPU (PSR) into supervisor mode, selects the interrupt stack and disables interrupts; it also clears all the user flags including the trace flag. Thus the start-up code must be found at address zero after a reset. This can be resident there, in an EPROM, or an EPROM can be arranged to switch in on reset and be switched out again when the system is running. Very often, the start-up program will do little more than read in another larger initialization program from disk, which will then set up the initial module table and the dedicated registers.

The processor registers are set up (or examined) by

```
LPRi    cpureg, src     ;copy src into CPU reg
SPRi    cpureg, dest    ;copy CPU reg into dest
```

where `cpureg` can be

UPSR	user PSR, just the low byte of the PSR
FP	frame pointer
SP	stack pointer
SB	static base register
PSR	both user and supervisor bytes of the PSR

INTBASE interrupt base register
MOD module register

If LPRi is used with UPSR, whatever operand length is demanded by i, only the low byte of the PSR will be affected and the operation is not privileged. Only the symbol SP is used for both SP0 and SP1; the choice of the register is made according to the state of the S bit in the PSR at the time. If either PSR or INTBASE is used as an operand, the instruction is privileged whatever the operand length is according to i.

Setting the initial values for both the interrupt stack and the user stack can be done by manipulating the S bit in the PSR (in supervisor mode, of course). There are two instructions, both in two lengths, which can be used to do this, as well as alter any other bits required. They are:

```
BISPSRi   src   ;set bits in PSR
BICPSRi   src   ;clear bits in PSR
```

In the form bispsrb and bicpsrb, they act on the user byte of the PSR alone and are not privileged; to change the S bit, they must be used in the forms bispsrw and bicpsrw, both of which are privileged. The operand, which has access class 'read' and may therefore be a constant, is a string of 16 (or 8) bits, the 1 bits determining which of the PSR bits are to be set or cleared. To set the S bit, bit 9 or the second bit in the upper byte you could use

```
bispsrw   =:20   ;set S bit for SP1
```

and to clear it

```
bicpsrw   =:20   ;clear S bit for SP0
```

The last register that needs to be set is the configuration register which shows whether an FPU or an MMU is available – if the appropriate bits are not set, and floating point or memory management instructions are used, an undefined trap will be caused. This register also has a bit showing whether an ICU is in use; if this bit is set, the maskable interrupts are vectored and their service routines must end with RETI instead of RETT. The instruction used, which is privileged, is

```
setcfg   cfglist
```

with cfglist being one or more of the letters i, f, m or c in brackets (c stands for Custom Slave Processor – a chip specially designed for the system with special instructions). To show that the system contains an FPU and an MMU but no ICU or custom slave, the instruction is

```
setcfg   [f, m]
```


Note that the NatSemi assembler does require brackets around the operands here and that the absence of FPU, MMU, ICU and slave processor is shown by empty brackets.

Supervisors, at the level invited by the 32000 series which is that of minis and mainframes, allow several different pieces of code to be executed seemingly simultaneously while not interfering with one another. These different pieces of code are called processes and in executing them a means of controlling access to data or code is needed so that while one process is accessing it another can be made to wait. This means is provided by the CBITi and SBITi instructions described in Chapter 3. Assuming that an area of memory has a lock on it controlled by the state of bit 0 of the byte variable lock, claiming the resource can be done by:

```
lock    dcb   0

        ...

        sbitb   =0, lock   ;check lock
        bfs   locked       ;already in use
```

The sbitb instruction sets the lock bit to 1 and copies the original setting into the flag bit of the PSR. If this is set, the process can be diverted by the bfs instruction and the bit is not changed, the other process still being in control. If the bit is clear, the process has already locked it and can go on to make use of it. No instructions other than string instructions can be broken into by an interrupt, so the whole operation will always be completed even if a clock interrupt occurs in the middle of it. The only problem with this arises if more than one CPU is using the same area in memory. In this case, though the operation is not interrupted, between the CPU cycles reading the state of the bit and writing a 1 into it, another CPU could have sneaked a write cycle to memory and changed it. To guard against this, there are versions of both SBITi and CBITi which activate the ILO (interlocked operation) pin on the CPU during the instruction and prevent other CPUs accessing memory until the interlocked operation is complete – if the hardware is so designed.

A supervisor has a number of tasks which it performs in an order determined by its scheduling algorithm. When all its tasks are done, there must still be something for it to do (the so-called backstop); doing nothing except keeping the supervisor busy until another task is ready to be activated. The 32000 provides something for this, an instruction called simply wait: execution stops until an interrupt starts one of the service routines. On return from the interrupt, execution starts at the instruction following wait; a null task could be a branch back to wait.

There are no input/output instructions as such in the 32000 series as peripherals are expected to communicate with the CPU by pretending to be a piece of memory. Peripherals will be attached to the system through various special-purpose chips with control and input/output registers which can be

read from or written to via system determined addresses. With the enormous memory space of the 32000 this sidesteps the question of how many input/output ports are needed and seems a simpler way to do it.

10.4 MEMORY MANAGEMENT

The NS32082 Memory Management Unit (MMU) is a 32000 co-processor working as closely with the CPU as the floating point unit. Its purpose, however, is to implement demand paged virtual memory – once the exclusive preserve of the mainframes and minis, now available on micros.

10.4.1 Virtual memory

In a 32000 system there may only be a few hundred kilobytes of memory physically present. Under normal circumstances this would limit the size of programs to be run on the system or the amount of data they could deal with unless a complicated overlay system was written for it. The MMU changes all this. No longer must the programmer struggle with overlays and their attendant bugs; it is all done by hardware.

The means used is similar to overlays; blocks of memory are exchanged with other blocks stored on disk, but, from the program's point of view, it is all in memory, loaded into the 16 Mbyte made available by a kindly manufacturer.

In virtual memory systems the blocks of memory are called pages and the 32000 defines pages as 512 bytes in length. The pages in memory are no longer addressed by the physical page they may be currently occupying but by a virtual address which bears no relation to the physical address. To get from the virtual to the physical, the CPU constructs a set of tables which the MMU uses to translate the virtual addresses sent to it by the CPU.

Of course, if a program uses more pages of virtual memory than there are physical pages available, there will be times when the MMU gets an address which it can see, from the tables, does not correspond to a physical page. This situation is called, somewhat pejoratively, a page fault.

At this point the MMU interrupts (ABT trap) the CPU which returns the registers to the state they were in when the interrupted instruction started and then calls the trap service routine. This routine will arrange to bring the page required into physical memory from disk, if necessary displacing a page already there; the CPU will now restart the instruction.

It might seem that the use of disk storage as an extension to memory would slow programs down to an intolerable degree, but experience has shown that, for comparatively long periods of time, programs use only a small set of pages and, while slightly slower than a program contained entirely in physical memory, the advantages of a large but virtual memory considerably outweigh the disadvantages.

This balance does rely on the size of the working set of pages being a

reasonable proportion of the physical pages available. Once the working set exceeds the number of physical pages to such an extent that the CPU is spending most of its time waiting for pages to be swapped, program execution is reduced to the speed of rocks weathering.

10.4.2 Address translation

The NS32082 supports a 16 Mbyte virtual address space; the NS32382, which has all sorts of goodies like support for an off-chip cache, will support the 4 Gbyte address space offered by the NS32332 and its successors.

Address translation takes place in two stages or levels: the first level page table contains up to 256 entries of 32 bits, each entry pointing to a second level pointer table; the pointer table has at most 128 entries, again of 32 bits, with each entry pointing to a physical page.

The 24-bit virtual address is divided into three fields (Fig. 10.4). `index1` is the number of the entry in the page table containing the address of a pointer table; `index2` is the number of an entry in the pointer table addressed by `index1` containing the number of a physical page. The final physical address is made up of the 15-bit page number in bits 9 to 23 and the 9-bit offset in bits 0 to 8. The only fields translated are `index1` and `index2`; `offset` remains the same, denoting the same byte in the physical page as it did in the virtual one.

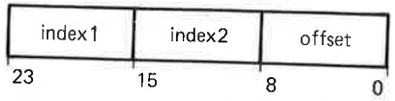

Fig. 10.4 Virtual address format.

If every virtual address needed two memory accesses (which may take an average of 20 clock cycles) to translate, the program would be slowed to a crawl with the overhead. To reduce the delay, the MMU has a 32-entry associative translation cache on the chip which can match a virtual address in one clock cycle. (The 32332 does away with even this delay by overlapping the translation cycle with the following data bus cycle.)

The cache contains the right entry around 98% of the time; when it doesn't, it must get the entry from memory, replacing one of the entries in the cache. The entry to be replaced is chosen to be that which has been least recently used...an algorithm which has been used to decide which page in memory to replace on a page fault but is made more efficient by being implemented in hardware.

The format of a page table entry (the same format is used for pointer tables as well) is shown in Fig. 10.5.

The M ('modified') bit is set when the page is in physical memory and is written to: if this page is chosen to be replaced by a page on disk needed by the CPU, it must be written back on to the disk to make sure that, if it is required again, the program isn't given an out-of-date copy.

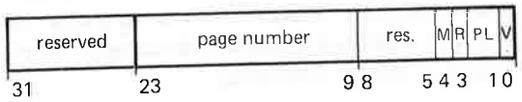

Fig. 10.5 Page table entry format.

The R bit is set whenever the page is referenced, either read or written. By periodically clearing this bit in both pointer and page tables, the operating system can work out which pages are frequently used so that it can swap out an infrequently used one instead.

The V bit, when set, shows that the page corresponding to the entry is in memory. If the V bit is clear in an entry the MMU tries to use for translation, it will abort the attempt, alerting the CPU to the need for a page swap. Note that page tables (not only pointer tables) have a V bit, allowing pointer tables to be kept on disk as well as program and data pages. This makes for a very flexible arrangement, with a section of program or even a complete utility permanently 'loaded' in memory and being swapped in with its pointer tables only when required.

The two PL bits implement protection on a page-by-page basis. To allow the supervisor overall access, the effect of the bits is different in the two modes (see Table 10.1).

Table 10.1 Effect of PL bits.

Protection level	*User mode*	*Supervisor mode*
00	no access	read only
01	no access	full access
10	read only	full access
11	full access	full access

10.4.3 Debugging with the MMU

The MMU provides two debugging operations: breakpointing and flow tracing.

Implementing breakpoints in a virtual memory system can cause a lot of problems. In testing supervisor code the breakpoints must be set at physical addresses which, as all addresses from the CPU are virtual, is impossible to do by conventional means. Again, getting a trap to occur at the desired instruction by overlaying it with code, while fine for non-virtual systems, entails a lot of overhead for a memory manager.

For these and other reasons, the designers of the MMU decided to build debugging support into the hardware. The implementation is based on two registers, BPR0 and BPR1; an MMU breakpoint will be caused when the address in either of these registers matches that on the address bus. Two addresses are

provided so that both destinations of a conditional branch can be breakpointed.

Several different types of match can be chosen: the addresses can be matched against either virtual addresses or translated physical addresses; the source of the address can be on an instruction fetch, on being written to or read from or any combination of these. It is also possible, by setting the Breakpoint Count register, to select how many times the address match with BPR0 occurs before the trap is taken – useful in stopping halfway through a loop. As this is all done in hardware, the effect on program execution times is negligible, running times will be much the same whether debugging or not – an important point when a bug bites after several minutes of execution.

Flow tracing provides a history of the last few instructions executed. Looking at the values of variables at the time the error makes itself felt is fine and dandy but not always sufficient. There are four registers in the MMU which take care of flow tracing: two program flow registers, PF0 and PF1, and two 16-bit sequential count registers, SC0 and SC1.

The MMU keeps a count of the number of instructions executed since the last non-sequential instruction and the address of the instruction in these registers. When a non-sequential instruction (branch, jump, call, return or interrupt) is met, PF0 is copied into PF1 and the address of the current instruction is put into PF0. SC0 is now copied into SC1 and SC0 is cleared. SC0 is incremented for every sequential instruction performed. At any time, therefore, the addresses of the last two non-sequential instructions are in PF0 and PF1 with the counts of sequential instructions in between held in SC0 and SC1.

Up to now, to get this level of debugging, you would have to get a logic analyser – now it is all part of the system.

10.4.4 The MMU registers

The main MMU register is the status register, MSR; this, like all the other registers, can be copied into a double word in memory by

```
smr   mmureg, dest
```

and set from a double word in memory by

```
lmr   mmureg, src
```

The set of acceptable symbols for mmureg is given in Appendix A under the instruction lmr.

The MSR has the format shown in Fig. 10.6 and the bits are set as follows.

- ERC, the error class field (bits 2 to 0), is set on an MMU exception to show the cause. Bit 0 is set on an address translation error. Bit 2 is set on an MMU breakpoint and cleared on a non-sequential trace interrupt (cf. bit 25, NT). Bit 1 is unused.

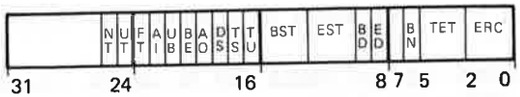

Fig. 10.6 MMU status register format.

- TET is the translation error flag: bit 3 is set on a protection level error (cf. PL bits in page table entry), bit 4 on an invalid level 1 page table entry and bit 5 on an invalid level 2 pointer table entry (cf. V bit in page table entry).
- BN is set to show whether the breakpoint address in `BPR0` (clear) was matched or that in `BPR1` (set).
- ED is the error data direction bit: if it is 1 the address translation error was caused by an attempt to read the contents of the address; otherwise the instruction was attempting to write to it.
- BD, the breakpoint direction, is similar to ED but shows how the breakpoint occurred. If BD is set, it occurred on an attempt to read from the address; otherwise it was a write operation.

Table 10.2 Bus status settings for EST and BST.

Bus status	*Operation*
`000`	The CPU is reading the next sequential word from the instruction stream.
`001`	Non-sequential instruction fetch: the CPU is performing the first fetch of an instruction after a branch, jump, call, return, interrupt, etc.
`010`	Data transfer: the CPU is reading or writing an operand.
`011`	Read `rmw` operand: the CPU is reading an operand which will be modified and rewritten.
`100`	Read for effective address calculation: the CPU is reading an address from memory to use in memory relative or external address mode.
`101`	Transfer slave processor operand: the CPU is transferring an operand to or from the FPU, MMU or custom slave processor or issuing an instruction to it.
`110`	Read slave processor status: an FPU, MMU or custom slave processor instruction is complete.
111	Broadcast slave ID: the CPU is initiating an FPU, MMU or custom slave processor instruction.

- EST is the error status flag and BST is the breakpoint status flag. EST is set to the three lower bits of the bus status on an address translation error and BST is set similarly on an MMU breakpoint. The values of interest are shown in Table 10.2. and this information, combined with the knowledge that a breakpoint or address translation error has occurred, will give the reason for the trap and the operand which caused it.
- TU is the translate user bit: when it is set, all addresses in user mode will be taken to be virtual addresses.
- TS is the translate supervisor bit: when set, all addresses in supervisor mode will be taken as virtual addresses. The operating system kernel will run in supervisor mode but must use physical addresses – on entry to the kernel this bit must be cleared. There are layers above the kernel which may well be allotted fixed addresses, permanently 'loaded' and kept, with their pointer table, on disk. Before calling one of these, the kernel would re-enable supervisor virtual addressing.
- DS is the dual space bit: if this bit is clear, the page table pointed to by `PTB0` will be used for both supervisor and user virtual addresses; otherwise `PTB0` will be used for supervisor addresses and `PTB1` for user addresses. This makes handling multiple processes very easy as the kernel simply needs to change `PTB1` before returning to user mode and also gives it a bit more flexibility in laying out supervisor and user address spaces. VAX/VMS allocates the upper 2 Gbytes of its address space to the supervisor and the lower 2 Gbytes to the user; that is, they share the same address space but partition it. With the dual space bit the supervisor and user can have the same addresses but achieve separation by having two different page tables. Designer's choice.
- AO is the access override bit: when set, a program in user mode may access all addresses, even those normally only accessible by the supervisor.
- BEN is the breakpoint enable bit: when set, it enables the MMU's breakpoint mechanism using `BPR0` and `BPR1`.
- UB is the user break bit: when set, breakpoints are enabled in user mode only. It is ignored when BEN is clear.
- AI is the abort or interrupt bit.
- FT if the flow trace bit: when set, flow tracing is enabled.
- UT is the user trace bit: when set, flow tracing takes place in user mode only. It is ignored if FT is clear.

- NT is the non-sequential trace bit: when set, the MMU interrupts the CPU (via NMI) on a branch, jump, call or return instruction. This can be used to implement an entry and exit trace for procedures.
- EIA, the error/invalidate address register, has two purposes. The CPU can put a virtual address into it to remove the corresponding entry in the MMU cache. It would need to do this when it had swapped out a page and had changed the page table, to make sure the MMU copy is changed as well. It is also used to hold the virtual address which caused an address translation error. Bit 31 of the EIA is set if PTB1 has the level 1 page table address which was used and is cleared if PTB0 was used. The cause of the error is given in the ERC and TET fields of the MSR.

The breakpoint registers, BPR0 and BPR1, have the layout shown in Fig. 10.7. Bits 26 to 31 determine which addresses will be compared to the breakpoint address (bits 0 to 32) and under what conditions the MMU will signal a break. The bits are as follows:

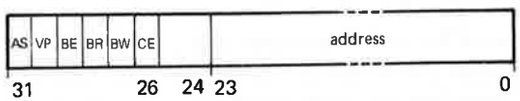

Fig. 10.7 MMU breakpoint register format.

- AS is the address space bit: if 0, the address will be compared against virtual addresses from the page table pointed at by PTB0; otherwise addresses from the PTB1 page table will be used.
- VP is the virtual/physical bit: if 0, the MMU will make the comparison with virtual addresses; otherwise the translation will be made with translated virtual addresses.
- BE is breakpoint execution: if set, the breakpoint is taken when the instruction at the address is executed. The first byte of the instruction must have the address for the breakpoint to occur.
- BR is breakpoint read: if set, the breakpoint will be taken if data is read from the breakpoint address.
- BW is breakpoint write: if set, the breakpoint will be taken on a write to the breakpoint address or when it is *read* as the first part of a read – modify – write operation.
- CE is the counter enable: this bit only has any effect in BPR0; when set, the breakpoint count register (BCNT) is decremented when the address and condition bits in BPR0 cause a break. If the result after decrementing is zero, the breakpoint will be taken; otherwise it will be ignored.

10.4.5 Finally

The MMU is a very complex chip and it has only been possible to sketch some of its uses here. A full discussion of its programming would require half a book rather than half a chapter.

The description of the MMU instructions

```
lmr     mmureg, src
smr     mmureg, dest
rdval   loc
wrval   loc
MOVSUi  src, dest
MOVUSi  src, dest
```

will be found in Appendix A.

APPENDIX A

32000 instruction reference

A.1 ADDRESSING MODES

There are 31 distinct addressing modes in nine groups:

- **Register** The operand is either a general register or a floating point register depending on the instruction and the operand. The register contains the operand and the length is given by the option letter on the mnemonic. Note that if this mode is used with access class address, the register is held to contain the *address* of the operand, not the operand itself. Access classes are discussed in Section A.3.
- **Register relative** The operand is written `disp(Rn)` where `n` is 0 to 7 and `disp` is a signed displacement (at present) in the range −16777215 to +16777215. The `disp` field is usually an expression involving a symbol. The register is taken to contain an address and the operand location is the sum of this address and `disp`. It can be thought of as an indirect address plus an offset.
- **Memory relative** This has three forms represented by `(disp2(disp1(br)))` where `br` may be the frame pointer (FP), the stack pointer (SP) or the static base register (SB). In action, the double word at `br+disp` is taken to be an address and `disp2` is added to it to get the location of the operand. An extremely useful mode, it allows access to an array of pointers which may then have an offset added to them to get the operand address.
- **Immediate** The operand is the integer or floating point constant given. This constant is stored in the instruction itself and, as it is a constant, it can only be combined with the 'read' access class. It may not be used as the base mode for scaled indexing.
- **Absolute** This is represented as `@disp` with `disp` a displacement representing an absolute address.
- **External** This mode is usually hidden under a reference to an imported or exported symbol reference. It accesses an entry in the link table and adds an offset to it.
- **Top of stack** This is represented by the reserved symbol TOS. Depending on the access class, the operand may be pushed on to the stack (write), popped from the stack (read) or neither (rmw).

- **Memory space** There are four modes in this group. One is `$+disp` (`*+disp` for the NatSemi assembler) addressing `disp` bytes from the current address in the program counter where `disp` is signed. The other three are of the form `disp(br)` where `br` is the frame pointer (FP), the stack pointer (SP) or the static base register (SB). The signed integer `disp` is added to the address in the register to get the location of the operand.
- **Scaled indexing** The general form is `basemode[Rn:i]` where `Rn` is one of `R0` to `R7` and `i` is `b` for scaling by 1, `w` to scale by 2, `d` to scale by 4 or `q` to scale by 8. The address given by the addressing mode expression `basemode` is indexed by the contents of `Rn` scaled by 1, 2, 4 or 8 depending on the element length qualifier. The addressing mode used for `basemode` may be any of the modes except immediate or scaled index mode. Note that in being applied with scaled indexing, the access class of `basemode` is changed to address with the changes to register and top of stack modes this entails.

A.2 MNEMONIC OPTIONS

The instruction mnemonics are presented in a general form in which the constant part of the mnemonic may be followed by an operand length option or condition code. There are two types of length option, one for integer operands and one for floating point operands. Some of the floating point instructions have one integer and one floating point operand and in their general form they will have both an integer and a floating point length option. The letter used for the integer length option is `i` and may be replaced by `b` to indicate a byte operand, `w` to indicate a word operand or `d` for a double word; the floating point length option is `f` and may be replaced by `F` for single precision floating point operand or `L` for double precision. The `MEIi` and `DEIi` instructions have a length option `2i` indicating that the operand is *twice* the length chosen.

Only two instructions have a condition code option; `Bcond` (branch on condition) and `Scondi` (save condition). In each case the letters which may be substituted for `cond` are given in the description of the instruction.

A.3 OPERAND TYPES

Most operands are general, coded `gen`, and may be given in any one of the allowable addressing modes. These operands also have an access class and an implied length.

The access classes used in the descriptions are

- `read` The operand is read only; an immediate mode operand can be used only with this access class. If top-of-stack mode is used, the stack pointer is post-incremented by the length of the operand, thus popping it.
- `write` The operand is written only; immediate mode may not be used. If top-of-stack mode is used, the stack pointer is pre-decremented by the length of the operand, thus pushing it.

- `rmw` The operand is read, modified and rewritten back in the original location; immediate mode may not be used with this access class. If top-of-stack mode is used the stack pointer is not changed; the operand is not popped and then pushed back again.
- `addr` The operand is treated as an address; if register mode is used, the register is assumed to hold the address of the operand not the operand itself. For this access class, the only difference between register mode and register relative is that register relative can add a displacement to the address in the register before it is used. Immediate mode may not be used with this access class. If top-of-stack mode is used the operand at the stack pointer is taken to be an address; the stack pointer is not changed. The operand addressed may be read, written or both read and written depending on the instruction being executed.
- `regaddr` Register/address: this class is only used to describe the base operand in the bit and bit field instructions. If base is the name of a general register, the bit or bit field is in the register; otherwise base denotes a byte from which, using offset, the bit or start of the bit field is found. Immediate mode may not be used with this access class. Like `addr`, the operand eventually accessed may be read, written or both read or written depending on the instruction.

The access class and implied length are usually combined in the description line as `rmw.i` or `read.f`.

Other operands do not have an addressing mode and are described by the following names.

- `reg` The operand is one of the general registers `R0` to `R7` and any one of these may be specified. The whole register is always read or read and written by the instruction.
- `quick` The operand is a signed value in the range −8 to 7; before use it is sign extended to the implied operand length given by the length option chosen for the mnemonic.
- `short` The operand is 4 bits long and its use will be given in the instruction description.
- `imm` The operand is a single byte immediate value; its use is given in the description of the instruction.
- `disp` The operand is an immediate integer value which is stored as part of the instruction: it comes in three sizes, 1, 2 and 4 bytes holding signed values rather less than the full range for the integer, 1 or 2 bits being set aside as a size code. The 1 byte `disp` can hold a value in the range −64 to +63, the word has a range of −8192 to 8191 and the double word goes from −16777215 to +16777215 (assembler limited). Note that this is only a 24-bit value; values outside this range are undefined at present.

A.4 DESCRIPTIONS

The general form of the instruction mnemonic appears against the left-hand margin and, if there are any, the expanded mnemonics are shown against the right-hand margin. The operands follow the general mnemonic using names to indicate their purpose. The second line gives the operand types, showing which addressing modes can be used or whether the operand is a constant. The abbreviations used are: `disp`, displacement; `gen`, general; `imm`, immediate; `reg`, register. The third line shows the operand's access class if it is not a constant, and following this there is a description of the action performed, the flags affected and the traps that may be caused.

```
ABSf    src,    dest                    absf
        gen     gen                     absl
        read.f  write.f
```

Action *Absolute value floating*: forces a positive sign on the floating point `src` operand and puts this into `dest`.

Flags No PSR flags: if `src` is a reserved operand, the TT field of the FSR will be set; otherwise it will be zeroed.

Traps FPU trap if `src` is a reserved operand; the TT field in the FSR will be set appropriately.

```
ABSi    src,    dest                    absb
        gen     gen                     absw
        read.i  write.i                 absd
```

Action *Absolute value*: puts the absolute value of the `src` operand into `dest`.

Flags F is set if `src` is the largest negative number for the size of integer; −128 for bytes, −32768 for words or −2147483648 for double words.

Traps None.

```
ACBi    inc,    index,  dest            acbb
        quick   gen     disp            acbw
                rmw.i                   acbd
```

Action *Add, compare and branch*: the sign extended `inc` value is added to the `index` operand with the sum being left in `index`. If the sum is not zero, the instruction branches to `dest`; otherwise execution continues at the instruction following `ACBi`.

Flags None.

Traps None.

```
ADDf    src,    dest        addf
        gen     gen         addl
        read.f  rmw.f
```

Action *Add floating*: the floating point `src` operand is added to `dest` and the result placed in `dest`.

Flags No PSR flags. UF in the FSR is set if an underflow occurs; IF is set on an inexact result and TT will be set to reflect any exception.

Traps UND trap if the F bit in the CFG is not set. FPU trap set on a floating point exception.

```
ADDi    src,    dest        addb
        gen     gen         addw
        read.i  rmw.i       addd
```

Action *Add*: the `src` operand and `dest` are added and the result placed in `dest`.

Flags C is set if there is a carry; F is set if there is an overflow.

Traps None.

```
ADDCi   src,    dest        addcb
        gen     gen         addcw
        read.i  rmw.i       addcd
```

Action *Add with carry*: the sum of `src`, `dest` and the carry flag is put into `dest`.

Flags C is set on a carry; F is set on overflow.

Traps None.

```
ADDPi   src,    dest        addpb
        gen     gen         addpw
        read.i  rmw.i       addpd
```


Action *Add packed decimal*: the sum of `src`, `dest` and the carry flag is put into `dest`. Each decimal digit is represented as the values 0 to 9 in a 4-bit nybble.

Flags C is set on a carry; F is cleared.

Traps None.

```
ADDQi   src,    dest        addqb
        quick   gen         addqw
                rmw.i       addqd
```

Action *Add quick integer*: the sum of `src` (sign extended to the length of `dest`) and `dest` is placed into `dest`.

Flags C is set on a carry; F is set on overflow.

Traps None.

```
ADDR    src,    dest
        gen     gen
        addr    write.D
```

Action *Compute effective address*: the address of `src` is put in `dest` as a double word. If `src` is an imported or exported procedure name, its descriptor from the link table is put into `dest` rather than its address.

Flags None.

Traps None.

```
ADJSPi  src             adjspb
        gen             adjspw
        read.i          adjspd
```

Action *Adjust stack pointer*: `src` is *subtracted* from the stack pointer, increasing the length of the stack if `src` is positive and decreasing it if `src` is negative. The `src` operand is sign extended to 32 bits before being used; the entire stack pointer is modified whatever the implied length of the operand.

Flags None.

Traps None.

```
ANDi    src,    dest                    andb
        gen     gen                     andw
        read.i  rmw.i                   andd
```

Action *And*: the bitwise AND of the `src` and `dest` operands is put into `dest`.

Flags None.

Traps None.

```
ASHi    count,  dest                    ashb
        gen     gen                     ashw
        read.B  rmw.i                   ashd
```

Action *Arithmetic shift*: `dest` is shifted by `count` bits with the sign bit being copied into vacated positions for a right shift and zeros being copied in for a left shift. The `count` operand is taken to be signed and the shift will be to the left if it is positive and to the right if negative; `count` must be in the range −7 to +7 for `ashb`, −15 to +15 for `ashw` and −31 to +31 for `ashd`.

Flags None.

Traps None.

```
BCOND   dest
        disp
```

Action *Conditional branch*: the instruction branches to `PC+disp` (PC is the address of the first byte of the instruction) if the condition is true; otherwise execution continues with the next sequential instruction. The conditions are:

`EQ`	Equal	Z flag set
`NE`	Not equal	Z flag clear
`CS`	Carry set	C flag set
`CC`	Carry clear	C flag clear
`HI`	Higher	L flag set
`LS`	Lower or same	L flag clear
`GT`	Greater than	N flag set
`LE`	Less than or equal	N flag clear
`FS`	Flag set	F flag set
`FC`	Flag clear	F flag clear
`LO`	Lower	Z and L flags clear
`HS`	Higher or same	Z or L flag set

`LT`	Less than	Z and N flags clear
`GE`	Greater than or equal	Z or N flag set

The conditions corresponding to the flag settings may seem a little perverse but they have been chosen so that in the code

```
cmpb    a, b
bgt     exceeds
```

the branch will be taken when `a>b`.

Flags None.

Traps None.

```
BICi    src,    dest                    bicb
        gen     gen                     bicw
        read.i  rmw.i                   bicd
```

Action *Bit clear*: the bits in `dest` corresponding to 1 bits in `src` are cleared.

Flags None.

Traps None.

```
BICPSRB src
        gen
        read.B
```

Action *Bit clear in PSR*: the bits in the user PSR (the low byte) corresponding to 1 bits in `src` are cleared. This is not a privileged instruction.

Flags Any flags corresponding to 1 bits in `src` are cleared.

Traps None.

```
BICPSRW src
        gen
        read.W
```

Action *Bit clear in PSR*: the bits in the whole PSR corresponding to 1 bits in `src` are cleared.

Flags Any flags corresponding to 1 bits in `src` are cleared.

Traps The illegal operation trap (ILL) is activated if the PSR U bit is set (user mode).

```
BISPSRB     src
            gen
            read.B
```

Action *Bit set in PSR*: the bits in the user PSR (the low byte) corresponding to 1 bits in `src` are set. This instruction is not privileged.

Flags Any flags corresponding to 1 bits in `src` are set.

Traps None.

```
BISPSRW     src
            gen
            read.W
```

Action *Bit set in PSR*: the bits in the whole PSR corresponding to 1 bits in `src` are set.

Flags Any flags corresponding to 1 bits in `src` are set.

Traps The illegal operation trap (ILL) is activated if the PSR U bit is set (user mode).

```
BPT
```

Action *Breakpoint trap*: this instruction activates the breakpoint trap (BPT); the return address on the stack is the address of the `bpt` instruction itself.

Flags None.

Traps The breakpoint trap is activated.

```
BR          dest
            disp
```

Action *Branch*: this instruction transfers control to `PC+dest` where PC is the address of the instruction itself.

Flags None.

Traps None.

```
BSR         dest
            disp
```

Action *Branch to subroutine*: this instruction transfers control to `PC+dest` where PC is the address of the instruction itself. It also pushes the address of the next sequential instruction on to the stack. The subroutine should return control with a `RET` instruction.

Flags None.

Traps None.

```
CASEi       src                                  caseb
            gen                                  casew
            read.i                               cased
```

Action *Case branch*: this instruction transfers control to `PC+src` where PC is the address of the instruction itself. The `src` operand is sign extended to 32 bits before use.

Flags None.

Traps None.

```
CBITi       offset, base                         cbitb
            gen     gen                          cbitw
            read.i  regaddr                      cbitd
```

Action *Clear bit*: the bit designated by `base` and `offset` is set to 0 after copying it to the F flag.

Flags F is set to the original value of the designated bit.

Traps None.

```
CBITIi      offset, base                         cbitib
            gen     gen                          cbitiw
            read.i  regaddr                      cbitid
```

Action *Clear bit interlocked*: the bit designated by `base` and `offset` is set to 0 after copying it to the F flag. The Interlocked Operation pin on the CPU is activated during this instruction and can be used to prevent another CPU modifying the bit while the instruction is in progress.

Flags F is set to the original value of the bit.

Traps None.

```
CHECKi    dest,    bounds,  src        checkb
          reg      gen      gen        checkw
                   addr     read.i     checkd
```

Action *Bounds check*: this instruction checks the `src` operand against the values in the bounds operand, setting F if `src` is outside the bounds. The instruction then subtracts the lower bound from `src`, putting the result as a zero extended 32-bit value into the `dest` register.

Flags F is set if `src` is out of bounds.

Traps None.

```
CMPf      src1,    src2        cmpf
          gen      gen         cmpl
          read.f   read.f
```

Action *Compare floating*: `src1` is compared with `src2` and the PSR Z and N flags are set to show the result (`src2-src1`). Positive and negative zero are taken to be equal.

Flags Z is set if `src1` and `src2` are equal; N is set if `src1`> `src2`; L is always cleared. The TT field of FSR is set to denote any floating point exception conditions.

Traps The undefined instruction trap is activated if the F bit in the CFG is clear. The FPU trap is activated if a floating point exception occurs.

```
CMPi      src1,    src2        cmpb
          gen      gen         cmpw
          read.i   read.i      cmpd
```

Action *Compare*: `src1` is compared with `src2` and the PSR Z, N and L bits set according to the result (`src2-src1`).

Flags Z is set if `src1` is equal to `src2`; N is set if `src1`>`src2` (signed comparison); L is set if `src1`>`src2` (unsigned comparison).

Traps None.

```
CMPMi     block1,  block2,  length     cmpmb
          gen      gen      disp       cmpmw
          addr     addr                cmpmd
```

Action *Compare multiple*: the contents of `block1` and `block2` are compared and the Z, N and L flags set according to the result.

Flags Z is set if every integer in `block1` is equal to the corresponding integer in `block2`; N is set if, for the first pair of unequal integers, the integer from `block1` is greater than the integer from `block2` (signed comparison); L is set under the same circumstances as N but the comparison is unsigned.

Traps None.

```
CMPQi     src1,    src2        cmpqb
          quick    gen         cmpqw
                   read.i      cmpqd
```

Action *Compare quick integer*: the `src1` operand (sign extended to the length of `src2`) is compared with `src2` and the Z, N and L flags set according to the result (`src2-src1`).

Flags Z is set if `src1` is equal to `src2`; N is set if `src1`>`src2` (signed comparison); L is set if `src1`>`src2` (unsigned comparison).

Traps None.

```
CMPSi     options     cmpsb
CMPST     options     cmpsw
                      cmpsd
```

Action *Compare strings*: the operands are in registers `R0` to `R4`:

`R0`	number of elements
`R1`	address of string1 element
`R2`	address of string2 element
`R3`	address of the translation table (`cmpst` only)
`R4`	match value (Until or While option)

The options are `b` (backward) – the addresses will be decremented; `u` – the comparison will continue until an element of `string1` matching the contents of `R4` is encountered; `w` – the comparison will continue while the elements from `string1` match the value in `R4`. The `cmpst` instruction will assume that the string elements are bytes and, before comparison, the `string1` element (as an unsigned byte value) will be added to the

translation table address and the byte at that address used for the comparison: the `string2` element is not translated.

Flags The Z, N and L flags are changed; the F flag is set if the instruction ended as a result of a while or until match. To check the result of the comparison, the flags should be looked at in the following order:

1. If the until or while option has been selected, check the F flag. If it is set, the instruction has ended because of an until or while match and the other flags are set to Z = 1, N = 0 and L = 0. `R1` contains the address of the `string1` element causing termination, `R2` is the address of the corresponding `string2` element and `R0` is the number of elements not yet compared *including* the one causing termination.
2. If F = 0, check the Z flag. If this is set, the instruction has exhausted the strings and they are identical. `R1` and `R2` hold the address of the element following the last element in `string1` and `string2`, `R0` is 0 and the N and L flags are both clear.
3. After the two tests above, this point is reached if the strings are unequal. `R0` has the number of remaining elements, including the element which compared unequal, and `R1` and `R2` hold the addresses of the two unequal elements. If N is set, the element in `string1` is greater than the element in `string2` (signed comparison); if L is set, the element in `string1` is greater than that in `string2` (unsigned comparison).

Traps None.

```
COMi    src,    dest                    comb
        gen     gen                     comw
        read.i  write.i                 comd
```

Action *Complement*: the one's complement of `src` is put into `dest`.

Flags None.

Traps None.

```
CVTP    offset, base,   dest
        reg     gen     gen
                addr    write.D
```

Action *Convert to bit pointer*: the absolute offset (offset from bit 0 of byte 0 in memory) of the bit designated by `base` and `offset` is put into `dest` as a 32-bit value.

Flags None.

Traps None.

```
CXP     index
        disp
```

Action *Call external procedure*: a procedure in another module is called via a descriptor in the current module's link table. The operand must be an imported procedure name and will be represented by the index of the entry in the link table.

Flags None.

Traps None.

```
CXPD    desc
        gen
        addr
```

Action *Call external procedure with descriptor*: a procedure in another module is called via a descriptor in the current module's memory space.

Flags None.

Traps None.

```
DEIi    src,    dest                    deib
        gen     gen                     deiw
        read.i  rmw.2i                  deid
```

Action *Divide extended integer*: the double length `dest` operand is divided by the single length `src` operand and the (single length) quotient is put into the high-order integer of `dest` with the remainder going into the low-order one. If the `dest` operand is a register, it must be given as the even register of an even – odd register pair with the even register holding the low-order integer of `dest` and the odd register the high-order integer. The remainder will go into the even register and the quotient into the following odd register.

Flags None.

Traps The divide by zero (DVZ) trap will be activated if `src` is zero.

```
DIVf    src,    dest                    divf
        gen     gen                     divl
        read.f  rmw.f
```

Action *Divide floating*: `dest` is divided by `src` with the quotient going into `dest`.

Flags No PSR flags; in the FSR, UF is set if an underflow occurs, IF is set if the result is inexact and the TT field is set to show any floating point exception.

Traps The undefined instruction trap (UND) is activated if the F bit in the CFG is clear; the floating point trap (FPU) will be activated if there is a floating point exception. If `src` is zero there will be a floating point divide by zero exception and if both `src` and `dest` are zero there will be an invalid operation exception.

```
DIVi    src,    dest                    divb
        gen     gen                     divw
        read.i  rmw.i                   divd
```

Action *Divide*: `dest` is divided by `src` and the result, rounded to the next lower or more negative integer, is put into `dest`.

Flags None.

Traps A divide by zero exception will occur if `src` is zero.

```
ENTER   reglist, constant
        imm     disp
```

Action *Enter new procedure context*: this instruction creates a frame on the stack which can hold local variables for the procedure. The `constant` operand gives the number of bytes of local storage to allocate; the `reglist` operand gives the names of the general registers to be saved. The contents of the frame pointer on entry are saved in the frame and the pointer is set to point to the old value, thus linking the stack frames together and giving a reference point to access parameters and local variables.

Flags None.

Traps None.

```
EXIT    reglist
        imm
```


Action *Exit procedure context*: this instruction is the converse of `enter`. It removes the frame constructed by `enter`, restoring the general registers given in the `reglist` operand and the old value of the frame pointer.

Flags None.

Traps None.

```
EXTi    offset, base,   dest,   length          extb
        reg     gen     gen     disp            extw
                regaddr write.i                 extd
```

Action *Extract field*: the bit field designated by `base`, `offset` and `length` is copied to the `dest` operand. The field is right justified in `dest` which has its high-order bits zero filled if the field is too short. If the field is longer than `dest`, the field's high-order bits are discarded. The `offset` operand is taken to be a 32-bit signed integer; the `length` operand must have a value in the range 1 to 32 and the bit field must lie within 4 bytes.

Flags None.

Traps None.

```
EXTSi   base,   dest,   offset, length          extsb
        gen     gen     imm.....                extsw
        regaddr write.i                         extsd
```

Action *Extract field short*: the bit field designated by `base`, `offset` and `length` is copied to the `dest` operand. The field is right justified in `dest` which has its high-order bits zero filled if the field is too short. If the field is longer than `dest`, the field's high-order bits are discarded. The `offset` and `length` operands are coded together as a single byte. This limits the `offset` to the range 0 to 7. The `length` operand must have a value in the range 1 to 32 and the bit field must lie within 4 bytes.

Flags None.

Traps None.

```
FFSi    base,   offset                          ffsb
        gen     gen                             ffsw
        read.i  rmw.B                           ffsd
```

Action *Find first set bit*: the instruction searches the `base` operand for a 1 bit

starting at the bit given by `offset`. The search ends at the first 1 bit or the end of the integer. If a 1 bit is found, `offset` is set to the number of the bit and F is cleared; otherwise F is set and `offset` is zeroed. The initial value of `offset` must be in the range 0 to 7 for `ffsb`, 0 to 15 for `ffsw` and 0 to 31 for `ffsd`.

Flags F is set if no 1 bit is found and cleared otherwise.

Traps None.

`FLAG`

Action *Trap on flag*: the flag trap (FLG) is activated if the F bit is set.

Flags None.

Traps The flag trap is activated if F is set.

```
FLOORfi  src,     dest                    floorfb floorlb
         gen      gen                     floorfw floorlw
         read.f   write.i                 floorfd floorld
```

Action *Floor floating to integer*: the `src` operand is rounded towards the integer less than or equal to it (towards negative infinity) and the result put into `dest` as a signed integer.

Flags No PSR flags. The FSR flag IF is set on an inexact result and the TT field will be set on an exception.

Traps The undefined trap (UND) will be activated if the F bit in the CFG is not set and the FPU trap will be activated on a floating point exception; in particular, the overflow exception; if the resulting integer is too big for `dest`.

```
IBITi    offset,  base                   ibitb
         gen      gen                    ibitw
         read.i   regaddr                ibitd
```

Action *Invert bit*: the bit designated by `base` and `offset` is copied into the PSR F bit and then it is inverted.

Flags F is set to the original value of the designated bit.

Traps None.

```
INDEXi   accum,   length,  index          indexb
         reg      gen      gen             indexw
                  read.i   read.i          indexd
```

Action *Calculate index*: this instruction performs the calculation

```
accum*(length+1)+index
```

one step in the iterative calculation of the index for a multidimensional array. The result of the calculation is put into the `accum` operand; the `length` operand is the difference of the upper and lower bounds for the current dimension (one less than the dimension's length) and `index` is the zero adjusted index for the current dimension. The `length` and `index` operands are unsigned integers and are zero extended to 32 bits before use; `accum` is taken to be an unsigned 32-bit integer.

Flags None.

Traps None.

```
INSi     offset,  src,    base,   length   insb
         reg      gen     gen     disp     insw
                  read.i  regaddr          insd
```

Action *Insert field*: the `src` operand is inserted into the bit field designated by `base`, `offset` and `length`. The operand is right justified in the field which, if the operand is shorter than the field, is zero extended to the right, and, if longer, is truncated on the right. `offset` is taken as a 32-bit signed integer; `length` must be in the range 1 to 32 and the bit field must lie within 4 bytes.

Flags None.

Traps None.

```
INSSi    src,     base,    offset,  length  inssb
         gen      gen      imm.....         inssw
         read.i   regaddr                   inssd
```

Action *Insert field short*: the `src` operand is inserted into the bit field designated by `base`, `offset` and `length`. The operand is right justified in the field which, if the operand is shorter than the field, is zero extended to the right, and, if longer, is truncated on the right. `offset` and `length` are coded in a byte, limiting `offset` to the range 0 to 7; `length` must be in the range 1 to 32 and the bit field must lie within 4 bytes.

Flags None.

Traps None.

```
JSR     dest
        gen
        addr
```

Action *Jump to subroutine*: this instruction transfers control to the location in the current module's memory space designated by the general operand `dest` which allows the jump to be indirect via a pointer. It also pushes the address of the following instruction on to the stack. The subroutine should return control with the `ret` instruction.

Flags None.

Traps None.

```
JUMP    dest
        gen
        addr
```

Action *Jump*: this instruction transfers control to the location in the current module's memory space designated by the general operand `dest` which allows the jump to be indirect via a pointer.

Flags None.

Traps None.

```
LFSR    src
        gen
        read.D
```

Action *Load floating point status register (FSR)*: the double word `src` operand is put into the FSR.

Flags No PSR flags. All FSR flags are affected.

Traps The undefined instruction trap (UND) occurs if the F bit in the CFG is clear.

```
LMR     mmureg, src
        short   gen
                read.D
```


Action *Load memory management register*: the double word `src` operand is copied into the MMU register given by `mmureg`. This may be:

`BPR0`	breakpoint register 0
`BPR1`	breakpoint register 1
`PF0`	program flow register 0
`PF1`	program flow register 1
`SC`	sequential count register
`MSR`	MMU status register
`BCNT`	breakpoint count register
`PTB0`	page table base register 0
`PTB1`	page table base register 1
`EIA`	error/invalidate address register

Flags None.

Traps The undefined instruction trap (UND) occurs if the M bit in the CFG register is clear. The illegal instruction trap (ILL) occurs if the U flag in the PSR is set (user mode).

```
LPRi    procreg, src                    lprb
        short    gen                    lprw
                 read.i                 lprd
```

Action *Load processor register*: the `src` operand is copied to the given CPU register. `procreg` may be given as:

`UPSR`	user PSR (low byte)
`FP`	frame pointer
`SP`	stack pointer
`SB`	static base pointer
`PSR`	processor status register
`INTBASE`	interrupt base register
`MOD`	module register

In registers other than PSR, the high-order bits are zero filled if `src` is shorter than the register. SP is interpreted as SP0 or SP1 according to the state of the S bit in the PSR; if UPSR is given as the register, only the low byte of the PSR is affected whatever the length of `src`.

Flags All PSR flags are affected if the register is PSR; only the N, Z, L, T and C flags are affected if the register is UPSR or PSR is used with `lprb`. Flags are not affected otherwise.

Traps The illegal operation trap occurs if the PSR U flag is set and PSR or INTBASE is the register.

```
LSHi    count,  dest        lshb
        gen     gen         lshw
        read.B  rmw.i       lshd
```

Action *Logical shift*: dest is shifted by count bits, vacated bits being filled by zeros. If count is positive the shift is to the left, if it is negative the shift is to the right. count must lie in the range −7 to +7 for lshb, −15 to +15 for lshw and −31 to +31 for lshd.

Flags None.

Traps None.

```
LXPD    src,    dest
        gen     gen
        addr    write.D
```

Action *Load external procedure descriptor*: this is not an instruction in its own right but a synonym for addr. The src operand must refer to an entry in the link table.

Flags None.

Traps None.

```
MEIi    src,    dest        meib
        gen     gen         meiw
        read.i  rmw.2i      meid
```

Action *Multiply extended integer*: this instruction multiplies src and the lower half of dest, putting the (double length) result into dest. The dest operand may be a pair of general-purpose registers in which case the integer in the even numbered register of the pair will be multiplied by src and the result will go into the even numbered register and the next consecutive (odd numbered) register. Note that two registers are always used even if the integer length is byte or word; the two parts of the result are placed in the low end of each register, the high-order half of the double length result in the odd register and the low-order half in the even register. If the top of stack addressing mode is used for dest, space must already have been allocated in the stack to hold the entire result.

Flags None.

Traps None.

```
MODi    src,    dest        modb
        gen     gen         modw
        read.i  rmw.i       modd
```

Action *Modulus*: the remainder of dest DIV src is placed in dest; the division is performed according to the rites of DIVi (q.v.). The division is rounded to the nearest integer less than or equal to the exact result; the result always takes the sign of the src operand.

Flags None.

Traps A divide by zero trap (DVZ) occurs if src is zero.

```
MOVf    src,    dest        movf
        gen     gen         movl
        read.f  write.f
```

Action *Move floating point*: the src operand is copied to dest.

Flags No PSR flags. The FSR TT field is set to zero.

Traps An undefined instruction trap (UND) occurs if the F bit in the CFG register is clear.

```
MOVi    src,    dest        movb
        gen     gen         movw
        read.i  write.i     movd
```

Action *Move*: the src operand is copied to dest.

Flags None.

Traps None.

```
MOVif   src,    dest        movbf movbl
        gen     gen         movwf movwl
        read.i  write.f     movdf movdl
```

Action *Move converting integer to floating point*: the src (integer) operand is converted to a single or double precision floating point number and put into dest. If the integer is too large for the significand, it will be rounded according to the setting of the rounding mode bits in the FSR.

Flags No PSR flags. In the FSR, IF is set on an inexact result (integer too large) and the TT field will be set to show any exception.

Traps An undefined instruction trap occurs if the F bit in the CFG is clear; a floating point trap will occur on a floating point exception, if an IF trap is enabled, for example. This can occur in the `movdf` form as the significand is shorter than a double word; integers greater than +16777215 or smaller than −16777216 will give an inexact result.

```
MOVFL    src,    dest
         gen     gen
         read.F  write.L
```

Action *Move floating to long floating*: `src` is converted to long floating and put into `dest`.

Flags No PSR flags; the TT field of FSR is set to show any floating point exception conditions.

Traps An undefined instruction trap occurs if the F bit in the CFG is clear. A floating point trap will occur on a floating point exception.

```
MOVLF    src,    dest
         gen     gen
         read.L  write.F
```

Action *Move long floating to floating*: the long floating operand (`src`) is converted to short and put into `dest`. `src` is rounded, if necessary, according to the setting of the rounding bits in the FSR.

Flags No PSR flags; UF is set in the FSR if an underflow occurs, IF is set on an inexact result and the TT field is set to show any exception condition.

Traps An undefined instruction trap occurs if the F bit in the CFG is clear; a floating point trap occurs on a floating point exception. This may be overflow, if the long value is outside the range of short floating point, or underflow, if enabled and `src` is closer to zero than short floating point can go, or inexact result if enabled and `src`'s significand must be shortened.

```
MOVMi    block1, block2, length          movmb
         gen     gen     disp            movmw
         addr    addr                    movmd
```

Action *Move multiple*: the contents of `block1` are copied to `block2`; `length` is the number of integers in the block and must be in the range 1 to 16 for byte integers, 1 to 8 for words and 1 to 4 for double words.

Flags None.

Traps None.

```
MOVQi    src,    dest            movqb
         quick   gen             movqw
                 write.i         movqd
```

Action *Move quick integer*: the `src` operand is sign extended to the length of `dest` and put in `dest`.

Flags None.

Traps None.

```
MOVSi    options                 movsb
                                 movsw
                                 movsd
MOVST    options
```

Action *Move strings*: the operands are in registers `R0` to `R4`:

`R0`	number of elements
`R1`	address of string1 element
`R2`	address of string2 element
`R3`	address of the translation table (`movst` only)
`R4`	match value (until or while option)

The options are `b` (backward) – the addresses will be decremented; `u` – `string1` will be moved element by element to `string2` until an element of `string1` matching the contents of `R4` is encountered; `w` – `string1` will be moved element by element to `string2` while the elements from `string1` match the value in `R4`. The `movst` instruction will assume that the string elements are bytes and, before being moved, the `string1` element (as an unsigned byte value) is added to the translation table address and the byte at that address moved to `string2` instead.

Flags The F flag is set if the instruction ended as a result of a while or until match.

Traps None.

```
MOVSUi   src,    dest            movsub
         gen     gen             movsuw
         addr    addr            movsud
```

Action *Move value from supervisor to user space*: the src operand in the supervisor space is moved to dest in the user space.

Flags None.

Traps An undefined instruction trap occurs if the M bit in the CFG is clear; an illegal operation trap occurs if the PSR U flag is set.

```
MOVUSi    src,     dest                    movusb
          gen      gen                     movusw
          addr     addr                    movusd
```

Action *Move value from user to supervisor space*: the src operand in user space is copied to dest in the supervisor space.

Flags None.

Traps An undefined instruction trap occurs if the M bit in the CFG is clear; an illegal operation trap occurs if the PSR U flag is set.

```
MOVXBD    src,     dest
          gen      gen
          read.B   write.D
```

Action *Move byte with sign extension to double word*: the byte at src is copied to the low byte of dest and then sign extended to fill it.

Flags None.

Traps None.

```
MOVXWD    src,     dest
          gen      gen
          read.W   write.D
```

Action *Move word with sign extension to double word*: the word at src is copied to the low word of dest and then sign extended to fill it.

Flags None.

Traps None.

```
MOVXBW    src,     dest
          gen      gen
          read.B   write.W
```


Action *Move byte with sign extension to word*: the byte at src is copied to the low byte of dest and then sign extended to fill it.

Flags None.

Traps None.

```
MOVZBD    src,     dest
          gen      gen
          read.B   write.D
```

Action *Move byte with zero extension to double word*: the byte at src is copied to the low byte of dest and then zero extended to fill it.

Flags None.

Traps None.

```
MOVZWD    src,     dest
          gen      gen
          read.W   write.D
```

Action *Move word with zero extension to double word*: the word at src is copied to the low word of dest and then zero extended to fill it.

Flags None.

Traps None.

```
MOVZBW    src,     dest
          gen      gen
          read.B   write.W
```

Action *Move byte with zero extension to word*: the byte at src is copied to the low byte of dest and then zero extended to fill it.

Flags None.

Traps None.

```
MULf      src,     dest                    mulf
          gen      gen                     mull
          read.f   rmw.f
```

Action *Multiply floating*: the product of src and dest is put into dest.

Flags No PSR flags: UF in FSR is set on an underflow, IF is set on an inexact result and TT is set to show an exception.

Traps An undefined instruction trap occurs if the F bit in the CFG is clear; a floating point trap occurs on a floating point exception.

```
MULi    src,    dest                    mulb
        gen     gen                     mulw
        read.i  rmw.i                   muld
```

Action *Multiply*: the product of src and dest is put into dest; if the product is too long, the high-order bits are truncated.

Flags None.

Traps None.

```
NEGf    src,    dest                    negf
        gen     gen                     negl
        read.f  write.f
```

Action *Negate floating*: the sign bit of src is complemented and the result put into dest.

Flags No PSR flags: TT in FSR is set to show any exception.

Traps An undefined instruction trap occurs if the F bit in the CFG is clear. A floating point trap occurs on a floating point exception.

```
NEGi    src,    dest                    negb
        gen     gen                     negw
        read.i  write.i                 negd
```

Action *Negate*: the two's complement of src is formed by subtracting it from zero and is put into dest.

Flags C is set on a borrow from the subtraction which always occurs except when src is zero. F is set on overflow which occurs when the largest negative integer of the given size is negated. The result on overflow is the original value of src.

Traps None.

```
NOP
```

Action *No operation*: control is passed to the following instruction with no operation performed.

Flags None.

Traps None.

```
NOTi    src,    dest                    notb
        gen     gen                     notw
        read.i  write.i                 notd
```

Action *Complement boolean*: src (considered as a boolean) has its least significant bit inverted. The value 1 becomes 0 and vice versa.

Flags None.

Traps None.

```
ORi     src,    dest                    orb
        gen     gen                     orw
        read.i  rmw.i                   ord
```

Action *OR*: a bitwise logical OR is performed on the src and dest operands and the result put in dest.

Flags None.

Traps None.

```
QUOi    src,    dest                    quob
        gen     gen                     quow
        read.i  rmw.i                   quod
```

Action *Quotient*: the dest operand is divided by src returning the nearest integer whose absolute value is less than or equal to the absolute value of the exact result. This result is placed in dest.

Flags None.

Traps A divide by zero (DVZ) trap occurs if src is zero.

```
RDVAL    loc
         gen
         addr
```

Action *Validate address for reading*: the F bit in the PSR is set if the user mode location `loc` is protected against reading.

Flags F is set if the location cannot be read.

Traps An undefined instruction trap occurs if the M bit in the CFG is clear; an illegal instruction trap occurs if the U bit in the PSR is set.

```
REMi     src,     dest                    remb
         gen      gen                     remw
         read.i   rmw.i                   remd
```

Action *Remainder*: `dest` is divided (using `QUOi`) by `src` and the remainder put into `dest`. The result always has the sign of `dest` unless zero which is always positive.

Flags None.

Traps A divide by zero trap (DVZ) occurs if `src` is zero.

```
RESTORE  reglist
         imm
```

Action *Restore general-purpose registers*: the registers given in `reglist` are restored from the stack. No check can be made that the registers in `reglist` are on the stack or that the stack pointer points to saved registers, so it had better be right.

Flags None.

Traps None.

```
RET      constant
         disp
```

Action *Return from subroutine*: a return address is taken from the stack and the stack also reduced by `constant` bytes, removing parameters put there before the procedure was called. Execution continues at the return address. This instruction should only be used to return from procedures called by either `bsr` or `jsr`.

Flags None.

Traps None.

```
RETI
```

Action *Return from interrupt*: this instruction returns control from an interrupt service routine for a vectored interrupt; it may only be used when the hardware includes an Interrupt Control Unit as it causes an End of Interrupt bus cycle informing the ICU that the service routine is ending.

Flags All flags are restored from the stack.

Traps An illegal instruction trap occurs if the U bit in the PSR is set.

```
RETT     constant
         disp
```

Action *Return from trap*: this instruction ends a trap or non-maskable or non-vectored interrupt service procedure; it must not be used to return from a vectored interrupt as notice of the service routine ending is not sent to the ICU. The `constant` operand is used to remove any parameters which may have been passed to the trap on the stack. The `svc` instruction is a likely candidate for this treatment.

Flags All flags are restored from the stack.

Traps An illegal instruction trap occurs if the U bit in the PSR is set.

```
ROTi     count,   dest                    rotb
         gen      gen                     rotw
         read.B   rmw.i                   rotd
```

Action *Rotate*: `dest` is rotated by `count` bits with the bits shifted off one end being moved to the vacated positions at the other. If `count` is positive the shift is to the left; if it is negative the shift is to the right. The `count` operand must be in the range −7 to +7 for `rotb`, −15 to +15 for `rotw` and −31 to +31 for `rotd`.

Flags None.

Traps None.

```
ROUNDfi   src,     dest                    roundfb roundlb
          gen      gen                     roundfw roundlw
          read.f   write.i                 roundfd roundld
```

Action *Round floating to integer*: `src` is rounded to an integer and the result put into `dest`. The rounding used is round to even and, if `src` is exactly halfway between two integers, `ROUNDfi` returns the even integer as the result.

Flags No PSR flags: IF in the FSR is set on an inexact result and the TT field shows any exception.

Traps An undefined instruction trap occurs if the F bit in the CFG is clear; a floating point trap occurs on a floating point exception. An overflow exception can occur if the integral part of `src` is too large for `dest`.

```
RXP       constant
          disp
```

Action *Return from external procedure*: this instruction returns control from a procedure called by `CXP`; `constant` is used to remove any parameters from the stack by adding the value to the stack pointer.

Flags None.

Traps None.

```
SCONDI    dest                            Scondb
          gen                             Scondw
          write.i                         Scondd
```

Action *Save condition as boolean*: `dest` is set to 1 if the condition is true and 0 if it is false. The conditions are the same as for `Bcond` and have the same interpretation.

Flags None.

Traps None.

```
SAVE      reglist
          imm
```

Action *Save general-purpose registers*: the registers given in `reglist` are pushed on to the stack.

Flags None.

Traps None.

```
SBITi     offset,  base                   sbitb
          gen      gen                    sbitw
          read.i   regaddr                sbitd
```

Action *Set bit*: the bit designated by `base` and `offset` is copied into the F bit in the PSR and is then set to 1.

Flags F is set to the original value of the designated bit.

Traps None.

```
SBITIi    offset,  base                   sbitib
          gen      gen                    sbitiw
          read.i   regaddr                sbitid
```

Action *Set bit interlocked*: the bit designated by `base` and `offset` is copied into the F bit in the PSR and then set to 1. During this instruction, the Interlocked Operation pin on the CPU is activated to allow other CPUs in a multiprocessor configuration to be stopped from accessing the same bit and changing it.

Flags F is set to the original value of the designated bit.

Traps None.

```
SETCFG    cfglist
          short
```

Action *Set configuration*: this instruction is used to set or clear bits in the configuration register so that the CPU knows whether to accept FPU and MMU instructions or not.

Flags None.

Traps An illegal instruction trap occurs if the U bit in the PSR is set.

```
SFSR      dest
          gen
          write.D
```

Action *Store floating point status register (FSR)*: the contents of the FSR are copied to the double word `dest`.

Flags None.

Traps An undefined instruction trap occurs if the F bit in the CFG is clear.

```
SKPSi    options                    skpsb
                                    skpsw
SKPST    options                    skpsd
```

Action *Skip string*: the operands are in registers `R0` to `R4`.

`R0`	number of elements
`R1`	address of string1 element
`R2`	not used
`R3`	address of the translation table (`skpst` only)
`R4`	match value (until or while option)

The options are: `b` (backward) – the addresses will be decremented; `u – string1` will be examined element by element until an element matching the contents of `R4` is encountered; `w – string1` will be examined element by element while the elements match the value in `R4`. The `skpst` instruction will assume that the string elements are bytes and, before being examined, the string byte is added (as an unsigned integer) to the translation table address and the byte at that address examined instead.

Flags The F flag is set if the instruction ended as a result of a while or until match.

Traps None.

```
SMR      mmureg, dest
         short   gen
                 write.D
```

Action *Store memory management register*: the contents of the MMU register specified by `mmureg` are put in the double word `dest`. The coding of `mmureg` is the same as for `lmr`.

Flags None.

Traps An undefined instruction trap occurs if the M bit in the CFG register is clear; an illegal instruction trap occurs if the U bit in the PSR is set.

```
SPRi     procreg, dest
         short   gen                sprb
                 write.i            sprw
                                    sprd
```

Action *Store processor register*: the contents of the dedicated CPU register `procreg` are stored in `dest`; high-order bits of the register are truncated if `dest` is shorter than the register and the high-order bits of `dest` are filled with zeros if the register is shorter than `dest`. The operand `procreg` is coded as for `LPRi`.

Flags None.

Traps An illegal instruction trap occurs if U in PSR is set.

```
SUBf     src,    dest               subf
         gen     gen                subl
         read.f  rmw.f
```

Action *Subtract floating*: `src` is subtracted from `dest` and the result put into `dest`.

Flags No PSR flags: in the FSR, UF is set on an underflow, IF is set on inexact result and TT is set to show an exception.

Traps An undefined instruction trap occurs if the F bit in the CFG is clear; a floating point trap occurs on a floating point exception.

```
SUBi     src,    dest               subb
         gen     gen                subw
         read.i  rmw.i              subd
```

Action *Subtract*: `src` is subtracted from `dest` and the result put into `dest`.

Flags C is set on a borrow; F is set on overflow.

Traps None.

```
SUBCi    src,    dest               subcb
         gen     gen                subcw
         read.i  rmw.i              subcd
```

Action *Subtract with carry*: the sum of `src` and the C flag is subtracted from `dest`, and the result is put into `dest`.

Flags C is set on a borrow; F is cleared.

Traps None.

```
SUBPi     src,     dest                    subpb
          gen      gen                     subpw
          read.i   rmw.i                   subpd
```

Action *Subtract packed decimal*: the sum of `src` and the C flag is subtracted from `dest` and the result put into `dest`.

Flags C is set on a borrow; F is set on overflow.

Traps None.

`SVC`

Action *Supervisor call*: this instruction causes the supervisor call trap; the return address pushed on to the stack is that of the `svc` itself.

Flags None.

Traps The supervisor call trap (SVC) occurs.

```
TBITi     offset,  base                    tbitb
          gen      gen                     tbitw
          read.i   regaddr                 tbitd
```

Action *Test bit*: the bit designated by `base` and `offset` is copied to the F bit in the PSR.

Flags F is set to the value of the bit.

Traps None.

```
TRUNCfi   src,     dest                    truncfb trunclb
          gen      gen                     truncfw trunclw
          read.f   write.i                 truncfd truncld
```

Action *Truncate floating to integer*: `src` is truncated to the nearest integer less than or equal to its absolute value (towards zero), the integer is put into `dest`.

Flags No PSR flags: IF in the FSR is set on an inexact result and TT is set to show any exception.

Traps An undefined instruction trap occurs if the F bit in the CFG is clear; a floating point trap occurs if there is a floating point exception. An overflow exception will occur if the integer is too large for `dest`.

`WAIT`

Action *Wait*: program execution is suspended until an interrupt occurs; the return address for the interrupt is the instruction following `wait`.

Flags None.

Traps None.

```
WRVAL     loc
          gen
          addr
```

Action *Validate address for writing*: the F flag is cleared if the address `loc` can be written to in user mode; otherwise the flag is set.

Flags F is set if `loc` is write protected.

Traps An undefined instruction trap occurs if the M bit in the CFG is clear; an illegal instruction trap occurs if the U bit in the PSR is set.

```
XORi      src,     dest                    xorb
          gen      gen                     xorw
          read.i   rmw.i                   xord
```

Action *Exclusive OR*: `src` is bitwise Exclusive ORed with `dest` and the result put into `dest`.

Flags None.

Traps None.

APPENDIX B

32000 instructions listed by function

B.1 INTEGER

Arithmetic

ADDi	src, dest	add
ADDQi	quick, dest	add quick integer
ADDCi	src, dest	add with carry
SUBi	src, dest	subtract
SUBCi	src, dest	subtract with carry
NEGi	src, dest	negate
ABSi	src, dest	absolute value
MULi	src, dest	multiply
MEIi	src, dest	multiply extended integer
DIVi	src, dest	divide
MODi	src, dest	modulus
QUOi	src, dest	quotient
REMi	src, dest	remainder
DEIi	src, dest	divide extended integer

Movement and conversion

MOVi	src, dest	move
MOVQi	quick, dest	move quick integer
MOVXBD	src, dest	sign extend byte to double
MOVXWD	src, dest	sign extend word to double
MOVXBW	src, dest	sign extend byte to word
MOVZBD	src, dest	zero extend byte to double
MOVZWD	src, dest	zero extend word to double
MOVZBW	src, dest	zero extend byte to word

Comparison

CMPi	src1, src2	compare
CMPQi	quick, src2	compare quick integer

B.2 PACKED DECIMAL

ADDPi	src, dest	add
SUBPi	src, dest	subtract

B.3 FLOATING POINT

ADDf	src, dest	add
SUBf	src, dest	subtract
MULf	src, dest	multiply
DIVf	src, dest	divide
NEGf	src, dest	negate
ABSf	src, dest	absolute value
CMPf	src1, src2	compare
MOVf	src, dest	move
MOVLF	src, dest	move long floating to floating
MOVFL	src, dest	move floating to long floating
MOVif	src, dest	move converting integer to floating point
ROUNDfi	src, dest	round floating to integer
TRUNCfi	src, dest	truncate floating to integer
FLOORfi	src, dest	floor floating to integer
LFSR	src	load floating point status register
SFSR	dest	store floating point status register

B.4 LOGICAL

Arithmetic

ANDi	src, dest	AND
ORi	src, dest	OR
BICi	src, dest	bit clear
XORi	src, dest	Exclusive OR
COMi	src, dest	complement

Shift

ASHi	count, dest	arithmetic shift
LSHi	count, dest	logical shift
ROTi	count, dest	rotate

Boolean

NOTi	src, dest	complement boolean
SCONDi	dest	save condition as boolean

B.5 BIT

TBITi	offset, base	test bit
SBITi	offset, base	set bit
SBITIi	offset, base	set bit interlocked
CBITi	offset, base	clear bit
CBITIi	offset, base	clear bit interlocked
IBITi	offset, base	invert bit
FFSi	base, offset	find first set bit
CVTP	offset, base, dest	convert to bit pointer

B.6 BIT FIELD

EXTi	offset, base, dest, length	extract field
EXTSi	base, dest, offset, length	extract field short
INSi	offset, src, base, length	insert field
INSSi	src, base, offset, length	insert field short

B.7 STRING

MOVSi	options	move string
MOVST	options	move byte string, translating
CMPSi	options	compare strings
CMPST	options	compare byte string, translating
SKPSi	options	skip string
SKPST	options	skip byte string, translating

B.8 BLOCK

MOVMi	block1, block2, length	move multiple
CMPMi	block1, block2, length	compare multiple

B.9 ARRAY

CHECKi	dest, bounds, src	bounds check
INDEXi	accum, length, index	calculate index

B.10 PROCESSOR CONTROL

Branches

JUMP	dest	jump
BCOND	dest	conditional branch
BR	dest	unconditional branch
CASEi	src	case branch
ACBi	quick, index, dest	add, compare and branch

Local procedure call/return

JSR	dest	jump to procedure
BSR	dest	branch to procedure
RET	const	return from procedure

External procedure call/return

CXP	index	call external procedure
CXPD	desc	call external procedure with descriptor
RXP	const	return from external procedure

Explicit traps

BPT		breakpoint trap
FLAG		trap on flag bit set
SVC		supervisor call

Trap/interrupt returns

RETT	const	return from trap
RETI		return from interrupt

B.11 PROCESSOR SERVICE

Effective address

ADDR	src, dest	compute effective address
LXPD	src, dest	load external procedure descriptor

Context instructions

SAVE	reglist	save registers
RESTORE	reglist	restore registers
ENTER	reglist, const	enter new procedure context
EXIT	reglist	exit procedure context

Register/stack manipulation

ADJSPi	src	adjust stack pointer
BICPSRB	src	bit clear in user PSR
BICPSRW	src	bit clear in PSR
BISPSRB	src	bit set in user PSR
BISPSRW	src	bit set in PSR
LPRi	procreg, src	load processor register
SPRi	procreg, dest	store processor register
SETCFG	cfglist	set configuration

Miscellaneous

NOP	no operation
WAIT	wait on interrupt

B.12 MEMORY MANAGEMENT

LMR	mmureg, src	load memory management register
SMR	mmureg, dest	store memory management register
RDVAL	loc	validate address for reading
WRVAL	loc	validate address for writing
MOVSUi	src, dest	move value from supervisor to user space
MOVUSi	src, dest	move value from user to supervisor space

APPENDIX C

Examples

C.1 INTRODUCTION

These example programs have been chosen to display the versatility of the NS32000 instruction set and the ease with which code can be constructed to perform a particular function.

The first example is a small string function taken from the C library. As well as showing how the string instructions can be used to perform quite complicated searching functions with minimal but clear code, it also serves as a real-life example of the layout of the stack for a procedure with parameters and local storage.

The second example is a utility and is mainly concerned with the use of Panos service routines for command line argument decoding, input and output (both to files and hardware devices) and copyable code using the Panos error handling routine.

A more elaborate procedure gives an example of a case jump and miscellaneous sections of useful code; the routine itself can be used as it is or extended quite easily. It is helpful in debugging assembler modules.

The last example is an interesting diversion.

The code has not just been copied from the source files into this text but has been interspersed with extended comments for which there was no convenient place in the source. These extended comments give background information on the choice of instructions, pitfalls avoided and the application of the program logic.

The routines are intended for 'owner use'; if they were to be used by the naïve, they would require more parameter checking. What checking has been included is there for the absent-minded knowledgeable user rather than as a safety net for incompetents.

C.2 MACROS

A number of macros have been written to automate the calls to the system routines used to display messages, strings and numbers and are extensively used in these examples. The details are not of any consequence as the routines called would be different running under other systems but some discussion of

the purpose of the macros will help in converting the programs or in understanding them better.

The macro `wrmsg` displays a constant string of characters on the screen. It expects the first byte of the string to be the count of bytes in the string. This is easily arranged by using the counted string form made available by the Acorn assembler.

The macro `wrstr` displays a string which has been returned by one of the Panos routines; this 'returned string' format has been standardized for convenience in writing the macros which return the string as well as the other macros which call routines taking the returned string as a parameter. The format is:

```
        dcd     0
string  dcb     maxlen
        allocb  maxlen
```

The address used as the string address is that of a byte containing the allocated length of the string (maximum length parameter when describing a string result). This is followed by the bytes of the string. The double word preceding the maximum length byte is for the actual string length returned by the routine.

The macros `wrcard` and `wrint` both call the string-to-integer routines and follow this with a call to `XBlockWrite` to print the resulting string. `wrcard` uses `CardinalToString` to do the conversion and `wrint` uses `IntegerToString`. The result string used for the string-to-integer calls is allocated within the macro (if so far undefined) in the static area.

The macro `newl` sends a CR–LF to the screen, again using `XBlockWrite` to do it. The CR–LF string is allocated by the macro if undefined so far.

The `stop` macro calls the system routine `Stop`: if no parameter is given, it returns a zero code to the system.

C.3 STRING SEARCH

This example is a function which is provided in the run-time libraries of some C compilers. The definition of the function is taken from *C: A Reference Manual* by Samuel P. Harbison and Guy L. Steele (Samuel P. Harbison and Guy L. Steele, Jr., *C: A Reference Manual*, © 1984, pp. 261–262. Adapted by permission of Prentice-Hall, Inc., Englewood Cliffs, NJ.).

The function takes two string parameters, `s` and `set`, and searches the string `s` for the first occurrence of any character in the string set. If a matching character is found, it returns a pointer to that character in `s`; otherwise it returns the null pointer.

All strings in C end with a zero byte as a terminator and the null pointer has the value zero; the pointer is assumed to be passed back to the calling program in `R0`.

The definition of the function is:

```
char    *strpbrk ( s, set )
        char *s, *set;
```

Assuming that the calling module uses `s` and `set` to label the first bytes of the appropriate strings, the function can be called by the following sequence:

```
addr    set, TOS
addr    s, TOS
importc strpbrk
cxp     strpbrk
```

The `importc` directive can be moved to the beginning of the calling module if desired.

The routine is presented as a module. Under normal circumstances, a module would not contain only one function but would include a group of similar functions to reduce the overhead of many separate modules and in the expectation of more than one of the group being used if one is requested.

The code starts:

```
module  CString
exportc strpbrk
```

The function name is called `strpbrk` (as suggested in the book) and the name must be exported to be able to satisfy linker requests. A static area for the module can be defined in which its personal tables and common variables can be kept without other modules being able to access them. This function does not need any as the table it uses is on the stack.

```
strpbrk
        enter   [R1,R2,R3,R4], 256
```

The function is kind to its callers; it saves the registers it uses. The `enter` instruction sets up a stack frame, allocates 256 bytes for the skip string translation table and saves the registers. The stack after `enter` is shown in Fig. C.1. This gives all the important addresses. Note that `table` seems to extend backwards; this is only because the stack lengthens into lower memory so that the last byte of `table` is 'lower' down the stack than the first byte. The table is used in the translation form of the skip string instruction. Each byte from `s` is translated using `table` before being compared against the value in `R4`. By setting all the bytes corresponding to the characters contained in `set` to 1 and zeroing all the others, the skip instruction will stop at the first character from `set` it discovers in `s` as only these characters can match the value in `R4`. To make sure that it will stop at the end of `s` when there are no matching characters, the

first byte in table (corresponding to the terminating zero byte) is also set to 1 – a simple test of the byte pointed at after the skip shows whether it is the end of s or not.

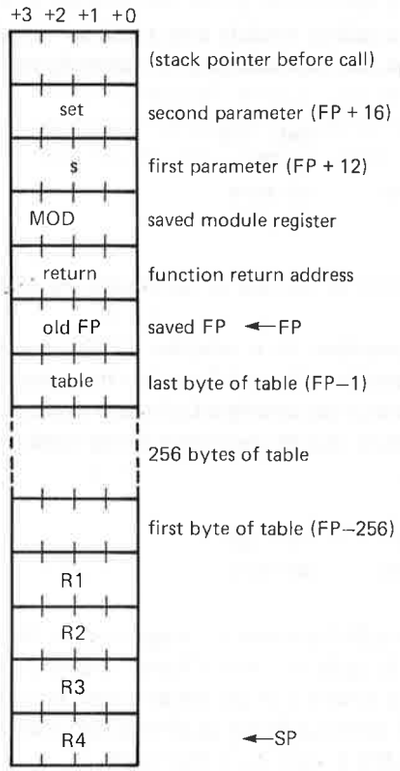

Fig. C.1 The stack on entry to strpbrk after enter.

The table is zeroed by an overlapping move: the first double word in table is set to zero and then 63 double words are moved from table to table+4. First the double word at table (which has been set to zero) is moved to table+4, then the double word at table+4 (which was set to zero by the first move) is moved to table+8, and so on.

```
;the table is set up in the current stack frame: the address
;of its first byte is -256(FP). Zero it first.
        movqd     0, -256(FP)     ;set the first double word
        movzbd    =256/4-1, R0    ;number of words to clear
        addr      -256(FP), R1    ;source (table)
        addr      -252(FP), R2    ;destination (table+4)
        movsd
```


After the table has been zeroed, the required bytes are changed to 1, starting with the zero byte. Each byte from set is brought into R1 and then, if it is not the zero byte ending set, is used as an index into table to insert the 1. Note that the code will not fail if set consists only of the zero byte (empty set): the string s will be searched but will continue until the terminating zero byte. If set is empty then the pointer returned will always be null.

Note that the byte from set must be expanded (with movzbd) into a double word as it is used in a scaled index addressing mode. All 32 bits of the register are used.

```
        movqb   1, -256(FP)             ;set zero byte to terminate
        movqd   0, R0                   ;byte index into set
SetTable
;address of set is in 16(FP)
        movzbd  0(16(FP))[R0:b], R1     ;get byte from set
        cmpqb   0, R1                   ;end of set?
        beq     Scan                    ;yes, now scan s
        movqb   1, -256(FP)[R1:b]       ;mark char from set
        acbd    1, R0, SetTable         ; roll on inexorably
```

In the next section, where the string s is scanned, the maximum length of the scan is set to 65536 bytes. If this function is used in an editor, or similar tool, which has a buffer length greater than this, the maximum will have to be increased.

```
Scan
        movd        =:10000, R0     ;increase if too little
;address of s is in 12(FP)
        movd        12(FP), R1      ;address of first byte to scan
        addr        -256(FP), R3    ;translation table address
        movqd       1, R4           ;stop when translated to 1
        skpst       [u]             ;scan until match
```

If the byte pointed at by R1 is zero, the string s has been searched without finding a character from set and zero is put into R1. Just before the end, R1 is transferred to R0 to be returned to the calling procedure. Make sure that the registers you use to return values to the caller are not included in the enter save list as they will be overwritten by exit! On the other hand, values can be passed back in registers by altering the saved values on the stack.

```
        cmpqb       0, 0(R1)        ;stopped on zero byte?
        bne         Found           ;no, return pointer in R1
        movqd       0, R1           ;no match, zero pointer
Found
        movd        R1, R0          ;returned pointer
```

exit restores the registers saved by enter. If the lists are not the same the

stack pointer will still be correct after leaving the procedure as the stack frame is cleared down to the frame pointer. If, however, there are more registers in the `exit` list than the `enter` list, they will all have the wrong values.

Finally, the call put two 4-byte parameters on the stack and the `rxp` is set up to remove them again. The stack pointer should have exactly the same address in it after the call as it did before the parameters were pushed on to it.

```
exit        [R1,R2,R3,R4]
rxp         8           ;remove 8 bytes of parameter
end
```

C.4 DETAB

This is an example program which replaces tabs with the equivalent number of spaces so the output is visually the same as the input but contains no tab characters.

The algorithm used comes from that excellent book *Software Tools in Pascal* by Brian W. Kernighan and P. J. Plauger (Addison-Wesley, 1981). The actual code embodying the algorithm is quite a small part of the program as it uses operating system facilities to get arguments from the command line, to open files for input and output and to interpret OS error codes.

The utility is called with the command line

```
detab Source Dest Tab n Tabs t1,t2,t3,...
```

with `Source` being the input file name, `Dest` being the output file name or printer (`lp:`) or VDU (`vdu:`). The two tab arguments are alternatives, `Tab` is for equispaced tabs (every `n` columns) and `Tabs` is for the other kind. There are also two arguments special to Panos: `Help`, which displays a reminder of the command line arguments and `Identify`, which causes the utility's version and date to be printed.

The code starts:

```
module    Detab

options   -mexp

get       'Write-mac'
get       'ArgDcd-mac'
get       'IOLib-mac'
```

The `options` directive stops macros being expanded and the three `get` directives bring in the macros in three libraries which are used to automate the calling of system routines.


```
          areadef   static, [write, data], double
          defsb     static
          area      static

i         dcd       0
j         dcd       0
n         dcd       0
InStrm    dcd       0
OutStrm   dcd       0
Eof       dcd       0
InChars   dcd       0
OutChars  dcd       0
InLines   dcd       0
Column    dcd       0
Byte      dcb       0
```

The utility's static area starts with three integers used in loops later in the program. Registers would be a more natural choice, but the Panos system routines don't save registers on entry and the code has the choice of using registers and saving them before each system call or using memory. The choice is easier as the form of an instruction using a register and the same instruction using memory is very much the same.

Here a number of constants representing characters of interest are defined:

```
TAB       equ       :09
NL        equ       :0a
CR        equ       :0d
BLANK     equ       ' '
```

The position of each tab is marked by setting a bit in an array, the maximum line length handled is set at 255 and an appropriate array of bytes is declared as the bit array. In the second form of tabs argument, the positions are entered as a sequence of numbers. This needs a storage area provided as `tabarray` which allows up to 31 tabs to be given.

```
MaxLine   equ       255

tabstops
          allocb    (MaxLine+7)>>3 ;1 bit/position
tabarray
          allocb    31
```

The Panos command line argument decoding routines require a template provided here by the string `Key`. The attributes of each argument are given by one or more code characters separated by `/` after the argument keyword. `/a` implies that the keyword must have at least one argument, `/c` shows that the argument is expected to be a cardinal (unsigned integer) and `/s` that the

argument is a state – if it is present the argument has the value TRUE otherwise it has the value FALSE. /? shows that the keyword can have any number of values. The keywords are only necessary if the arguments are not given on the command line in the order they are in the key string: Tab or Tabs, however, must be given as they are alternatives and the state keywords must be present to enter the value TRUE.

```
Key    dcb     "source/a dest/a tab/c tabs/c/? Identify/s Help/s"
```

When the decoding routines are called to return the value for a particular argument, the keyword must be passed as one of the parameters. The list of keywords follows next.

```
srcarg    dcb         "source"
destarg   dcb         "dest"
tabarg    dcb         "tab"
tabsarg   dcb         "tabs"
```

The static area is closed by the 'area' directive which starts the code area, entry causes the current code address to be used as the entry point after the program is loaded.

```
          area
          entry
```

The first action is to bring the command line into the program for the decoding routines to work on. The macro argstr calls the system routine Arguments which passes it back as a string Arguments: the macro declares this string if it has not yet been defined.

```
detab
          argstr      Arguments
```

Next DecodeInit is called to initialize the decoding routines, showing them where the command line is and returning a Handle to use in the subsequent system calls.

```
;DecodeInit Key, Arguments, Handle
          dcdinit     Key, Arguments, Handle, ErrorCode
          cmpqd       0, R0
          bgt         GetError
```

A return code is passed back in R0. If this is negative, an error has occurred and another system routine is called to interpret it so that useful messages can be displayed.

The specification for Help says that the routine should exit immediately after displaying helpful information so it is tested for first. If it succeeds, the

message is displayed and the decoding process brought to a tidy end before terminating.

```
Help
;HelpRequired          Handle, ErrorCode, Result
          hlpreq      Handle, ErrorCode
          cmpqd       0, R0
          beq         identify
          area        static
HelpMsg   dcb         "Detab Source Dest Tab n/Tabs t1, t2, ..."
          area
          wrmsg       HelpMsg
          newl
          dcdend      Handle
          br          stop
```

If no help information is required, a check is made to see if identification is needed. This is the normal state of the system and the result returned for Identify depends not only on whether the Identify keyword is present but on the state of the system variable Program$Verbose: if Program$Verbose is non-zero the Identify result will be true whether the keyword is present or not.

```
;IdentifyRequired Handle, ErrorCode, Result
identify
          idreq       Handle, Error Code
          cmpqd       0, R0
          beq         getargs
          area        static
identmsg  dcb         "Detab v1.00 4.Jun 85 15:00"
          area
          wrmsg       identmsg
          newl
```

Next the source file name and the destination file name or device code are collected and put into the strings Source and Dest:

```
getargs
          getstr      srcarg, =1, Handle, Source

          getstr      destarg, =1, Handle, Dest
```

The =1 is entered as the index of the value to be returned. As arguments may have more than one value associated with them, the one required must be designated.

Now the optional arguments are collected. First, the number of values provided is obtained; If this is zero, the argument is missing:

```
            getnum      tabarg, Handle, ntab
            cmpqd       0, ntab
            bge         GetTabs
```

If it is present (there is only one value to Tab, the tab spacing), get the value and ignore any Tabs argument:

```
            getcard     tabarg, =1, Handle, Tab
```

If there is no Tab argument, try for a Tabs; get the number of values and, if it is 1 or more, get the values and put them into Tabarray:

```
GetTabs
            getnum      tabsarg, Handle, ntabs
            cmpqd       0, ntabs
            bge         DecideOnTabs

            movqd       1, i
            movd        ntabs, n
NextTab
            getcard     tabsarg, i, Handle, Tab
            movd        i, R1
            movb        Tab, tabarray-1[R1:b]
            addqd       1, i
            acbd        -1, n, NextTab
            newl
```

This part of the program looks to see whether a tab argument has been provided and, if it has, sets up the bit array TabStops depending on which argument has been given. If neither argument is present (both ntab and ntabs zero), an argument of tab 8 is assumed. In the *Software Tools* description, much stress was laid on having the procedure setting up the tab stops and the procedure checking them separated from the main program on the principle that the main program need not know about the TabStops data structure. Here, the code setting up TabStops and the code checking it is so small that the overhead of putting it into a procedure would be unnecessary. Thus, it has been inserted in-line. The section of code starting with ZeroTabs simply zeros the array. Due to the way the array has been declared, it can not be assumed that it is an even number of words or double words, so it is zeroed a byte at a time.

```
DecideOnTabs
            dcdend      Handle
```



```
            cmpqd       0, ntab
            bne         ZeroTabs
            cmpqd       0, ntabs
            bne         ZeroTabs
            movqd       1, ntab
            movzbd      =8, Tab
ZeroTabs
            movzbd      =(MaxLine+7)>>3, R1
zero        movqb       0, tabstops-1[R1:b]
            acbd        -1, R1, zero
```

There are two loops setting TabStops corresponding to the Tab (SetTabs) and the Tabs (ArrayTabs) arguments. SetTabs simply adds the tab spacing to the index, starting at 0, setting bits at each increment. ArrayTabs takes the values from the array TabArray, using them as indices to set the bits in TabStops. The use of the instructions sbitd and sbitb may be a trifle confusing; sbitd takes a double word offset (here in R0) while sbitb contents itself with a byte from TabArray. If TabStops were to increase in size this byte would have to change to a word.

```
SetTabs
            cmpqd       0, ntab
            beq         ArrayTabs
            movd        Tab, R0
set         sbitd       R0, tabstops
            addd        Tab, R0
            cmpd        =MaxLine, R0
            bgt         set
            br          OpenFile

ArrayTabs
            movd        ntabs, R0
seta        movb        tabarray-1[R0:b], R1
            sbitb       R1, tabstops
            acbd        -1, R0, seta
```

Having set up TabStops, the input and output files must be opened. Here a special twist comes in as the Panos file name must be converted into a name satisfying the rules of the filing system in use. To do this, the Panos name appearing in the command line is passed to the system routine PhysicalFile Name which, from global variable information, converts it into the filing system equivalent. Source is here assumed to be a real file and, if PhysicalFileName returns an error, it is assumed to be a real error and control is passed to the error handler. The output file, however, may be the printer or VDU which PhysicalFileName rejects as errors. For Dest then, an error is assumed to be due to a hardware device being called on and it is opened as its unconverted name. Any error comes then from the system call FindOutput, which may not give immense satisfaction and may need to be changed with experience.

```
OpenFile
          realfn      Source, FS, PSource, GetError
          findin      PSource, InStrm, GetError
;the error code from realfn is left in R0 when no error
;label is given.
          realfn      Dest, FS, PDest
          cmpqd       0, R0
          ble         CheckOpen
;on an error, assume that a real device is to be used
          findout     Dest, OutStrm, GetError
          br          repeat

CheckOpen
          findout     PDest, OutStrm, GetError
```

After all this, the code given in *Software Tools* comes into play. The code following is an exact representation of the code on pages 24 and 25 of *Software Tools*, with the proviso that the code for `tabpos` is in-line rather than in a procedure.

```
          movqd       1, Column
repeat
          srdbyte     InStrm, Byte, CheckEOF
          addqd       1, InChars
          cmpb        =TAB, R0          ;is it a TAB
          bne         CheckEOL
TabExpand
          swrbyte     OutStrm, =BLANK, GetError
          addqd       1, OutChars
          addqd       1, Column
          cmpb        =MaxLine, Column  ;past end of line?
          bls         repeat            ;yes !!unsigned!!
          tbitb       Column, tabstops  ;at tabstop yet?
          bfc         TabExpand         ;until next tab
          br          repeat
```

Note the anguished 'unsigned' comment to the comparison between `MaxLine` and `Column`. `MaxLine` is represented as a byte and with value 255 is −1 as a signed byte; to work correctly an unsigned comparison must be made.

Finally, an extension to the original code. I found that some files of interest to me used BBC (`&0D`) or CP/M (`&0A&0D`) as their end-of-line markers. As I was interested in transferring these files to the 32016 second processor, I added the following code which converts CR−LF, LF−CR and CR alone to LF as an end-of-line. The system routine `SCurrentByte` (embodied in the macro `snxtbyte`) returns the next byte from the input stream without advancing the pointer, thus giving a measure of lookahead.


```
CheckEOL
          cmpb        =CR, R0
          bne         CheckNL
          snxtbyte    InStrm            ;byte left in R0
          cmpb        =NL, R0
          bne         CROnly
;End of line is CR-LF
CRLF      srdbyte     InStrm, Byte      ;remove second byte of pair
          br          EOL
CROnly    br          EOL

CheckNL
          cmpb        =NL, R0
          bne         Ordinary          ;not unusual
          snxtbyte    InStrm            ;byte left in R0
          cmpb        =CR, R0
          beq         CRLF
```

The branches to `EOL` at `CRLF` and `CROnly` are not both necessary; they are left to allow other actions to be taken when a specific end-of-line marker is encountered.

When the kind of end-of-line marker has been ascertained (and the second byte of a pair read away), the standard end-of-line marker can be written out to the destination file:

```
NLOnly
EOL       swrbyte     OutStrm, =NL, GetError
          addqd       1, OutChars
          addqd       1, InLines
          movqd       1, Column
          br          repeat
```

Not all bytes are interesting; ordinary ones just get copied to the output.

```
Ordinary
          swrbyte     OutStrm, Byte, GetError
          addqd       1, OutChars
          addqd       1, Column
          br          repeat
```

The read at the beginning of the 'repeat' loop checks `R0` for an error code and, if it finds it, comes here to decide whether it is an end-of-file or a proper error; end-of-file is tested for by the system routine `EndOfFile`, the call to which is manufactured by the `eof` macro. The second read, at the label `CRLF`, doesn't need to check for an error as `SCurrentByte` has already done that for it.

```
CheckEOF
;if the error is EOF, finish neatly; otherwise issue message
          movd        R0, ErrorCode
```

```
            eof         InStrm, Eof, GetError
            cmpqd       1, R0
            beq         EOF
            movd        ErrorCode, R0
            br          GetError
```

At the end of the routine, when it has reached the end of the input file, it prints a count of the characters and lines read and the characters written; the same number of lines will be written as read, but any expanded tabs will increase the number of characters going out.

```
            area        static
endmsg      dcb         "Copy complete.*0a*0d"
incmsg      dcb         "characters,"
inlmsg      dcb         "lines read:"
outcmsg     dcb         "characters written.*0a*0d"
            area
EOF
            wrmsg       endmsg
            wrcard      InChars
            wrmsg       incmsg
            wrcard      InLines
            wrmsg       inlmsg
            wrcard      OutChars
            wrmsg       outcmsg
            close       InStrm, GetError
            close       OutStrm, GetError

stop
            stop
```

Finally the routine exits to the system by the `Stop` call. This sets a return code; the macro `stop` sets this to 0 by default.

The final section of `Detab` is concerned with issuing a message for any error code returned from any of the system calls which check for it. The system routine `XGetErrorMessage` is passed an error code and returns three strings. The first is the name of the library module in which the error was detected (Detecting Facility), the second is the name of the library module actually called (in case this passed the call on to a lower level routine) and the third is the message with any names and values filled in. The name of the interfacing facility is introduced by `"Called by:"` and the detecting facility by `"Found in:"`. The stack parameters for the returned strings are explained earlier in Section 9.4.2.

```
GetError
            movd        R0, TOS         ;the error code
            area        static
            dcd         0
```



```
ErrorMsg    dcb         40
            allocb      40
            dcd         0
Inter       dcb         40
            allocb      40
            dcd         0
Detect      dcb         40
            allocb      40
CalledBy
            dcb         "Called by:"
FoundIn     dcb         "Found in :"
ErrorIs     dcb         "Error is :"
            area
            addr        ErrorMsg-4, TOS
            movzbd      ErrorMsg, TOS
            addr        ErrorMsg+1, TOS
            addr        Inter-4, TOS
            movzbd      Inter, TOS
            addr        Inter+1, TOS
            addr        Detect-4, TOS
            movzbd      Detect, TOS
            addr        Detect+1, TOS
            ifndef      XGetErrorMessage
            importc     XGetErrorMessage
            fi
            cxp         XGetErrorMessage
            wrmsg       CalledBy
            wrstr       Inter
            newl
            wrmsg       FoundIn
            wrstr       Detect
            newl
            wrmsg       ErrorIs
            wrstr       ErrorMsg
            newl
            stop        -10

            end
```

C.5 SIMPLE `printf`

This is a procedure designed to be used while debugging modules. To this end it preserves all the registers on entry and restores them all on exit.

As in the real `printf`, a much used part of the C language system, the routine has a variable number of parameters, the first of which must be a string. This string contains ordinary characters interspersed with formatting codes introduced by a `%`. Ordinary characters are output until a `%` is encountered. If this is followed by one of the special codes, the next parameter

is converted according to the code and the % and following code replaced by the formatted result. The codes accepted by printf allow 16- and 32-bit integers to be output in decimal, octal or hexadecimal and provide for single and double precision floating point as well. In addition, several non-printing characters are represented by 'escaped' lower-case letters, newline being written as \n, for instance.

This utility only provides for 32-bit integer output in decimal or hexadecimal, strings in a particular format or registers. The parameters are put on the stack in reverse order, with the address of the integer or string corresponding to the *last* format code being sent first followed by that for the next to last, with the address of the format string being put on the stack immediately before the count of parameters. The value of the registers is taken from the procedure's stack frame and is the only format code for which there is no corresponding parameter passed.

In this rather primitive version, no provision is made for a newline character or any other non-printing characters; the macro written to set up the call simply adds :0a, :0d, 0 to the end of the format string presented to it.

To print a format string containing no format codes the call would be:

```
fmt1    dcb         'A string without formatting codes.',:0a,:0d,0

        ...

        addr        fmt1, TOS       ;the format string
        movqd       0, TOS          ;no. of parameters
        cxp         printf
```

The end of the format string is marked by a zero byte, in the C tradition.

To print a string with a single integer format code (%i, print in decimal) the call would be

```
int                 dcd             12345
fmt2                dcb             'The integer is %i.',:0a,:0d,0

        ...

        addr        int, TOS
        addr        fmt2, TOS
        movqd       1, TOS
        cxp         printf
```

and so on.

The formatting codes recognized by this printf are:

i	signed integer
u	unsigned integer
x	unsigned integer in hexadecimal
s	string
Rn	register with $0 \leqslant n \leqslant 7$

The string must be in a special format which has been used throughout a set of macros used to generate the calling sequences for the Panos system routines; it is:

```
        dcd         0
string  dcb         strlen          ;actual string length
        allocb      strlen          ;max string length
                                    ;the string bytes
```

This format has proved convenient in constructing macros to handle strings returned as results from some of the Panos routines. The calling sequence expects the address of an unsigned integer, followed by the maximum length of the string and the address of the first byte of the string on the stack for each result string. With the above format this can be done by the instructions

```
addr    string-4, TOS
movzbd  string, TOS
addr    string+1, TOS
```

with the parameter string alone being needed by the macro.

The code exports the entry point and uses the write macros to output integers, unsigned integers (Cardinals) and strings to the default output stream – normally the screen.

The code starts:

```
        module      PrintFormatted
        exportc     printf

        options     -mexp
        get         'Write-mac'

        areadef     static, [write, data], double
        area        static
MAXFMT  equ         80          ;maximum number of bytes in Fmt
nparams dcd         0           ;number of parameters
Fmt     dcd         0           ;address of next byte of format string
FmtRem  dcd         0           ;bytes remaining in Fmt
param   dcd         0           ;address of next parameter
Code    dcb         5, 1        ;bounds of legal format codes
digit   dcb         '7', '0'    ;legal register numbers
i       dcd         0           ;holds integer parameters
StrAdr  dcd         0           ;first byte of string to print
StrLen  dcd         0           ;no. of bytes of string to print
```

The stack picture is similar to that for strpbrk. With the calling sequence given, the number of parameters is lowest on the stack at 12(FP), the address of the format string is at 16(FP) with the first parameter (if there is one) at 20(FP), the second at 24(FP) and so on.

```
            area
;Call is:   printf(nparams, fmt, param1,..., paramn )
            ;nparams is in 12(FP)
            ;fmt addr is in 16(FP)
            ;params from 20(FP) up

            area
printf
;save all registers for possible print out.
            enter       [R0, R1, R2, R3, R4, R5, R6, R7], 0
            movd        12(FP), nparams
            movd        16(FP), Fmt
            addr        20-4(FP), param ;allowing for initial incr
```

After the two fixed parameters and the first variable one have been transferred to local storage, the actual length of the format string is found avoiding the need to check for the terminator byte while scanning for the next %. The maximum acceptable length has been set to 80 bytes as the format string is entered in-line and would spoil the listing format if it was too long;

```
;scan for format string length, up to MAXFMT bytes
            movzbd      =MAXFMT, R0
            movd        Fmt, R1
            movqd       0, R4           ;string terminator
            skpsb       [u]
            movzbd      =MAXFMT, R2
            subd        R0, R2          ;fmt string length
            movd        R2, FmtRem
            cmpqd       0, R2           ;empty fmt string?
            beq         Finish          ;yes
```

The length is calculated as the initial setting of the number of bytes to search for the zero byte terminator less the number remaining when it is eventually found; the count is kept in FmtRem and decremented as bytes from the string are used. It even checks for an empty string!

The main work is done by the routine PrtFmt, which, theoretically, outputs each character from the string until it encounters a %. In fact, it scans the string from the current byte, for the number of bytes remaining (in FmtRem), up to the next % or end of string, prints this and then goes on to examine the putative format code. This code, if legal, is then returned as a number from 1 to 5 in R0. On return, the parameter address is incremented and the count decremented. Before a parameter is printed, the count is checked and if it

shows that no corresponding parameter has been passed then the print is skipped. If a register code is found, for which there is no corresponding parameter, then the loop is re-entered at SeekParam to leave the parameter address and count unchanged.

```
;print all ordinary characters in the fmt string up to
;the next % or end of string: return a code in R0
FmtLoop
            addqd       4, param        ;increment parameter address
            addqd       -1, nparams
SeekParam                               ;entry after register, no param increment
            bsr         PrtFmt
;R0         returned ...  for
;           0           end of fmt string
;           1           %i found
;           2           %u found
;           3           %x found
;           4           %s found
;           5           %Rn found, 0 ≤ n ≤ 7 returned in R1
```

At this point a code has been returned from PrtFmt and, for the convenience of the various routines handling the resulting output, the current parameter address is put into R7:

```
;move parameter address into R7
            movd        param, R7
```

The code value returned will be an illegal value (zero) if PrtFmt has output all the bytes to the end of the string and there are no bytes or format codes left. This is picked up by the checkb instruction working with the legal code bounds in the 2 bytes at Code. When the end of the string is reached, F will be set and the routine will terminate at Finish. The checkb also guards against inconsistencies in the course of development when a new code is introduced in PrtFmt but the corresponding routine to handle it forgotten.

```
;check and adjust code value in R0
            checkb      R6, Code, R0
            bfs         Finish          ;end of Fmt string

Case        casew       Jumps[R6:w]
Jumps       dcw         WrInt-Case      ;code 1
            dcw         WrCard-Case     ;code 2
            dcw         WrHex-Case      ;code 3
            dcw         WrStr-Case      ;code 4
            dcw         WrReg-Case      ;code 5
```

The case jump winds up a nice bit of illustrative code. Note that with casew, the scaled index must be w and the table entries must be dcw.

Each handler checks for the presence of its parameter by checking the value of nparam. As this is initially decremented, its original range 1...*n* becomes 0...*n*−1 and, to be valid, it must be greater than or equal to zero.

```
;print an integer
WrInt
;does parameter exist (nparams ≥ 0)?
          cmpqd       0, nparams
          bgt         FmtLoop         ;no more parameters
          movd        0(R7), R1       ;get address
          movd        0(R1), i        ;get the integer
          wrint       i
          br          FmtLoop
```

As the parameters are represented by their addresses, a doubly indirect fetch must be done to get the value. There are addressing modes which do double indirection. The one which comes to mind in this connection is Memory Relative; to get the value of the first parameter in one instruction you could write

```
          movd        0(20(FP)), i
```

but here both displacements are fixed in the instruction and cannot be used if the distance from FP is not known when the program is assembled.

```
;print an unsigned integer (Cardinal)
WrCard
;does parameter exist (nparams ≥ 0)?
          cmpqd       0, nparams
          bgt         FmtLoop         ;no more parameters
          movd        0(R7), R1
          movd        0(R1), i
          wrcard      i
          br          FmtLoop

;print an unsigned integer in hex
WrHex
;does parameter exist (nparams ≥ 0)?
cmpqd     0, nparams
          bgt         FmtLoop         ;no more parameters
          movd        0(R7), R1
          movd        0(R1), i
          wrcard      i, 16
          br          FmtLoop

;print a string in 'returned string' format:
;         dcd         0               ;actual length
;str      dcb         n               ;maximum length
```



```
;         allocb      n               ;string bytes
;the parameter passed to printf is the address of
;the maximum length byte
WrStr
;does parameter exist (nparams ≥ 0)?
          cmpqd       0, nparams
          bgt         FmtLoop         ;no more parameters
          movd        0(R7), R1
          wrstri      R1
          br          FmtLoop
```

Printing registers is slightly different. The values here are held on this side of the frame pointer with R0 in the double word immediately below it, its least significant byte being in the *lowest* address of this double word; that is, at -4(FP). R7, 7 double words further down, is at -32(FP). The register number, put by PrtFmt into R1, must be negated for the scaled indexing to work as R0 corresponds to 0-4(FP) and R7 to -7*4-4(FP).

```
;print a register: it is held as a double word on the stack
;with R0 at -4(FP) going down to R7 at -32(FP). The number of
;the register required is in R1. No parameter corresponds
;to this code
;the register number in R1 (0..7) needs to be converted into 0..-7
WrReg     negd        R1, R1
          movd        -4(FP)[R1:d], i ;get the reg
          wrcard      i, 16
          br          SeekParam
```

As mentioned before, the loop entry after printing a register skips the parameter adjustment.

The termination code is much as one would expect except that the parameters on the stack must be removed by an ADJSPi instruction. rxp can only remove a constant number of bytes from the stack:

```
;end of Fmt string - restore the registers and return;
;the calling program will adjust the stack
Finish    exit        [R0, R1, R2, R3, R4, R5, R6, R7]
          rxp         0
```

PrtFmt does most of the work in printf. It scans the format string looking for a % character stopping at the end of the string. If it finds the character it will check to see if it is one of the legal codes i, u, x, s or the combination Rn; if it is none of these, it will simply print whatever character it finds after the % character and continue. This both shows up errors in the input and allows the % character itself to be printed by placing %% in the string where a % is to be printed. Before the scan is started though, the number of bytes remaining in the string is checked and, if it is zero, a special exit is taken leaving the zero byte count in R0 to serve as the end-of-string code, zero.

```
;scan the Fmt string for a %
PrtFmt
          movd      FmtRem, R0      ;no of bytes remaining
          cmpqd     0, R0
          beq       Quit
          movd      Fmt, R1         ;byte address
ScanPC    movzbd    ='%', R4        ;byte to match
          skpsb     [u]
          movd      FmtRem, StrLen
          subd      R0, StrLen      ;no of bytes to print
          movd      Fmt, StrAdr     ;1st byte to print
          movd      R0, FmtRem      ;update bytes remaining
          movd      R1, Fmt         ;current Fmt byte address
```

When the scan is complete, the number in R0 can be subtracted from FmtLen, the number of bytes to the end of the string, to find the number of bytes to print. The address of the first byte to print is the byte at which the scan started, Fmt. Having set up the printing parameters, StrAdr and StrLen, the format string parameters, Fmt and FmtRem can be up-dated to point to the first unscanned byte and the number of bytes remaining to be scanned.

The 'ordinary' bytes are printed using the Panos routine to output a block of bytes, checking first that there is something to print:

```
;print ordinary string bytes
          cmpqd     0, StrLen
          beq       NoBytes
          movd      StrAdr, TOS
          movd      StrLen, TOS
          cxp       XBlockWrite
NoBytes
```

The number of bytes remaining in the format string is then checked. If it is zero, the special exit is taken to show that printf is finished.

```
;did it stop at the end of the format string?
          movd      FmtRem, R0
cmpqd     0, R0                     ;no more?
          bne       CheckPC
Quit                                ;end routine
          ret       0
```

In the next section, the next format byte address and the count of bytes remaining is updated and, if there prove to be no more bytes in the string, the Quit exit is taken. Note that a % immediately followed by the end of the string might be considered an error but, as we want printout not errors, it is simply ignored.


```
;check the % code
CheckPC
          addqd     -1, R0          ;move over %
          cmpqd     0, R0           ;end of fmt string?
          beq       Quit            ;yes, go no further
;now update Fmt and FmtRem to point to the byte after %
          movd      R0, FmtRem
          addqd     1, Fmt
          movd      Fmt, R1
```

It is now established that a byte follows the %: the address of the next byte to be scanned and the remaining byte count are updated here, to avoid having to do it at the end of each exit sequence, and the character following % is investigated. If it turns out to be another % the second % is displayed using Panos' XWriteByte, the next byte and byte count are updated and the scan loop re-entered.

```
;R1 points to the byte following %
;now move Fmt and FmtRem to point to the byte following
;the code byte after %, for a register code they now point
;to the register digit
          addqd     -1, FmtRem
          addqd     1, Fmt

          movzbd    0(R1), R0       ;get code byte after %
          cmpb      ='%', R0        ;is it an escaped %?
          bne       PCCode

;% character represented by '%%'-write % and move on.
WrByte    movd      R0, TOS
          importc   XWriteByte
          cxp       XWriteByte
          movd      FmtRem, R0
          movd      Fmt, R1
          addqd     -1, R0          ;pass over 2nd %
          addqd     1, R1
          br        ScanPC
```

At this point in the code a character which is not % has been found to follow the % format marker; a series of tests finds out whether is is one of the legal characters or not. On discovering a legal format character, the integer corresponding to it is left in R0 and control is returned to the main printf loop.

```
;get % code and return the appropriate integer in R0
PCCode    cmpb      ='i', R0
          bne       TryU
          movqd     1, R0
          ret       0
```

```
TryU    cmpb    ='u', R0
        bne     TryX
        movqd   2, R0
        ret     0

TryX    cmpb    ='x', R0
        bne     TryS
        movqd   3, R0
        ret     0

TryS    cmpb    ='s', R0
        bne     TryR
        movqd   4, R0
        ret     0
```

If the code is a register format marker, the digit following must be picked up and, according to the arrangement made by the calling code, returned in R1. The checkb instruction is used again to make sure that the digit is in the acceptable range: if it is not, the digit character (which remains unaltered in R2 – the adjusted character is left in R1) is printed by the same piece of code which deals with doubled % format markers and unrecognized format characters.

```
TryR    cmpb    ='R', R0
        bne     WrByte          ;error: print byte

;get register number
        movd    Fmt, R1         ;already points at digit

;move Fmt and FmtRem over the register digit
        addqd   -1, FmtRem
        addqd   1, Fmt

        movb    0(R1), R2       ;pick up reg digit
        checkb  R1, digit, R2
        bfs     Not0to7         ;out of bounds

;%R0 to %R7: leave 5 in R0 - digit is already in R1
        movqd   5, R0
        ret     0

;%Rx - x not 0..7
Not0to7 movb    R2, R0
        br      WrByte

        end
```

A number of improvements spring to mind. The lower-case i, u and x could represent byte values while upper-case I, U and X could be brought in to represent 32-bit integer quantities. A further improvement could be made by making the wrxxx macros take a stream parameter so that, if a stream variable was set non-zero, output could be to a stream rather than the default output stream as at present. This would allow the code to be used by an fprintf entry which would increase its utility considerably. It would also be a good idea to introduce the abbreviation \n for newline, which would make it more flexible.

C.6 AND FINALLY ...

In June 1964 the learned journal *Mathematics of Computation*, previously *Mathematical Tables and Aids to Computation*, published a short note by E. and U. Karst announcing that, on 1 January 1964, they discovered the first string of 8 zeros in a power of two. The power was 14007 and 'a fast program' running on an IBM 1620 took 1 hour 18 minutes to find it.

The following example sets out to better that discovery by finding the first power containing a string of 9 zeros. The program was developed by way of finding, successively, the first occurrences of 4, 5, 6 and 7 zeros (at the powers 377, 1491, 1492 and 6801) and went on to find the string of 8 zeros in 45 seconds. This development program was a more general one than that given here, in that it counted the number of zeros it found and only printed the count and power if it was one better than the previous count. The program was then left to run for 13 hours 40 minutes to complete the search for consecutive zeros up to the power 217706 (65536 digits) discovering the 9 zeros on the way; it appears that ten consecutive zeros lies further up.

This program has been optimized to search for exactly 9 zeros, ignoring all others, and completed its task in just over 2 hours 42 minutes on a Cambridge second processor (6 MHz 32016).

Since the power had to be in decimal and it would have been extremely costly to convert it back and forth, the calculation was done using the decimal add instruction and extending the precision (from the maximum 8 digits in a double word) by adding in the carry after adding a component pair of double words. In fact, this is done automatically by the ADDPi instruction and it is only necessary to stop the addition after it has passed the most significant digit. In the development stage, the code given at the end of Chapter 6 was used to print the powers as they were formed for checking.

```
;Find first power of two containing nine consecutive zeros
        module  PowersOfTwo
        options -mexp
```

The write and time and date macros are used to print the starting and ending time and the elapsed time:

```
        get     'Write-mac'
        get     'TimDat-mac'
```

```
            areadef     static, [write,data], double
            area        static

PowerOfTwo
            dcd         0

digits      equ         65536           ;to 2^217706
two         allocd      digits>>3
twoms       equ         $-1             ;most significant digit
            dcd         0               ;barrier

psr         dcb         0               ;usr PSR
```

PowerOfTwo is the count of the number of times the addition has been performed so far, and the packed digits of the power calculated are kept in two. twoms is used to stop the calculation when the power is just about to overflow; the most significant digit is 5 or more. The barrier is a bit of a bodge, the progression of the most significant non-zero digit is calculated and the addition done up to and including the double word above the double word containing it. To avoid overwriting useful things, an empty double word was put above the last word actually used. psr keeps the user byte of the PSR after the addition of two double words, the instructions following it in the loop cleared the carry which was needed for the next addition.

On entry, the time (provided by Panos) is taken using the BinaryTime service routine; this is converted into 'standard time' by StandardTimeOfBinary Time and displayed:

```
            area

            entry
powers
            btim        start
            stimb       start, STime
            wrstr       STime+1, STime-4
```

Now the power of two is initialized to 1 by setting the least significant double word to 1 and all the others to 0; there is no sign to worry about.

```
            movqd       1, two
            movd        =-(digits>>3)+1, R0
            movqd       0, PowerOfTwo
            addr        two+4, R1
init        movqd       0, 0(R1)
            addqd       4, R1
            acbd        1, R0, init
```

The position of the most significant non-zero digit is initialized. As the addition is by double words (for efficiency) the position of the most significant

digit is taken as the double word containing it and, because eventually there will be a carry out of this double word, the addition is continued to the next higher double word.

```
            addr        two+1, R2       ;ms zero byte
```

This is the addition and search loop: the value of PowerOfTwo is incremented, the value in psr is set to zero which, later in the loop, will clear the carry bit in the user PSR and the address of the first double word to add goes into R1. Note that the double word is added to itself with the result replacing its original contents.

```
clrC        addqd       1, PowerOfTwo
            movqb       0, psr          ;clear Carry
            addr        two, R1

;no. of double words to add = (ms byte+1+3) DIV 4
            movd        R2, R0          ;two+n
            subd        R1, R0          ;-two
            addqd       4, R0           ;+1+3
            ashd        =-2, R0         ;DIV 4
add         lprb        UPSR, psr       ;restore carry
            addpd       0(R1), 0(R1)
            sprb        UPSR, psr       ;save carry
            addqd       4, R1           ;!! clears Carry !!
            acbd        -1, R0, add
```

Carry must be saved, in the form of the user byte of the PSR, because addqd clears it.

The address of the most significant digit of the power of two is kept as the address of the double word above that which actually contains it. This word should therefore be zero; if it is no longer so, it is time to move up.

```
;if the present ms byte is now non-zero, increase the
;address

set_ms      cmpqb       0, 0(R2)
            beq         search
            addqd       1, R2
```

Now the search is made for a zero byte using the string 'skip until' instruction,

```
search      save        [R2]            ;save ms byte address
            addr        two, R1         ;start scan here
            movd        R2, R0          ;no. of bytes to scan
            subd        R1, R0
;continue the search after a zero byte has been investigated
```

```
;and there are more than 4 bytes remaining
continue_search
          movqd     0, R4          ;scan for zero byte
          skpsb     [u]

;if no (further) zero byte is found (F clear),
;continue with the next power of two
          bfc       contin
```

Nine consecutive zero digits must consist of four consecutive zero bytes with either a zero nybble before the first one or after the last; note that the skpsb above could have been changed to skpsw, speeding up the search a bit as the 32016 can deal with a word at a time.

```
;quick search: after finding the zero byte, check forwards
;for 3 more zero bytes. If they are all found (8 contiguous
;zeros), check for a zero nybble just before the first zero
;byte or just after the last
          cmpqb     0, 1(R1)
          bne       recheck1
          cmpqb     0, 2(R1)
          bne       recheck2
          cmpqb     0, 3(R1)
          bne       recheck3
          extsb     4(R1), R2, 0, 4  ;upstream nybble
          cmpqb     0, R2
          beq       print_nine
          extsb     -1(R1), R2, 4, 4 ;downstream nybble
          bne       recheck4

;Nine zeros found: take the time, print the message and
;close
print_nine
          btim      finish
          wrstr     ninez
          wrcard    PowerOfTwo
          newl
          br        ends

          area      static
ninez     dcb       "Nine zeros found at 2^"
          area
```

This is the recovery section after the search, depending on how many consecutive zero bytes have been found, advance the scan pointer in R1, decrement the byte count and then, if there are more than 4 bytes left to scan, continue the loop.


```
;restart after 1 zero byte
recheck1
          addqd     -1, R0
          addqd     1, R1
          br        check4

;restart after 2 zero bytes
recheck2
          addqd     -2, R0
          addqd     2, R1
          br        check4

;restart after 3 zero bytes
recheck3
          addqd     -3, R0
          addqd     3, R1
          br        check4

;restart after 4 zero bytes
recheck4
          addqd     -4, R0
          addqd     4, R1

;continue the search of the string if there are more
;than 4 bytes left
check4    cmpqd     4, R0
          blt       continue_search
```

The scan is complete (and presumably failed). Check the top digit in the top double word of two and stop if the next addition will overflow beyond the allotted length:

```
contin    restore   [R2]
;continue until the leading digit is 5 or greater
          extsb     twoms, R0, 4, 4  ;ms digit
          cmpqb     5, R0
          bgt       clrC
;stopped by leading digit
          btim      finish
          wrstr     leadmsg
          wrcard    PowerOfTwo
          newl
```

The time, in binary centiseconds, is returned in two double words with the most significant in the higher address.

```
          area      static
diff      dcd       0, 0             ;BTim record
          area
```

```
;calculate the elapsed time and print it in centiseconds
;and as a standard time
ends        movd        finish+4, diff+4
            subd        start+4, diff+4
            movd        finish, diff
            subcd       start, diff

            wrcard      diff
            wrstr       csecmsg
            newl

            stimb       finish, STime
            wrstr       STime+1, STime-4
            newl
            stop

            area        static
leadmsg     dcb         "Leading digit 5 or more: stopping at 2^"
csecmsg     dcb         "csecs."

            end
```

INDEX