R.N. Ibbett and
N.P. Topham

Architecture of High Performance Computers Volume I

Springer Science+Business Media, LLC

Architecture of High Performance Computers

Volume I

R. N. Ibbett and N. P. Topham

Department of Computer Science
University of Edinburgh
Edinburgh
Scotland EH9 3JZ

Architecture of High Performance Computers

Volume I

Uniprocessors and vector processors

Springer Science+Business Media, LLC

Originally published by Springer-Verlag New York Inc. in 1989

ISBN 978-1-4899-6714-5 ISBN 978-1-4899-6712-1 (eBook)
DOI 10.1007/978-1-4899-6712-1

Contents

Preface

This second edition of *The Architecture of High Performance Computers* has been produced as a two volume set. Volume I is a revised, updated and slightly expanded version of the first edition, dealing mainly with techniques used in uniprocessor architectures to attain high performance. Many of these techniques involve some form of parallelism, but this parallelism is largely hidden from the user. Machines which are explicitly parallel in nature are dealt with in Volume II, which concentrates on the architecture of systems in which a number of processors operate in concert to achieve high performance. The high performance structures described in Volume I are naturally applicable to the design of the elements within parallel processors, and therefore Volume II also represents a historical progression from Volume I.

Computer architecture is an extensive subject, with a large body of mostly descriptive literature, and any treatment of the subject is necessarily incomplete. There are many high performance architectures, both on the market and within research environments, far too many to cover in a student text. We have therefore attempted to extract the fundamental principles of high performance architectures and set them in perspective with case studies. Where possible we have used commercially available machines as our examples. The two volumes of this book are designed to accompany undergraduate courses in computer architecture, and constitute a core of material presented in third and fourth year courses in the Computer Science Department at Edinburgh University.

Many people gave advice and assistance in the preparation of the first edition, particularly former colleagues at the University of Manchester, and much of this has carried over into the second edition. Computer maintenance engineers at various sites willingly answered obscure questions about the machines in their charge, and staff at Cray Research and Control Data Corporation, particularly Chuck Purcell, vetted parts of the manuscript and provided much useful information. In preparing this first volume of the second edition the authors are indebted to William White of Cray Research for his comments on the content of the manuscript. Preparation of the manuscript involved many hours at computer terminals and the authors would like to thank Alison Fleming for her assistance and expert advice on the use of LaTeX.

Roland Ibbett
Nigel Topham

1 *Introduction*

Computer architecture has been defined in a number of ways by different authors. Amdahl, Blaauw and Brooks [ABB64], for example, the designers of the IBM/360 architecture, used the term to "describe the attributes of a system as seen by the programmer, i.e. the conceptual structure and functional behaviour, as distinct from the organisation of the data flow and controls, the logical design and the physical implementation." Stone [Sto75], on the other hand, states that "the study of computer architecture is the study of the organisation and interconnection of components of computer systems." The material presented here is better described by this wider definition, but is particularly concerned with ways in which the hardware of a computer can be organised so as to maximise performance, as measured by, for example, average instruction execution time. Thus the architect of a high performance system seeks techniques whereby judicious use of increased cost and complexity in the hardware will give a significant increase in overall system performance.

1.1 Historical developments

Designers of the earliest computers, such as the Manchester University/ Ferranti Mark 1 (first produced commercially in 1951 [Lav75]), were constrained by the available technology (valves and Williams Tube storage, for example, with their inherent problems of heat dissipation and component reliability) to build (logically) small and relatively simple systems. Even so, the introduction of B-lines and fast hardware multiplication in the Mark 1 were significant steps in the direction of cost-effective hardware enhancement of the basic design. At Manchester this trend was developed further in the Mercury computer, with the introduction of hardware to carry out floating-point addition and multiplication. This increased logical complexity was made possible by the use of semiconductor diodes and the availability of smaller and more reliable valves than those used in the Mark 1, both of which helped to reduce power consumption (and hence heat dissipation) and increase overall reliability. A further increase in reliability was made in the commercial version of Mercury (first produced in 1957) by the use of the then newly developed ferrite core store.

The limitations on computer design imposed by the problems of heat dissipation and component reliability were eased dramatically in the late

1950s and early 1960s by the commercial availability of transistors, and the first generation of 'supercomputers' such as Atlas [KELS62], Stretch [Buc62], MULTICS [Org72] and the CDC 6600 [Tho70] appeared. These machines incorporated features which have influenced the design of subsequent generations of computers (the paging/virtual memory system of Atlas and the multiple functional units of the 6600, for example), and also highlighted the need for sophisticated software, in the form of an operating system intimately concerned with activities within the hardware from which the user (particularly in a multi-user environment) required protection, and vice versa! In this book we shall follow the developments of some of the ideas from these early supercomputers into present day computer designs.

1.2 Techniques for improving performance

Improvements in computer architecture arise from three principal factors

1. technological improvements
2. more effective use of existing technology
3. improved software-hardware communication.

Technological improvements include increases in the speed of logic circuits and storage devices, as well as increases in reliability. Thus the use of transistors in Atlas, for example, not only allowed individual circuits to operate faster than those in Mercury, but also allowed the use of parallel adders, which involved many more logic circuits but which produced results very much more quickly than the simple serial adders used in earlier machines. Corresponding changes can be seen today as the scale of integration of integrated circuits continues to increase, allowing ever more complex logical operations to be carried out within one silicon chip, and hence allowing greater complexity to be introduced into systems while keeping the chip speed and overall system reliability more or less constant.

Making more effective use of existing technology is the chief concern of computer architecture, and almost all of the techniques used can be classified under the headings of either storage hierarchies or concurrency. Storage devices range from being small and fast (but expensive) to being large and cheap (but slow); in a storage hierarchy several types, sizes and speeds of storage device are combined together, using both hardware and software mechanisms, with the aim of presenting the user with the illusion that he has a fast, large (and cheap) store at his disposal.

Concurrency occurs in various forms and at various levels of system design. Low-level concurrency is typified by pipeline techniques, interleaved storage and parallel functional units, for example, all of which are largely

invisible to software. High-level concurrency, on the other hand, typically involves the use of a number of processors connected together in some form of array or network, so as to act in parallel on a given computational task. Various authors have attempted to classify computers on the basis of the type of concurrency they exhibit. The most well known of these taxonomies is that proposed by Flynn [Fly72], who classified processors as Single Instruction Single Data (SISD) machines, Single Instruction Multiple Data (SIMD) vector machines, Single Instruction Multiple Data array processors, and Multiple Instruction Multiple Data (MIMD) machines. Volume I of this book deals mainly with techniques which are applicable within the design of SISD computers and with SIMD vector machines. SIMD array architectures and MIMD (multiprocessor) machines are dealt with in Volume II.

Communication between software and hardware takes place through the order code (or instruction set) of the computer, and the efficiency with which this takes place can significantly affect the overall performance of a computer system. One of the difficulties with some 'powerful' computers is that the 'power' is obtained through hardware features which can only be fully exploited either by hand coding or by the use of complex and time-consuming algorithms in compilers. This fact was recognised early in the Manchester University MU5 project, for example, where an instruction set was sought which would permit the generation of efficient object code by high-level language compilers. The design of this order code was therefore influenced not only by the anticipated organisation of the hardware, which was itself influenced by the available technology, but also by the nature of existing high-level languages. Thus the order code, hardware organisation, and available technology all interact together to produce the overall architecture of a given system. An example of such an interaction is given in the next section, based on some of the early design considerations for MU5 [MI79].

1.3 An architectural design example

At the time when the MU5 project was started (1966/7), the fastest production technology available for its construction was that being used by ICT (later to become ICL) for their 1906A computers. In this technology, MECL 2.5 small scale integrated circuits are mounted individually or in pairs on printed circuit modules, together with discrete load resistors, and up to 200 of these modules are interconnected via multi-layer platters. Platters are mounted in groups of six or nine within logic bays, with adjacent platters being joined by pressure connectors. Inter-group and inter-bay connections are via co-axial cables. The circuit delay for this technology is 2 ns, but the wiring delay of 2 ns/ft (through matched transmission line interconnec-

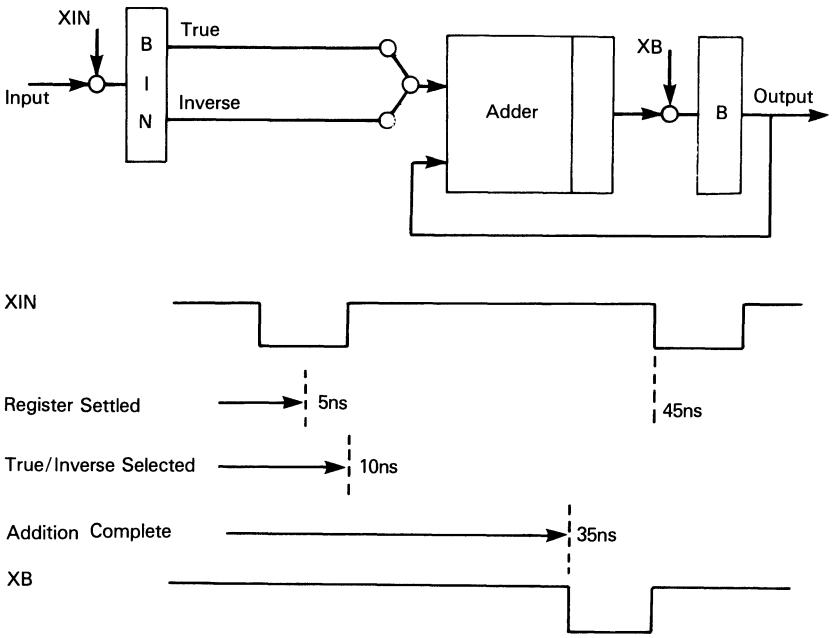

Figure 1.1 An index arithmetic unit

tions) must also be taken into account. An average connection between two integrated circuits mounted on modules on the same platter involves a 1" connection on each module and a 4" connection between modules, thus totalling 6" and giving an extra 1 ns delay. Some additional delay also occurs because of the loading of an output circuit with following input circuits, and so for design purposes a figure of 5 ns was assumed for the delay between the inputs of successive gates.

A 32-bit fixed-point adder/subtracter constructed in this technology requires five gate delays through the adder, plus one gate delay to select the TRUE or INVERSE of one of the inputs to allow for subtraction, giving a total delay of 30 ns. This adder/subtracter can be incorporated into an index arithmetic unit as shown in figure 1.1. A 10 ns strobe XIN copies new data into register BIN, the output from which is steady after 5 ns. During the addition, information is strobed into a register which forms part of the adder, and the 10 ns strobe XB, which copies the result of the addition into the index register B, starts immediately after this internal adder strobe has finished. The earliest time at which the next XIN strobe can start is at the end of XB, so that the minimum time between successive add/subtract

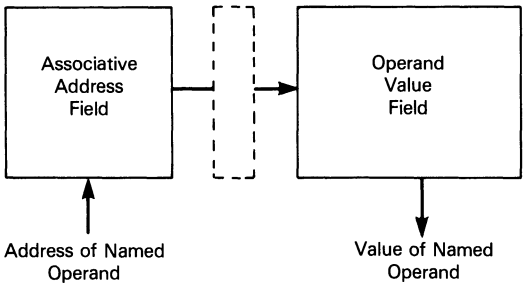

Figure 1.2 The MU5 name store

operations in this unit is 45 ns.

This time is very much less than the access time to the main store of MU5, a plated-wire store with a 260 ns cycle time. Thus, in the absence of some additional techniques, there would be a severe mis-match between the operand accessing rate and the arithmetic execution rate for index arithmetic instructions. An examination of the operands used in high-level languages, and studies of programs run on Atlas, indicated that over a large range of programs, 80 per cent of all operand accesses were to named scalar variables, of which only a small number were in use at any one time. Thus, a system which kept these operands in fast programmable registers would be able to achieve high performance. Such a technique is now in widespread use. However, this is exactly the sort of hardware features which causes compiler complexity, and which the designers of MU5 sought to avoid.

The alternative solution adopted in MU5 involves the identification within each instruction of the kind of operand involved, and the inclusion in the hardware of an associatively addressed buffer store for named variables, known as the Name Store. This store has to operate in conjunction with the main store of the processor, thus immediately introducing the need for a storage hierarchy in the system. Having observed this implication, further discussion of storage hierarchies will be left until chapter 3: there are additional implications for the system architecture to be considered here, based on the timing of Name Store operations.

The Name Store consists of two parts, as shown in figure 1.2; the associative address field and the operand value field. During the execution of an instruction involving a named variable, the address of the variable is presented to the associative field of the Name Store, and if a match is found in one of its 32 associative registers, the value of the variable can be read from the corresponding register in the Name Store value field. The time required for association is 25 ns, and similarly for reading. Thus, in order for

the index arithmetic unit to operate at its maximum rate, the association time, reading time and addition time for successive instructions must all be overlapped (by introducing a buffer register, such as that shown dotted in figure 1.2). In making provision for this overlap, however, another architectural feature has been introduced – the organisation of the processor as a pipeline. Further discussion of pipelines will be left until chapter 4, but it should now be clear that there is considerable interaction between the various facets of the architecture of a given computer system.

2 *Instructions and Addresses*

An important characteristic of the architecture of a computer is the number of addresses contained in its instruction format. Arithmetic operations generally require two input operands and produce one result, so that a three-address instruction format would seem natural. However, there are arguments against this arrangement, and decisions about the actual number of addresses to be contained within one instruction are generally based on the intuitive feelings of the designer(s) in relation to economic considerations, the expected nature of implementation, and the type of operand address and its size. An important distinction exists between register addresses and store addresses, for example; if the instruction for a particular computer contains only register addresses, so that its main store is addressed indirectly through some of these registers, then up to three addresses can be accommodated in one instruction. On the other hand, where full store addresses are used, multiple-address instructions are generally regarded as prohibitively expensive both in terms of machine complexity and in terms of the static and dynamic code requirements. Thus one store address per instruction is usually the limit (in which case arithmetic operations are performed between the content of the store location and the content of an implicit accumulator), although some computers have variable-sized instructions and allow up to two full store addresses in a long instruction format. In this chapter we shall introduce examples of computer systems which have three, two, one and zero-address instruction formats, and discuss the relationships which exist between each of these arrangements and the corresponding hardware organisation.

2.1 Three-address systems — the CDC 6600 and 7600

The Control Data Corporation 6600 computer first appeared in the early 1960s, and was superseded in the late 1960s by the 7600 system (now renamed CYBER 70 Model 76). The latter is machine code compatible upward from the former, so that the basic instruction formats of the two systems are identical, but the 7600 is about four times faster that the 6600. The 6600 remains an important system for students of computer architecture, however, since it has been particularly well documented [Tho70]. An understanding of its organisation and operation provides a proper background not only for an appreciation of the design of the 7600 system, but

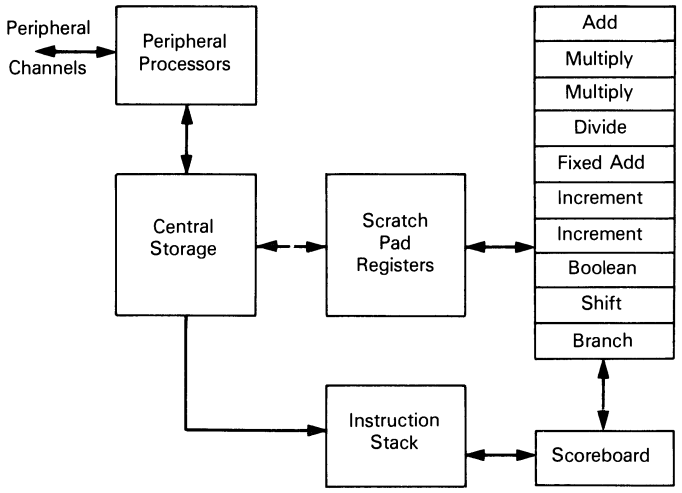

Figure 2.1 CDC 6600 processor organisation

also that of the CRAY-1, which can be seen as a logical extension of the 6600/7600 concepts from scalar to vector operation. The design of all these machines will be discussed in more detail in chapters 6 and 7.

The CDC 6600 was designed to solve problems substantially beyond contemporary computer capability and, in order to achieve this end, a high degree of functional parallelism was introduced into the design of the central processor. This in turn required an instruction set and processor organisation which could exploit this parallelism, while at the same time maintaining at least the illusion of strictly sequential execution of instructions. A three-address instruction format provides this possibility, since successive instructions can refer to totally independent input and result operands. This would be quite impossible with a one-address instruction format, for example, where one of the inputs for an arithmetic operation is normally taken from, and the result returned to, a single implicit accumulator. Despite the potential for instruction overlap, dependencies between instructions can still occur in a three-address system. For example, where one instruction requires as its input the result of an immediately preceding instruction, the hardware must ensure that these are strictly maintained. This would be difficult if full store addresses were involved, but the use of three full store addresses would, in any case, have made 6600 instructions prohibitively long. There were, in addition, strong arguments in favour of having a *scratch-pad* of fast registers in the 6600 which could match the operand processing rate of the functional units.

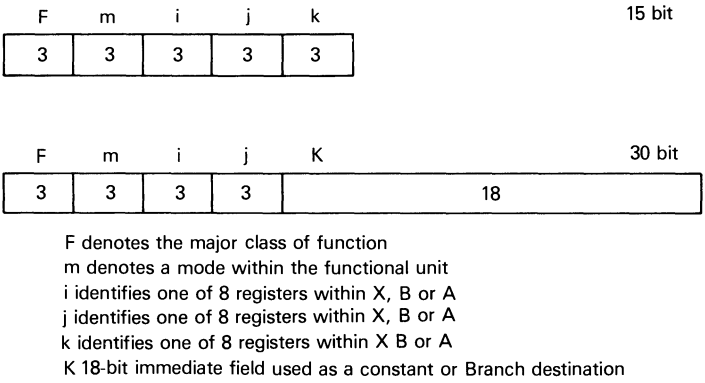

Figure 2.2 CDC 6600 instruction formats

Thus the overall design of the 6600 processor is as shown in figure 2.1, and the instruction formats are as shown in figure 2.2. There are three groups of scratch-pad registers, eight 60-bit operand registers (X), eight 18-bit address registers (A), and eight 18-bit index registers (B). 15-bit computational instructions take operands from the two X registers identified by j and k, and return a result to the X register identified by i. Operands are transferred between X registers and Central Storage by instructions which load an address into registers A1-A5 (thus causing the corresponding operand to be read from that address in store and loaded into X1-X5), or into registers A6 or A7 (thus causing the contents of X6 or X7 to be transferred to that address in store). The contents of an A register can also be indexed by the contents of a selected B register, and the B registers can also be used to hold fixed-point integers, floating-point exponents, shift counts, etc. The 30-bit instruction format (identified by particular combinations of F and m, as are the register groups) is used where a literal (*immediate*) operand is to be used in an operation involving an A or B register, or in control transfer (*branch*) instructions, where the value in the K field is used as the jump-to, or destination, address.

The issuing of instructions to the functional units and the subsequent updating of result registers is controlled by the Scoreboard (section 6.1), which takes instructions in sequence from the Instruction Stack (section 5.2.1). This in turn receives instructions from Central Storage. The order code of the central processor does not include any peripheral handling instructions, since all peripheral operations are independently controlled by the ten Peripheral Processors, which organise data transfers between Central Storage and input/output and bulk storage devices attached to

the peripheral channels. This relieves the central processor of the burden of dealing with input/output, a further contribution towards high performance.

2.2 Two-address systems — the IBM System/360 and /370

The design objectives for the IBM System/360 [ABB64,IBM] (introduced in 1964) were very different from those for the CDC 6600. Here the intention was to produce a line of processors (initially six) with compatible order codes, but a performance range factor of 50. Furthermore, whereas the CDC 6600 was intended specifically for scientific and technological work, System/360 machines were required to be configurable for a variety of tasks covering both scientific and data processing applications. The former are dominated by floating-point operations, where fast access to intermediate values requires the provision of one or more registers close to the arithmetic unit(s) in the central processor. Data processing applications, on the other hand, frequently involve the movement (and fairly simple manipulation) of long strings of data, and direct two-address storage to storage operations are much more appropriate than operations involving three store addresses or operations involving the use of intermediate values held in registers. Furthermore, in machines at the low performance end of the range, logical registers are normally implemented within main storage, and the extra store accesses required for these registers would actually slow down this type of operation.

Store addressing is itself a major problem in the design of a compatible range of computers; to quote from IBM, in turn quoting from DEC [CP78,BS76]

> "There is only one mistake... that is difficult to recover from – not providing enough address bits..."

In the design of the IBM System/360 large models were seen to require storage capacities of the order of millions of characters, whereas small models required short addresses in order to minimise code space requirements and instruction access time. This problem was overcome by the use of index registers to supply base addresses covering the entire 24-bit address space, and small displacement addresses within instructions. This can be seen in figure 2.3, which shows the five basic instruction formats: register to register operations (RR), register to indexed storage operations (RX), register to storage operations (RS), storage and immediate operand operations (SI), and storage to storage operations (SS). All operand addresses consist of a 4-bit B field specifying which of the 16 general registers provided is to be used as a base value to form the store address. In the case of the RX format,

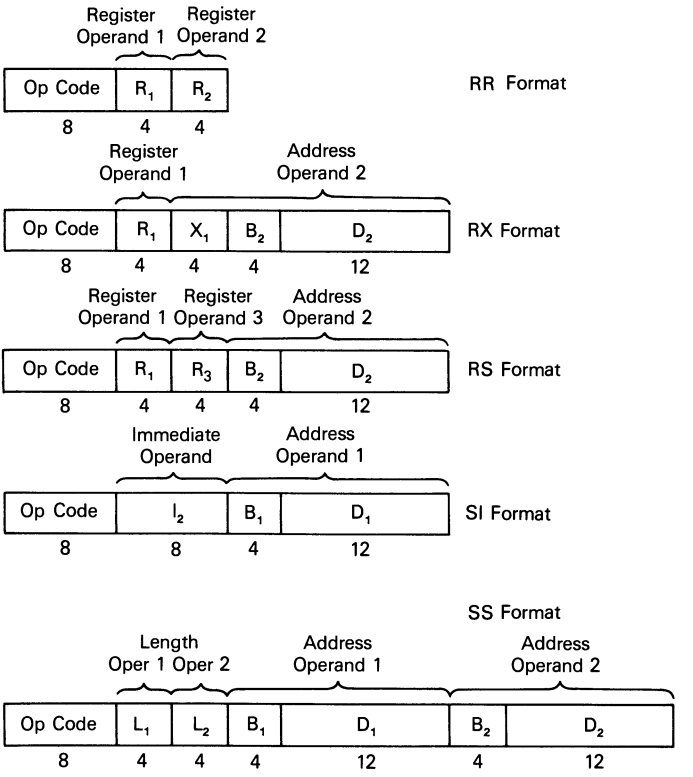

Figure 2.3 IBM System/360 basic instruction formats

a further general register is specified by the X field, for use as a conventional index register allowing access to an element within an array, for example. In the RS format this field becomes R3, used in the specification of the number of registers involved in the simultaneous transfer of a multiplicity of register contents to or from store. A typical instruction using the RR format, for example, involves the addition of the content of the register specified by R2 to the content of register R1, the result being returned to R1.

The number of registers to be provided was determined from an extrapolation, at the time when the System/360 was designed, of contemporary technological trends. These indicated the probable availability of small high-speed storage devices, and the design therefore incorporated what was then considered to be a substantial number of logically identifiable registers. These registers were to be physically realised in core storage, local high-speed storage or discrete transistors, according to the model. There

are, in fact, two sets of addressable registers, sixteen 32-bit general purpose registers used both as index registers and fixed-point accumulators, and four 64-bit floating-point registers. Both sets occupy the same logical address space, the operation code determining which type of register is to be used in a given operation. The choice of 16 addressable register locations also allows the specification of two registers in a single 8-bit byte, the smallest addressable unit of information in the System/360. The separation of the two sets, using operation code information, allows instruction execution in larger models to be carried out in a separate unit from, and concurrently with, the preparation of subsequent instructions, thus leading to higher performance (section 4.4).

Store addresses generated by the central processor of almost all System/360 machines are *real* addresses, each referring directly to a physical location in main store. (The same is also true of CDC 6600 addresses.) A move away from this situation occurred with the introduction of the System/370 in 1970 [CP78]. IBM System/370 machines have essentially the same order code as System/360 machines, with minor modifications including, for example, the introduction of some one-address instructions. Addresses in System/370 machines are all *virtual* addresses, however, which do not refer directly to physical locations in the main store, but are mapped on to these locations by a dynamic address translation mechanism. Such a system was first introduced in Atlas, and is described in some detail in chapter 3.

2.3 One-address systems

Few high performance computers have strictly one-address instruction formats, although these are fairly common among mini and microcomputers where economy of bits in instructions is vital. The DEC PDP-10 [BKHH78], for example, has a single store address within each instruction (figure 2.4), but also specifies one of sixteen identical, general purpose registers, rather than assuming a single implicit accumulator. *One-and-a-half-address* might be a more appropriate designation for this type of instruction, and clearly the IBM System/360 RX format (figure 2.3) would fall into this category. Instructions in MU5 (section 2.5), on the other hand, and its commercial derivative, the ICL 2900 Series [Buc78], can be regarded as strictly one-address. There are only a few registers, all serving different dedicated purposes, and the register specification is therefore implied by the function code, rather than being explicitly addressed within the instruction format.

The Atlas computer [KELS62] had a single full-length (48-bit) accumulator, closely associated with the main floating-point arithmetic unit, and so its A-codes were strictly one-address, with double B-modification (fig-

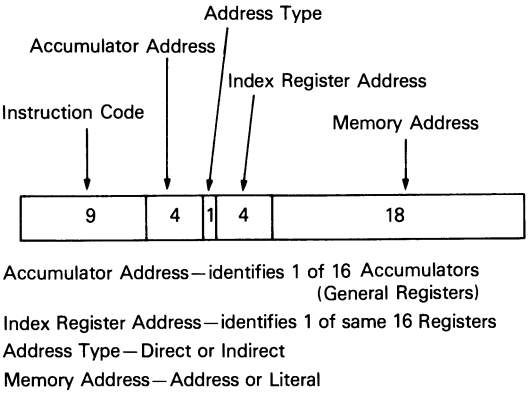

Figure 2.4 DEC PDP-10 instruction format

ure 2.5). The processor was organised in a manner which allowed operands to be accessed from the 2 μs cycle-time core store at a rate matching the execution rate of the main arithmetic unit (of the order of 0.5M instructions per second), thereby avoiding the need for additional full-word buffer registers. This organisation (figure 2.6) involved having a separate 24-bit fixed-point B-arithmetic unit closely associated with a 128-line B-line store, separate access mechanisms to the individual units of the core store (this is known as *interleaving*, described in more detail in section 3.1), and extensive overlapping of operations (this is known as *pipelining*, described in chapter 4).

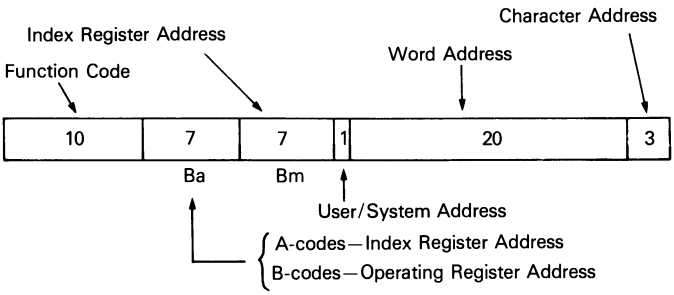

Figure 2.5 Atlas instruction format

B-codes in Atlas were of the one-and-a-half-address type, however, since they involved operations on one of the 128 B-lines specified by the Ba field in the instruction, Bm being used to specify a single modifier. The B-

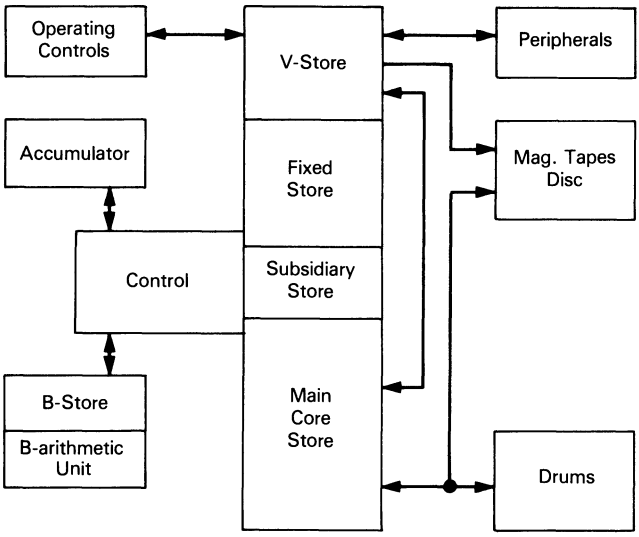

Figure 2.6 The Atlas computer system

store was implemented as a fast core store (0.7 μs cycle time) of 120 24-bit words, together with eight flip-flop registers which served special purposes. These included the floating-point accumulator exponent, for example, and the three control registers (or program counters) used for user programs, extracode and interrupt control. The existence of these three control registers allowed the machine to switch rapidly from one context to another, while their inclusion within the addressable B-store avoided the need for separate control transfer functions. Extracodes were a set of functions which gave access to some 250 built-in subroutines held in the high-speed (0.5 μs) read-only fixed store. These included the more complex mathematical functions such as sin, log, etc., peripheral control orders, input/output conversion routines, etc. The extracodes made use of 30 of the B-lines, leaving 90 for use as index registers or fixed-point accumulators in user programs.

User addresses in Atlas (defined by a 0 in their most significant address bit) referred to a 1M word virtual address space, and were translated into real addresses through a set of page registers (as will be described in chapter 3). The real store in the prototype machine consisted of a total of 112K words of core and drum storage, combined together through the paging mechanism so as to appear to the user as a *one-level* store. Thus a user's total working store requirements could not exceed this amount, but code and data could be placed in any convenient part of the virtual ad-

dress space. This led to an informal segmentation of the address space, a concept developed formally in MULTICS [Org72]. Formal segmentation, or *2-dimensional addressing*, is used in MU5 and the ICL 2900 Series (section 2.5), and has appeared more recently in minicomputers such as the Data General ECLIPSE MV/8000 and microcomputers such as the Zilog Z8000 and Motorola M68010/M68020.

Atlas addresses with a 1 in the most significant bit were system addresses rather than user virtual addresses, and referred to the fixed store (containing the extracodes), the subsidiary store (used as working space by the extracodes and operating system), or the V-store. The latter contained the various registers needed to control the tape decks, input/output devices, paging mechanism, etc., thus avoiding the need for special functions for this purpose. This technique of incorporating peripherals into the address space was subsequently used in a number of computer systems, notably the DEC PDP-11, and is now in common use as 'memory-mapped I/O'.

2.4 Zero-address systems

If we consider a program statement of the form

```
RESULT := (A + B) * ((C + D) / (E + F))
```

then this could be compiled straightforwardly into one-address instructions as

```
LOAD A
ADD B
STORE TEMP0
LOAD C
ADD D
STORE TEMP1
LOAD E
ADD F
REVERSE DIVIDE TEMP1
MULTIPLY TEMP0
STORE RESULT
```

We note that two temporary variables have to be allocated and that these are re-used in the opposite order to that in which they were created. This corresponds to a last-in, first-out, or push-down stack arrangement, and where a multiplicity of base registers is provided, this stack is usually implemented by using one of the base registers as a stack pointer. Indeed, on the PDP-11, the auto-increment and auto-decrement addressing modes are specifically designed for this purpose.

An alternative arrangement is for the accumulator itself to be implicitly treated as the top-of-stack location, and for successive load orders to push previously loaded operands down into the stack automatically. Arithmetic orders then operate between the two most recently loaded operands, and leave their result on the stack, so that no address specification is required. These are therefore zero-address instructions, and systems which use this arrangement are frequently known as stacking machines.

Stacking machines offer particular advantages to compiler writers, since the algorithm for translating into Reverse Polish notation, and thence into stacking machine code, is very simple. The expression in the example above becomes, in Reverse Polish

```
A B + C D + E F + / * RESULT =>
```

Thus operands carry over into Reverse Polish without any relative change of position, and the rule for arithmetic operators is that the operator follows the two arithmetic values on which it is designated to work. Burroughs computers from the B5500 upwards are based on this system [Org73]. The push-down stack hardware consists of two or three registers at the top of the stack and a contiguous area of memory that permits the stack to extend beyond the registers.

Instructions in these machines are 12 bits long (packed four to a word), and fall into four categories: literal calls, operand calls and address calls (all of which involve pushing words into the stack), and operators. The operand calling mechanism is complex, since the actions which occur depend on the type of object found in store at the operand address (formed by treating part of the instruction or the top-of-stack word as an index, and adding it to the contents of a base register). The full word length is 51 bits, 48 bits of operand, instructions, etc, together with a 3-bit tag field which specifies the kind of information contained within the word. As examples, a word accessed by an operand call may be an operand (in which case it is simply stacked), a data descriptor (in which case the word addressed by the descriptor is stacked), or a program descriptor (in which case the program branches to a subroutine). Operators also use the tag field and treat words according to their tagged nature, so that there is only one ADD operator, for example, which takes any combination of single and double precision, integer or floating-point operands, and produces the appropriate result. Some operators require particular word types as operands, however, for which hardware checks are carried out, and an interrupt generated in case of error.

The designers of both the System/360 and the PDP-10 gave serious consideration to the use of a stacking machine, but both rejected it on a number of grounds. The IBM team argued that the performance advantage of a

push-down stack derived principally from the presence of several fast registers, rather than the way in which they were used or specified. Furthermore, they claimed that the proportion of occasions on which the top-of-stack location contained the item actually needed next was only about one-half in general use, so that operations such as TOP (extract from within stack) and SWAP (exchange top two stack items) would be needed. As a result, the static code requirements would be comparable with that obtained from a system with a small set of addressable registers.

The DEC team were concerned about high cost and possible poor performance in executing the operating system, LISP and FORTRAN. Burroughs' commitment to the essentially Algol-orientated stacking architecture was largely an act of faith, based on a belief held in several quarters in the early 1960s, that FORTRAN would disappear. It did not, has not, and probably will not for many years to come. It has been a constant source of difficulty for computer designers, and on Burroughs' machines most FORTRAN programs execute more slowly than corresponding Algol-like programs.

2.5 The MU5 instruction set

During the design of the MU5 processor the prime target was fast, efficient processing of high-level language programs in an interactive environment. This requirement led to the need for an instruction set which satisfied the following conditions

1. Generation of efficient code by compilers must be easy.
2. Programs must be compact.
3. The instruction set must allow a pipeline organisation of the processor leading to a fast execution rate.
4. Information on the nature of operands should be available to allow optimal buffering of operands.

Algol and FORTRAN were taken as being typical of two major classes of programming languages, distinguished by the use of recursive routines (or procedures) and dynamic storage allocation in the former, and non-recursive routines and compile time storage allocation in the latter, but having the following features in common

(a) Each routine has local working space consisting of named variables of integer (fixed-point) and real (floating-point) types, and structured data sets (arrays, for example) which are also named.

(b) A routine may refer to non-local names either of an enclosing routine or in a workspace common to all routines.

(c) Statements involve an arbitrary number of operands which may be real names, array elements, function calls or constants.

In order to reconcile item (c) with conditions (1) and (4), it was decided that there would be an address form corresponding to each of the different operand forms. However, it was felt that to have more than one such operand specification per instructions would conflict with conditions (2) and (3) (although subsequent simulation experiments (section 2.6) have shown that this is not necessarily true), and the use of addressable fast registers was also rejected. There were two arguments against registers. The first was the desire to simplify the software by eliminating the need for optimising compilers. The second was the desire to avoid the need to dump and restore register values during procedure entry and exit and process changes. The alternative arrangement proposed for MU5 was a small associatively addressed Name Store, as described briefly in section 1.3, and which is considered in more detail in chapter 3.

The choice of instruction format was therefore limited to zero or one-address. A zero-address system was rejected on two main grounds. Firstly, the hand-coded library procedures which would be used for input/output handling, mathematical functions, etc., almost all required more code in zero-address than in one-address instructions. This was because the main calculation, the address calculations and control counting tended to interfere with each other on the stack. Secondly, execution of statements such as

```
A := B + C
```

would actually run more slowly on a stacking machine with the proposed hardware organisation for MU5. This statement would be coded in one-address and zero-address instructions as follows

```
ACC = B          STACK B
ACC + C          STACK C
```

If the operands, A, B and C, came from main store, then operand accessing would dominate the execution times of these sequences, and both would be approximately the same. However, using the Name Store, the access times to these operands and to the stack would be the same, and the zero-address sequence would involve six stack accesses (the ADD involves two reads and a write) in addition to the operand accesses.

The instruction format eventually chosen for MU5 represents a merger of single address and stacking machine concepts. All the arithmetic and logical operations involve the use of an accumulator and an operand specified in the instruction, but there is a variant of the load order (*=) which causes the accumulator to be stacked before being re-loaded. In addition, there is

an address form (STACK) which unstacks the last stacked quantity. Thus the expression

```
RESULT := (A + B) * ((C + D)/(E + F))
```

would compile into

```
ACC = A
ACC + B
ACC *= C
ACC + D
ACC *= E
ACC + F
ACC /:  STACK
ACC * STACK
ACC => RESULT
```

The function /: is REVERSE DIVIDE; if the operand to the left of an operator is stacked, it subsequently appears as the right hand side of a machine function (as the dividend in this case). Thus, for the non-commutative operations - and /, the reverse operations -: and /: have to be provided. In pure stacking machines (such as Burroughs), all subtractions and divisons are implicitly reverse operations.

The instruction set is summarised in figure 2.7. There are two basic formats, one for computational functions and one for organisational functions (control transfers, base register manipulations, etc.). They are distinguished by the *type of function* bits and use four and six bits respectively to specify the function. The remaining nine or seven bits specify the *primary* operand. The *k* field indicates the kind of operand, and determines how the *n* field is interpreted. Where the *k* field indicates the Extended Operand Specification, the instruction is extended by an additional 16 bits in the case of a name, or by 16, 32, or 64 bits in the case of a literal.

The formal segmentation of the virtual address space, and the unique identification of store words belonging to different processes are achieved by splitting the address into three separate fields (thus producing 3-dimensional addressing), a 4-bit Process Number, a 14-bit Segment Number, and a 16-bit address defining a 32-bit word within a segment. Scalar variables are normally held in a single segment of a process at addresses formed by adding the 6 or 16-bit name in the instruction to the contents of the Name Base register (NB). The name segment number and the process number (both held in special registers in the processor) are concatenated with these addresses to form the full virtual addresses required for store accessing. The value held in NB is unique to the name space associated with each procedure, and is altered at each procedure change. Access to names of other

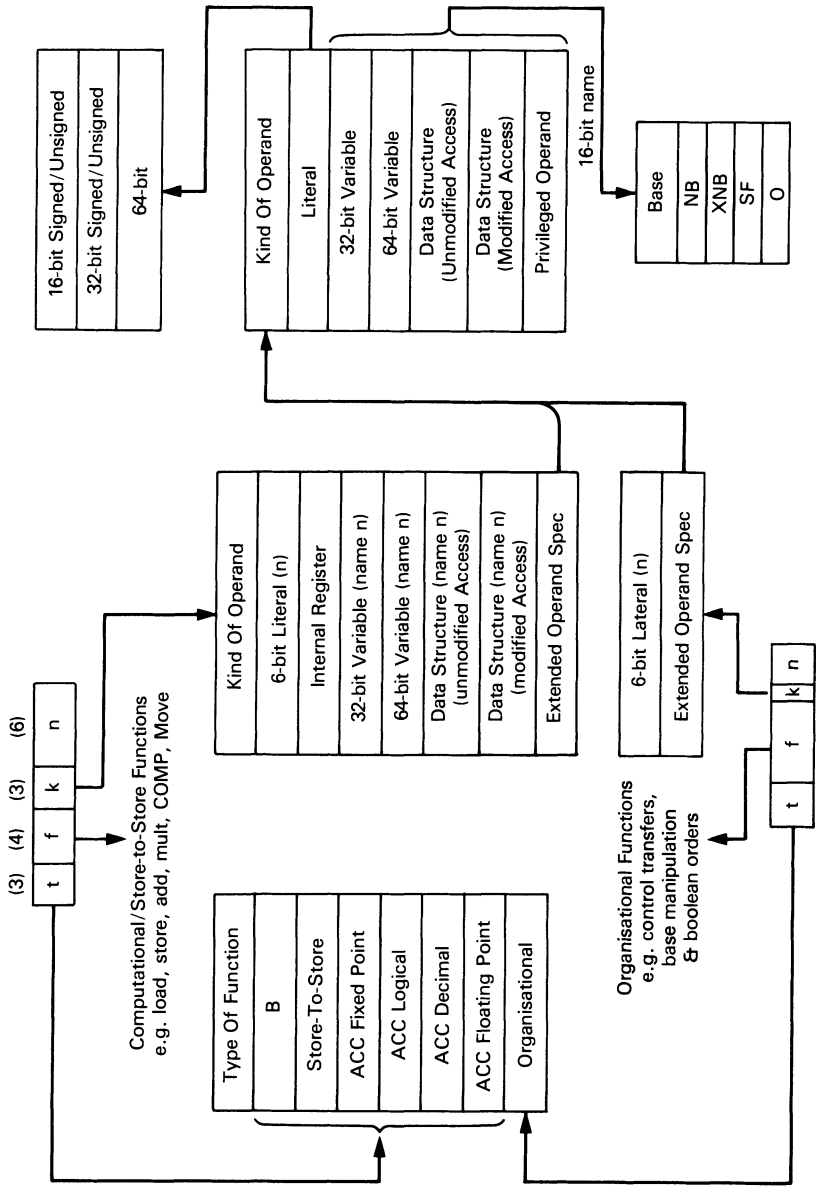

Figure 2.7 The MU5 instruction set

procedures and to common variables is made using the Extra Name Base (XNB), while using zero as base allows absolute addressing into the Name Segment (to access the Name Base values of different procedures, for example). The Stack Front register (SF) points to the area immediately beyond the name space of the current procedure, so that stacked variables are contained within the same address space as the names. SF is automatically incremented or decremented during the execution of a stacking function or a STACK operand access respectively.

Access to array elements is achieved through a descriptor mechanism. When a data structure element is specified by the *k* field, the corresponding primary operand is a 64-bit descriptor. This descriptor is sent to a D-unit within the processor, together with the contents of the B register if the *k* field specifies modification. The D-unit interprets the descriptor and makes a *secondary* store access for the element. It is this detachment of secondary addresses from instructions, and the use of names within instructions, which allow the full 30-bit virtual address space available to a process to be referenced by 16-bit instructions.

There are two main types of descriptor

String descriptor:	Ts	Length	Origin
Vector descriptor:	Tv	Bound	Origin
	8	24	32

String descriptors describe strings of bytes. The length field specifies the number of bytes, and the origin specifies the byte address of the first byte. Strings are normally used in conjunction with string processing orders (which may be two-address operations involving the use of two descriptors). These orders provide facilities for the manipulations of data records in languages such as COBOL, PL/1 and Algol 68 (move, compare, logically combine, etc.).

The size of the elements in a vector is specified within the type bits (Tv) and may be between 1 and 64 bits. If the operand access is modified, the modifier must be greater than or equal to zero and less than the bound. This check is carried out automatically by hardware with virtually no time penalty, and removes the need for the costly software bound check which is often required, particularly during program development. The descriptor accessing mechanism will be discussed in more detail in chapter 8 in relation to vector processing techniques.

2.6 Comparing instruction formats

Comparisons of instruction formats are often based on rather short sequences of hand-coded instructions which show the numbers of instructions required to obey various program statements. The expression

```
x := (a * b) = (c * d)
```

for example, could be coded in zero, one, two and three-address instructions (assuming all addresses are store addresses), as follows

0	1	2	3
LOAD a	LOAD a	TEMP = a	TEMP = a * b
LOAD b	* b	TEMP * b	x = c * d
*	=> TEMP	x = c	x = x + TEMP
LOAD c	LOAD c	x * d	
LOAD d	* d	x + TEMP	
*	+ TEMP		
+	=> x		
=> x			

Clearly the number of instructions to be obeyed decreases with increasing numbers of addresses in the instructions format, but if we assume that all function codes are 8 bits long and all addresses 16 bits long, then for this example the code space requirements are 18, 21, 25 and 21 bytes respectively for the zero, one, two and three-address instruction versions. Comparisons such as these tend to be misleading (especially when operands may, or may have to be in registers rather than store locations), and they do not give an accurate indication of the performance to be expected when complete programs are considered. A realistic comparison requires programs to be run on systems which have different numbers of addresses in their instruction formats, but are otherwise similar. Such machines do not exist in practice, but the ISPS architectural research facility developed at Carnegie-Mellon University [BSG*77] allows measurements to be made during the (simulated) running of programs on software models of different computer architectures, and hence allows for proper comparisons to be made.

ISPS is an implementation as a computer language of the ISP (Instruction Set Processor) notation, originally proposed by Bell and Newell [Bel71] for documentation purposes. The process of using the ISPS facility involves writing a description in the ISPS language of the actions of each instruction, and the registers and store locations on which they operate. This description forms the input to the ISPS compiler, which runs on a DEC PDP-10 computer and produces a PDP-10 program which is a simulator for the

architecture under investigation. Programs and data can be loaded into the memory of this model, and the programs can be executed. During execution the simulator keeps count of the number of bytes of information transferred in and out of the registers and store of the model, and the number of times labelled statements and procedures in the original description are executed. Various measurements can be derived from these counts, in particular the number of instructions obeyed by the model during the simulated execution of a program, and the number of instruction bytes transferred from memory (the dynamic code requirement).

This system has been used to evaluate both a two-address and a three-address version of the MU5 instruction set (designated MU5/2 and MU5/3 and having minimum instruction sizes of 24 and 32 bits respectively) against the one-address version [DIS80]. Standard test programs were used in the evaluation, the Wichmann synthetic Algol scientific benchmark [CW76] and the Computer Family Architecture (CFA) test programs developed at Carnegie-Mellon [FBSL77]. The results of these experiments are summarised in figures 2.8 and 2.9.

				Ratio to MU5	
	MU5	MU5/2	MU5/3	MU5/2	MU5/3
Code Space (in bytes)	1914	2013	1804	1.05	0.94
Instructions Executed	9633	7121	4934	0.74	0.51
Bytes Executed	21282	22220	20592	1.04	0.97

Figure 2.8 Results for CFA programs

The static code space requirements for the CFA programs run on MU5/2 and MU5/3 were not found to be significantly different from those for MU5, whereas the numbers of instructions executed were. The two-address version required roughly three-quarters as many instructions to be obeyed as the one-address version, and the three-address version only half as many. However, the two-address and three-address instructions are each proportionally longer than the one-address instructions, and counts of the numbers of instruction bytes executed showed that MU5/2 required 1.04 times as many bytes as MU5 and MU5/3 0.97 times as many. Clearly, neither of these figures differs significantly from the one-address figures, although the differences which do exist are in close agreement with the static code requirements.

For the Wichmann benchmark the code space requirements were found to be higher for both MU5/2 and MU5/3 with ratios of 1.18 and 1.22 respectively, and the numbers of instruction bytes executed were also found

	MU5	MU5/2	MU5/3	Ratio to MU5 MU5/2	MU5/3
Code Space (in bytes)	1411	1671	1720	1.18	1.22
Instructions Executed	107662	90527	75183	0.84	0.70
Bytes Executed	338854	386900	392082	1.14	1.16

Figure 2.9 Results for Wichmann benchmark

to be relatively higher than for the CFA programs, being 1.14 times higher for MU5/2 and 1.16 times higher for MU5/3. The numbers of obeyed instructions for the two and three-address versions again showed significant decreases relative to the one-address version, although these were less dramatic than for the CFA programs, with ratios of 0.84 and 0.70 respectively. The Wichmann benchmark contains many relatively long program statements with typically four arithmetic operators and an assignment operator (which can be coded more efficiently in one-address instructions than in two or three-address instructions) whereas the CFA programs are more typical of programs generally and contain a much higher proportion of short statements.

For both sets of measurements it is clear that the static and dynamic code requirements are approximately the same for all three systems, while the numbers of instructions obeyed show decreases as the number of addresses in the instruction format increases. In terms of overall system performance, however, these differences could only be translated into significant improvements if the instruction and operand accessing rates could be improved correspondingly, since we can assume that the arithmetic speeds would be the same in each case. Thus the instruction byte accessing rate would need to be around 1.5 times higher for the two-address system and nearly twice as high for the three-address system. Furthermore, one three-address instruction clearly requires three operands, and there would be a significant engineering problem in providing three operands at a rate sufficient to maintain the same instruction execution rate as that of a one-address system. Given the same instruction operand accessing rates, all three systems would have very similar performances, and we would be back where we started, with the intuitive feelings of the designer deciding the ultimate choice. Flynn *et al.* [FJW85] argue that the important issue is *transformational completeness* from language statements to instructions and show that this can best be achieved using a mixed format instruction set including both three-address and stack types.

2.7 Reduced instruction sets

Some of the instruction sets we have examined contain quite complex instructions, and we shall see in succeeding chapters that this can lead to considerable hardware complication in their implementation. Some designers [Pat85] have questioned the cost-effectiveness of this approach, arguing that in Reduced Instruction Set Computer (RISC) architectures all (of a small number of) instructions can be made to execute more quickly than any one instruction in a Complex Instruction Set Computer (CISC), and that it is in any case difficult for compilers to use CISCs effectively. To some extent this argument makes a virtue out of a necessity for designers of single chip computers, in which silicon area is at a premium and finding space for the decoding logic or microprogram control-store of a complex instruction set is difficult. The complexity is instead transferred to the compiler used with a RISC architecture, which often has to generate sequences of instructions corresponding to constructs which could be handled directly in a CISC architecture. Performance figures for RISCs are impressive, but it is difficult to make objective comparisons between RISC and CISC architectures, or even to find agreement on a definition of terms. Colwell *et al.* [CHJ*85] attempt to disentangle the terminology, folk-lore and hype, and they present a very useful comparison between RISC and CISC architecture.

For some RISC architectures performance comes from having overlapping sets of registers (ones which can be accessed with equal ease by a calling and a called procedure, for example), rather than from the nature of the instruction set. As the number of registers is limited, these registers are much more effective for some kinds of program than they are for others. What is important for both the compiler writer and the hardware designer is that the instruction set be easy to compile into and easy to decode out of; this has more to do with the instruction *format* having a regular structure than with the complexity of the instructions themselves.

3 *Storage Hierarchies*

Storage hierarchies have existed for as long as stored program computers themselves. The Ferranti Mark 1 installed at Manchester University in February 1951 had a main store made up of eight Williams Tubes each containing 32 40-bit words and a drum backing store having a capacity of 3.75 Kwords. Initially users were required to organise their own store transfers between a selected drum track and a selected pair of Williams Tubes, but the Mark 1 Autocode system introduced in 1954 carried out these tasks automatically on behalf of users and so made the two levels of storage appear to them as a one-level store. This arrangement was possible because the Mark 1 was a single user machine, and its performance was acceptable because the time required to transfer one drum track was roughly the same as the time for a programmed floating-point addition. Later developments in both architecture and technology led to the need for more sophisticated systems and in this chapter we shall consider the virtual memory and paging systems used in Atlas, the cache stores used in some models in the IBM System/360 and System/370 ranges, and the MU5 storage hierarchy. Before discussing these systems in detail we introduce the technique of store interleaving, a technique used to obtain increased performance in these and many other computer systems.

3.1 Store interleaving

Until parallel arithmetic hardware units were introduced in the late 1950s, the speed of random access storage devices was not a limiting factor on processor performance. The time for a floating-point addition in Mercury, for example, was 180 μs, while the core store cycle time was 10 μs. In Atlas, on the other hand, the time for a floating-point addition as 1.6 μs, while the core store cycle time was 2 μs. Thus, in the absence of some special technique, the time required to access both an instruction and its operand would have amounted to 4 μs, leaving the floating-point unit idle for much of its time. Part of the solution to this problem was to overlap arithmetic and store accessing operations for successive instructions, a development which we shall consider in more detail in chapter 4. In addition, the core store was made up of a number of independent stacks, each with its own access circuits, so that if successive accesses were to different stacks, one access would be able to proceed immediately after another without being

held up for a complete store cycle. Had it been possible to ensure that instructions and operands always occupied different stacks, then given the appropriate degree of overlap, instructions could have been executed at a rate of one per core cycle. Even in the absence of this possibility, benefit could still be gained from the independent access facilities of the stacks. The addressing of words in the four stacks of the 16 Kword version of Atlas installed at Manchester University was arranged in the following manner

STACK 0	STACK 1
00000	00001
00002	00003
.	.
.	.
08190	08191
STACK 2	**STACK 3**
08192	08193
08194	08195
.	.
.	.
16382	16383

Since instructions more often than not follow in sequence, and since successive instructions occupied adjacent stacks in this arrangement, instructions could be accessed in pairs in Atlas, so that two instructions were accessed (and subsequently buffered) in one core cycle. Assuming a relatively small program occupying stacks 0 and 1, then the two operands required could come from any of the stacks, giving different limitations on performance. In a sequence of instructions in which the two operands accessed by each pair of instructions were all in either stack 0 or stack 1, for example, then the minimum time for the execution of the pair would be 6 μs (2 μs for the instruction pair access plus 2 μs for each operand); for a sequence in which alternate operands were in stacks 0 and 1, the minimum execution time would be 4 μs, and for a sequence in which alternate operands were in stacks 2 and 3, the minimum execution time would be 2 μs (with the instruction pair and operand accesses fully overlapped). Assuming a random distribution of operands, however, we must consider all possibilities, as in the following table, and take an average to get an indication of the expected performance.

OPERAND PAIR DISTRIBUTION		EXECUTION TIME
0	0	6
0	1	4
0	2	4
0	3	4
1	0	4
1	1	6
1	2	4
1	3	4
2	0	4
2	1	4
2	2	4
2	3	2
3	0	4
3	1	4
3	2	2
3	3	4

The total of the execution times for these sixteen possibilities is 64 μs, corresponding to an average of 4 μs for the execution of a pair of instructions or 2 μs per instruction. For sequences of computational instructions this figure was in fact achieved in Atlas. The overall average instruction time was higher, but this was due to the effects of control transfer instructions, as we shall see in chapter 5, and not to store clashes. Thus, by having a store made up of four independent stacks with two-way interleaving of addresses, the execution time of computational instructions was improved by a factor of two.

In Atlas the single floating-point arithmetic unit was used not only for addition and subtraction, but also for lengthier operations such as multiplication and division (which took 4.9 μs and between 10.7 and 29.8 μs respectively), and the average rate at which this unit could execute arithmetic operations was well matched to the average store accessing rate. In the CDC 6600, on the other hand, the use of multiple functional units meant that instructions could be issued at the much higher maximum rate of one every 100 ns. The cycle time of main storage modules (or *banks* in CDC terminology) exceeded the basic processor clock period by a factor of 10 and, since the 24 computational registers offered a relatively small amount of buffering, a much higher level of store interleaving was required to sustain the instruction execution rate. This same ratio of processor clock period to store cycle occurs in the CDC 7600 (27.5 ns and 275 ns respectively) and in order to minimise the effects of store clashes on overall performance, the 60-bit word main stores of both these machines are 32-way interleaved.

Because of the factor of 10 difference between the processor clock and store cycle time, a maximum of 10 storage banks can be in operation at one time. This situation only arises during block copy operations between the Small Core Memory (SCM) and Large Core Memory (LCM) in the 7600 and between Central Storage and the Extended Core Storage in the 6600. In random addressing of the 7600 SCM for instructions, program data and input-output channel data, an average of four SCM banks in operation at one time is more normal. This means that there is a high probability that any single request will be to a free storage bank, and will be serviced immediately, and so much of the benefit of this storage arrangement comes not so much from interleaving *per se*, but from the fact that the store is made up of a large number of relatively small independent modules. The interleaving of these modules has a significant effect on performance when many successive accesses occur at sequential addresses, such as during instruction fetch sequences, and more importantly during the processing of long vectors. Hence, interleaving is extended to even higher levels in the CDC CYBER 200 Series of vector processors. The 32-bit wide storage banks of the CYBER 205, for example, are 256-way interleaved, and evidence for the necessity for such a high level of interleaving comes from the fact that reducing this interleaving to 128-way can cause a reduction in performance of nearly 40 per cent for some types of operation.

3.2 The Atlas paging system

In early computer systems peripheral transfers took place under direct program control using input/output instructions which normally required several milliseconds for their completion. By the time the Atlas computer was developed, the disparity between processor and peripheral speeds had become so great (of the order of 1000:1) that it was essential that peripherals be operated on a time division basis, with each peripheral interrupting the main program when it required attention. Since the routines controlling these peripherals were required to be resident in store concurrently with the main user program, and since the system was extended to allow several user programs to co-exist in store, it was no longer feasible either to optimise programmed transfers between the core and drum stores (so as to eliminate the drum access time of 6 ms) or even to allow users to program transfers within a shared physical store when they had no knowledge of other users' requirements. Consideration of these problems led to the idea of providing users with a *virtual memory* and to the development of a paging system to map virtual addresses on to the real, physical storage devices. Thus the integration of the Atlas core-drum combination into an apparently *one-level store* was a function of the machine itself, rather than a software facility, as it had been on earlier machines such as Mark 1 and Mercury.

3.2.1 Real and virtual addresses

In a virtual addressing system the programmer is allowed to use all available store addresses, but these virtual addresses do not refer directly to real store addresses, and each user programmer can assume that he or she has sole use of the entire address range. In the hardware the virtual address space is considered to be split up into a series of contiguous pages of some convenient size (512 words in Atlas), which may be represented as follows

PAGE	0	1	2	3	4	5	6	7	8	. .
WORD	0	512	1024	1536	2048	. .				

Not all virtual pages contain information, since virtual addressing allows users to locate their code and data in areas which they find convenient, and which need not be contiguous. Thus the core and drum in Atlas might have contained, at some stage in the execution of a particular user program, the following virtual pages

CORE BLOCK	0	1	2	3	4	5	. .
VIRTUAL PAGE	1	4	7	**	40	0	. .

DRUM BLOCK	32	33	34	35	36	37	38	39	. .
VIRTUAL PAGE	12	13	3	9	2	6	5	41	. .

where ** indicates an empty core block. If the program called for a virtual address in page 7, for example, the hardware would have been required to translate this into a request to a real address in block 2 of the core store. This required the existence of a page table relating virtual and real addresses, as follows

Virtual	Real
0	5
1	0
2	36
3	34
4	1
5	38
6	37
.	.
.	.
.	.

3.2.2 Page address registers

In developing the hardware to implement the translation between virtual and real addresses, the designers of Atlas made the following observations

1. Every virtual address generated within the processor has to be translated into a real address by a table look-up, and this involves an extra contribution to the store access time.

2. If the number of virtual pages is large, the table is large and therefore slow to access.

3. The requested virtual addresses must be in the core store most of the time if the system is to operate efficiently; a page on the drum must be moved into core before the processor can access individual words within it. Thus a small table containing only the entries corresponding to pages in core can be looked at initially, and the full table accessed only when the small table does not contain the required page address.

The system of Page Address Registers (PARs) used to implement this table is shown in figure 3.1. To locate a page, the 11 PAGE digits of the required address were presented to an associative store containing one register for each of the 32 blocks of core store. These registers were implemented in such a way that a check for equivalence between the presented PAGE bits and the contents of each register took place in parallel, thereby minimising the address translation overhead. If an equivalence occurred between the required page and a page address stored in one of the associative registers, that register indicated a *1* while the others indicated a *0*. Thus for virtual page 7 in our example, line 2 would have contained the value 7 and given equivalence when an address in page 7 was requested. The 32 outputs from the associative registers were encoded into a 5-bit binary number and concatenated with the 9 LINE digits from the required virtual address to produce the real address to be sent to the core store.

If no equivalence occurred, the required page was not in the core store, and the program could not continue until it was. A page fault interrupt was therefore generated and the assistance of the operating systems (known as the Supervisor in Atlas) invoked. The Supervisor used the full page table in order to locate the required page, initiated the necessary transfer from the drum into an empty core block, and set up the corresponding Page Address Register. It also initiated the transfer of an existing page from the core back to the drum in order to maintain an empty core block in readiness for the next page fault.

The choice of page to transfer out of core was made by a *learning program* which attempted to optimise these transfers such that pages currently in use

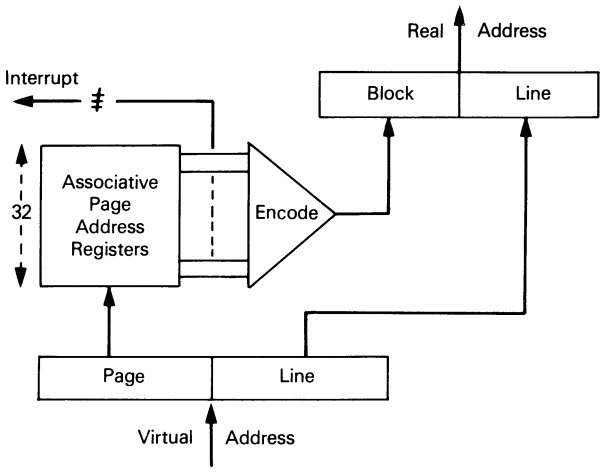

Figure 3.1 The Atlas page address registers

were not chosen for rejection. The operation of this program was assisted by the provision of a *use* digit in each page register, which was set every time the corresponding page was accessed. All 32 digits were read and reset at regular intervals (normally every 1024 instructions), and their values added to a set of running totals maintained and used by the learning program. This program was the first example of a page rejection algorithm. Page rejection algorithms now form an important part of most operating systems and will not be considered further here. The interested reader should consult [Lis84], for example.

Each of the Page Address Registers in Atlas also contained another digit, the *lock-out* digit, which prevented user programs from accessing a page even though it was core-resident. This served two purposes. Firstly, it prevented programs from accessing pages to which drum, magnetic tape or peripheral transfers were taking place; with the lock-out digit set, access to that page was only permitted if the address had been generated by the drum or tape systems, or as a result of a peripheral transfer. Secondly, it prevented interference between programs co-existing in the core store; whenever the Supervisor caused a change from one program to another, it protected the pages belonging to the suspended program by setting their lock-out digits. In the subsequent Manchester University computer (MU5), this inter-program protection was extended by the inclusion of a 4-bit process number in each paging register. The *current process number* was held in a register within the processor and updated by the operating systems at a process change. The contents of this register were concatenated with each

address generated by the processor, so that addresses could only produce equivalence in those paging registers that contained the appropriate process number.

The Atlas virtual memory/paging system, and the concept of dynamic address translation which it introduced, have had a profound influence on the design of many other computer systems, among which are the DEC PDP-10, the IBM System/370 series, the CDC STAR-100, and their successors. The full implications of virtual memory and its attendant advantages and disadvantages are still subjects of controversy. An excellent survey of the issues involved can be found in [Lis84].

3.3 IBM cache stores

3.3.1 The System/360 Model 85 cache

A different form of storage hierarchy from that used in Atlas involves cache storage, first introduced into the IBM System/360 series with the Model 85 [Lip68]. Here all program-generated addresses are real addresses referring to main (core) store locations, and the semiconductor cache store, which is invisible to the programmer, is used to hold the contents of those portions of main storage currently in use by the program. Thus it is provided solely as a means of improving performance, rather than as a means of solving the problems involved in multi-programming, and may be thought of as a real address paging system.

The cache mechanism operates by dividing both cache and main storage into logical sectors, each consisting of 1024 contiguous bytes starting on 1 Kbyte boundaries. During operation, cache sectors are assigned to main storage sectors in current use, and a sector address register associated with each cache sector contains the 14-bit address of the main storage sector to which it is assigned (figure 3.2). Since the number of cache sectors (16-32, depending on the configuration) is smaller than the number of main storage sectors (512-4096, depending on configuration), most main storage sectors do not have a cache sector assigned to them. However, the localised nature of store accessing exhibited by most programs means that most processor accesses are handled by the cache (which operates at the 80 ns processor rate rather than by main storage (which has a 1.04 μs cycle time).

Each sector within the cache is made up of 16 blocks of 64 bytes and each block is marked with a *validity* bit. Whenever a cache sector is re-assigned to a different main storage sector, all its validity bits are re-set, and the block containing the required store word in the new sector is accessed from main storage. The validity bit for this block is then set and the sector address register updated. Further blocks are accessed and their validity bits set as required.

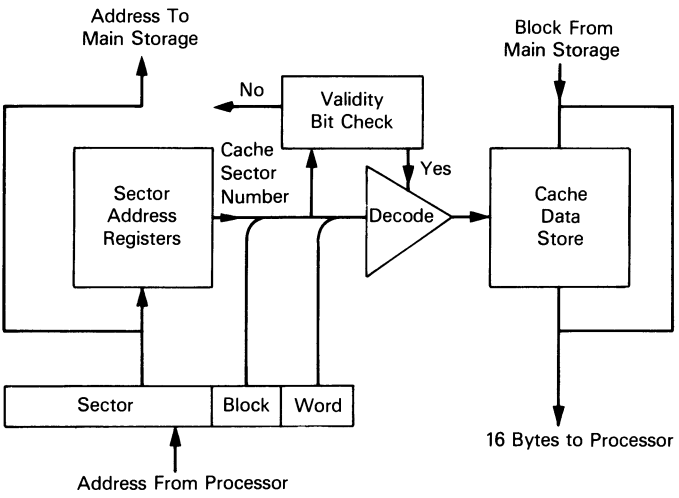

Figure 3.2 IBM System/360 Model 85 cache organisation

The sector address registers constitute an associative store. Whenever an address is generated which requires an operand to be fetched from store, the sector bits within the address are presented for association. If a match occurs a 4-bit tag is produced indicating the cache sector address, and this is used in conjunction with the block bits to select the appropriate validity bit for examination. If a match occurs in the sector field, and the validity bit is set, the appropriate 16-byte word is read out in the next machine cycle and returned to the processor. Throughput is maintained at one access per cycle by overlapping association and reading for successive instructions.

If a match occurs in the sector field, but the validity bit is not set, the required block is read from main storage. Reading one 64-byte block involves one access to each of the four interleaved storage modules making up the main store, since each is 16 bytes wide. The delay experienced by the processor as a result of these main storage accesses is minimised by always accessing the required 16-byte word first in the cycle of four, and by sending this word directly to the processor at the same time as loading it into the cache.

If a match does not occur, then a cache sector must be re-assigned to the main storage sector containing the failing address. A mechanism is therefore required to select a cache sector for re-assignment. The problem is similar to that involved in page rejection algorithms, except that in order to maintain performance, the algorithm must be implemented in hardware. The Model 85 cache implements a *least recently used* algorithm by maintaining an activity list with an entry for each cache sector. Whenever a

sector is referenced it is moved to the top of the list by having its entry in the activity list set to the maximum value, while all intervening entries are decremented by one. The sector with the lowest activity list value is the one chosen for re-assignment.

Re-assignment does not involve copying back to main storage values in the cache updated by the action of *write-to-store* orders. Whenever such an order is executed, both the value in the cache and that in main storage are updated, a technique known as *store-through*. Furthermore, if the word being updated is not held in the cache, the cache is not affected at all, since no sector re-assignment or block fetching takes place under these circumstances. While the store-through technique has the advantage of not requiring any copying back of cache values at a sector re-assignment, it also has the disadvantage of limiting the execution rate of a sequence of write orders to that imposed by the main storage cycle time. The store-through principle was rejected by the designers of MU5, as we shall see in section 3.4, and abandoned by IBM in the design of the 3090.

Simulation studies of the Model 85 cache showed that the average performance of the cache/main storage hierarchy over 19 different programs corresponded to 81 per cent of a hypothetical system in which all accesses found their data in the cache. The average probability of an access finding its data in the cache for these programs (the hit rate) was 96.8 per cent. There do not appear to have been any subsequent measurements on real Model 85 systems to confirm or deny these predictions, but certainly a different arrangement was introduced in the System/360 Model 195 [MW70] and carried through into the System/370 Models 165 and 168. The average hit rate in the Model 195 cache was shown to be 99.6 per cent over a range of 17 job segments covering a wide variety of processing activities.

3.3.2 The System/370 Model 165 cache

The Model 85 cache is characterised by having a relatively small number of fairly large contiguous units of buffer store. While this can perform well in a single program environment, it is unlikely to do so in a multi-access multiprogramming environment where rapid switching between a number of store co-resident programs is essential. What is needed in these circumstances is a relatively large number of (necessarily) small contiguous units of buffer store. Smaller versions of the Model 165, for example, have 8-Kbyte cache stores consisting of 256 blocks each of 32 bytes. Accessing these blocks through a 256-word 19-bit associative store is not practicable in an 80 ns processor cycle, however, so cache and main storage are logically divided into 64 columns (figure 3.3).

Each column in the cache store contains 4 blocks and each column in main storage 512 blocks (in a 1 Mbyte version of the model). The block

address registers constitute an address array made up of 64 columns, each containing four 13-bit entries. Thus of the 19 bits of an incoming address used to access a block, the 6 low order bits are used to select a column, and the 13 high order bits are presented for association in the selected column of the address array. This arrangement is referred to as set associative mapping; two adjacent blocks from main storage contained within the cache will be referenced by identical entries in adjacent columns of the address array.

A validity bit associated with each block indicates whether or not an address giving a match on association refers to valid data; if not the required data is accessed from main storage. Apart from the factor of two difference in store widths, data paths and block sizes, the arrangements for transferring data between the cache and main storage are identical in the 370 Model 165 to those in the 360 Model 85. The cache replacement algorithm is also

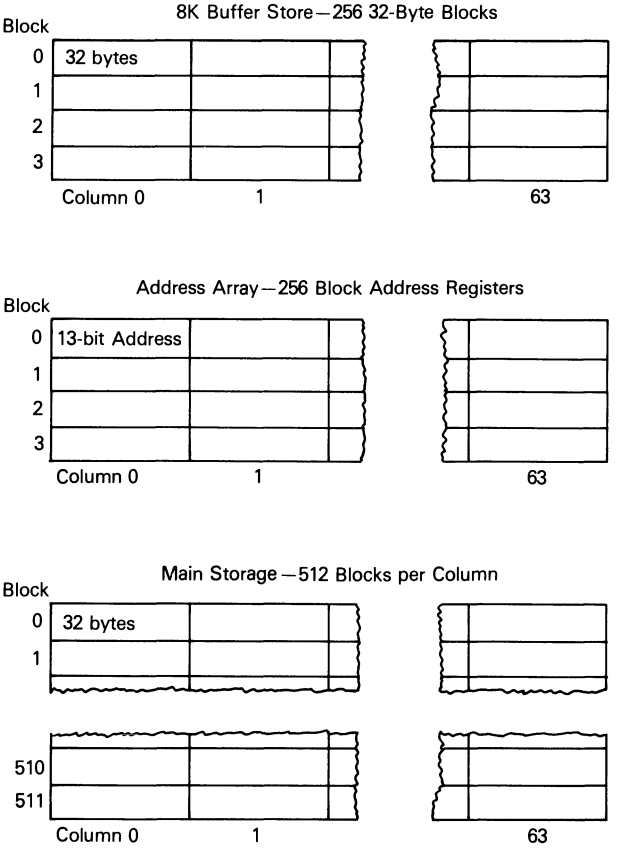

Figure 3.3 IBM System/370 Model 165 store arrangement

the same, except that in the 370 Model 165 a replacement array is required containing a separate activity list for each of the 64 columns.

3.3.3 The 3090 cache

By the early 1980s processor cycle times in IBM machines had been reduced so much in relation to central storage cycle times that the penalty of storing through on every store action had become intolerable. Thus in the 308X and the 3090 multiprocessor system [Tuc86] a *store-in-cache* design is used. In the 3090 a store operand is written into the cache of the processor performing the store operation, but it is not sent to central storage. A cache line (the unit of transfer between the cache and central storage) is kept solely in that cache until another processor requests that line or until the cache location is required for a different line.

Access to central storage from the processors in a 3090 system is provided through a *system control element*. Using the store-in-cache principle, any lines in a cache that have been stored into become, in effect, part of the central storage system. Thus the latest copy of a variable may be in the processor's local cache, in central storage, or in the cache of another processor. The design of the system control element must therefore take this into account. Further discussion of the problems of cache consistency in multiprocessor systems can be found in Volume II.

3.4 The MU5 Name Store

In the late 1960s a quite different approach to high-speed buffering was taken in the design of the MU5 computer. As we saw in section 1.3, the use of fast programmable registers in MU5 was rejected in favour of a small associatively addressed buffer store containing only named scalar variables and forming part of a one-level store with the Local Store of the processor [IH77,MI79]. Simulation studies of this *Name Store* indicated that a hit-rate of around 99 per cent would be obtained with 32 words of store, a number which it was technologically and economically feasible to construct and operate at a 50 ns rate, and which could be conveniently fitted on to two of the platters used in the construction of MU5.

Thus the address and value fields of the Name Store (figure 3.4) each occupy one of these platters, and form two adjacent stages of the Primary Operand Unit (PROP) pipeline, through which instructions move in a series of beats (section 4.2). At each beat a virtual address generated in the previous two stages of the pipeline is copied into the Interrogate Register (IN), and concatenated with the contents of the Process Number register (PN), for presentation to the address field of the Name Store. A full virtual address in MU5 consists of a 4-bit Process Number, a 14-bit Segment and 16

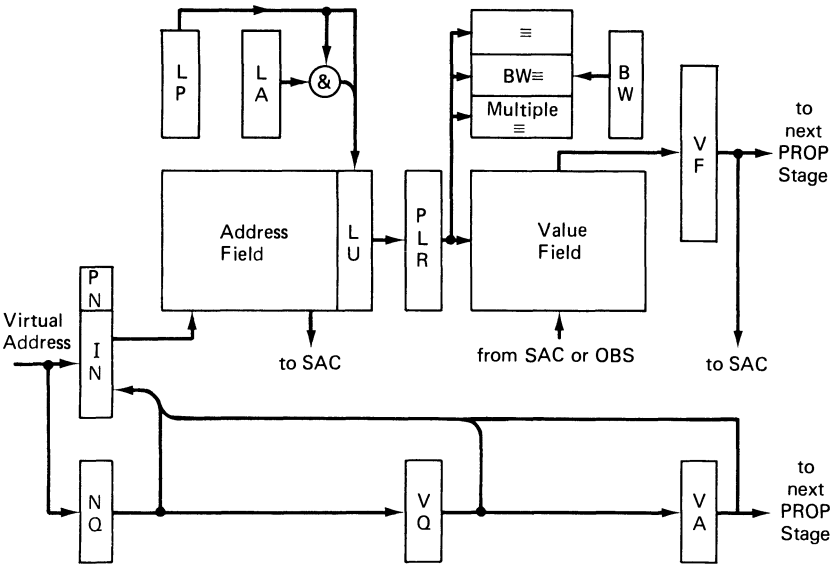

Figure 3.4 The MU5 PROP Name Store

bits which identify a 32-bit word within a segment. Addresses presented to the Name Store do not contain the Segment Number, however, since it was assumed at the design stage that the Name Segment would always be zero, and only 15 of the word address bits are used, referring to 64-bit operands. Where necessary, a 32-bit operand is selected from within a 64-bit word in a later stage of the pipeline.

3.4.1 Normal operation

If the presented virtual address gives a match in the associative store, and the corresponding Line Used digit (equivalent to the *validity bit* in an IBM cache store) is set to 1, then an equivalence has occurred, and on the next pipeline beat a digit is set in the PROP Line Register, PLR. The digit in PLR then selects a register in the Value Field, and the 64-bit word is read out and copied into the Value Field Register (VF) by the next beat. At the same time, checks are made to determine whether

1. an equivalence occurred
2. a *B Write Equivalence* occurred
3. multiple equivalence occurred.

The check for equivalence simply requires an *OR* operation on the digits in PLR. If no digit is set, however, this indicates non-equivalence, and the Name Store is updated by transferring a new word into it and discarding an old one. It would clearly be inefficient to enter software to organise this one-word transfer, so the transfer is controlled directly by hardware. A *B Write Equivalence* occurs if the line giving equivalence is the target line of an outstanding $B \Rightarrow$ *name* order, and this causes a pipeline hold-up (section 4.2.2) until the operand value has been returned from the B-unit (figure 3.5) and written into the Name Store. A multiple equivalence occurs if a hardware malfunction in the associative field causes more than one line to give equivalence. In this case an interrupt is generated.

When a Name Store entry is replaced the hardware must take into account the effect of store orders. To maintain the speed advantage of the Name Store, store orders only update the value of an operand in a Name Store, rather than operating on the store-through principle used in IBM System/360 and System/370 cache stores. The Name Store contains values which would be held in registers in these IBM machines, and it would clearly be inappropriate for these values to be written back to the main store each

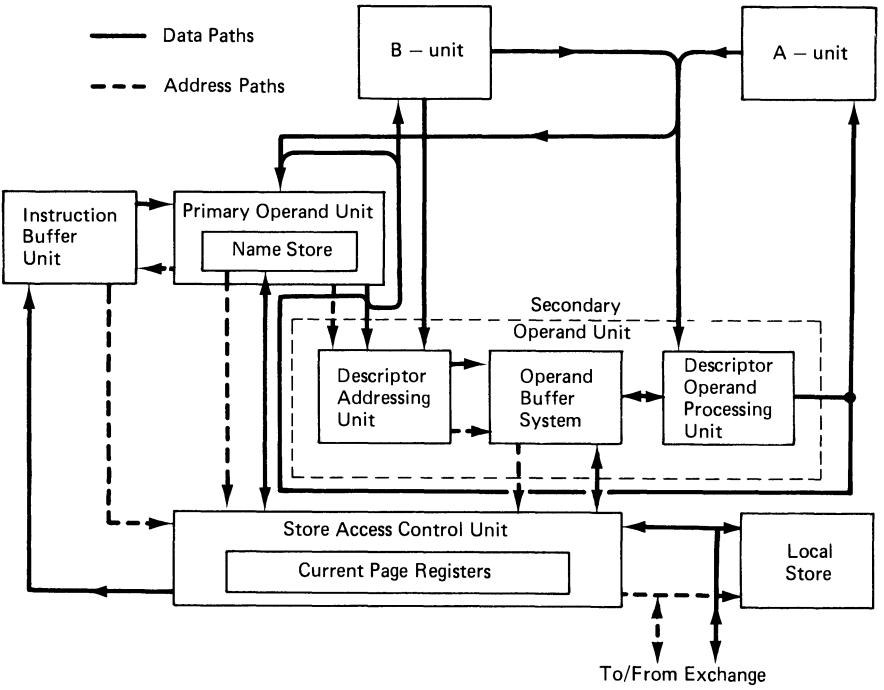

Figure 3.5 The MU5 processor

time they were altered. Thus the old word may have to be copied back to the Local Store before it is overwritten. The decision concerning which line to replace requires the use of a replacement algorithm, and the effects of various replacement algorithms were studied by simulation before the Name Store was built. These varied from a simple cyclic replacement algorithm, requiring a minimum of additional hardware for its implementation, to a multiple-use digit algorithm requiring, for a 32-line store, 32 5-digit counters. Very little difference in performance was found among these different algorithms, and the simple cyclic one was therefore chosen.

The actions which take place when a non-equivalence is detected also depend upon whether the instruction is an Accumulator (ACC) order destined for the A-unit and whether the required operand is already in an additional Name Store situated close to the A-unit. In order to improve the accessing of vector elements in MU5 (section 8.1), the Secondary operand Unit (SEOP) incorporates an Operand Buffering System (OBS) in addition to the two parts of the D-Unit (the Descriptor Addressing Unit and Descriptor Operand Processing Unit in figure 3.5). As a result there are some ten pipeline stages between the end of PROP and the A-unit through which all instructions destined for the A-unit must pass (in order to maintain the correct program sequence). Thus if a name held in the PROP Name Store were to be used to accumulate a total calculated by ACC orders in a small program loop, the order reading the total from the Name Store would have to be held up until the value calculated by the previous pass through the loop had been returned. The solution to this problem was the provision of 24 lines of Name Store in OBS.

The OBS Name Store is meant to keep names used by ACC functions and the PROP Name Store to keep those used by non-ACC functions. Thus for a non-ACC order the normal situation is for equivalence to occur in the PROP Name Store, while for an ACC order the normal situation is for a non-equivalence to occur in the PROP Name Store and equivalence to occur in the OBS Name Store. However, the same name might be accessed by both kinds of orders, and the hardware must guard against the possibility that a name is in the *wrong* Name Store.

3.4.2 Non-equivalence actions

When a PROP Name Store non-equivalence occurs a *non-equivalence* digit is set in the function register associated with VF. This causes the normal control signals decoded for the instruction in the next pipeline stage to be overridden. As a result, a special order is sent through the pipeline to OBS carrying the non-equivalence address (including the Name Segment number). OBS accesses its Name Store, and if it finds equivalence, returns the 64-bit store word to PROP. If it does not find equivalence, it makes an

access via the Store Access Control Unit (SAC) to the Local Store on behalf of PROP, so that the 64-bit store word will be returned direct to PROP from the Local Store.

In PROP the occurrence of a Name Store non-equivalence causes the initiation of the next pipeline beat to be held up until the appropriate actions have been completed (figure 3.6). The first actions are the preparation of a line in the Name Store to receive the new address and store word and the copying of the new address, currently held in register VA, into the Interrogate Register (IN). The Name Store line to be replaced is indicated by the Line Pointer Register (LP), which contains one bit corresponding to each line in the Name Store and has, at any time, only one digit set to a 1. Thus LP is simply copied into register PLR in order to select the line for replacement. Two possible conditions are checked, however, and if necessary dealt with, before the line is ready to be overwritten with the new address and value. The first is that the selected line may be the target line of an outstanding B => *name* order (section 4.2.2). If it is, LP is moved on to select the next line. This is done by first copying the contents of PLR back into LP, and then copying LP into PLR. The outputs from PLR are routed back into the Line Pointer with a 1 digit shift, thus implementing the simple cyclic replacement algorithm. The second condition is that the selected line may have been altered by the action of a store order. This is checked using the appropriate digit in the Line Altered register. If the line has been altered, the virtual address and value are read out, and a write request is made to SAC. The selected line is then ready for overwriting.

The next action which occurs depends on whether the store word is returned from SAC or OBS, or on whether a CPR (Current Page Register) non-equivalence occurs, in which case no store word is returned, but the interrupt sequence is entered instead. If the store word comes from SAC, then when it arrives the address and value are written into the Name Store, the contents of PLR are copied back into the Line Pointer in preparation for the next non-equivalence, and the corresponding bits in LU and LA are set to 1 and 0 respectively.

Although the actions needed to update the Name Store are complete at this point, the contents of the PROP Line Register and Interrogate Register no longer correspond to the orders in the pipeline, and must be restored. PLR is restored first by copying the address in register VQ into IN, and after a delay to allow for association, PLR is re-strobed[1]. The address in register NQ is then copied into IN, and the actions are complete. These complications could have been avoided if a longer PROP pipeline beat time had been adopted in the design.

[1]Preserving a previous copy of PLR for use at this time would not be satisfactory, since it might be the line newly overwritten.

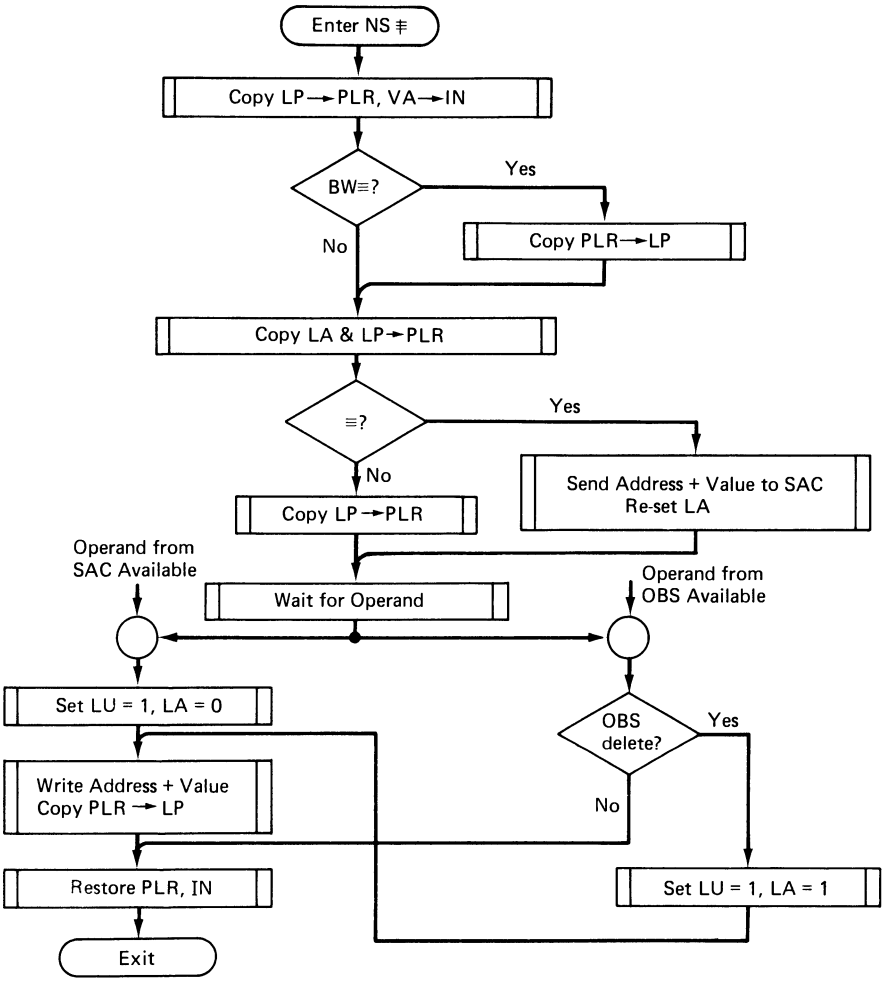

Figure 3.6 The Name Store non-equivalence routine

In the case where OBS indicates that the required store word is in its Name Store, then if all 64 bits are to be overwritten by a store order, the store word is not actually returned to PROP. In this case OBS simply deletes its copy and PROP assumes it to have arrived immediately, thereby reducing the non-equivalence delay time. In cases where the operand is not deleted from the OBS Name Store, the only further action required in PROP is the restoration of PLR and IN before the routine is complete[2]. If

[2]The clearing of a line earlier in the routine only copies its contents back to the Local Store and sets the line to be *unaltered*, so the Name Store is not disturbed.

the operand has been deleted from OBS and returned to PROP, the actions are as for an operand from SAC.

3.4.3 Actions for ACC orders

Although the normal situation for an ACC order using a named operand is for non-equivalence to occur in the PROP Name Store and equivalence to occur in the OBS Name Store, equivalence may be found in the PROP Name Store on some occasions. For example, a 32-bit variable required for an ACC order might be contained in the same 64-bit store word as a 32-bit variable already in use by B orders. Unless the ACC order finding equivalence is a store order, the operand is read out as for a non-ACC order and carried through to OBS in the same way as a literal operand, so that no access is made to the OBS Name Store. If the ACC order finding equivalence is a store order, however (*ACC Write Equivalence*) then action is taken to delete the word from the PROP Name Store.

The actions for an *ACC Write Equivalence* are initiated in a similar way to those for a non-equivalence for a non-ACC order. These actions are similar to those involved in preparing a line for a non-equivalence and in restoring the pipeline at the end. The appropriate address and value are read out of the Name Store and sent to SAC, and the appropriate digits in the LU and LA digits are then set to zero, in order to mark the line empty. When the order reaches OBS its operand will not, of course, be found in the OBS Name Store, and the normal OBS non-equivalence actions will be initiated.

3.4.4 Name Store performance

The management of the Name Stores of MU5 is clearly complex by comparison with the cache approach. This complication was thought to be acceptable because it was expected to give rise to a high *hit-rate* and a high rate of instruction execution between *misses*. Measurements of Name Store performance were made for a set of 95 programs containing both FORTRAN and Algol jobs ranging in complexity from simple student exercises to large scientific programs. For most programs it was found that 80 per cent ($\pm$ 5 per cent) of operand accesses were to named variables, that no more than 120 names were used in any one program, and that in all programs 95 per cent of name accesses were to fewer than 35 per cent of the names used. These figures confirmed the Atlas results which inspired the idea of using a Name Store, but the figures for hit-rates were not as good as had been anticipated. Thus although 96.1 per cent of name accesses found their operands in one or other Name Store, only 86 per cent of name accesses found their operands in the correct Name Store.

The comparatively high, and largely unforeseen, amount of interaction between the two Name Stores was found to be due to the procedure calling mechanism used in MU5. Parameters are normally passed between procedures by stacking them in the PROP Name Store, but in many cases they may subsequently be used with ACC orders. Conversely, it is possible for a particular word required to be in the OBS Name Store in one procedure to be required to be in the PROP Name Store in a subsequent procedure. It is by no means clear what is the best solution to this problem.

3.5 Data transfers in the MU5 storage hierarchy

Operand accesses which cannot be satisfied by the buffer stores within the MU5 processor cause requests to the Store Access Control Unit (SAC). SAC contains 32 Current Page Registers (CPRs) which translate the presented virtual address into a real address to be sent to the Local Store. Current Page Registers differ from Page Address Registers as used in Atlas, for example (section 3.2), in that they do not provide full cover of the addressable real store, and do not, therefore, correspond one for one with the pages in real store. Thus, whereas in Atlas a real address was obtained by encoding the 32 Page Address Register outputs into a 5-bit binary number, this cannot be done with CPRs. Instead there is a real address register corresponding to each virtual address register, and a real address is obtained from the CPRs in exactly the same way as an operand is obtained from the Name Store. Unlike the Name Store, however, software action is invoked in the event of a CPR non-equivalence (frequently referred to as a page fault). Either a block of data must be moved into the Local Store and a CPR set up to address it, or a CPR may simply be set up to address an existing Local Store page.

The MU5 *one-level* store extends over two further levels of real store, a 2.5 μs cycle-time Mass (core) Store and a Fixed-head Disc Store. Pages are brought into the Local Store from either the Mass Store or the Fixed-head Disc Store, but are rejected first to Mass and later to the Fixed-head Disc. The *one-level* store contains all the working space and files needed by current terminal users and active background jobs. Most of the files, however, are stored on large capacity discs attached to a separate PDP-11/10 minicomputer.

3.5.1 The Exchange

In order to provide a completely general and flexible interconnection scheme, allowing for data transfers between all these various stores and processors, an *Exchange* was developed as part of the MU5 project [LTE77,MI79]. Logically it is a multiple-width *OR* gate operated as a packet switching system

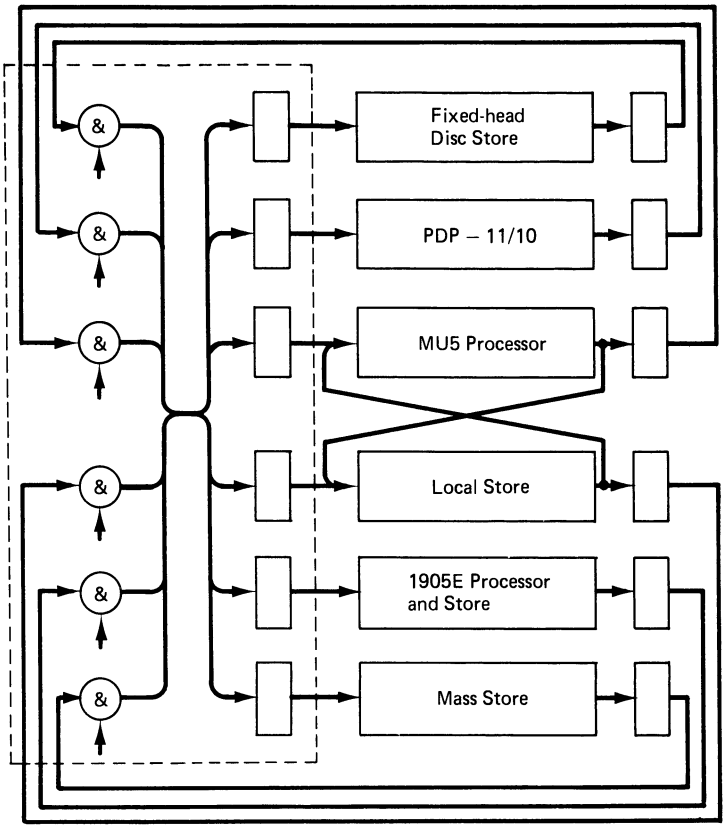

Figure 3.7 The MU5 Exchange system

at the star point of the interconnections. This configuration involves only a very short common path for transfers between the various units, allowing a much higher data rate than would be possible with a distributed highway or *bus* system. Thus transfers can occur at a rate of one every 100 ns, and each can involve a 64-bit data word together with address and control bits.

Each unit attached to the Exchange (figure 3.7) provides a set of parallel inputs to the *OR* gate and each is connected, via its own buffer register, to the output of this *OR* gate. The Exchange operates by time sharing the *OR* gate between the units. Thus the transfer of a block of words from the Fixed-head Disc Unit to the MU5 Local Store, for example, involves a succession of 64-bit word transfers from the Disc, as *Sending Unit*, to the Local Store as *Receiving Unit*, with the *OR* gate connecting these units for

the duration of each word transfer rather than for the whole duration of the block transfer. Other transfers can therefore be accommodated during this period, so that the 1905E computer, for example, can make read requests to the Mass Store. Two transfers are required for a read request, one in which, in this case, the 1905E as Sending Unit sends the address and appropriate control information through the Exchange to the Mass Store as Receiving Unit, and a subsequent *data available* request from the Mass Store as Sending Unit to the 1905E as Receiving Unit in order to return the data read out from the specified location in the Mass Store.

The requests from different Sending Units arrive at the Exchange completely asynchronously, and much of the control logic within the Exchange is concerned with scheduling transfers through the *OR* gate on a priority basis (section 3.5.2). A substantial proportion of these transfers are organised by a Block Transfer Unit (BTU). The BTU is activated by the MU5 processor, which sends to one of its four channels the starting address for the transfer in each store, together with the block size and a start command. The processor is then free to continue computation, while the BTU generates the necessary requests, via the Exchange, to the Mass and Local stores to carry out the transfers. At the end of the block transfer the BTU activates a processor interrupt.

The width of the data path through the Exchange *OR* gate is 8 bytes (plus one parity bit per byte) corresponding to the width of the data paths within the MU5 processor, and also exceeding the minimum width necessary for some of the units to be able to communicate at all. The Fixed-head Disc, for example has an effective data rate of 0.5 μs/byte, while the storage modules constituting the Mass Store have a cycle time of 2.5 μs, and both these devices and the data path between them must therefore be capable of dealing with at least 5 bytes per transfer for communication to be possible.

The address path contains 27 bits (including 3 byte-parity bits and one bit which distinguishes user and system addresses). The control field contains some information which is copied directly through the *OR* gate from the Sending Unit to the Receiving Unit, some information which is copied through the *OR* gate and is also used by the Exchange control logic, and some information (the Unit Number) which is modified by the control logic before being sent to the Receiving Unit. In all, 14 control digits pass between units via the Exchange, making the total width of the *OR* gate 113 bits, and some additional control signals pass between the Exchange control logic and each of the units. The Strobe Outwards (SO) signal (figure 3.8), for example, is sent from a Sending Unit to the Exchange to initiate a transfer, and timed to arrive as soon as the data, address and control information in the sending Unit output buffer have become valid at the Exchange (allowing for cable length, etc.). This output buffer is required because, at

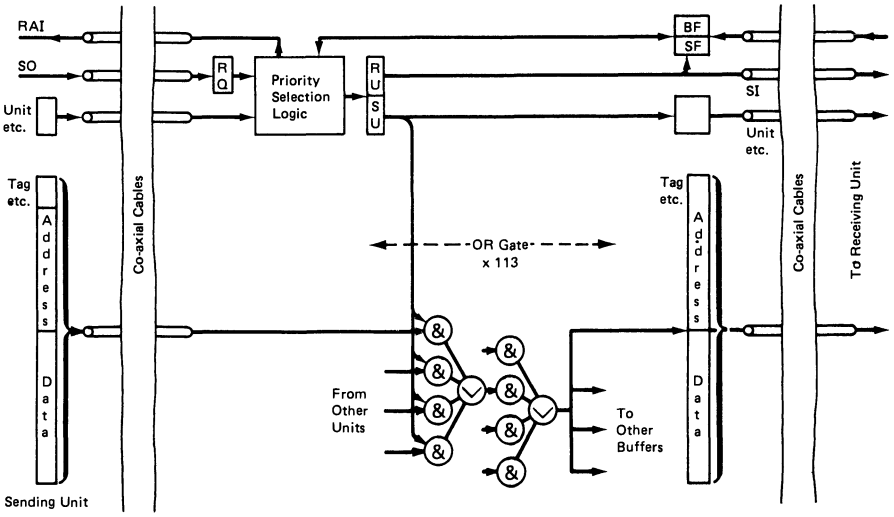

Figure 3.8 Exchange control and data paths

the time SO is sent, either the Receiving Unit may not be free, or a higher priority transfer may be in progress, and the Sending Unit therefore has no means of knowing when the transfer will actually occur.

The timing control logic of the Exchange itself is governed by a free running oscillator, so that the Exchange operates synchronously, at a rate of one transfer per 100 ns. Each transfer requires two 100 ns periods or *slots* for its completion, one for the actual transfer through the Exchange *OR* gate, and a previous one in which the Exchange control logic determines which of the incoming requests to service. Within the Exchange these two activities are overlapped for successive transfers. When a request has been selected for servicing in one time slot, the information from the Sending Unit is gated into the *OR* gate in the next time slot by the appropriate decoded output from the Select Unit Register (SU in figure 3.8). The output signals from the *OR* gate then propagate to the input buffers of all the Units but only the buffer corresponding to the Receiving Unit of the current transfer is strobed, at the end of the slot. A Strobe Inwards pulse (SI) is then sent to the appropriate Receiving Unit, thereby completing the transfer as far as the Exchange is concerned. The Receiving Unit, on receipt of SI, deals with the data in its buffer at its own convenience and then returns a signal to the Exchange indicating that its buffer is free to be overwritten by a further Exchange transfer.

3.5.2 The Exchange priority system

In any time-shared communication system to which a multiplicity of source units is connected there has to be some mechanism for establishing the order in which simultaneous requests are serviced. In the case of the MU5 Exchange the units can be classified into four categories: Peripheral Processing Units (PPUs), Central Processing Units (CPUs), Stores (Mass and Local), and the Block Transfer Unit. PPUs have highest priority since they are generally concerned with and only make occasional requests to stores via the Exchange. Apart from PPU transfers, most of the store transfers are paging transfers between the Mass and Local Stores organised by the Block Transfer Unit. Since this Unit can control up to four block transfers simultaneously, it can easily saturate the Mass Store, and although CPUs have a crisis time extending to infinity, it would be unreasonable to hold up their requests for the duration of a block transfer. Thus CPUs have the second highest priority and the Block Transfer Unit has the lowest.

Exchange priorities are assigned in inverse proportion to the Unit Number of the port to which a device is attached, so that the Fixed-head Disc, for example, attached as Unit 0, has highest priority. The priority of a request is normally determined by the priority of the Sending Unit, but in the case of a *data available* request occurring in response to a read request from a crisis time PPU, the priority of the request is determined by that of the Receiving Unit. All transfers to or from the Fixed-head Disc, for example, are organised by the Disc's own block transfer unit, and must all have top priority. Failure to transfer a request to or from the Disc within its 4 μs crisis time would invalidate the whole block transfer, and it would have to be aborted and re-started.

4 *Pipelines*

In any computer the execution of a single instruction requires various activities to be performed, such as instruction accessing, instruction interpretation, operand accessing and arithmetic. If separate hardware units carry out these activities their operations can be overlapped to give an increased rate of completion of instructions. This technique, first introduced in computers such as Atlas and Stretch, has become known as *pipeline concurrency*. In a pipelined computer several partially completed instructions are in progress concurrently, and although the time to complete any one instruction is still limited by the sum of the times for the various activities, the rate at which instructions progress through the pipeline is only limited by the time for an individual activity. In Atlas and Stretch the number of concurrent operations was of the order of four. In more recent designs the pipeline concurrency principle has been extended to several tens of instructions and used in both arithmetic and instruction processing units. In this chapter we shall discuss the principles of pipeline design, and then consider the actual design of the MU5 Primary Operand Unit as an example of instruction pipelining and Texas Instruments' Advanced Scientific Computer (TI ASC) as an example of arithmetic pipelining. In addition we shall consider some techniques used in the IBM System/360 Model 195 to overcome the problems which arise in pipelined systems.

4.1 Principles of pipeline concurrency

Pipelining is an implementation technique for high performance architectures which is normally invisible to the programmer. However, the use of pipelining may often be reflected in the order code of a high performance machine, as we shall see in the following section. Consider an arbitrary computing structure for evaluating a function F, on a stream of input values $x_1 \ldots x_n$, to produce a stream of output values $F(x_1) \ldots F(x_n)$. The performance of this computing structure can be characterised firstly in terms of its *latency*, and secondly in terms of its *throughput*. The latency is defined as the delay between supplying input value x_i and receiving at the output of F the value $F(x_i)$. The throughput is defined simply as the *rate* at which output values are produced by F. In the simplest case, where output i appears at the same time that input $i+1$ is presented to F, the throughput of F will be the reciprocal of the latency.

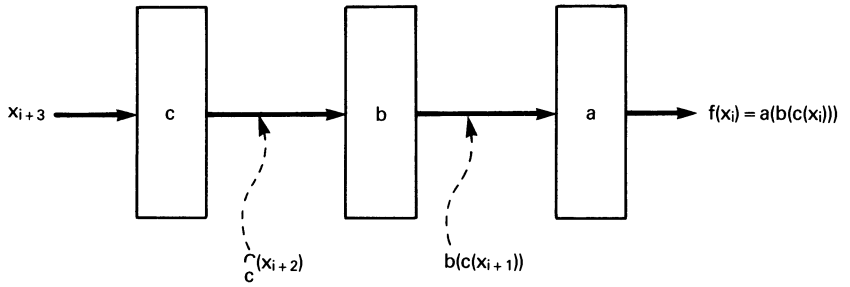

Figure 4.1 A pipelined structure

Improving the performance of such a computing structure means increasing the throughput. One way to do this is to reduce the latency, usually by implementing F using faster logic circuits. However, there are limitations on the performance improvements attainable in this way, and so other techniques have been sought. If the function F consists of a sequence of sub-functions, for example a, b and c, such that $F(x_i) = a(b(c(x_i)))$, then it is possible to implement F using pipelined logic. The decomposition of F into three sub-functions is illustrated in figure 4.1.

Each *stage* in the pipeline contains some function evaluation logic and between the stages there are latches for storing the intermediate values. Since the latency of each sub-function is less than the latency of F the *rate* at which new input values can be presented to each stage is greater than the rate at which they can be presented to a non-pipelined implementation of F. If a common clock is used to latch the intermediate values at all stages in the pipeline, and the logic at stage i has a latency of τ_i, then assuming a latch delay of ϕ, the *clock period* of a k-stage pipeline τ can be defined as

$$\tau = max\{\tau_i\}_{i=1}^{k} + \phi \quad = \tau_m + \phi$$

The activity within a pipeline can be represented pictorially using a *space-time* diagram, as shown in figure 4.2 where the space-time behaviour of a four-stage pipeline is depicted for a sequence of six input values $(x_1 \ldots x_6)$. From figure 4.2 it can be seen that for a k-stage pipeline a time equivalent to k clock periods elapses before the pipeline is full, after which one result appears at the output every clock period. Therefore, if a sequence of n data items is processed by a k-stage pipeline, it will take a time of $k\tau$ to produce the first result, and a time $(n-1)\tau$ to produce the remaining $(n-1)$ results. Hence, the total time to process n data items in a k-stage pipeline, T_k is

$$T_k = k\tau + (n-1)\tau \tag{4.1}$$

If we now make the assumption that a non-pipelined implementation of F has a latency of $k\tau$, then the same n data items will take a time

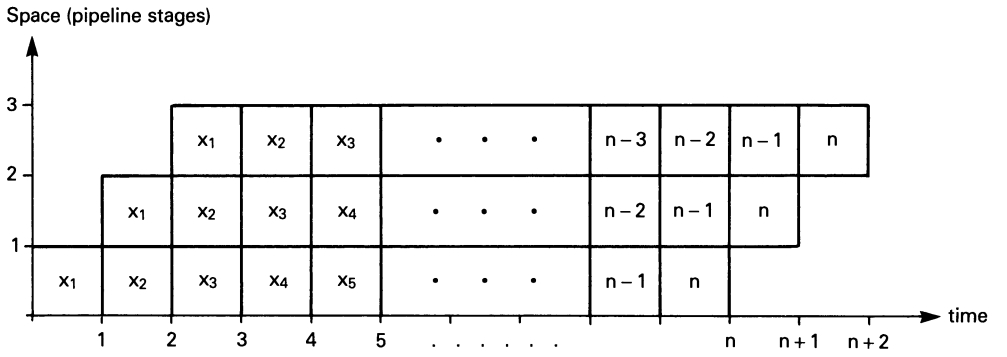

Figure 4.2 Space-time diagram showing pipeline behaviour

$T_1 = nk\tau$ when processed on a non-pipelined processor, effectively a one-stage pipeline, which evaluates an equivalent function.

The speedup resulting from the use of a k-stage pipelined implementation is defined as $S_k = T_1/T_k$, and this is given by

$$S_k = \frac{T_1}{T_k} = \frac{nk}{k + (n-1)} \tag{4.2}$$

and in the limit, as the number of data items tends to infinity

$$\lim_{n \to \infty} \frac{T_1}{T_k} = k$$

This defines the asymptotic maximum speedup, in terms of throughput, and this is equal to the number of pipeline stages. In general, of course, the flow of data items in any real pipeline is subject to disruptions of one kind or another, and maximum throughput cannot be maintained continuously. For a pipeline which processes a finite stream of data items between disruptions we can define two further parameters which characterise its performance: *efficiency* and *net throughput*. The efficiency, η, of a k-stage pipeline processing n data items is defined as

$$\eta = \frac{\text{speedup}}{\text{number of stages}} = \frac{n}{k + (n-1)} \tag{4.3}$$

while the net throughput under these conditions, r, is defined as the actual number of results produced per second, and therefore

$$r = \text{efficiency} \times \text{peak processing rate} = \frac{\eta}{\tau}$$

When the number of data items being processed in a pipeline approaches infinity $\eta \to 1$, and $r \to r_\infty = \tau^{-1}$. The interested reader may like to apply these formulae to the various pipelines described in this book.

Given that a pipeline cannot be maintained in a continuously busy state, there are a number of conditions which the designer should attempt to satisfy in order to maintain throughput as far as possible. These are

1. The latency of all pipeline stages should be equal.
2. The computations at each stage in the pipeline should be naturally sequential, and should be independent of each other.
3. A continuous flow of input values should be presented to the pipeline, and the resulting output values should be processed by the succeeding logic at the rate at which they are produced.

In considering the design of real pipelines we shall see not only how the designers have attempted to satisfy these conditions, but also how a variety of problems arises to thwart their best intentions. We turn first to an example of an instruction pipeline, the MU5 Primary Operand Unit.

4.2 The MU5 Primary Operand Unit pipeline

We saw in sections 1.3 and 3.4 how processor performance considerations led to the adoption of a pipeline structure for the MU5 Primary Operand Unit (PROP). PROP is concerned with accessing the operand specified directly by an instruction (the primary operand) and routing the instruction, together with its primary operand, to the appropriate following unit for execution or further processing. If the primary operand is a named variable or literal, for example, the instruction is ready for execution at the end of PROP, whereas an instruction involving a data structure must be sent to the Secondary Operand Unit (SEOP) where the accompanying descriptor, accessed as a primary operand by PROP, is used to specify the secondary access for the data structure element.

Figure 4.3 shows the basic hardware required in the Primary Operand Unit to implement the primary operand accessing facilities of the instruction set described in section 2.5 and the various stages of operation involved in processing a typical instruction. Instructions are received from an Instruction Buffer Unit (IBU) into registers DF (function) and DN (name). The first action is the decoding of the instruction to select the contents of the appropriate base register (NB, XNB or SF) and the name part of the instruction. For access to a 32-bit variable, the name is shifted down one place relative to the base and the least significant digit is used later to select the appropriate half of the 64-bit word obtained from the Name Store.

In the second stage the name and base are added together to form the address of a 64-bit word. This address is concatenated with the 4-bit Process Number (PN) and presented to the Associative Field of the Name

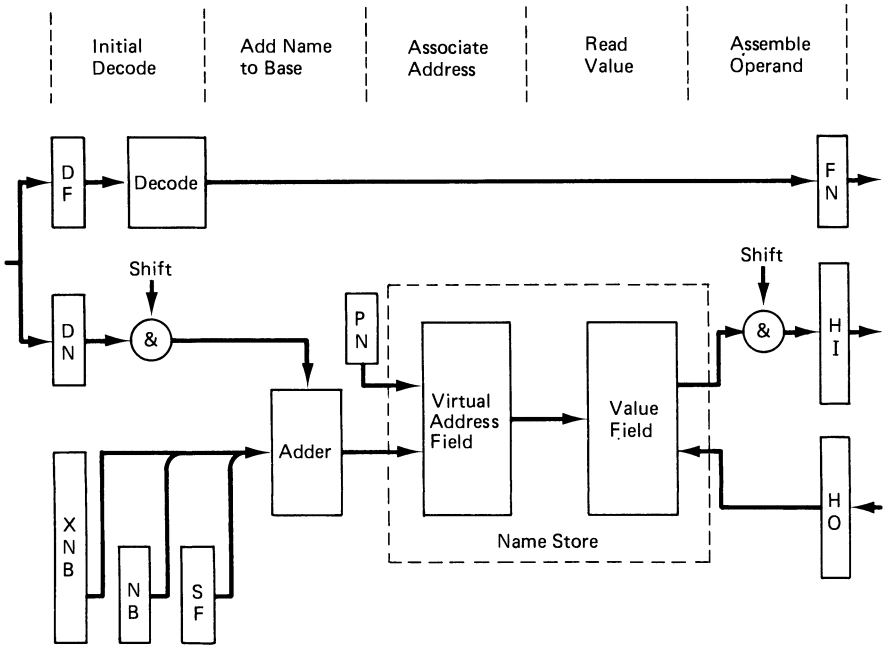

Figure 4.3 The basic components of the Primary Operand Unit

Store in the third stage. If the address is found, access is made in the fourth stage to the 64-bit word in the Name Store Value Field[1]. The fifth stage of processing is the assembly of the operand. A 32-bit integer, for example, may be taken from either half of the 64-bit store word, but must always appear at the least significant end of the data highway when presented to a succeeding unit. Registers FN and HI form the input to the highway which links PROP to other units in the processor (figure 3.5) and register HO is connected to one of the outputs of this highway in order to receive operands resulting from store orders.

The pipelining of the five stages in PROP is achieved by staticising the information obtained at the end of each stage in a buffer register (figure 4.4). An important aspect of the design of a pipeline is the timing of the strobes to the buffer registers, because some outputs from a stage (the function bits, for example) will derive directly from its inputs. With edge-triggered or master/slave flip-flops the various registers could be strobed simultaneously, but in the D-type flip-flops used in MU5 the information on the D input

[1]If the required address is not in the associative store, the non-equivalence actions described in section 3.4.2 must be initiated.

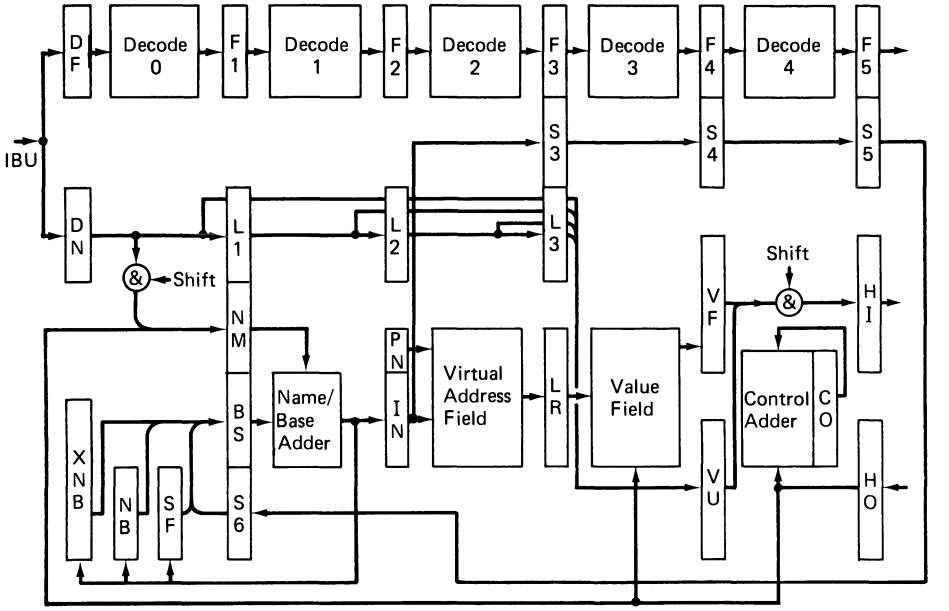

Figure 4.4 The complete Primary Operand Unit

is copied through to the output as long as the clock is held low and a different technique must therefore be used. Thus the result obtained at the end of any one stage is only copied into its buffer register when the result of the following stage has itself been staticised. The strobes used to copy information into the buffer registers are therefore staggered, as shown in figure 4.5. The shaded portions of figure 4.5 show the progress of one instruction through the PROP pipeline. It is first copied into DF and DN (function and name respectively) and the Decode 0 logic carries out the decoding of the instruction necessary to control the first stage. The decoding logic of figure 4.3 is spread out in the pipelined version into separate decoders for each stage. In many cases, however, the necessary decoding cannot be carried out in sufficient time to control the action of a given stage. In these cases it is carried out in the previous stage, and the various control signals appear as additional function digits, along with the original function, as inputs to the stage requiring them.

The next pipeline strobe is timed to arrive no earlier than when the outputs of the first stage have settled and are ready to be strobed into the registers F1 (function), NM (name) and BS (base). The addition of

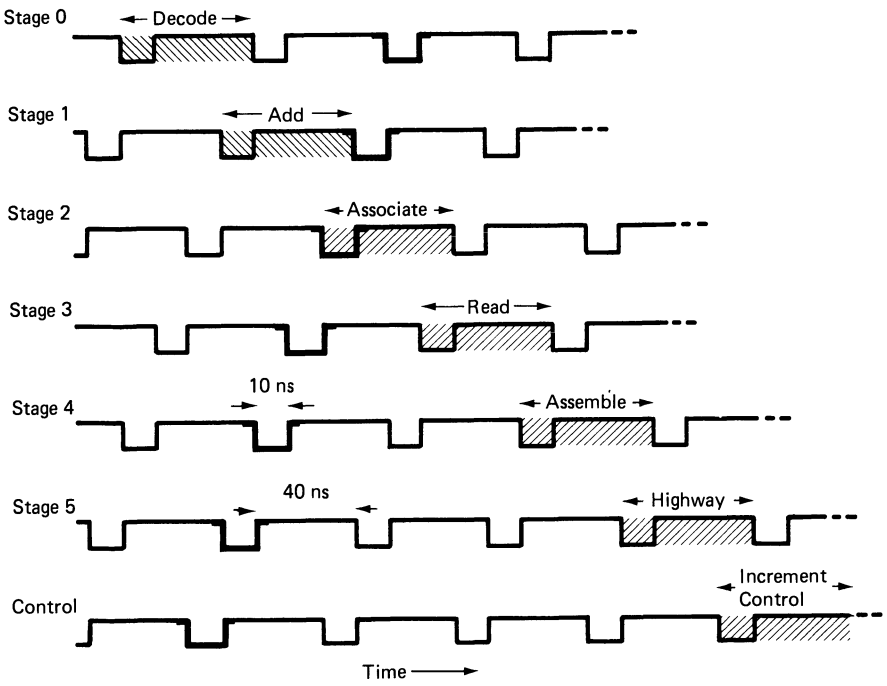

Figure 4.5 The basic PROP timing diagram

name and base now takes place and when the next pipeline strobe arrives, the result is copied into IN, the Interrogate Register. The output of IN is concatenated with PN, the Process Number, to form the input to the associative field of the Name Store. The result of the association is then copied into the PROP Line Register (PLR), the output of which accesses the line in the Value Field containing the required operand. The Value Field output is copied into the Value Field register (VF), and thence, after assembly, into the Highway Input register (HI).

Once an instruction has reached HI, PROP must wait until it has been accepted by another unit before taking any further action. When it has been accepted the Control Register is updated for that instruction and all other instructions in the pipeline move along one stage. Instructions therefore proceed through PROP in series of *beats*, the rate at which these beats occur being determined by the maximum operating rate of PROP and the acceptance rates of the succeeding units. The generation of a beat is initiated by the setting of a *data gone* flip-flop (figure 4.6) which when any other necessary conditions at the end of the PROP pipeline have been

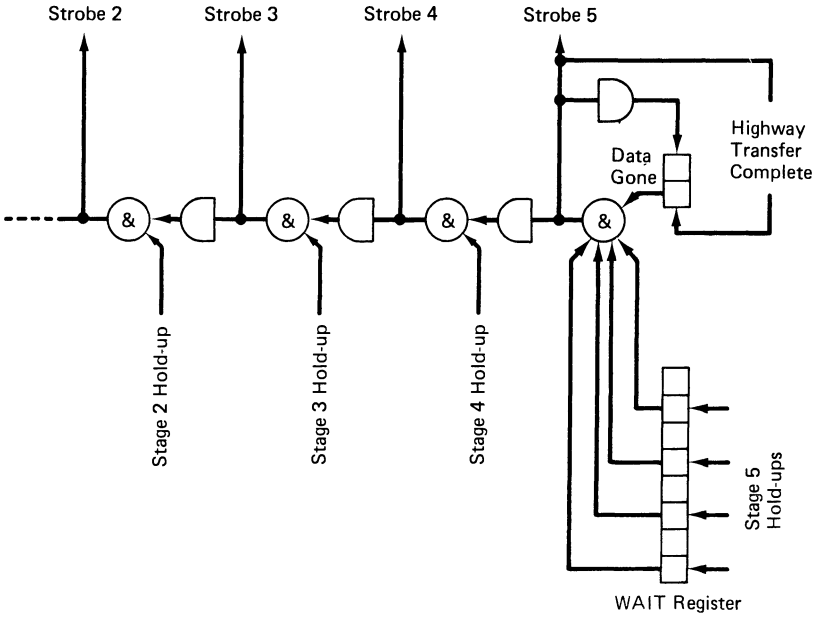

Figure 4.6 The pipeline delay chain

satisfied, allows a 20 ns pulse to propagate through the pipeline delay chain. The pulses from the chain drive 10 ns pulse-forming modules at each stage, and the delays in the chain are adjusted to produce a 10 ns stagger between stages. The progress of one pulse through the pipeline is shown heavily drawn in figure 4.5.

Some problems arise as a result of the physical dimensions of the Processor and the layout of the platters within it. Thus it is not feasible to locate all registers pertaining to one stage in close proximity either to each other or to the timing control logic. As a result, all the registers of one stage cannot be strobed simultaneously, since *far* strobes would have to be sent out in advance of *near* strobes by up to 20 ns. Alternatively, all *far* registers could be strobed late (the strobes and data must all travel the same distance) but some control signals must travel back from the far registers to the near registers, and the double delay would slow the pipeline down. The problem is overcome in practice by deriving far strobes from the earliest level of fan-out in each section of the delay chain, and by designing for only three levels of logic in paths which involve data travelling from near to far registers. Thus the 50 ns within each stage is typically made up as follows

Input Buffer Settling Time	5 ns
Operation Within Stage	30 ns
Inter-platter Cable Delays	5 ns
Output Buffer Strobe	10 ns
TOTAL	50 ns

4.2.1 Synchronous and asynchronous timing

PROP is an example of a synchronous pipeline in which the operation time of each stage is the same and the time relationship between the buffer register strobes is fixed. However, it is asynchronously connected to other units in the processor, and a *handshake* system between PROP and the B-unit, for example, determines when the next PROP beat is to be generated once a B-order has entered the PROP output registers. Figure 4.7 shows the hardware used to implement a typical handshake system between two units and a schematic timing diagram of its operation. Information is passed from one unit to another when the sending unit has the data available and when the receiving unit is not busy. For a given rate of operation, the cable length L cannot exceed a certain maximum. Assuming a delay of 5 ns per circuit and a co-axial cable delay of 1.8 ns per foot, for example, the maximum distance between units for a 50 ns operating rate is approximately 8 feet. This might seem long, but in a large system such as MU5 cable lengths between units can easily be of this order, and the cable length between PROP and the B-unit is in fact almost twice this figure. A double handshake loop which relies on the presence of an input buffer in the B-unit is therefore used to link these two units.

In MU5 all interactions between the separate functional units are asynchronous. This type of operation allows information transfers to take place as and when required and offers greater speed than a completely synchronous system (where transfers only take place at fixed times), particularly when the units concerned are not heavily used or where the different operations carried out within a particular unit take significantly different times (add, multiply and divide, for example, in the floating-point unit). Where a number of communicating pipeline stages are heavily used, however, the time penalty incurred by the handshake becomes a dominant factor, and a synchronous system such as that used in PROP may be preferred. In many high performance systems (the CRAY-1 for example) the central processor is completely synchronous. Asynchronous systems can also give rise to additional problems, particularly when a unit can accept requests from a number of sources. In MU5 this problem occurs, for example, at the input to the Store Access Control (SAC). Here three different functional units may request a store cycle at any time, and in the event of a clash,

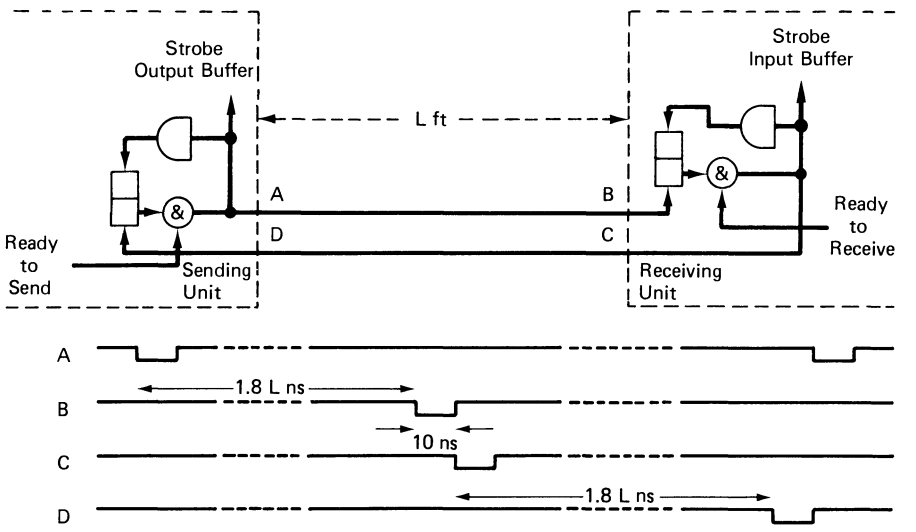

Figure 4.7 An asynchronous handshake system

SAC must decide which request to accept. Each incoming request pulse is staticised on a flip-flop, and the outputs of these three flip-flops drive a combinational priority logic circuit. The outputs from this circuit, only one of which can be a 1, form the inputs to a set of decision flip-flops. These flip-flops are strobed when SAC is free to accept a request *and* sufficient time has elapsed for the priority circuit to have settled after receipt of the first request.

Because a second request may occur a short time after the first request, however, it is possible for the inputs to the decision flip-flops to change state just before the end of the strobe, leaving the outputs somewhere between a 0 and a 1 level. Under these conditions the time taken for the flip-flop outputs to reach proper logic levels may be long compared with the normal propagation delay, and it is possible for subsequent circuits to operate inconsistently. Sufficient time must therefore be allowed for the decision flip-flops to settle to a constant level, or failure of the control circuits may occur [KW76]. For ECL flip-flops with a propagation delay of 2.2 ns, over 30 ns settling time must be allowed in order to maintain this failure rate at 1 per month. In MU5 a tunnel diode flip-flop with a decision time of the order of 100 ps was used in critical circuits, giving an MTBF of 136 years for a settling time of 3 ns.

4.2.2 Variations among instruction types

In describing the operation of the PROP pipeline, it was assumed that each stage of the pipeline contained one 16-bit order. However, circumstances arise which prevent full utilisation of each stage. For example, the order code allows for multi-length orders and some orders require multiple accesses (those involving the stack mechanism, for example). These can normally be dealt with by a purely logical control mechanism which does not affect the main pipeline timing, but may involve the creation of gaps (unused stages) in the pipeline. These gaps, or *dummy orders*, are distinguished by means of an additional function digit which, when set, inhibits the actions of each stage, including the Control Register updating.

The stacking mechanism is used for storing and retrieving partial results and for procedure links and parameters. For example, during the evaluation of expressions such as

$$a = b * c + d * e$$

partial results are stacked by the use of the *= (stack and load) function. They are later unstacked by use of the operand form STACK (section 2.5). Stacked operands are therefore contained in the MU5 Processor storage system in exactly the same way as names, their addresses being generated relative to the Stack Front register (SF), which points to the most recently stacked operand within the Name Segment. Thus SF is advanced by both the *= function and functions concerned with procedure entry (STACK and STACK LINK), and all these functions require two operand accesses to be made. Hence they are divided into two phases.

For the STACK function, for example, an access is first made for the specified operand followed by an access to the stack, while for the *= order the first access is to the stack, in order to store the content of the specified register, and the second is for the operand. For the stack writes the name/base adder is used to create the address SF+2 and at the same time SF is updated to this new value. The two phases of these orders are distinguished by extra digits carried through the pipeline with the function. These digits override the normal operand accessing mechanisms when access to the stack is required and also prevent the incrementing of the Control Register when the first phase reaches HI.

The implementation of this stack mechanism within a pipeline gives rise to additional problems in relation to control transfer orders. An order implicitly changing SF does so while there are still several orders ahead of it in the pipeline. Some of these will not have reached the Control Point (the point in the pipeline at which the Control Register is updated), and are therefore not completed. Any one of these orders could be a control transfer order requiring that the partially processed orders behind it in

the pipeline, including the order which modifies SF, be abandoned (section 4.2.3). Should this situation occur, the SF Register may contain an incorrect value. The correct SF value could be maintained by preventing overlap in such situations, but this would cause serious degradation of the performance of the pipeline. The alternative solution adopted is to allow the SF register to change as and when required and to propagate forward any new values for SF in registers S3, S4 and S5 (see figure 4.4). When the Control Register is updated for each order, the value in S5 is copied into S6. Therefore when a control transfer occurs the value in S6 is known to be correct and can be used to restore SF.

In a non-pipelined computer this situation would not arise since each instruction would be completed, and all register values would have been updated, before the next instruction could begin. A particular consequence of this is that an order which writes the contents of a programmable register to store in a non-pipelined computer can proceed without delay, since the register contents are guaranteed to be correct and available at the start of the order. This is not generally true in a pipelined machine where operand accesses are made several pipeline stages ahead of the programmable register into which the results of the operations involving those operands will be loaded. For example, in MU5 the Name Store is several stages ahead of the B-unit, so that when a store order of the type B ⇒ name accesses the Name Store there can be several orders ahead of it in the pipeline which have themselves not yet reached the B-unit and operated on the B register content. Thus the B register value cannot be made available immediately, and in the absence of any additional technique, the B ⇒ name order would have to wait at the Name Store stage of the pipeline until the B register contents could be guaranteed to be correct and available. This would involve a delay equivalent to at least four pipeline stages, and because these orders constitute 5 to 10 per cent of all orders, special action is taken to avoid this hold-up[2].

Whenever a B ⇒ name order enters Stage 4 of the PROP pipeline, the content of the PROP Line Register (PLR in figure 4.4) is preserved, for later use, in an additional register BW, and the order proceeds normally through the pipeline without causing a hold-up and without impeding orders following, except as described below. Execution of a ⇒ order in the B-unit involves sending the result to the HO register in PROP. When it arrives the PROP pipeline is held up before the next beat is generated, and the information held in register BW is used to select the appropriate line in

[2]Store orders involving registers within PROP (NB ⇒, etc.) or SEOP (DR ⇒, etc.) occur much less frequently and no special action is taken for them. ACC ⇒ name orders are dealt with separately, but in a similar manner to B ⇒ name orders, in the Secondary Operand Unit pipeline.

the Value Field of the Name Store so that the value in HO can be written into it. The additional information needed to select one-half of the line for overwriting is held in the F2 Function Register, together with a $B \Rightarrow$ *outstanding* digit which indicates that the BW Register is in use. When the action of writing into the store has been completed, the $B \Rightarrow$ *outstanding* digit is re-set and the pipeline is re-started.

While the $B \Rightarrow$ *outstanding* digit is set, two pipeline hold-ups can occur, one at Stage 2 and one at Stage 4. These hold-ups are typical of a number which can arise in the pipeline because some necessary information (such as the B register value for the B ⇒ name order) is not immediately available. Apart from hold-ups which arise from the fifth stage, they cannot be detected in time for the next beat of the pipeline to be inhibited and therefore operate independently of the beat generating logic, by simply preventing subsequent beats from propagating back beyond the stage from which they arise (figure 4.6), and by causing dummy orders to be propagated forwards. The hold-up at Stage 2 occurs if a second B ⇒ name instruction enters that stage, while the $B \Rightarrow$ *outstanding* digit is set, because the BW register can only deal with one outstanding order at a time. This hold-up prevents subsequent beats from propagating back beyond the input registers to Stage 3 (F3, etc.) and causes dummy orders to be copied into Stage 3. The hold-up at Stage 4 occurs if an instruction tries to access the same line in the Name Store as that indicated by BW. This is an example of the *read after write* problem found in almost all instruction processing pipelines [RL77] and is a further consequence of the fact that the B register value is not available for writing into store when the B ⇒ name order reaches the Name Store. The hold-up prevents subsequent beats from propagating back beyond the input registers to Stage 5 (F5, etc.) and causes dummy orders to be copied into Stage 5. Both these hold-ups are automatically released when the $B \Rightarrow$ *outstanding* digit is re-set, or if the contents of the pipeline are discarded by the action of a control transfer before the B ⇒ order has left the end of PROP. If a control transfer occurs after a B ⇒ order has left the end of PROP, then the store updating action must still be carried out since the Control Register will have been incremented for this order.

Hold-ups arising from the fifth stage are normally those involving complex actions within PROP and possibly interactions with another unit (as in the case of a Name Store non-equivalence, for example), and which require the pipeline to be stopped. The need for one of these hold-ups is indicated by the setting of one or more bits in a *WAIT* Register as the instruction is copied into F5 (figure 4.6). On completion of the highway transfer the *data-gone* flip-flop is set as usual, but the next beat is prevented from being released by the presence of the digit in the WAIT Register. Instead, a hardware routine is entered appropriate to the most significant digit in the

WAIT Register. When the routine is completed the corresponding WAIT digit is re-set and either the beat is released or another hardware routine is entered appropriate to the next most significant digit in the WAIT Register. A typical WAIT routine is that used for base register manipulation. This manipulation is carried out using the same adder as that used for address calculations (figure 4.4). If the order is of the type `NB +`, then the base forms one input to the adder, via the same route as that used for address calculations, and the operand forms the other input, in place of the name. The adder output is routed back to the inputs of all the base registers and when the addition is complete the appropriate one is updated. These orders must be completed before a succeeding instruction can be allowed into register `DF`, since it may use the content of `NB` to form an address. In these cases the decoding logic in Stage 0 of PROP sets a *no-overlap* digit which prevents further instructions from being copied into `DF` until the base manipulation order has been executed.

4.2.3 Control transfers

In computers with a fixed instruction length the Control Register (or Program Counter) needs only to be incremented by one at the completion of each instruction and a simple counter may be used for this purpose. In computers such as MU5, however, where the instruction length is variable, a full adder is required to accommodate the correspondingly variable increment. In a simple implementation this adder would be shared with other operations, but in a high performance system a separate dedicated adder is required. This same adder can therefore be used to execute control transfer instructions. Thus for a relative control transfer in MU5 the Control Adder in PROP (figure 4.4) is used to add the operand to the current value in the Control Register (CO), while for absolute transfers the operand passes through the adder with the Control Register input to the adder held at zero.

Control transfers (or *branches*) are a major problem in pipelined instruction processors, since if the branch is taken all the instructions pre-fetched and partially executed in the pipeline must normally be discarded. This can not only cause serious delays in instruction execution, but also requires that the designer ensure that no irrecoverable action is taken during the partial execution of instructions ahead of the Control Point. We have already seen an example of the latter in the implementation of the stacking mechanism in PROP. Various techniques have been used in high performance computers to overcome the delay problems caused by control transfers and we shall examine these in detail in chapter 5. We note here, however, that the MU5 Instruction Buffer Unit incorporates a prediction mechanism which, following the first execution of a particular control transfer, attempts to supply

the correct sequence of instructions to the pipeline behind subsequent occurrences of that control transfer, so that if the branch is taken there is no delay in continuing instruction execution. This system offers a performance advantage, because unconditional transfers always branch and a high proportion of conditional control transfers are loop closing instructions which branch more often than not.

When a control transfer instruction reaches the end of PROP an *out-of-sequence* digit accompanying the following instruction is examined. If set this digit indicates that the instruction is the start of a predicted *jump-to* sequence, and if the control transfer is unconditional or conditional and the condition is met, instruction execution can proceed as soon as the Control Register has been updated. This also happens if the *out-of-sequence* digit is not set and the condition for a conditional transfer is not met. If the *out-of-sequence* digit is not set following an unconditional transfer or a conditional transfer for which the condition is met, however, or if the *out-of-sequence* digit is set and the condition for a conditional transfer is not met, then all instructions in the pipeline before the Control Point (those in PROP and in the Instruction Buffer Unit) must be discarded, by the setting of their dummy order bits, and a long delay is incurred while the new jump-to sequence of instructions is fetched.

4.2.4 Performance measurements

The overall performance of a pipelined instruction processing unit is affected by the relative frequencies of occurrence of different instruction types and the additional delays which they incur. Measurements of these frequencies have been made for MU5 [YIH77] using a hardware performance monitor, for a number of benchmark programs, and the results are shown in columns three and five of table 4.1 for program execution and compilation respectively. Column two gives the time required in excess of the 50 ns beat time for the execution of each type of order, and columns four and six give the net additional contribution of each type of order to the execution time of the average order.

For both execution and compilation the longest delays are due to unpredicted control transfers and Name Store non-equivalences. The significant difference in performance between these two types of activity is due largely to the data-dependent nature of the compilation process which involves many alternative processing sequences. Name Store delays are largely a consequence of the organisation of the storage hierarchy (section 3.4), whereas the delays caused by control transfers, B-store orders and base manipulation orders are directly attributable to the difficulties inherent in processing sequentially dependent instructions in a pipeline, and those due to multi-length orders are consequences of the particular organisation chosen for the

Table 4.1 PROP performance

Type of Order	Excess Time (ns)	Execution Frequency (%)	Execution Net Time Added (ns)	Compilation Frequency (%)	Compilation Net Time Added (ns)
Multi-length	50	56.0	27.9	45.7	22.8
B Store	100	6.2	6.2	9.1	9.1
Base Manipulation	450	5.1	23.0	8.4	37.8
Predicted Control Transfer	100	8.7	8.7	8.0	8.0
Unpredicted Control Transfer	1350	4.5	60.7	10.5	141.8
Name Store NEQ	1180	3.5	41.3	11.9	140.4

IBU-PROP interface. In each case, however, improvements could be made which would reduce these delays. Many of the decisions taken during the design of MU5 were based, inevitably, on inadequate information about instruction usage, and in a commercial environment changes almost certainly would have been made to subsequent models in the light of experience gained from what was essentially a prototype design.

4.3 Arithmetic pipelines — the TI ASC

Although instruction and operand accessing are highly pipelined in MU5, no attempt is made to pipeline the activities of the main floating-point arithmetic unit. Floating-point addition, for example, consists of four distinct operations (exponent subtraction, mantissa shifting, mantissa addition, normalisation) which can be pipelined in a very straightforward manner. In a one-address system such as MU5 no benefit would be gained from this arrangement. Since one of the input operands to an addition operation is always the Accumulator, it cannot be used as an input to subsequent operation until the current operation is finished. This implies that tempo-

ral overlap of two successive additions is not possible if the result must always pass through the Accumulator. For successful operation of a pipelined arithmetic unit each instruction must reference at least two and preferably three operands, and even when this requirement is satisfied the possibility of dependencies between successive instructions requires complex control mechanisms and may cause hold-ups in instruction execution. Examples of these control techniques are the data forwarding arrangement used in the IBM System/360 Model 91 (section 4.4) and the Scoreboard used in the CDC 6600 (section 6.1). The ideal arrangement is for one instruction to cause two independent operand streams to be combined in the arithmetic unit to form a third independent result stream. This is achieved in vector processing systems such as the CDC Cyber 200 Series (section 9.2), the CRAY-1 (section 7.1) and the Texas Instruments' Advanced Scientific Computer (TI ASC).

Texas Instruments has traditionally been a supplier of instrumentation to the oil industry, and it was this industry's need for a powerful seismic processing capability that led to the development from 1966 onwards of the Advanced Scientific Computer [Wat72]. A significant feature of this type of processing is the frequent use of triple-nested indexing loops, and an important characteristic of the ASC is the provision of three levels of indexing within a single vector instruction.

The basic structure of the central processor of the ASC is shown in figure 4.8. The Memory Control Unit allows two-way traffic between eight

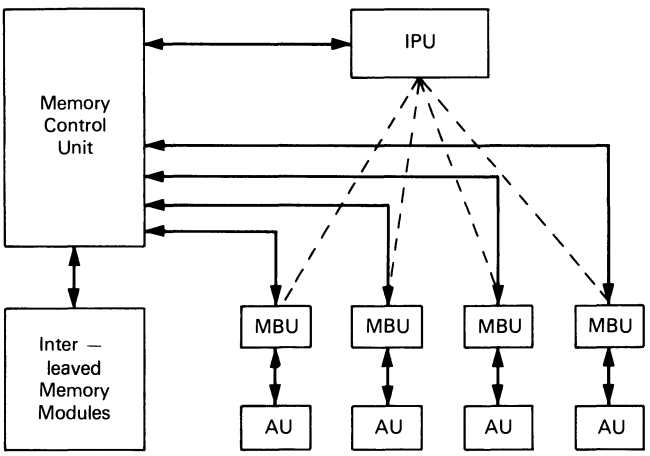

Figure 4.8 TI ASC central processor structure

256-bit wide processor ports and eight interleaved memory modules (plus an optional memory extension), with each processor port having full access to the 24-bit address space. The central processor design is such that one, two, three or four pipelined arithmetic units (AU) can be provided. Each AU is linked to the Memory Control Unit through its own Memory Buffer Unit (MBU), the primary function of the latter being to supply, from memory, a continuous stream of operands to the arithmetic unit, and to return to memory a continuous stream of arithmetic results. Addresses for sustaining this continuity are computed in the MBU using parameters supplied through the Vector Parameter File at the start of a vector instruction, and double buffering (in units of eight 32-bit words) is provided for each of the operand streams. The Vector Parameter File contains a starting address field for each of the three vectors involved in a typical vector instruction, a vector length, inner and outer loop counts, and increments to be added to each of the three starting addresses at the completion of each inner and outer loop. Information is supplied to the Vector Parameter File by the Instruction Processing Unit (IPU), which is mainly concerned with initiating operations in the MBU-AU vector processing pipelines, but also acts as a conventional scalar processor dealing with instruction fetching and decoding, address generation, control transfers, etc.

The arithmetic units are 64-bit parallel fixed and floating-point units split into two halves of 32 bits each. Double length operations are carried out using both 32-bit halves in parallel, while single length operations use only the most significant half of an AU. The ASC differs from the CDC Cyber 200 Series in this respect, since the 64-bit arithmetic units in the latter split into two identical 32-bit units which can work in parallel. Numbers are represented in the ASC in IBM format. Thus fixed-point numbers are represented as 2's complement integers, while floating-point numbers are represented in sign and magnitude form with a base 16 (hexadecimal) exponent represented by an excess 64 binary number, as shown in figure 4.9.

Each AU is made up of eight distinct sections, each of which performs a separate arithmetic or logical operation (figure 4.10). Each section can be connected to any other section to allow the correct sequence of operations to be executed for a particular instruction, with the appropriate configuration being established at the start of each vector instruction. In any given configuration the various sections form a pipeline into which a new pair of operands can, in principle, be entered at each 60 ns clock, and after a start-up time, corresponding to as many clock periods as there are sections in use, result operands emerge at a rate of one per clock period. At the end of a vector instruction there is a similar run-down time between the entry of the last operand pair and the emergence of the corresponding result.

Floating-point addition, for example, requires the use of the Receiver Register, Exponent Subtract, Align, Add, Normalise and Output sections, connected as shown by the solid line in figure 4.10. Pairs of operands from the MBU are first copied into the Receiver Register, the cable delays between the MBU and AU effectively forming a complete stage in the overall pipeline arrangement. The Exponent Subtract section then performs a 7-bit subtraction to determine the difference between the exponents of the two floating-point operands, or in the case of equal exponents uses logic to determine which of the fractional mantissae is larger (this logic is also used by those instructions which test for greater than, less than or equal to, in order to avoid duplication of hardware).

The exponent difference is used in the Align section to shift right the mantissa of the operand with the smaller exponent. In one cycle any shift which is a multiple of four can be carried out, which is all that is required for floating-point numbers represented in base 16. (Fixed-point right shifts require two cycles, one shifting by the largest multiple of four in the shift value, and a second in which the result of the first is re-entered and shifted by the residue of 0, 1, 2 or 3.)

Having been correctly aligned, the fractional parts of the two floating-point numbers are added in the Add section, and the result is passed on to the Normalise section. This section closely resembles the Align section in that floating-point operations only require one cycle, while the fixed-point left shifts which it also carries out require two. The major difference between these two sections is that Align receives information concerning the length of shift required in floating-point operations, while the Normalise section has to compute the shift length by determining which four-bit group contains the most significant digit. It also contains an adder to update the exponent value when a normalisation shift occurs. The results of all arithmetic operations pass through the Output section before being returned to

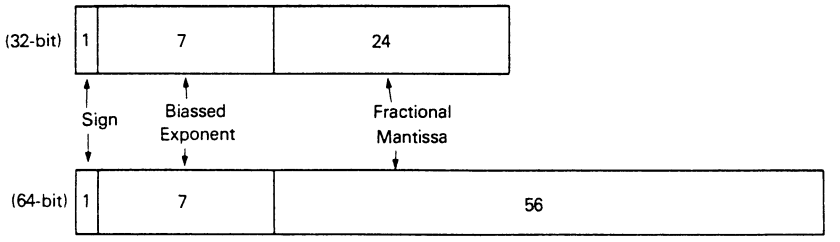

Figure 4.9 ASC floating-point formats

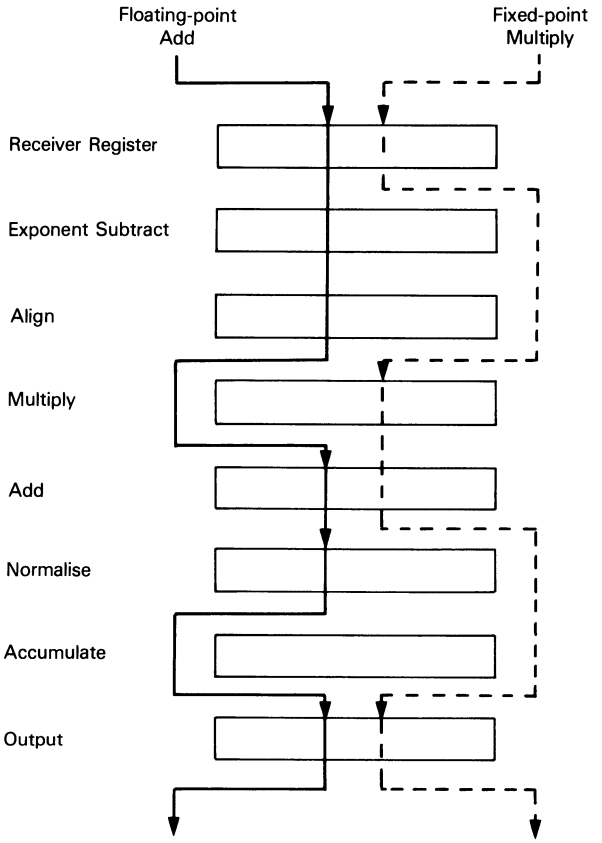

Figure 4.10 ASC arithmetic pipeline

the Memory Buffer Unit. The partitioning of the arithmetic unit into these various sections is primarily intended to give high throughput of floating-point addition and subtraction. Each section is capable of operating on double length operands so that vector double length instructions can proceed at the clock rate. Double length multiplication, and all divides (which are performed by an iterative technique), proceed more slowly.

The dashed line in figure 4.10 shows the interconnection used for fixed-point multiplication. The Multiply section can perform a 32 by 32-bit multiplication in one clock period, so that the results of both fixed-point and single-length floating-point multiplication are available after one pass through the multiplier. Because a carry-save addition technique is used, the output of the Multiply section consists of a 64-bit pseudo-sum and a 64-bit

pseudo-carry. These must be added in the Add unit to produce the true result. Double-length multiplication requires three separate 32 by 32-bit multiplications to be performed and these can therefore proceed at a rate of only one every three clocks. After passing through the Add section the three separate results are added together in their proper bit positions in the Accumulate section.

The Accumulate section is similar to the Add section and is used in all instructions which require a running total to be maintained. An important example of this type of instruction is the Vector Dot Product, which is used repeatedly, for example, in matrix multiplication. Pairs of operands are multiplied together in this instruction and a single scalar result, equal to the sum of the products of the pairs, is produced. Because the running total is maintained in the arithmetic unit, the *read after write* problems which occur in scalar implementations of this operation are avoided in the ASC. In MU5, for example, the sequence of instructions required to produce, on the stack, the vector dot (or *scalar*) product of two vectors defined by descriptors VEC1 and VEC2, each of length LIMIT, is

```
      B  = 0
    ACC  = 0
L1: ACC *= VEC1[B]
    ACC  * VEC2[B]
      B CINC LIMIT
    ACC  + STACK
     IF /=, -> L1
```

where [B] implies modification of the descriptor Origin by the contents of the index register B, and the function *= is processed in two stages, the first stacking the contents of the specified register (the floating-point accumulator in this case) and the second re-loading it with the specified operand (the Bth element of VEC1 in this case). Thus there are only two Accumulator instructions between the instruction which writes a new value from the Accumulator into the top-of-stack location held in the Name Store (section 3.4.1) and the instruction (ACC + STACK) which reads it out again. In the MU5 pipeline the Name Store is several stages earlier than the Accumulator, and a similar mechanism to that used for B write orders (section 4.2.2) is used to prevent ACC write orders from creating an immediate hold-up. The hardware must guard against an operand being read out before it has been updated by an outstanding write order, however, and in so doing it does cause a hold-up in the passage of instructions through the pipeline. Exactly the same *read after write* problem occurs in pipelined, multi-address register machines such as the high-performance models in the IBM System/360 series; in the case of Models 91 and 195, however, a *data forwarding* mechanism was developed in order to avoid this hold-up.

4.4 The IBM System/360 Model 91 Common Data Bus

The System/360 Model 91 was developed in the mid 1960s with the primary aim of providing the highest performance capability that advanced design philosophy and System/360 circuit technology extensions could achieve within a balanced development schedule [FL71]. Performance was taken to mean general computer availability and high-speed execution of general problem programs, and a factor of between one and two orders of magnitude over the earlier 7090 was achieved, depending on the nature of the problem. Only a small number of Model 91s were actually produced, but most of its architectural features were carried over into the commercially more successful Model 195.

One of the problems facing the designers of high-speed computer systems is the difficulty of achieving the fastest possible execution times for a particular technology in universal execution units. Circuitry designed to carry out both multiplication and addition, for example, will do neither as fast as two units each limited to one kind of operation. Thus not only did the Model 91 contain separate fixed-point and floating-point execution areas (figure 4.11), but the floating-point area contained separate add and multiply/divide units capable of concurrent operation. Both units were pipelined; the add unit as a two-stage pipeline capable of starting a new operation in each 60 ns clock cycle, and the multiply/divide unit capable of starting a second operation three cycles after the start of a previous multiply or twelve cycles after a previous divide.

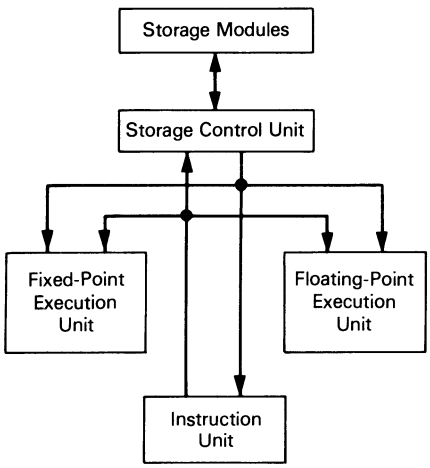

Figure 4.11 IBM System/360 Model 91 central processor

Independence of fixed-point and floating-point operations is guaranteed in all System/360 machines by the separation in the instruction format of the two sets of general accumulator registers (section 2.2). Within the floating-point unit, however, the hardware must preserve essential instruction sequence dependencies while allowing the greatest possible overlap of independent operations. This same problem arises in a number of machine designs, particularly the CDC 6600, with which we shall deal in some detail in chapter 6. The control mechanism used to solve this problem in the Model 91 shows some similarities to the technique used in the CDC 6600, although the organisation of the data paths is quite different.

The organisation of the Model 91 floating-point unit [Tom71] is shown in figure 4.12. Instructions are prepared for this unit by the Instruction Unit pipeline and entered in sequence, at a maximum rate of one per clock cycle, into the Floating-point Operand Stack (FLOS). Instructions are taken from the FLOS in the same sequence, decoded, and routed to the appropriate execution unit. The Instruction Unit maps both storage-to-register and register-to-register instructions into a pseudo-register-to-register format, in which the equivalent of the R1 field (figure 2.3) always refers to one of the four Floating-point Registers (FLR), while R2 can be a Floating-point Register, a Floating-point Buffer (into which operands are received from store), or a Store Data Buffer (from which operands are written to store). In the first two cases R2 defines the source of an operand; in the last case it defines a sink.

The most significant feature of this floating-point system is the Common Data Bus (CDB). The CDB is fed by all units which can alter a register, and itself feeds the floating-point registers, the store data buffers and all units which can have a register as an input operand. The latter connections allow data produced as the result of any operation to be forwarded directly to the next execution unit without first going through a floating-point register, thus reducing the effective pipeline length for *read after write* dependencies, as found, for example, in the scalar product loop. The running total in this loop would not actually appear in a floating-point register in the Model 91 until the last execution of the loop.

The operation of the CDB is controlled by a tagging mechanism. A tag is a 4-bit number generated by the CDB control logic to identify separately each of the eleven sources which can feed the CDB. Thus there are six floating-point buffers, three parallel *reservation stations* (containing input buffer registers) associated with the adder, and two parallel reservation stations associated with the multiplier/divider. Tag registers are associated with each of the four floating-point registers, with the source and sink input registers of each of the five reservation stations, and with each of the three store data buffers. There is also a busy bit associated with each of

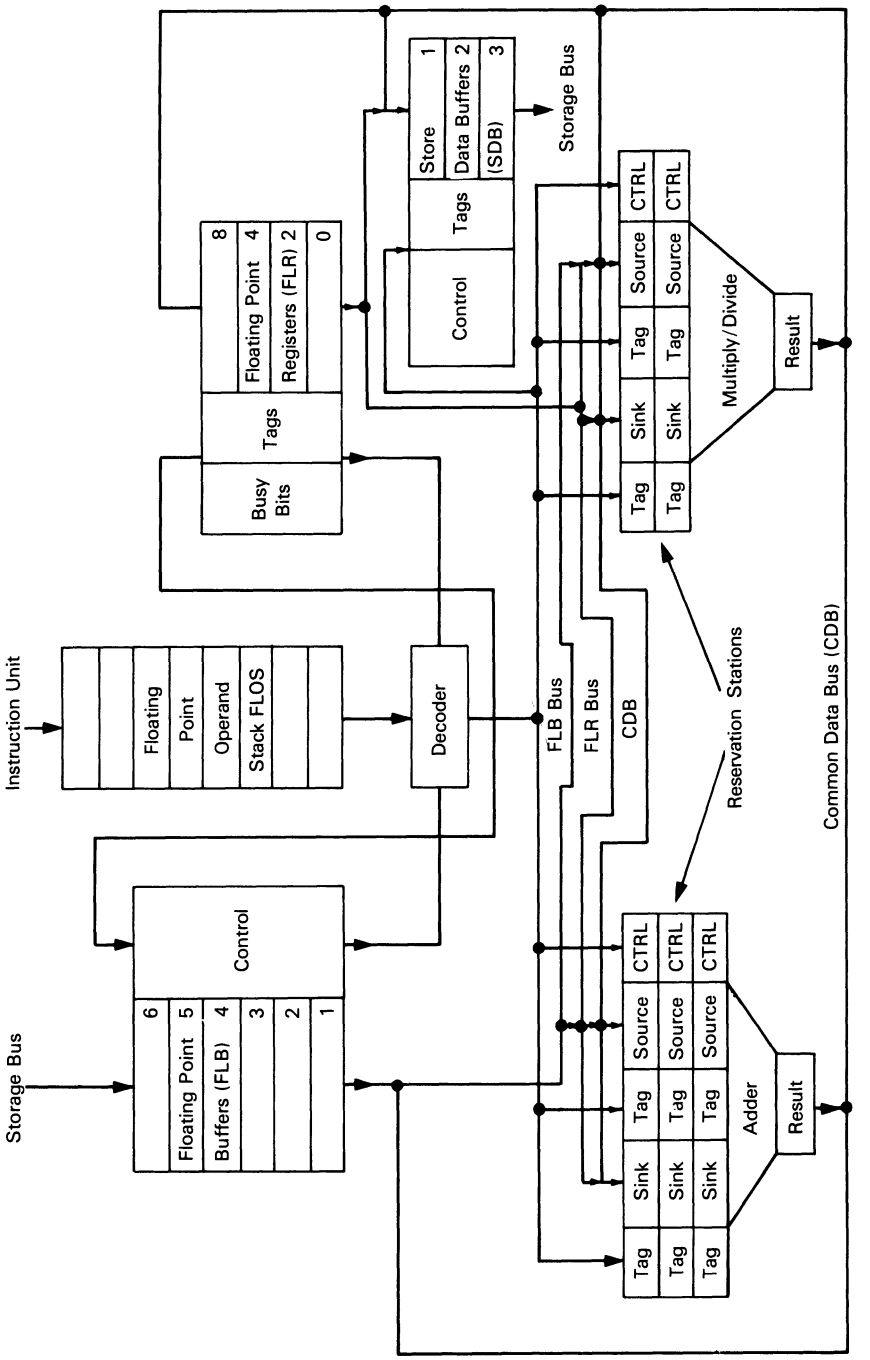

Figure 4.12 The IBM System/360 Model 91 floating-point unit

the floating-point registers. This bit is set whenever the FLOS issues an instruction designating the corresponding register as a sink, and is re-set when a result is returned to the register.

Whenever the FLOS decodes an instruction it checks the busy bit of each of the specified floating-point registers. If the bit is zero, the contents of the register are sent to the selected reservation station via the Floating-point Register (FLR) Bus. On issuing the instruction the FLOS sets the busy bit of the designated sink register, and enters into its tag register the tag number of the selected execution unit. If the FLOS finds a busy bit set, however, it does not transmit the register contents to the reservation station, but instead transmits the current value of the corresponding tag register, and enters into that tag register the appropriate new tag number. Thus the tag register of a busy floating-point register identifies the last unit (in proper program sequence) which will produce a result for it.

Whenever a result appears on the CDB, the tag corresponding to its source is broadcast to all destinations. Each active reservation station (selected but awaiting a register operand) compares its sink and source tags with the CDB tag. If a match occurs (a sink is also a source in the System/360 two-address instruction format), the reservation station takes the data from the CDB. In a similar manner, the CDB tag is compared with the contents of the tag register associated with each busy floating-point register. All busy registers with tags matching that on the CDB are set to the value on the CDB and their busy bits are re-set.

Issuing an instruction in this system only requires that a reservation station be available for whichever execution unit is required. If a source register is awaiting the result of a previously issued, but as yet uncompleted instruction, or if a floating-point buffer register is awaiting an operand from store, the tag associated with that register is transmitted instead to the reservation station, which then waits for that tag to appear at its input. Thus it is the reservation stations which do the waiting for operands, rather than the execution circuitry, which is free to be engaged by whichever reservation station fills first. Execution of an instruction starts when a reservation station has received both operands. It may receive one or both operands from either the CDB or FLR bus in the case of a register-to-register instruction, while in the case of storage-to-register instructions the source operand is transmitted by the FLB bus. Both the FLR and FLB busses could have been incorporated, in principle, into the CDB, but in practice this would have caused a reduction in performance due to conflicts over the common facility. The use of the CDB and the associated tagging mechanism has been shown to reduce the execution times of the inner loops of programs used to solve partial differential equations, for example, by about one-third.

5 *Instruction Buffers*

In dealing with operand accessing in earlier chapters we considered various techniques used to overcome the disparity between processing speed and main store accessing rate. This problem also impinges on instruction accessing, since for efficient operation instructions must also be supplied to the processor at a rate matching its execution rate. In the case of instruction accessing, however, the problem is ameliorated by the fact that most instructions are obeyed sequentially and the main store word size is normally such that one word fetched from main store can contain several instructions. Furthermore, with an interleaved store, successive accesses for sequential instructions reference each stack in turn and are not held up by cycle time effects. Thus store requests can be made in advance of the corresponding instruction being required and the replies buffered until they are needed for execution. This *pre-fetching* technique is used in almost all high performance pipelined processors. A significant proportion of instructions cause control transfers, however, and each such transfer requires a request to be made to the store for a new sequence of instructions. Thus although the accessing rate for instructions can normally be matched satisfactorily to the processing rate, the access time for the first instruction of a new sequence can result in a long delay to the processor. Techniques for overcoming this problem rely on the fact that the cause of many control transfers is a branch back from the end to the start of a loop of instructions, and *loop-catching* buffers are incorporated into a number of processors. In this chapter we shall consider the instruction accessing and buffering techniques used in the IBM System/360 Model 195, the CDC 6600 and 7600, MU5 and the CRAY-1.

5.1 The IBM System/360 Model 195 instruction processor

The organisation of the System/360 Model 195 Central Processor is very similar to that of the Model 91 (figure 4.11) and, as in the case of the Common Data Bus discussed at the end of chapter 4, the instruction buffering technique used in the Instruction Processor of the Model 195 derives directly from that used in the Model 91 [AST71]. The Model 195 Instruction Processor is concerned with fetching and buffering instructions from storage, fetching the operands which those instructions specify, issuing instructions to the appropriate execution units, handling interrupts, and executing all

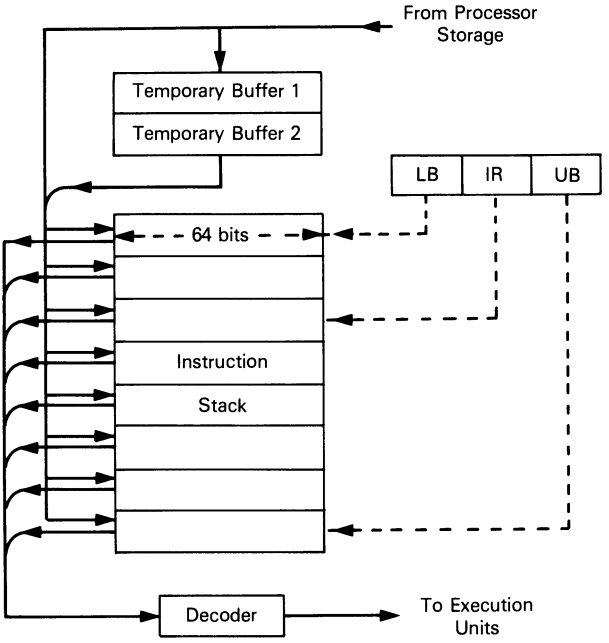

Figure 5.1 The IBM System/360 Model 195 instruction buffer

branching (control transfer), status switching and input/output instructions.

Instructions fetched from store are buffered in an eight-doubleword (64-bit) Instruction Stack (figure 5.1). The instruction fetching mechanism is controlled by three registers: the Instruction Register (IR) which addresses the instruction currently being decoded, the Upper Bound Register (UB) which points to the most recent doubleword brought into the stack, and the Lower Bound Register (LB) which points to the earliest doubleword in the stack. During normal operation the stack contains the current instruction doubleword, some doublewords ahead of the current instruction and a copy of some instructions which have already been issued.

5.1.1 Sequential instruction fetching

Pre-fetching of instructions is controlled by the UB register. When instruction fetching is initiated following an interrupt, for example, the Instruction Stack is declared empty and the main storage address of the first instruction doubleword is loaded into UB and LB. The instruction fetching mechanism associated with UB then accesses this doubleword and loads it into the

location in the Instruction Stack addressed by the three least significant doubleword address bits in UB. Initially this location is also addressed by IR, which selects each instruction in sequence for decoding and processing. After an instruction has been decoded and passed to the next stage in the processor pipeline, IR is incremented by the number of half-words in that instruction and the next instruction is selected.

Once the first instruction access has been sent to store, the instruction fetching mechanism increments UB and continues to make sequential store accesses until prevented from doing so either because the address in UB is seven doublewords higher than that in IR (and any further accesses would cause instructions not yet decoded to be overwritten), or because the Instruction Processor has detected a condition giving rise to a change in the instruction sequence (a branch instruction or an interrupt, for example).

During normal operation the instruction fetching mechanism continually attempts to increment UB and fetch instruction doublewords from store, while the instruction decoding mechanism continually increments IR as instructions are decoded and passed along the processor pipeline. Once IR has been incremented beyond the address in LB, instructions in the first doubleword fetched into the stack can be overwritten with new information. Provided IR remains ahead of LB, then when incrementing UB would cause its three least significant doubleword address bits to match the corresponding bits in LB, both these registers are incremented together. Thus at each instruction access the oldest doubleword in the stack is replaced by the latest doubleword fetched from store.

Use of this pre-fetching mechanism allows a continuous sequence of instructions to be supplied to the processor at a rate approaching one per machine clock cycle, and thus roughly matching the instruction execution rate[1]. When a new sequence of instructions is required as a result of the branch being taken in a branch instruction, however, the start-up delay is of the order of six clock cycles, and in the absence of some additional technique the average performance of the processor would be seriously degraded. Conditional branches cause even further problems since the branch decision depends on the outcome of a previously issued, but not necessarily completed arithmetic instruction, and an additional delay may be incurred in awaiting this outcome. This problem is discussed further in section 5.5. In the Model 195 two techniques are used to ameliorate the problems caused by branches, one involving the establishment of a *Conditional Mode* of operation, and the other a *Loop Mode*.

[1]Although the processor pipeline was designed to execute instructions at a rate of one per clock cycle, instruction dependencies, storage conflicts and the frequency of operations requiring multi-cycle execution combine to reduce the average rate to about half this figure.

5.1.2 Conditional mode

Conditional branch instructions interrogate a 2-bit Condition Code at their point of execution in order to determine whether or not the branch is to be taken. The Condition Code is set by a variety of instructions, but only the last of these issued before a conditional branch must be allowed to affect its outcome. This is accomplished by tagging at decode time each instruction which will set the Condition Code. At the same time a signal is forwarded through the pipeline to remove the tags from any previously issued but uncompleted instructions. Only a tagged instruction may set the Condition Code, at which point its tag is removed, and a conditional branch instruction can only execute when there are no outstanding tags in the processor.

Since in general the Condition Code will not be valid when a conditional branch is decoded, the hardware always assumes this to be the case and establishes Conditional Mode. In Conditional Mode further sequential instruction accesses are inhibited, but rather than hold up further activity entirely, processing of the remaining instructions in the Instruction Stack proceeds as far as possible (until a further branch is decoded or the pipeline becomes full, for example), with the instructions being marked as *conditional.* Conditional instructions are decoded, their operand fetches are initiated, and they are forwarded to the relevant execution units in the normal way. The conditional tag inhibits the execution units from actually completing them, however, and once the first such instruction reaches the point of execution, further processing is held up until the Condition Code is set and the branching action determined. If the branch is not taken, the conditional tags are re-set and the pipeline is re-started without further delay.

If the branch is taken, the conditional instructions must be abandoned and a fresh start made with a new sequence. The delay incurred in refilling the pipeline from the decoder onwards is unavoidable, but the delay in accessing the first instruction at the target address of the new sequence is minimised in the Model 195 because the hardware assumes at the start of Conditional Mode that either outcome is equally likely and fetches the first two instruction doublewords at the branch target address immediately. These two doublewords are loaded into the two Temporary Buffers shown in figure 5.1, in order that the Instruction Stack remain unaffected if the branch is not taken. Clearly these instruction fetches will have been made unnecessarily on many occasions, and since instruction accesses have priority over operand accesses on the store address path, some performance degradation can occur due to interference with operand accesses for the conditional instructions. This disadvantage is more than offset, however, by the advantage gained, when the branch does occur, of the access time

for the target instructions having been overlapped with the wait for the Condition Code. In the case of an unconditional branch to an instruction not in the Instruction Stack, there is, of course, no need to wait for the Condition Code to become valid. As in the conditional case, the target instruction sequence is requested immediately, but unless the execution unit pipelines are also held up (as a result of divide operations, for example) the six clock cycle start-up delay inevitably causes a gap to occur in the instruction processing sequence.

The primary purpose of the whole conditional philosophy was the circumvention of storage delays, and in retrospect the designers felt that the complications of the system, which involve numerous interlocks throughout the processor, would become increasingly difficult to justify as storage access times decrease.

5.1.3 Loop mode

Without the use of branch target instruction pre-fetching in Conditional Mode, the time lost when the branch is taken would be roughly equal to the sum of the time spent waiting for the Condition Code to be set and the storage access time. With pre-fetching the time lost becomes equal to only the greater of these two, but even so, where the branch is closing a short loop of instructions, this loss can severely limit overall processor performance. Thus for short loops a different philosophy is adopted whereby the entire loop is contained within the Instruction Stack and storage accesses are avoided altogether until the program exits from the loop. Clearly, the longer the loop, the smaller the proportion of time lost as a result of the branch, and the choice of eight doublewords as the capacity of the stack represents a compromise between hardware cost and performance in Loop Mode.

Loop Mode is entered whenever a branch backwards is taken to a target address within eight doublewords of the current instruction. The Instruction Stack is immediately re-initialised to contain the appropriate eight doublewords, after which instruction fetching ceases and the address path to store is fully available for operand fetching throughout execution of the loop. Loop Mode is controlled by two additional registers, one containing the loop target address (SLT) and the other the value of IR corresponding to the loop closing instruction (SLCIR). Once in Loop Mode the address of any branch instruction being decoded is compared with that in SLCIR, and if it is the same the branch is made immediately to the target address held in SLT. Thus the rôle of Conditional Mode is reversed, since it is assumed that the branch will be taken, and instructions are therefore decoded from the target path rather than the straight-through path. Furthermore, no fetches are made to the Temporary Buffers in Loop Mode.

Loop Mode is normally turned off because an exit is taken from the loop. This can happen in a variety of ways. If the branch closing the loop is not taken, for example, IR will run off the end of the instructions held in the stack and require a store access. Alternatively some other branch within the loop may be taken to a target outside the stack, or the address in SLCIR may be invalidated. This can happen if the base or index register specified in the instruction which caused SLCIR to be set up is altered. A record of these registers is kept with SLCIR and a check made against this record if any instruction in the loop alters a fixed-point register.

5.2 Instruction buffering in CDC computers

5.2.1 The CDC 6600 instruction stack

The organisation of the CDC 6600 Central Processor [Tho70] was discussed in section 2.1 in connection with the 6600 instruction set. We noted that instructions are fetched from Central Storage and placed in an Instruction Stack before being decoded and issued to the appropriate functional unit under control of the Scoreboard. The Instruction Stack itself consists of eight 60-bit registers (I0-I7 in figure 5.2) which operate as a push-up stack and which can contain instruction loops. Programs are initiated in the 6600 by an *Exchange Jump* in which the contents of all addressable registers in the central processor are interchanged with the contents of a designated store area. Following such an Exchange Jump the new contents of the program address register are used to access the first instruction word. This word is received from Central Storage into an Input Register and then loaded into the bottom register of the Instruction Stack.

Instruction words are made up of four 15-bit *parcels* and as the first instruction word enters the bottom register of the stack (I0), the first two parcels within the word (starting from the left) are transferred into a series of instruction registers within the Scoreboard control logic (section 6.1). At the same time a further instruction fetch is initiated. Two parcels are taken to allow for long format (30-bit) instructions. If the first parcel is a short format (15-bit) instruction, the second parcel is ignored and in the next processor minor cycle the second and third parcels are taken from I0. When a long instruction is encountered an extra minor cycle is spent skipping over the second half, so that dealing with one complete instruction word never takes less than four 100 ns minor cycles. This matches the rate at which instructions move into the stack. Whenever a new instruction fetch is initiated, the contents of the stack ripple upwards one register every half minor cycle, with the topmost location (I7) being overwritten first. At the end of four minor cycles the contents of I0 are moved up and I0 is then ready to receive a new instruction from the Input Register.

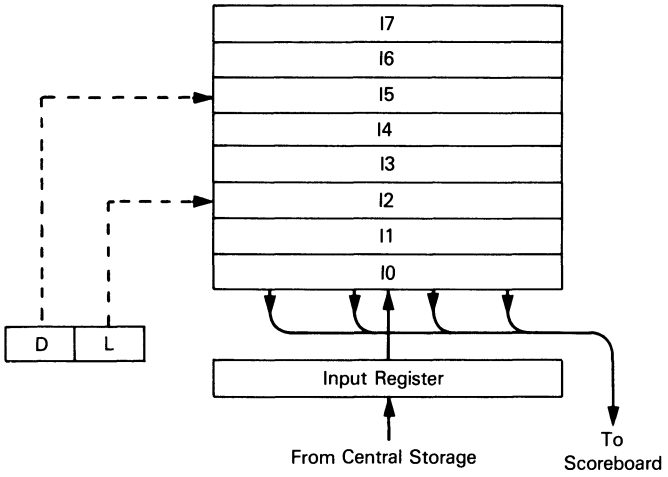

Figure 5.2 The CDC 6600 instruction stack

In practice the rate at which instructions could be accessed from Central Storage turned out to be lower than anticipated at the design stage, which meant that a new instruction word would not actually be available until after a total of eight minor cycles. Since the rate at which instructions are issued to the functional units cannot often be maintained at one per minor cycle, however, the overall effect of this delay on performance is not quite so bad as might be imagined, and when executing loops which can be contained within the stack, no Central Storage accesses for instructions are required at all.

Information about the contents of the stack is contained in two registers: the Depth (D), which measures the number of valid instruction words in the stack, and the Locator (L), which specifies the location in the stack of the instruction word currently in use. During execution of a loop held entirely in the stack the instructions remain in fixed locations and the program address register can point to any one of the stack registers within a distance D from the bottom. D is re-set to zero whenever a branch out of the stack is taken, and is incremented by one for every new instruction word brought in. When the stack is full, D remains equal to seven.

When a conditional branch is decoded a test for *jump within stack* is made. This involves subtracting the current program address from the branch address. If the absolute value of the result is less than seven words, and if the values in D and L indicate that the branch is to a location within the stack, no further store accesses are made for instruction words until instruction parcels are again taken from I0. Thus a branch may jump

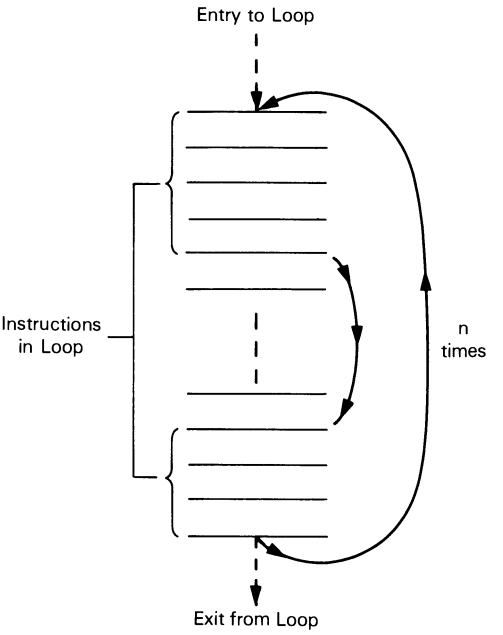

Figure 5.3 A non-contiguous instruction loop

forwards or backwards within the stack and loops may be held in the stack in various forms.

A very similar Instruction Stack was used in the STAR-100 computer, CDC's first commercially produced vector processor. The STAR-100 had a much longer instruction format than the 6600 so that its Instruction Stack was larger, being made up of sixteen 128-bit registers, but it used essentially the same control mechanisms. The main drawback of both the 6600 system and that used in the IBM System/360 Model 195 is that where the total number of instructions being obeyed in a loop will fit into the stack, but the code is actually made up of a number of non-contiguous segments (as in figure 5.3, for example), the loop may not be caught in the stack. With machine code programming this situation can normally be avoided, but it is a common occurrence in compiler generated code and the increasing emphasis on high-level language programming has caused processor designers to seek alternative solutions. The CDC 7600 and CYBER 205, for example (successors, respectively, to the 6600 and STAR-100), both use associatively addressed buffers, with the 205 Instruction Stack again being correspondingly larger. Further discussion of the CYBER 205 is left until chapter 9.

5.2.2 The CDC 7600 instruction stack

The CDC 7600 [CDC77] was designed to be machine code upward compatible with the 6600, but to provide a substantial increase in performance. The ways in which this was achieved will be discussed in more detail in section 6.2. For now we will note that the organisation of the 7600 central processor is very similar to that of the 6600. It contains nine parallel functional units, a scratch-pad of eight X registers, eight A registers and eight B registers, and an Instruction Stack. The 7600 Instruction Word Stack is made up of twelve 60-bit registers, however, compared with the eight used in the 6600, and each register also has its own 18-bit associative address register in an Instruction Address Stack (figure 5.4).

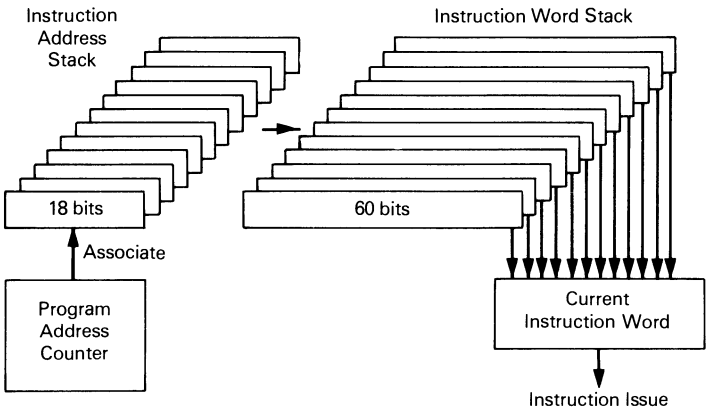

Figure 5.4 The CDC 7600 instruction stack

The Instruction Stack is filled two words ahead of the instruction currently being executed, thus giving a greater degree of pre-fetching than was possible in the 6600, and hence overcoming the storage access delay for sequential instructions. Furthermore, instructions are obeyed from a Current Instruction Word register, rather than from the bottom stack register, and a complete 60-bit word is transferred from the Instruction Stack into this register whenever the word address changes in the program address counter. This transfer can be made from any of the twelve registers in the Instruction Word Stack, allowing a considerable degree of flexibility in pre-fetching and loop catching. Whenever a new word is required in the Current Instruction Word register the address in the program address counter is compared with the entries in the Instruction Address Stack, and if a coincidence occurs for any of these entries, the contents of the corresponding register in the

Instruction Word Stack are transferred into the Current Instruction Word register.

When obeying sequential code the required word will normally be in one of the bottom two registers. When a branch instruction is executed and the branch taken, the required word may already be in one of the top ten registers, obviating the need for a store access, and giving improved performance. If the required word is not in the stack, the first two words at the target address are immediately requested from store and instruction execution continues when the first of these is received. Whenever an instruction word is received from store all the entries in the Instruction Word Stack and the Instruction Address Stack are simultaneously moved up one position, with the new address and instruction word being entered at the bottom of the stack and the oldest entry being lost. Entries in the stack are only invalidated by the execution of a subroutine call or Exchange Jump, and not by normal branch instructions, so that a program may branch back and forth between short sequences of non-contiguous code held in the stack.

Although this stack is larger than that of the 6600, it is still relatively small, and considerable effort is frequently required to reduce the amount of code in program loops in order that they may fit into it. A quite different scheme from any of those considered so far is required if loops of unrestricted size are to be accommodated. Such a scheme is to be found in MU5.

5.3 The MU5 Instruction Buffer Unit

The need for sequential instruction pre-fetching, and the disruptive effects of control transfers, become increasingly apparent as the degree of pipelining is increased. In the MU5 computer the Primary Operand Unit (section 4.2) has a maximum execution rate of one 16-bit instruction per 50 ns, while the plated-wire Local Store has a 260 ns cycle time and is four-way interleaved. The Local Store can therefore supply successive 128-bit words at 65 ns intervals and so there is no problem in supplying sequential instructions at the required rate. Because the access time is much longer than 65 ns, however, instruction requests have to be sent out well in advance of their being required by the Primary Operand Unit, and buffered in an Instruction Buffer Unit. This Instruction Buffer Unit (figure 5.5) contains three 128-bit buffer registers which constitute its Data Flow section. The Data Flow control logic unpacks instructions from the first buffer and assembles them in the second and third buffers ready for PROP to take them as required. The necessary store requests are made by the Store Request System, which issues store addresses formed by a counter at a rate matched to that at which instructions are taken from the Data Flow by PROP.

This system operates satisfactorily until a control transfer occurs as a result of either an unconditional control transfer instruction, or a conditional

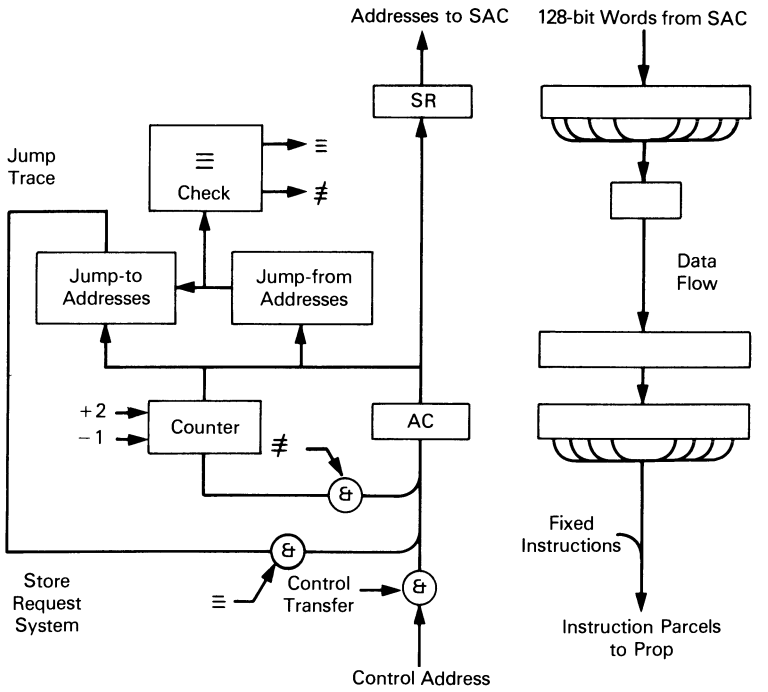

Figure 5.5 The MU5 instruction buffer unit

control transfer instruction for which the condition is met. Then all the prefetched instructions must be abandoned, and the correct new instruction cannot be sent to PROP until the store has been accessed, using the new control address, and the instruction has passed through the Data Flow. As a result the total time between the execution of the control transfer and the first instruction of a new sequence is 950 ns.

Measurements made during the execution of a number of benchmark programs run on MU5 showed that control transfers constitute around 13 per cent of obeyed instructions [YIH77]. Thus without some attempt being made to overcome the effects of these control transfers, the average instruction execution time would be

$$50 * 87/100 + 950 * 13/100 = 158.5 \text{ ns}$$

which represents a reduction by a factor of over three from the peak execution rate.

A system was initially considered which had buffer registers containing the first few instructions at the destination or *jump-to* addresses of recently obeyed control transfers. Access to these instructions was to have been

via an associative search on their addresses. The pre-fetching mechanism would proceed normally until a control transfer occurred, and the destination address would then be presented to the associative store. If a match was found, the instructions in the corresponding buffer register would be read out and sent to the Primary Operand Unit (PROP). If no match was found, one of the set of associative and buffer registers would be updated when the instructions had been obtained from store.

Simulation studies of this technique showed that only eight lines of store would be needed to trap 80 per cent of jump instructions and that increasing the number of lines to sixteen would only produce an extra 1 per cent improvement. The problem in implementing this scheme was the width of the buffer store. In order to allow the pre-fetching mechanism to catch up after a control transfer, each line of the buffer would need to hold up to 950 ns worth of instructions. At 50 ns per 16-bit instruction the number of bits needed in each line would have amounted to over 300.

In order to retain the advantage obtained by using an associatively addressed store, without incurring the cost of buffering large amounts of data, the system actually used in the MU5 Instruction Buffer Unit (IBU) includes an eight-line associatively addressed *Jump Trace* store which attempts to predict the outcome of an impending control transfer. Whenever a new instruction address is generated by the IBU it is presented to the associative *jump-from* address store before being sent to the Local Store via the Store Access Control Unit (SAC). If an equivalence is found, this address is replaced by the corresponding *jump-to* address, so that pre-fetching of the new sequence takes place instead. When the control transfer instruction which gave equivalence in the trace is sent to PROP, it is accompanied by a bit indicating that the instructions following it are *out of sequence.* This bit is used in PROP to determine the action after execution of the control transfer. If the following instructions have been correctly predicted, execution of instructions continues uninterrupted. If the instructions are not out of sequence, but should have been, a request is made to SAC for the instructions at the *jump-to* address, and at the same time a line in the Jump Trace is loaded with the *jump-from* and *jump-to* addresses[2]. Thus when the *jump-from* address re-occurs within the IBU, the instructions at the *jump-to* address are automatically pre-fetched.

Simulation studies of this system indicated that about 75 per cent of control transfers could be trapped using an eight-line store, and that, as before, increasing the number of lines in the store did not significantly improve the performance. The apparent drop in performance as compared

[2]This line is selected using a cyclic replacement algorithm and as each li e is overwritten a *use* digit associated with it is set. The *use* digits are normally only re-set, and the Trace thereby cleared, at a process change.

with the first system considered occurs because the prediction mechanism sometimes predicts a transfer which does not occur. No attempt is made to correct the Jump Trace when a predicted branch does not occur, however, since the drop in performance is more than offset by the fact that the prediction mechanism allows useful overlapping of instructions to continue in PROP when the prediction is correct.

5.3.1 Sequential instruction fetching

The Store Request System (figure 5.5) is responsible for initiating requests for 128-bit words from SAC at the required intervals. Two different types of request may occur according to circumstances: ordinary requests and priority requests. Priority requests are made whenever PROP signals a discontinuity in the instruction stream, that is, when the instructions following the one currently being executed must be replaced by a different sequence. Following such an event, the instruction address received from PROP is loaded into both the Advanced Control Register, AC, and the Store Request Register, SR, and a priority request is made to SAC. AC is then incremented at 40 ns intervals (it operates at a slightly faster rate than PROP) and whenever a carry across the 128-bit word address boundary occurs, the new address is copied into SR and an ordinary request is made to SAC. In addition, each new address generated in AC is checked against the contents of the associative jump-from field of the Jump Trace. If an equivalence occurs the corresponding jump-to address is read out and used instead. The Store Request System then continues to make ordinary requests starting from this new address.

Ordinary requests normally continue to be made until an instruction sequence discontinuity arises. This happens when an interrupt occurs, for example, or when the IBU has sent an incorrect sequence of instructions to PROP. This can occur either because an unpredicted control transfer instruction causes a jump, or because a predicted control transfer does not cause a jump. In either of these cases the priority mechanism is re-invoked when the address of the required instruction is sent from PROP. In the case of an interrupt two fixed (hard-wired) instructions are read from within the IBU. The first of these preserves essential linking information in store and the second causes a control transfer to the start of the appropriate interrupt routine. This again invokes the IBU priority mechanism.

The operation of the Store Request System can also be temporarily halted as a result of interlocks incorporated into the IBU to ensure that no loss of information occurs as a result of asynchronous operation. These interlocks simply cause time delays before the next ordinary request can be sent. For example, the rate of issuing of requests from the IBU to SAC is normally geared to the maximum rate at which PROP can process

instructions. When PROP executes an instruction which requires a long interval of time for its completion, however, a hold-up occurs and this hold-up propagates back through PROP to the IBU Data Flow buffers. Since the IBU is obliged to accept data from SAC as soon as it becomes available, sufficient space must be maintained in these buffers to receive it. In order to meet this condition and to be able to maintain the maximum throughput rate, IBU ordinary requests may be held up within SAC until the IBU can guarantee to accept the requested instructions.

5.3.2 Instruction effects in the Jump Trace

Although the Jump Trace is in principle a fairly straightforward system, a number of features of the instruction set complicated its implementation. For example, although most control transfers use a literal operand (which is invariant in a machine like MU5 which uses *pure procedure* code), some do not, and in these cases the jump-to address will vary on some subsequent executions of the instruction. No attempt is made in MU5 to take advantage of cases where this variation occurs infrequently, however, since the designers felt that the extra hardware complication needed to check the correctness of the predicted jump-to address would not be cost-effective. Instead, the problem of variable jump-to addresses is avoided by only loading the Jump Trace for those control transfers which use a literal operand.

A further problem arises from the variable instruction length. The Control Register always addresses the first parcel of a multi-length instruction, but clearly jump-from addresses must always correspond to the last. This address must therefore be computed specially, since it is not otherwise required[3]. In MU5 the value in the Control Register after the execution of a conditional control transfer is either the address of the first parcel of the next instruction in sequence (the jump-from address + 1), or some quite different address (the jump-to address). The first alternative can be generated immediately, since it requires no operand, whereas the second must await the arrival of the operand and the result of the condition. The Control Adder associated with the Control Register in PROP can therefore perform two cycles with no loss of performance, and when an unpredicted control transfer using a literal operand occurs, both values are sent to the IBU. The first is loaded into register AC (figure 5.5) and then decremented by the address counter before being used to load a line in the jump-from field. The second is loaded into AC and thence into the jump-to field and also into SR to be sent to SAC as a priority request.

[3]This complication could be seen as one argument in favour of RISC architectures (section 2.7).

5.4 The CRAY-1 instruction buffers

The need for instruction pre-fetching and loop catching techniques arises from the impracticability of having a large high-speed random access store close to the central processor, and all the instruction buffering techniques with which we have dealt so far have been aimed at circumventing this problem. With the advent of MSI and LSI storage technologies, however, different hardware organisations have been made possible. An example of this can be seen in the instruction buffering mechanism used in the CRAY-1 [Rus78], a machine with which we shall deal in more detail in chapter 7.

The overall structure and instruction set of the CRAY-1 are largely derived from the CDC 6600 and 7600, with instructions being executed by a set of parallel functional units. The processor obtains all its instructions directly from a set of instruction buffers, however, and does not send a stream of instruction requests to the main store. These buffers are organised as shown in figure 5.6. Each of the four buffers holds 64 consecutive 16-bit instruction parcels, and if an instruction request cannot be satisfied from within these buffers, a full 64-parcel block of instructions is transferred from main store into one of them.

The CRAY-1 uses a 22-bit instruction address and the first instruction parcel in a buffer always has an address starting on a 64-parcel address boundary. Any one buffer is therefore defined by the 16 most significant bits of a parcel address, and for each buffer there is a 16-bit starting address register containing this value. At each clock cycle the high order bits of the program address counter are compared with the contents of these registers, and if a match occurs the required instruction parcel is selected from within the appropriate buffer either immediately, if the buffer concerned is the same as the one which supplied the previous parcel, or after a two clock period delay if a change of buffers is involved.

If no match occurs, instructions must be loaded into one of the instruction buffers before execution can continue. A two-bit counter is used to determine which buffer is to be loaded; this counter is incremented by one whenever a load operation occurs, thus implementing a cyclic replacement algorithm. The 64-bit main store in the CRAY-1 is an 8-way or 16-way interleaved bipolar semiconductor store having a 50 ns cycle time. During a block transfer all other store requests are inhibited, and sequential accesses can be made at a rate of one per 12.5 ns clock period. In the case of transfers to an instruction buffer, four storage banks can be accessed in parallel, giving access to 16 instruction parcels in one cycle and allowing all 16 banks in a 16-bank configuration to be accessed in four clock periods. Since the cycle time is also equal to four clock periods, the first four banks are then ready to accept a further request, and a complete block transfer

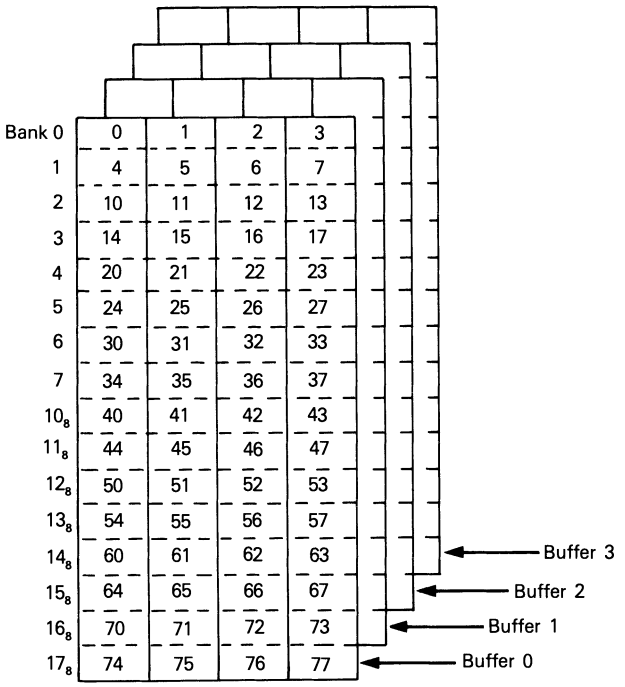

Figure 5.6 CRAY-1 instruction buffers

to an instruction buffer occupies four cycles of each bank. The total time required to access the first group of instruction parcels is nevertheless quite long, and a 14 clock period delay is incurred whenever a buffer has to be loaded. This delay is constant regardless of the position of the first parcel required from the buffer, since a technique is employed similar to that in the IBM System/360 Model 85 cache, whereby the first group of 16 parcels delivered to the buffers is always the one required immediately by the processor. Subsequent groups arrive at a rate of 16 parcels per clock period and fill the buffer circularly.

When a branch is taken the new value in the program address counter is compared with the contents of the buffer starting address registers in exactly the same way as it is following execution of instructions in sequence. If a match occurs the required instruction is selected from the appropriate buffer, and if not a block transfer is initiated. Separate subroutines, or even non-contiguous segments of code within a loop, may be held concurrently in separate buffers. The buffer contents are only invalidated, by having their starting addresses set to all ones, when an Exchange Jump occurs.

5.5 Position of the Control Point

In any computer executing instructions in a sequential manner, there is a register (called variously the Program Counter, Instruction Address Register, or Control Register) that contains the address of the instruction currently being executed. In a pipelined processor this Control Register points to an instruction at one particular stage of the pipeline, the *Control Point.* All instructions before the Control Point are being decoded, having their operands fetched, and so on, in such a way that they can be abandoned at any time, without causing any irreversible change to the state of the machine. Those after the Control Point have had the Control Register incremented past their address, and so must be guaranteed to complete their execution, even in the event of an interrupt.

Depending on the position of the Control Point, the pipeline can be described as *early control*, with the Control Point at an early stage in the pipeline (often the first), or *late control*, with the Control Point several stages further on. The position of the Control Point depends on a number of factors including the instruction set, hardware organisation, and the nature of program addresses (real or virtual, for example). Thus the parallelism of the functional units in machines like the CDC 6600 indicates an early Control Point, while the pre-processing required for an MU5 virtual address is easier to perform with a late control pipeline. Within these guidelines, there is a degree of flexibility in the position of the Control Point, and designers have to take other factors into consideration.

The effects of control transfers are particularly important. Control transfers can be divided into conditional control transfers, where the branch only occurs if some condition within the processor is satisfied, and unconditional control transfers, which always branch. Clearly, a conditional control transfer instruction may be successful or unsuccessful, depending on the state of the condition when it is executed, and the relative proportions of successful conditional, unsuccessful conditional, and unconditional control transfers are important to the performance of the instruction fetching strategy used in the processor.

Control transfer instructions are normally executed at the Control Point stage in the pipeline. Since control transfers are often followed down the pipeline by wrong sequences of instructions, the delay incurred when these instructions have to be discarded increases in proportion to the number of stages in the pipeline between the main store and the Control Point. A late control pipeline would therefore seem to imply long delays for control transfer instructions. However, a closer examination reveals a more complicated situation.

The decoding of any instruction in the pipeline normally takes place before the final execution of some preceding instructions. This produces

an information gap, or *Gulf of Ignorance*, between instructions entering the pipeline and those completing execution. This particularly affects conditional control transfers, since their execution depends on a value or condition determined by some previous instruction, or, in the case of machines like the CDC 6600, by the control transfer instruction itself. If this value is calculated by an instruction within the Gulf of Ignorance, then the control transfer is *unresolvable* and cannot proceed until the required instruction completes execution. *Resolvable* control transfers are those which are unconditional or which depend on the outcome of a long-passed instruction, outside the Gulf of Ignorance.

Thus the proportion of control transfers that are resolvable is very important, as a resolvable control transfer can be obeyed as soon as it is decoded. Unresolvable control transfers must normally be held at the Control Point until the required condition or result has been evaluated, usually at, or near, the end of the pipeline in one of the arithmetic units, and a gap will occur in the flow of instructions through the pipeline. The length of this gap is determined by the number of pipeline stages between the Control Point and the later stage where the arithmetic is performed[4].

Once the condition has been resolved, the control transfer can be obeyed. Unless the appropriate sequence is already following the control transfer down the pipeline, however, a further delay will be incurred while the new sequence of instructions is requested from store (or the instruction buffer). This delay will be equal to the instruction access time plus the time to pass through the pipeline from the store to the Control Point. Thus, for a conditional control transfer followed by the wrong sequence of instructions, the delay subsequently incurred depends on the numbers of pipeline stages both between the main store and the Control Point and between the Control Point and the arithmetic unit, that is, the total pipeline length. However, if the correct instruction sequence following the conditional control transfer has been supplied to the pipeline, then this delay is only proportional to the number of stages in the pipeline beyond the Control Point.

On at least some occasions conditional control transfer instructions will be followed by the correct sequence (when they are unsuccessful in a simple system), and in these cases the pipeline delays are reduced if the Control Point is as late as possible in the pipeline. In the absence of any special techniques for supplying correct instructions after unconditional transfers, however, the pipeline delays for unconditional transfers are reduced by placing the Control Point as early in the pipeline as possible. Clearly, the relative

[4]It is this gap which the Conditional Mode of the IBM System/360 Model 195 is designed to overcome, in cases where the outcome of the conditional transfer has been correctly predicted. When it has not the Instruction Address Register at the Control Point has to be wound back to take account of the instructions issued but not actually obeyed.

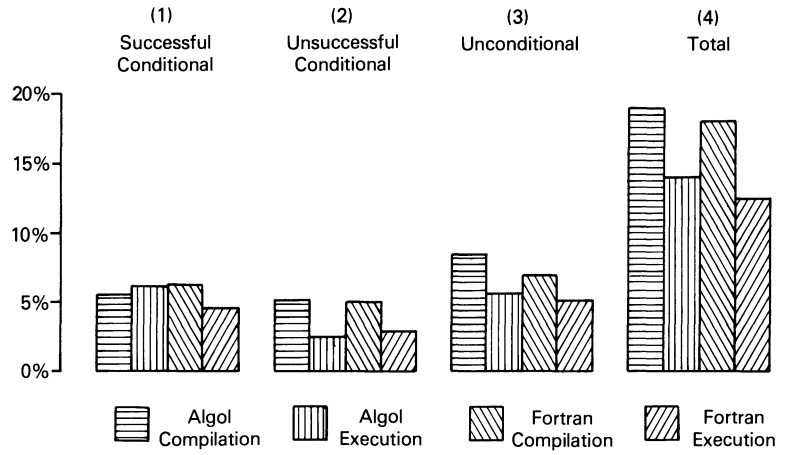

Figure 5.7 Percentages of control transfers

proportions of these instructions are important. Measurements made using the MU5 hardware performance monitor produced the results shown in figure 5.7 for the proportions of different types of control transfer instruction obeyed during compilation and execution of 10 Algol and 15 FORTRAN benchmark programs [HI80].

During Algol execution 14.0 per cent of all instructions obeyed are control transfers, made up of 6.0 per cent successful conditional, 2.4 per cent unsuccessful conditional and 5.6 per cent unconditional transfers. For FORTRAN execution 12.5 per cent of all instructions are control transfers, made up of 4.5 per cent successful conditional, 2.8 per cent unsuccessful conditional, and 5.2 per cent unconditional transfers. During both Algol and FORTRAN compilation the proportions of control transfers are higher, and the success rates of the conditional transfers are lower. Both of these differences arise from the nature of the tasks. The execution figures include large numbers of loops of numerical calculations ended by successful conditional control transfers, while the data-dependent nature of the compilation task involves many unsuccessful tests for particular values of the input data.

The relative numbers of predictable and unpredictable control transfers are also important in MU5, since only predictable control transfers (those which use a literal operand) cause a Jump Trace entry to be made following their first successful execution. The results in column 1 of figure 5.8 show that the percentage of these predictable control transfers (including conditional and unconditional) is fairly constant at over 85 per cent for all the types of program considered.

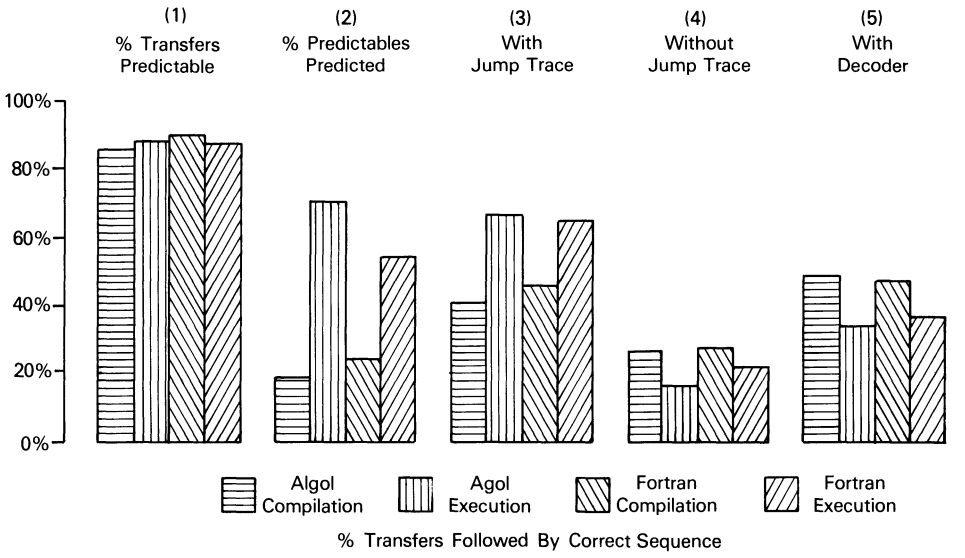

Figure 5.8 Control transfer prediction rates

Column 2 of figure 5.8 shows the proportion of predictable transfers that are correctly predicted by the Jump Trace. The figures for execution are much better than for compilation, and again the nature of the tasks accounts for this difference. During execution, the Jump Trace appears to offer less advantage for FORTRAN than Algol programs, even though many of the benchmarks actually correspond to the same task carried out in each language. When the percentages of control transfers followed by correct sequences of instructions are considered, however, as in column 3, the lower success rate of conditional transfers in FORTRAN compensates for this effect. The figures in column 3 may be compared with those in column 4, which shows the percentages of control transfers what would be followed by correct sequences in the absence of the Jump Trace.

An alternative system that might seem to produce useful improvements in the numbers of correct sequences is one that decodes and executes predictable unconditional control transfers at an earlier stage in the pipeline than the Control Point. Such a scheme is used in the more powerful machines in the ICL 2900 series of computers. Using MU5 figures for the numbers of unsuccessful conditional control transfers and the predictable unconditional transfers, the percentages of control transfers followed by correct sequences using the decoder scheme alone, would be as shown in column 5 of figure 5.8. The effects for compilation are remarkably similar to those obtained using the Jump Trace, though clearly for execution this

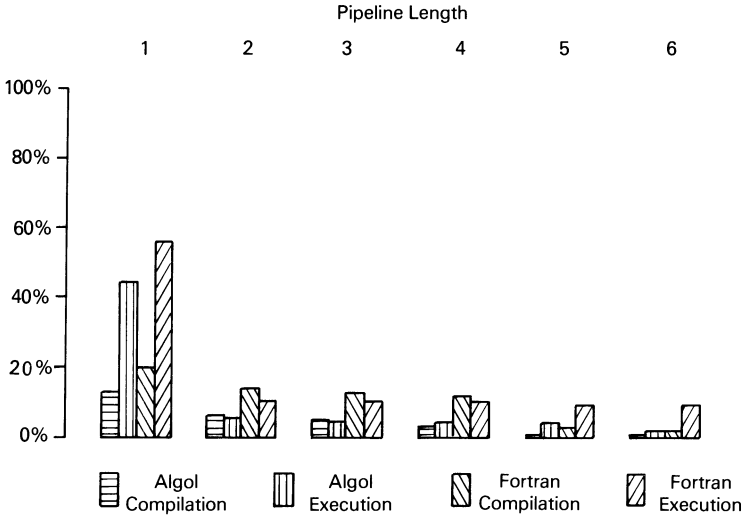

Figure 5.9 Percentages of resolvable control transfers

scheme is less attractive. Furthermore, large gaps will occur in the pipeline unless the decoding and execution of an unconditional control transfer occurs sufficiently far in advance of the Control Point for the new sequence to be accessed from store and supplied to the pipeline before the preceding instructions have all been processed.

A further set of measurements taken from MU5 concerns resolvability. Conditional control transfers in MU5 depend on the outcome of a previously executed COMPARE order, the result of which is held in a separate Test Register, and measurements of the numbers of conditional control transfers separated by a given number of instructions from the preceding COMPARE order were used to determine the percentages of conditional control transfers that would be resolvable for a given pipeline length (figure 5.9). Clearly, for a pipeline separation of more than one stage between the Control Point and the arithmetic unit carrying out the comparison, most conditional control transfers are unresolvable, even though the MU5 compiler writers attempted to separate comparison and control transfer orders wherever possible in the compiled code. If all conditional control transfers are assumed to be unresolvable in MU5, their proportion of the total number of control transfers amounts to very close to 60 per cent for each type of program considered. This figure agrees very closely with that obtained by Flynn from measurements taken from an IBM System/360 machine [Fly74]. Flynn found the overall proportion of control transfers higher, at 9 per cent resolvable and

16.5 per cent unresolvable (27.5 per cent in all), but the proportion of control transfers which were unresolvable was also 60 per cent.

The performance of pipelined processors is critically dependent on the effects of control transfers, as we saw in section 5.3, for example. What is important is the total length of the pipeline, and simply designing for greater overlap within a processor will not necessarily improve performance. For a given pipeline length, however, performance can be improved by placing the Control Point as late as possible in the pipeline and supplying the pipeline with the correct sequence of instructions after control transfers as frequently as possible. This necessarily involves the prediction of the outcome of a control transfer, and this must normally be made as early as possible in order to overcome store access delays. A system such as the MU5 Jump Trace, which bases its prediction on the addresses used to pre-fetch instructions, seems to offer the best chance of success in this situation, and indeed IBM now use a similar mechanism, in the 3090 [Tuc86]. Here "a decode-history-table scheme is used, in which a table keeps the history of branches. Just as past references are used to predict the data that should be kept in a cache, so past branching results can be used to predict future branch behaviour. The location of the branch is used to reference the table, and, if the table indicates that the last branch was successful, the current branch is predicted to be successful."

6 *Parallel Functional Units*

In chapter 4 we saw an example of the use of parallel functional units in the IBM System/360 Model 91. In this machine the parallelism was introduced as a means of enhancing the performance of a processor at the top end of a range of general purpose computers. In the case of the CDC 6600, performance was the principal criterion of the design, and as we saw in chapter 2, the use of parallel functional units with an instruction set capable of exploiting this parallelism was a key feature. In this chapter we shall consider the design of the CDC 6600 central processor, with particular emphasis on the Scoreboard, the mechanism used to control the operation of these functional units. We shall then go on to consider some of the design modifications introduced in the CDC 7600, the successor to the 6600. Limitations inherent in the architectures of both the 6600 and 7600 led to the design of the CRAY-1, which we shall consider in chapter 7.

6.1 The CDC 6600 central processor

The overall design of the CDC 6600 central processor was introduced in section 2.1, and is shown again in figure 6.1. Instructions are taken in sequence from the Instruction Stack and issued by the Scoreboard to the appropriate execution unit. Each unit takes its input operands from among the 24 scratch-pad registers (eight 60-bit X (operand) registers, eight 18-bit A (address) registers, and eight 18-bit B (index) registers) and returns its result to one of these registers. The maximum rate of issue is one instruction per minor clock cycle (100 ns), while the units take typically 300 or 400 ns to complete their operations. Thus while the units can in principle all operate simultaneously, simultaneous operation of three or four units is more typical in practice. The issuing of instructions is not straightforward since instruction dependencies can require that some instructions be held up until the completion of other, previously issued, instructions. In order to maximise the amount of concurrent processing, the Scoreboard is designed to issue each instruction as early as possible, within the limits set by these dependencies, so as to allow the following instruction to be issued, hopefully to a different unit. Dependency conflicts are resolved by the Scoreboard through control signals which link it to each unit and by the use of information buffered about the registers in respect of each unit, and about the units in respect of each register. Before discussing the details of

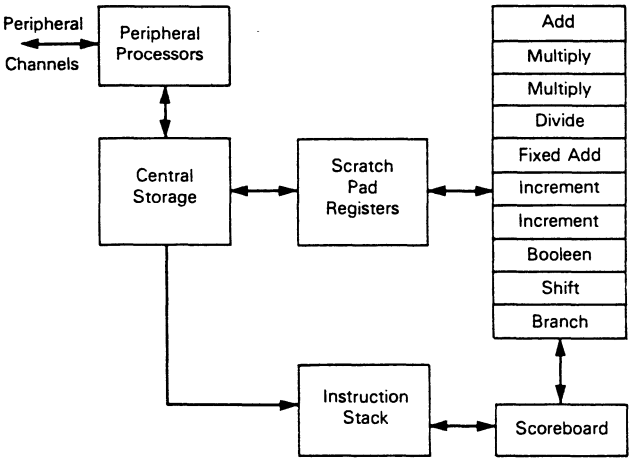

Figure 6.1 CDC 6600 central processor organisation

this system, however, we shall consider the characteristics of the units.

6.1.1 Functional units in the CDC 6600

Boolean Unit

The Boolean Unit performs logic operations of the type

$$X_i = X_j \wedge X_k$$
$$X_i = X_j \vee X_k$$

It operates on each pair of input digits in parallel and returns its result to the designated result register in 300 ns.

Fixed Add Unit

The Fixed Add Unit performs the operations

$$X_i = X_j + X_k$$
$$X_i = X_j - X_k$$

It treats the input operands as 1's complement numbers and uses a 60-bit parallel adder which allows it to return its result to the designated result register in 300 ns. It is also used as a partner to the Branch Unit when the latter is executing a conditional branch instruction which depends on a value in an X register.

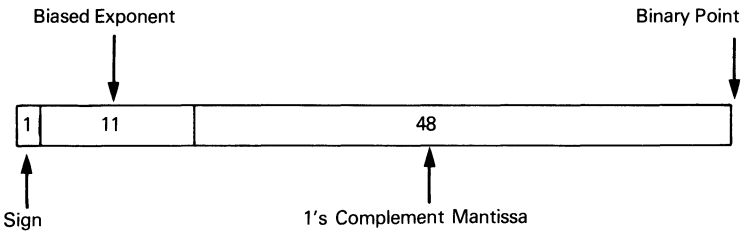

Figure 6.2 CDC 6600 floating-point format

Shift Unit

The Shift Unit performs a variety of operations. The shifter itself is organised as a 6 logic level network capable of performing left circular and right arithmetic shifts to any position in 300 ns. It executes the operations Pack and Unpack, which respectively couple and separate the exponent and mantissa of a floating-point number using registers Xi, Bj, and Xk, and Mask, which forms a string of ones for use by the Boolean Unit in *AND* operations used for masking. The Shift Unit also executes the Normalise operation which takes 400 ns. In most computers the results of floating-point operations are automatically normalised, but this increases the time required for these operations. Normalisation is treated as an optional extra in the 6600, for which the user only pays the time penalty when the facility is used.

Floating Add Unit

The Floating Add Unit also uses 1's complement representation for the 48-bit mantissae of floating-point numbers, but uses an 11-bit biassed binary exponent (an exponent value of 0 is represented as 10000000000). The mantissae are assumed, unusually, to be integers rather than fractions. This has the advantage of allowing fixed-point integers to be converted to floating-point numbers by simply *OR*ing in the exponent bias. Thus the floating-point number format is as shown in figure 6.2.

Floating-point addition and subtraction can be carried out in 400 ns and the result may be a rounded or unrounded single-length result, or the upper or lower half of a double-length result.

Floating Multiply Unit

The 6600 boasts two identical Floating Multiply Units, each capable of producing a rounded or unrounded single-length or the upper or lower half of a

double-length floating-point result in 1000 ns. This unit is a very interesting example of a high performance arithmetic unit, combining as it does the techniques of carry-save addition, multiplier pairs and split multiplier operation. However, we shall not be considering the details of arithmetic unit design in this book; the interested reader is referred to Thornton [Tho70] for details of the 6600 arithmetic units, or to Gosling [Gos80] for the design of arithmetic units generally.

Floating Divide Unit

Floating-point division is the longest arithmetic operation in the 6600, as it is in most computers, and requires 2900 ns to return a result to the designated result register.

Increment Unit

Each of the two Increment Units performs fixed-point addition or subtraction on 18-bit numbers in 300 ns. These units are used for indexing and for the loading of A registers (thereby causing transfers of operands between the corresponding X registers and Central Storage) in operations such as

$$A_i = A_j + K \qquad B_i = A_j + K$$
$$A_i = A_j - B_k \qquad B_i = A_j - B_k$$

Either of these units may also be used as a partner to the Branch Unit when the latter is executing a conditional branch instruction which depends on a value in a B register.

Branch Unit

The result register of the Branch Unit is the Program Address register, which is overwritten by an unconditional branch or a conditional branch for which the condition is met. Determination of the condition is carried out by a partner unit, an Increment Unit in the case of a B value being involved or the Fixed Add Unit in the case of an X value. The Branch Unit itself performs the *jump within stack* test (section 5.2.1) to determine whether or not the target instruction of a branch is in the Instruction Stack. Branch instructions take 800 or 900 ns to complete if the target instruction is in the Instruction Stack, and 1400 or 1500 ns if not. They also hold up the issuing of further instructions until they have completed.

6.1.2 Instruction dependencies

Dependency conflicts between successive instructions in the 6600 are classified by Thornton [Tho70] into three types: first order, second order and

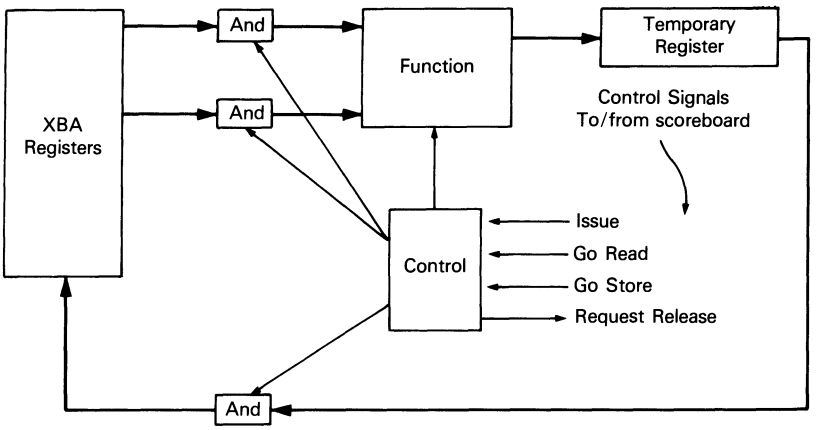

Figure 6.3 Control schematic for a 6600 functional unit

third order. The Scoreboard resolves these conflicts using the control signals which link it to each unit. The Scoreboard can send an Issue, Go Read and Go Store signal to each unit separately, and can receive from each unit a Request Release signal (figure 6.3). The Issue signal enters information into the unit identifying the source of its input operands and the mode in which it is required to operate, while Go Read causes the operands themselves to be copied into the unit. When a unit completes its operation it sends a Request Release signal to the Scoreboard, which responds with Go Store, a signal which allows the result of the operation to be copied out of the unit's Temporary Register and into the appropriate result register.

First Order Conflicts

A first order conflict occurs whenever an instruction which is about to be issued requires the use of an arithmetic unit or a result register which is already in use or has been reserved by a previously issued, but as yet uncompleted instruction. Each instruction in the following pair, for example, requires the use of the Floating Add Unit

```
X6  =  X1 + X2
X5  =  X3 + X4
```

while in the next example each instruction requires X6 as its result register

```
X6  =  X1 * X2
X6  =  X4 + X5
```

Although this latter example is unlikely to arise in normal programming practice, it must nevertheless give the correct result. Without proper interlocks the add operation would complete first and the result in X6 would then be overwritten by that of the multiplication.

Both these conflicts are resolved by holding up the Issue signal for the second instruction until the first has completed. Issuing an instruction involves a sequence of four separate actions. Firstly the functional unit required by the instruction is reserved by the setting of its *busy flag*, and the operating mode, derived from the m field of the instruction, is entered into it. No subsequent instruction can be sent to this unit until its busy flag has been re-set at the completion of the instruction which reserved it. Secondly register designators Fi, Fj and Fk, derived from the i, j and k fields of the instruction, are copied into the functional unit in order to identify the operand and result registers which it will use. Thirdly the Scoreboard copies into the current unit two numbers, Qj and Qk, taken from the identifier registers associated with these operand registers, and finally the identifier register associated with the result register is loaded with the unit number of the current unit. Thus a subsequent instruction which requires the content of this result register as an input operand will receive this unit number as a Q number, and a subsequent instruction which also requires this register as a result register will not be issued until the identifier has been cleared. The importance of Q numbers becomes clear when second order conflicts are considered.

Second Order Conflicts

A second order conflict occurs whenever an instruction which is about to be issued requires the result of a previously issued but as yet uncompleted instruction. An example of such a conflict is the following

```
X6  =  X1 + X2
X7  =  X5 / X6
```

Here the second instruction can be issued, but must not be allowed to start until the result of the first instruction has been entered into X6. This is achieved by holding up the Go Read signal. The first action within a functional unit at the start of any operation is the simultaneous copying into the unit of the two input operands, and this can only occur when Go Read is set. Go Read is the logical *AND* of the Read Flags associated with each of the two input operands.

The sequence of events for the example above is shown in figure 6.4. The unit number of the Floating Add Unit is entered into the identifier register of X6 when the first instruction is issued. When the second instruction is issued this unit number is copied from the X6 identifier register into the

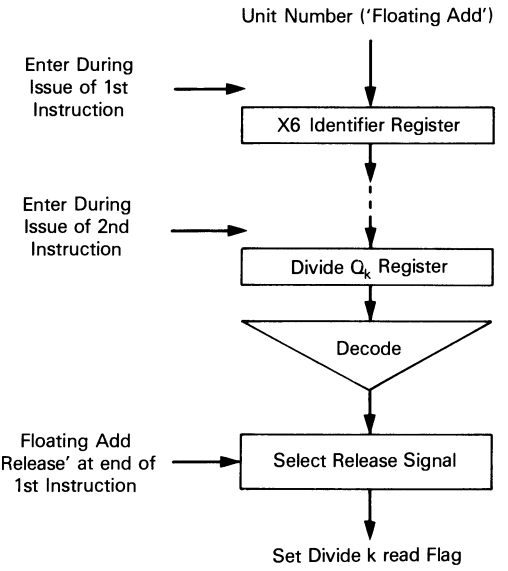

Figure 6.4 Second order conflict event sequence

Qk register in the Divide Unit. When the Floating Add Unit completes its operation it sends its Request Release signal to the Scoreboard, and on receiving a Go Store response, sends a Release signal to all the other functional units. Because the Divide Unit has the unit number of the Floating Add Unit set in its Qk register, it detects this Release signal and sets the k operand Read Flag. Assuming that the Qj register contains zero (indicating that there was no outstanding result destined for X5 at the time when the divide instruction was issued), the Read Flag for the j operand would have been set immediately, and the divide operation therefore starts as soon as the Read Flag for the k operand is set.

Third Order Conflicts

A third order conflict occurs when an instruction which has just completed its operation wishes to store its result in a register which is waiting to supply an input operand for a previously issued, but as yet unstarted instruction. Such a conflict occurs in the following sequence

```
X3 = X1 / X2
X5 = X4 * X3
X4 = X0 + X6
```

The third order conflict here is on register X4, and arises because of the second order conflict on X3. The second instruction can be issued immediately after the first, but is held up for its Go Read signal because X3 cannot be read until the Divide Unit completes its operation. The third instruction can likewise be issued immediately after the second, and since there are no result reservations on its input operands, the Floating Add Unit also receives Go Read immediately and starts its operation. The floating-point add operation completes in very much less time than division, however, and the Floating Add Unit is therefore ready to store its result in X4 before the Multiply Unit has read the current value in X4. Thus the Floating Add Unit sends its Request Release signal to the Scoreboard, but the Scoreboard holds up the Go Store response until after the multiplication has started.

This interlock is achieved through use of the Read Flags. When the multiplication instruction is issued there are no reservations on X4, and its Read Flag is set immediately. This Read Flag remains set until the X3 Read Flag is set at the end of the division operation, after which the multiplication is started and both Read Flags are re-set. Thus the fact that the value in a particular register is required as an input operand by a previously issued, but as yet unstarted, instruction is indicated by the presence of a Read Flag for that register. Conversely, if there are no Read Flags set for a particular register, that register can be described as *All Clear* and a Go Store can be returned to a unit which is ready to send a result to it. The All Clear signal is produced by decoding the Fj and Fk operand designators for each unit and *AND*ing the decoder outputs with the corresponding Read Flag signal to produce an indication of whether or not the unit has a Read Flag set for a given register. For each register the signals from all the units are then combined to produce the All Clear signal indicating whether or not there are any Read Flags set for it. The Go Store signal for a particular unit is formed by decoding its Fi (result register) designator and combining together the results of *AND* operations between these decoded Fi signals and the corresponding All Clear signals for the registers.

In the example above the Fi designator for the Floating Add Unit identifies register X4, and until the multiplication starts the Multiply Unit has a Read Flag set for X4, inhibiting the X4 All Clear signal. When the multiplication starts, this Read Flag is re-set, the X4 All Clear signal becomes true, and the Floating Add Unit receives Go Store. It then sends its result and Fi value along a data highway to the XBA registers where the Fi value is decoded to control entry of the result into the appropriate register, and the clearing of the result reservation in the corresponding identifier register.

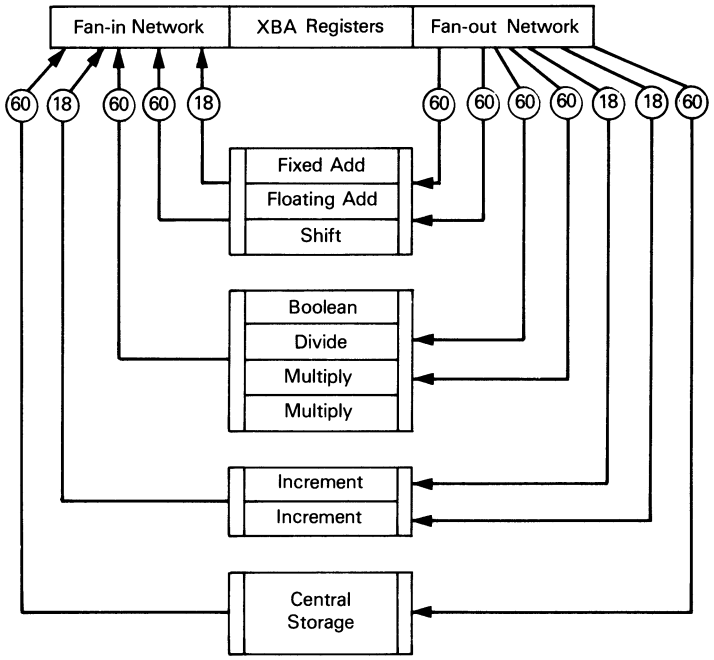

Figure 6.5 Data highways in the CDC 6600

6.1.3 Data highways

The instruction times quoted in section 6.1.1 refer to the time between the issuing of an instruction and the availability of its result in a register. Included in these figures is the time required to transfer operands to a unit at the start of an operation and the time to transfer the result back to the appropriate X, B or A register at the end. These times add up to 100 ns typically, so that the *highway* time amounts to as much as one-third of the total time required to execute short operations. Furthermore, because the Go Read signal may be held up for a particular instruction, the transfer of operands into the unit does not necessarily take place concurrently with the issuing of an instruction to that unit, and may be required to occur concurrently with the transfer of operands for a subsequent instruction issued to a different unit. Thus in order to maintain the maximum benefit of functional unit parallelism, a multiplicity of highways is required between the XBA registers and the functional units.

Figure 6.5 shows the data highways used in the 6600 Central Processor. The functional units are grouped together according to physical layout and performance criteria and each group is served by a separate highway. The

two Increment Units, for example, can cause a traffic of one operand on a highway every two minor cycles, on average, while the two Multiply Units and the Divide Unit can together cause an average traffic of approximately one operand on a highway every four minor cycles, as can the Boolean Unit. Thus the requirements of the Multiply, Divide and Boolean Units taken together roughly match those of the Increment Units. A dual 18-bit highway is therefore used to supply operands to the Increment Units, while two dual 60-bit highways are used to supply each of the other two groups of units, and a single 60-bit highway is used to return operand values from the XBA registers to Central Storage.

The XBA registers are supplied with operands from Central Storage via a single 60-bit highway, and can receive results from the Multiply group of units via a 60-bit highway, from the Increment Units via an 18-bit highway, and from the Add group of units via a dual 18-bit plus 60-bit highway. These latter highways would be used in an Unpack instruction, for example, where the mantissa of a floating-point number is returned to an X register, and the exponent to a B register.

The highways form a considerable part of the processor hardware and represent a significant cost penalty to be offset against the performance advantage obtained from the functional unit parallelism. The Scoreboard control logic also represents a cost penalty, but the amount of hardware which it involves is rather less than that in an average functional unit.

6.2 The CDC 7600 central processor

As we observed in chapter 5 when discussing instruction buffering techniques, the CDC 7600 was designed to be machine code upward compatible with the 6600, but to provide a substantial increase in performance. This increase was achieved principally by a reduction of the minor cycle clock period from 100 ns to 27.5 ns (made possible by advances in technology), together with a corresponding reduction of the major cycle clock period from 1000 ns to 275 ns, and by architectural changes such as the modifications to the Instruction Stack which we have already encountered, and the segmentation, or pipelining, of the functional units.

The overall design of the 7600 (figure 6.6) is in many ways similar to that of the 6600. Instructions are fetched from the Small Core Memory (SCM) and placed in the 12-word Instruction Stack before being copied into the Current Instruction Word (CIW) register for decoding and issue to one of the nine functional units. The control logic associated with the CIW register (section 6.2.2) serves the same purpose as the Scoreboard in the 6600, but is somewhat less complex. On early models of the 7600 the SCM consisted of either 32K or 64K words of 275 ns cycle-time core store made up from 16 or 32 independent 2K word banks. On current models the SCM consists

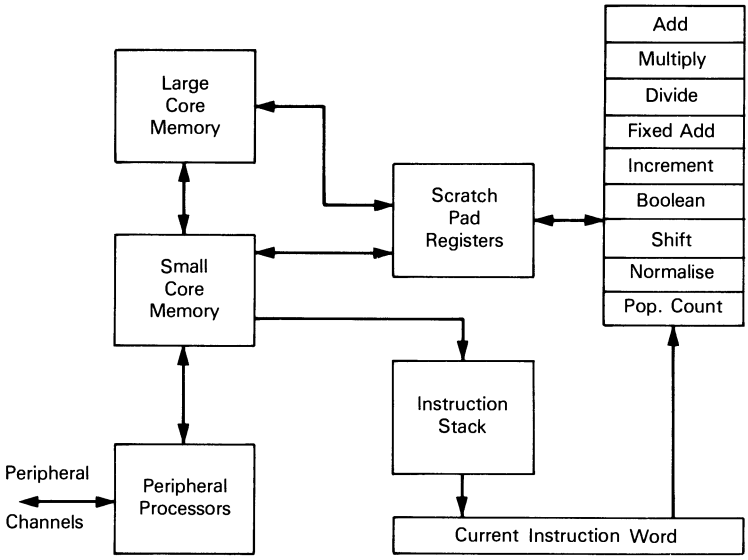

Figure 6.6 CDC 7600 central processor organisation

of either 64K or 128K words of semiconductor memory made up from 32 or 64 independent 4K word banks. Although quite different internally these memories appear virtually identical to the rest of the system and because of the interleaving both can transfer sequentially addressed data at a rate of one word every 27.5 ns.

All instruction accesses and most operand accesses are made to the SCM, and the SCM also communicates with the Large Core Memory (LCM) and with external devices through the Peripheral Processors. The LCM serves as a first level of backing store for the SCM, with blocks of code and data being exchanged between these two, when required, under direct program control. The LCM consists of 256K or 512K 60-bit words in four or eight banks of 1760 ns, word-organised (2D) core store. Each bank operates independently and is eight words wide; once an access has been made to a particular bank the data read out remains available in an output buffer until a subsequent access is made to the same bank. In an eight-bank model this arrangement allows words to be accessed at a rate of 64 per LCM cycle. This time (1760 ns) is equal to 64 minor cycle clock periods, and block copy operations between the LCM and the SCM (in which LCM banks are accessed sequentially) can therefore proceed at the SCM maximum transfer rate of one word per minor cycle.

6.2.1 Functional units in the CDC 7600

Most of the functional units in the CDC 7600 perform identical functions to those in the 6600. The pipelining of 7600 units obviates the need for the duplication of units found in the 6600, however, and control transfers are dealt with by the control logic associated with the CIW register rather than in a separate Branch unit. Two extra units are included, since the normalise operation is carried out in a separate unit from the shift operations, and a separate Population unit is used to carry out the population count function. In the 6600 this function was implemented within the Divide Unit.

Boolean, Fixed Add, Shift and Increment Units

The Boolean, Fixed Add, Shift and Increment Units in the 7600 carry out similar operations to those in the equivalent units in the 6600, but each is pipelined into two stages. Input operands are copied into one of these units at the start of one minor cycle clock period, and a second pair of operands may be copied in at the start of the next minor cycle. At the end of this second minor cycle the result of the first operation is copied into the designated result register.

Floating Add Unit

The Floating Add Unit operates as a four-stage pipeline, but in comparison with the TI ASC arithmetic pipeline the partitioning is expedient rather than elegant. Thus whereas the TI ASC pipeline consists of a number of functionally distinguishable sections, the pipeline stages in the 7600 Floating Add Unit each contain as much logic as can be accommodated, in terms of logic gate delays, within one clock period. The mantissa shifting logic and the mantissa adder, for example, are each split between successive pipeline stages. It is interesting to note that although a new floating-point addition can be started at each minor cycle clock period, the time for any one addition occupies four minor cycles, just as in the 6600.

Floating Multiply Unit

In the 6600 Multiply Unit the mantissa multiplier was split into two parts, each of which carried out a 24-bit by 48-bit multiplication. The results of these two parallel operations were then combined to form a 96-bit product. The 7600 Multiply Unit contains the equivalent of only one of these half multipliers, and this is used twice in a two-pass mode of operation. In the first pass the lower 24 bits of Xj are multiplied by all 48 bits of Xk to form a partial product. This result is then shifted right 24 places and fed back into the multiplier to be combined with the result of multiplying the

upper 24 bits of Xj by all 48 bits of Xk in the second pass. Thus although the Multiply Unit is pipelined in a similar fashion to the other functional units in the 7600, and clocked at every minor cycle clock period, a new multiplication can only be started in every alternate clock period. The total time for any one multiplication is 5 minor cycles, representing a factor of two improvement over the 6600 Multiply Units, each of which took 10 minor cycles to complete a multiplication.

Divide Unit

Floating-point division is executed by a repeated subtract and test algorithm which cannot be pipelined. This operation therefore has a very much longer execution time than any other operation, amounting to 20 minor cycles. A second division can, however, be started 18 minor cycles after the start of a previous division.

Normalise Unit

Normalisation is carried out in a separate functional unit in the 7600, rather than within the Shift Unit as in the 6600. This is because normalisation requires a three-stage pipeline, while shifting and mask generation can be carried out in a two-stage pipeline.

Population unit

The population count function counts the number of ones in the operand taken from Xk and returns this number as a result to Xi. In the 6600 this operation was carried out by hardware contained within the Divide Unit, but since it only requires two clock periods for its execution, a separate Population Count Unit, implemented as a two-stage pipeline, is used for this purpose in the 7600.

6.2.2 Instruction issue

The introduction of pipelining into the functional units in the 7600 had important consequences for the instruction issuing mechanism. Thus whereas the Scoreboard in the 6600 issued instructions as soon as possible, and used the Go Read and Go Store interlocks to maintain proper sequential dependencies, the hold-ups which these interlocks caused could not be tolerated in the functional unit pipelines of the 7600. An instruction is therefore issued from the CIW register in the 7600 only when the conditions in the functional units and operating registers are such that the instruction may be carried through to completion without conflicting with a previously is-

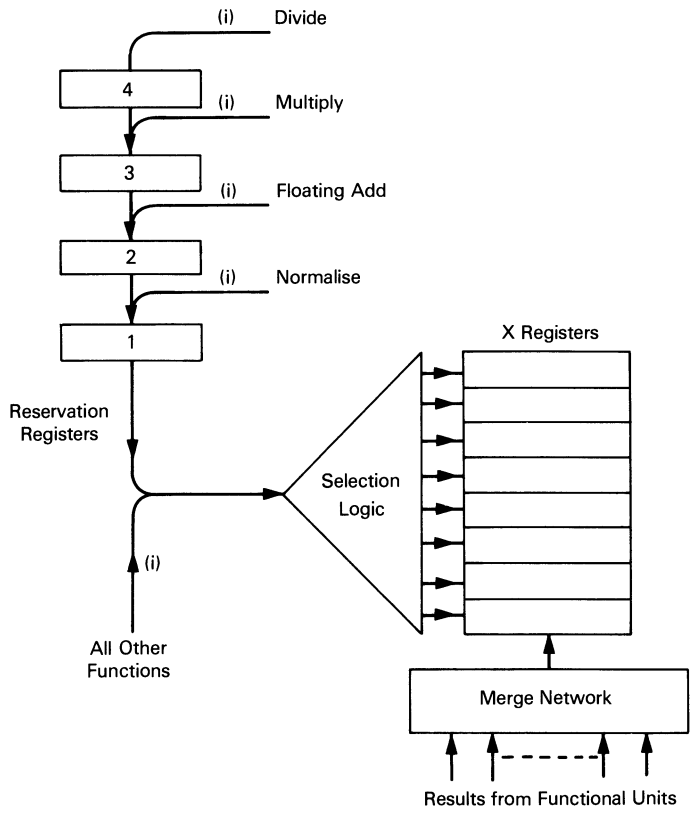

Figure 6.7 X register access control logic in the 7600

sued, but as yet uncompleted instruction. This condition occurs when the two signals *Registers Free* and *Enable Issue* are both in the TRUE state.

Associated with each of the X, B and A registers is a register busy flag. A busy flag is set to reserve a specific register when an instruction which will deliver a result to that register is issued from the CIW register; the flag is re-set when the result appears and is copied into the register. Whenever an instruction in the CIW register is about to be issued, its operand and result register designators are used to select the busy flags associated with these registers. The states of these selected busy flags are then combined to form the Registers Free signal. This signal is only TRUE (allowing the instruction to issue) when there are no busy flags set for the selected registers; if a busy flag is set the issuing of the instruction is held up until the flag has been re-set. This mechanism prevents an instruction from reading or overwriting

any register which is due to be updated by a previously issued, but as yet uncompleted, and possibly slower instruction.

The Enable Issue signal indicates that the states of the functional units, register destination paths and storage access path are such as to allow the current instruction in the CIW register to go to completion. The Multiply Unit, for example, cannot accept a new instruction until two minor cycles after a previous instruction, and the Divide Unit until after 18 cycles. The major problem to be overcome, however, is the occurrence of conflicts on the entry of results into the X registers, which can only accept one result per clock period. If all the functional units took the same time to produce their results this problem would not arise, of course, and does not arise in the case of the A registers, which only receive results from the Increment Unit. The X registers can receive results from any unit, however. Five of the units produce their results in two clock periods, one in three, one in four, one in five and one in 20 clock periods.

The entry of operands into the X registers is controlled by the X Register Access Control Logic, which contains four 4-bit Reservation Registers (figure 6.7). The normal situation is for a functional unit to produce a result two clock periods after receiving an instruction, and for its 3-bit *i* (result register) designator to be sent directly to the Selection Logic for use at the appropriate time. In the case of a normalise instruction, however, three clock periods are required, and instead of its *i* designator being sent directly to the Selection Logic, it is copied into Reservation Register 1, together with a fourth, *valid* bit which indicates its presence. This causes the necessary extra clock period delay to be inserted into the path of the *i* designator en route to the Selection Logic. For a floating-point addition the *i* designator is entered into Reservation Register 2, and for multiplication it is entered into Reservation Register 3. For division the *i* designator is held in the Divide Unit until the 16th cycle in the divide sequence, at which point it is entered into Reservation Register 4. At each minor cycle clock period information is copied through the Reservation Register chain towards the Selection Logic, with the input to any one of these registers being taken either from the previous register in the chain or from the CIW register when a new *i* designator is to be entered into it.

The Enable Issue logic uses the information held in the Reservation Registers to ensure mutual exclusivity of the two sources of information available to each Reservation Register. If a floating-point addition is issued in one clock period, for example, and the next instruction in sequence is a normalise instruction, then in the next clock period the presence in Reservation Register 1 of the valid floating-point add *i* designator inhibits the Enable Issue signal for the normalise instruction and prevents its being issued until one clock period later. If it were not held up, it would produce

a result in the same clock period as the floating-point add, and would also require entry into Reservation Register 1 for its i designator while that register contained the designator for the floating-point add. In the same way, a two-clock period instruction is prevented from issuing in the minor cycle immediately following the issue of the normalise instruction.

6.3 Performance

Since the real work of the computer is carried out on floating-point numbers, with the fixed-point numbers being involved mainly in *house-keeping* operations, it has become fashionable to gauge performance in terms of the rate of execution of floating-point operations, normally measured in millions of such operations per second (MFLOPS). In order to compare the architectures of the 6600 and 7600, however, it is more convenient to consider the number of floating-point operations which each can execute in one clock period (FLOPS/CLOCK). In the case of the 6600, floating-point addition/subtraction requires 4 clock periods, multiplication 10 and division 29. (Division may be ignored in this comparison; it always takes a long time, and computer designers have usually assumed that users will have the good sense to avoid using it as much as possible.) Thus, neglecting division, and allowing for the fact that there are two multiply units, the 6600 can perform 9 floating-point operations (5 additions and 4 multiplications) in 20 clock periods (corresponding to 0.45 FLOPS/CLOCK or 4.5 MFLOPS), with all three of the corresponding functional units being fully occupied. Since the Scoreboard is capable of issuing up to 20 instructions in this time, the right mix of instructions would allow this rate to be achieved. This rate also corresponds to the sum of the maximum rates of execution of long sequences of additions (0.25 FLOPS/CLOCK) and multiplications (0.2 FLOPS/CLOCK) occurring separately.

Maintaining these rates requires that the appropriate operands be available in the X registers, of course. Where these have first to be fetched from store, additional delays are incurred. However, the separation of the operand accessing and function execution facilities in the instruction set allows the possibility, at least, of programs (or compilers) organising appropriate pre-fetching of operands.

In the 7600 an individual floating-point addition or subtraction still takes 4 clock periods, as in the 6600 but, because the units are pipelined, the maximum execution *rate* for these operations is 1 per clock period (1 FLOP/CLOCK). Individual floating-point multiplications take 5 clock periods, but as we have seen, the multiply unit is also pipelined and can perform multiplications at the rate of 1 every 2 clock periods (0.5 FLOPS/CLOCK).

The sum of these two rates produces a total maximum execution rate of 1.5 FLOPS/CLOCK. This rate cannot be achieved, however, since in-

structions can only be issued, and results entered into the X registers, at a rate of 1 per clock period at most. Thus the maximum floating-point execution rate is 1 FLOP/CLOCK, equivalent to 36.4 MFLOPS and made up, for example, of a sequence of additions (in which case the add unit is fully occupied and the multiply unit idle) or a sequence of alternate multiplications and additions (in which case the multiply unit is fully occupied and the add unit 50 per cent occupied). Even sustaining this rate for any length of time is virtually impossible, of course, since it does not allow for the execution of other instructions such as operand accesses and control transfers. However, being able to sustain the maximum rates for addition and/or multiplication for any period of time gives a performance bonus over the 6600 additional to the improvement in clock rate, and CDC claim that the overall performance of the 7600 is 15 million instructions per second.

The first CDC 7600 was delivered in 1969. By the mid 1970s technological advances offered the possibility of increasing the clock rate by a factor of about two, but in seeking to provide an increase in performance over that of the 7600 comparable with that which the 7600 had offered over the 6600, the designers (principally Seymour Cray) were faced with the problem of overcoming the instruction issuing bottleneck in the 7600 design. The solution was found in vector processing, and the architecture which resulted appeared commercially as the CRAY-1. This machine and its successors form the topic of chapter 7.

7 *The CRAY Series*

The need to provide facilities for processing sequences of vector elements was recognised in the very early days of digital computer design. The von Neumann concept, for example, included the notion of allowing instructions to be treated as data, which meant that the address part of an instruction accessing a vector element could be incremented during the execution of a program loop and thus produce the effect of processing a vector. In practice, however, this technique allows so much scope for program error that even the very first stored program computer (the Manchester Mark 1) used B-lines instead, and this latter technique has been used almost universally ever since. Thus virtually any digital computer can be used to process vectors. The differences between machines lie in the addressing facilities which they provide to support accesses to data structures, and whether or not they include instructions which implicitly process a sequence of vector elements. Computers with this latter facility have been described by Flynn [Fly72] as Single Instruction Multiple Data (SIMD) arrangements, in contrast with the Single Instruction Single Data (SISD) arrangement of conventional computers. In fact SIMD systems may be further subdivided into Vector Processors and Array Processors. Systems in this latter category are dealt with in Volume II of this book. We have already met one example of an SIMD vector machine in the TI ASC (section 4.3); in this chapter we shall examine the CRAY series of computers.

7.1 The CRAY-1

The CRAY-1 [Rus78] is the logical successor to the CDC 7600 (section 6.2). The instruction issue bottleneck in the 7600 which prevents floating-point operations from being executed at a rate in excess of 1 per clock is overcome in the CRAY-1 processor by the use of vector orders, which cause streams of up to 64 data elements to be processed as a result of one instruction issue. Vectors are contained in a set of eight V registers, each capable of holding 64 elements (each of 64 bits), and a typical vector instruction causes sets of operands to be taken from two V registers and the results to be returned to a third. In the following instruction sequence

$$\begin{aligned} V0 &\leftarrow V1 + V2 \\ V3 &\leftarrow V4 * V5 \end{aligned}$$

the second instruction uses different registers and a different functional unit from the first and can be issued one clock period after the first instruction. Subsequent to the pipeline start-up delays in the functional units (each of which can carry out operations at a rate of one per clock period), a floating-point result will appear from both the adder and the multiplier in each successive clock period. Thus if performance is estimated in terms only of floating-point addition and multiplication, the maximum floating-point execution rate is 2 FLOPS/CLOCK. Furthermore, around 60 other instructions can be issued before these units require further instructions to keep them busy. With a clock period of 12.5 ns, 2 FLOPS/CLOCK corresponds to 160 MFLOPS.

The other serious bottleneck in the CDC 7600 architecture is the entry of results into the X registers, which is also limited to one per clock period. In the CRAY-1 each vector register has its own input multiplexer circuitry for selecting results from among the seven functional units which can produce vector results and, correspondingly, each vector functional unit has its own input multiplexers for selecting vector register operands. Without these circuits the CRAY-1 would also be limited to 1 FLOP/CLOCK.

The overall design of the CRAY-1 processor is shown in figure 7.1. In addition to the eight 64-element V registers, there are eight 64-bit S (scalar) registers and eight 24-bit A (address) registers (corresponding to the X and A registers in the CDC 6600 and 7600), together with 64 B registers (each of 24 bits) and 64 T registers (each of 64 bits). The B and T registers are used in a different way from any of the registers in the 6600 and 7600, however, in that they act as buffer stores for A and S register values, respectively. The functional units take their input operands from the A, S and V registers only, and only return results to these registers.

The A registers are used primarily as address and index registers for scalar and vector memory references, but are also used for loop control, input-output operations and to provide values for shift counts. An A register can be loaded either from a B register or direct from memory, while the B register contents can be transferred to or from memory in block copy operations which proceed at a rate of one per clock period. The S registers contain scalar operands, which may be used in scalar operations in the same way that X register values are used in the 6600 and 7600, but an S register in the CRAY-1 may also supply a scalar value required for a vector operation. S register values may be transferred to or from memory or the T registers, the latter allowing intermediate results of complex computations to be held in fast buffers rather than main memory. T register values can be transferred to or from memory in the same way as B register values.

The instruction format used in the CRAY-1 (figure 7.2) is very similar to that used in the CDC 6600 and 7600, except that the major function

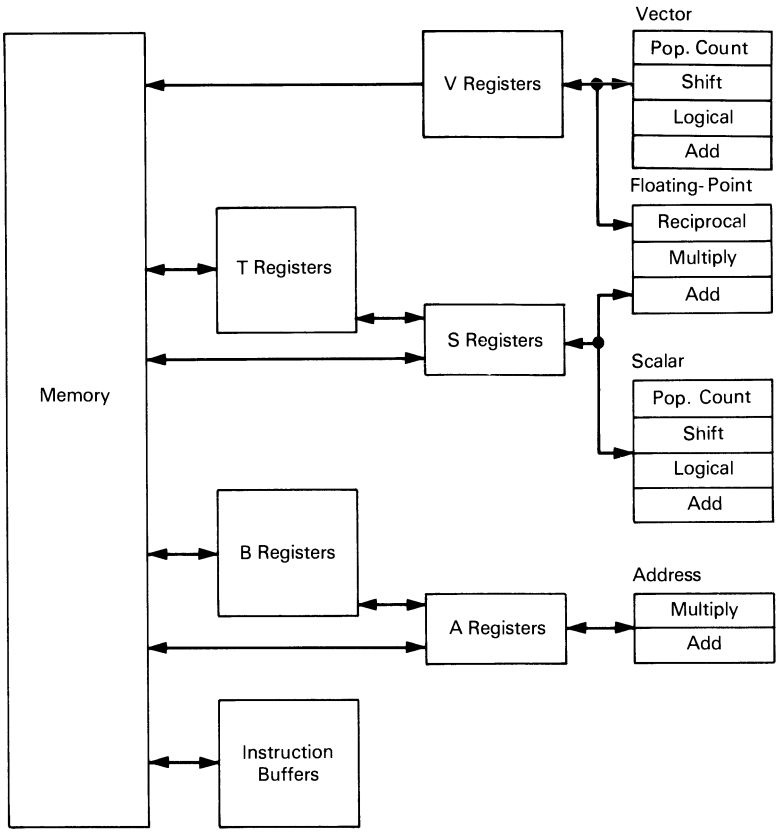

Figure 7.1 CRAY-1 processor organisation

field (g) contains four bits rather than three, and instructions are therefore 16 or 32 bits long rather than 15 or 30. The extra function bit allows vector as well as scalar operations to be specified, and a typical vector instruction takes the form

$$Vi \leftarrow Vj + Vk$$

implying that successive elements of Vk are to be added to successive elements of Vj, and the results returned as successive elements of Vi. Instructions which cause the transfer of an operand between A and B or S and T registers use the combined j and k fields to specify the B or T register. The j and k fields are also combined to produce shift counts in shift instructions.

Instructions which use immediate (literal) operands use the 32-bit format and combine the j, k and m fields to produce a 22-bit literal value.

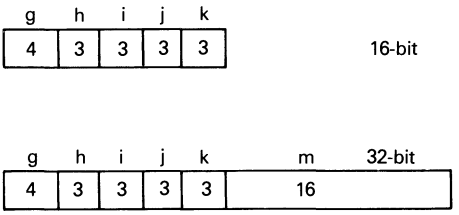

Figure 7.2 CRAY-1 instruction format

Memory referencing instructions similarly combine the j, k and m fields to produce a 22-bit memory address, and also use the h field to specify an A register for indexing. (The memory itself is an 8-way or 16-way interleaved 50 ns cycle-time semiconductor store containing 0.5M, 1M, 2M or 4M words according to the configuration.) Branch instructions combine the i, j, k and m fields to produce a 24-bit memory address field, allowing any 16-bit instruction parcel within a 64-bit word to be specified.

In addition to the operating and buffer registers, the CRAY-1 processor also contains several additional registers which support the control of program execution. These include the program counter, datum and limit addressing registers, interrupt registers, a Vector Mask register (VM) and a Vector Length register (VL). The VM register contains 64 bits, one per element position in the vector registers. In merge operations each bit in VM is used to select the corresponding element of one or other source vector for copying into the destination vector, while in test operations bits in VM are set according to whether or not corresponding elements in a source vector satisfy the chosen condition. The VL register contains a number in the range 0 to 64 and determines how many vector elements take part in an operation. In the case of an operation on a 150-element vector, for example, the hardware would be required to treat this as two successive 64-element operations (with VL = 64) followed by a 22-element operation (with VL = 22).

7.1.1 Functional units in the CRAY-1

The thirteen functional units in the CRAY-1 can be classified into four groups according to the kind of operations they perform and the operating registers to which they are connected. These groups are address, scalar, vector and floating-point. Each unit is pipelined into single clock period segments, so that each can start an operation on a new pair of input operands in each successive clock period.

Address Units

The Address Add and Address Multiply Units perform 24-bit 2's complement integer arithmetic on operands obtained from the A registers and both return their results to an A register. The Address Add Unit (which performs both addition and subtraction) is the equivalent of the Increment Unit in the CDC 7600, and is similarly pipelined into two stages. The Address Multiply Unit is pipelined into six stages and produces as its result the least significant 24 bits of the integer product of two 24-bit operands. Address multiplication is frequently used in the handling of multi-dimensional arrays, and whereas the number format used in the CDC 7600 allowed integer multiplication to be carried out quite straightforwardly in the floating-point multiply unit, the use of a sign and magnitude, fractional mantissa in the CRAY-1 makes the use of this option much less desirable. Furthermore, the floating-point multiply unit in the CRAY-1 is frequently reserved for long periods by vector operations, and a separate address multiplier is therefore essential for performance reasons.

Scalar Units

The Scalar Add, Scalar Shift and Scalar Logical Units perform operations on 64-bit operands taken from S registers and each delivers a 64-bit result to an S register. The Scalar Add Unit performs 2's complement integer addition and subtraction and is pipelined into three stages. The Scalar Shift Unit performs single and double-length shifts on one or two S register operands, the former requiring two clock periods and the latter three. The Scalar Logical Unit is the equivalent of the Boolean Unit in the CDC 6600 and 7600, but it produces its results in one clock period rather than the two required in the 7600.

The fourth unit in this group is the Population & Leading Zero Count Unit which takes a 64-bit operand from an S register and returns a 7-bit result, equal to the number of ones in the operand or the number of zeros preceding the most significant 1 in the operand, to an A register. The first of these operations requires four clock periods for its execution, and the second three.

Vector Units

The Vector Add, Vector Shift and Vector Logical Units take operands from two V registers and return their results to a V register. Successive operand pairs are transmitted to a vector unit in successive clock periods, and after a start-up delay equal to the pipeline length of the unit, results are also copied back to the result register in successive clock periods. The Vector Add Unit performs 64-bit 2's complement integer addition and subtraction

and is pipelined into three stages. The Vector Shift Unit is a four-stage pipeline which performs single-length shifts on individual elements of a V register or double-length shifts on consecutive pairs of V register elements. The Vector Logical Unit performs operations similar to those in the Scalar Logical Unit, but acts on operands taken from V registers rather than S registers, and is implemented as a two-stage pipeline. There is also a Vector Population Unit which operates on vectors in a manner similar to that in which the Scalar Population Unit operates on scalars.

None of the address, scalar or vector arithmetic units detects overflows; CRAY-1 users are supposed to know what they are doing and to write error-free programs. Floating-point out-of-range errors are detected, however, in the Floating-point Units.

Floating-point Units

The Floating-point Add, Floating-point Multiply and Reciprocal Approximation Units perform floating-point arithmetic for both scalar and vector operations. For scalar instructions the operands are obtained from S registers and the results returned to an S register, while for vector instructions the operands are obtained from a pair of V registers or a V register and an S register, and results returned to a V register. When executing vector instructions successive operand pairs are transmitted to a unit in successive clock periods, and results are similarly obtained.

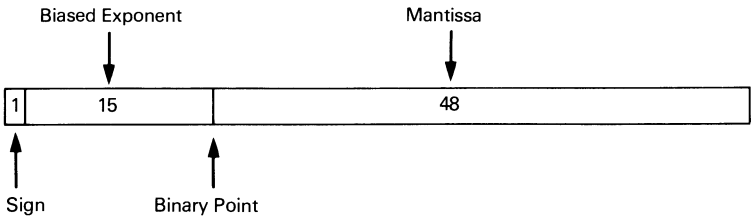

Figure 7.3 CRAY-1 floating-point format

The Floating-point Add Unit performs addition or subtraction of 64-bit operands in floating-point format (figure 7.3) and always produces normalised results. This is a departure from the 6600 and 7600 tradition where normalisation was an 'optional extra' which had to be paid for in extra instructions. The consequence is a longer pipeline in the CRAY-1 Floating-point Add Unit, involving six stages rather than the four used in the 7600.

The Floating-point Multiply Unit is pipelined into single clock period

segments, like all other functional units in the CRAY-1, and in contrast to the CDC 7600 Multiply Unit, which has a two clock period segment time. The CRAY-1 Multiply Unit uses a multiply pyramid to produce the mantissa product, and requires seven clock periods to produce any one result, rather than the five required in the 7600.

The CRAY-1 Floating-point Multiply Unit also participates in division. The standard *subtract and test* algorithm used in most machines to implement division cannot easily be pipelined, and a Newton-Raphson iteration algorithm is therefore used in the CRAY-1, similar to that used in the IBM System/360 Model 91 and in MU5. In the CRAY-1, however, division is not implemented directly, but instead an approximation of the reciprocal of the divisor is formed in the Reciprocal Approximation Unit, and a separate instruction must be used to multiply the result by the dividend in the Floating-point Multiply Unit. Furthermore, the Reciprocal Approximation Unit produces a result accurate to only 30 bits (in a 14-stage pipeline), and in order to produce a result accurate to 47 bits (which is still one bit short of the full mantissa), an additional iteration must be performed, again using the Floating-point Multiply Unit in a separate instruction. Thus a scalar quotient is normally computed in 29 clock periods, and forming an n-element vector quotient requires approximately $3n$ clock periods.

7.1.2 Instruction issue

As we saw in section 5.4, all instructions in the CRAY-1 are accessed from within the instruction buffers; if an instruction request cannot be satisfied by any of the four buffers, a 64-parcel block of instructions is transferred from memory into one of them. A new instruction is accessed whenever the P register (program counter) is updated. For sequential instructions this occurs as an instruction parcel enters the Next Instruction Parcel register (NIP in figure 7.4). From NIP the instruction parcel is copied into the Current Instruction Parcel register (CIP), where it waits to be issued. In the case of a 32-bit instruction the second parcel is contained in the Lower Instruction Parcel register (LIP) which is loaded in parallel with NIP.

As in the case of the CDC 7600, an instruction is only issued when the conditions in the functional units and operating registers are such that the instruction can be carried through to completion without conflicting with any previously issued, but as yet uncompleted instructions. Thus although any number of the V registers can, in principle, accept results in a single clock period, the A and S registers are similar to the X registers in the 7600; only one A register and one S register can accept a result in any one clock period. Issue of an instruction is therefore delayed if it would cause a result to arrive at either of these sets of registers at the same time as a result from a previously issued instruction.

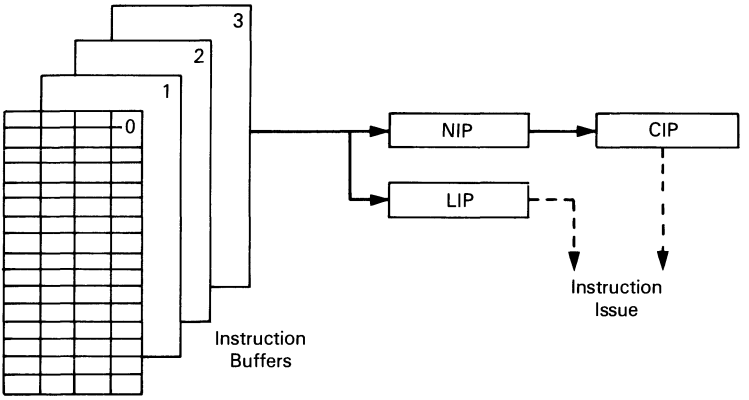

Figure 7.4 Instruction issue registers in the CRAY-1

The CRAY-1 also uses a reservation mechanism similar to that in the CDC 7600. When an instruction is issued which will deliver a new result to an A, S or V register, a reservation is set for that register which prevents the issuing of any subsequent instruction requiring the use of that register until the result has been delivered. In the case of a V register the reservation is for the whole register, rather than individual elements, and furthermore, during the execution of a vector operation, reservations are placed on the operand V registers as well as on the result register. These reservations do not apply to an S register taking part in a vector operation, however, or to the VL register, since their values are copied into the unit carrying out the operation as the instruction is issued. The registers themselves are then immediately free to participate in subsequent instructions.

The need to reserve the operand V registers arises from the nature of the integrated circuits used in the construction of the V registers. These each contain 16 x 4 bits, representing 4 data bits in each of 16 vector register elements, and only one set of 4 bits can be accessed in any one clock period. If two vector instructions using the same operand V registers were in progress at the same time, they would require access to two different elements simultaneously. The same argument applies to the result register; it is impossible to read one element from within a vector register while a new value is being written into another element. The only exception to this rule occurs when an element value which is being delivered to a vector register can, in the same clock period, be routed back into another functional unit as an input operand. This arrangement allows *chaining* of vector operations, and thus allows much more effective use to be made of the parallel functional units than was possible in the CDC 6600 or 7600.

7.1.3 Chaining

Chaining starts when a match occurs between one of the V register operand designators of an instruction awaiting issue in CIP, and the V register result designator of a previously issued instruction which has not yet returned its first result element. When this element becomes available for delivery to the result register, the instruction in CIP is issued (provided there are no other hold-ups) and the result element is forwarded with this instruction to the appropriate functional unit. Successive elements follow until the whole vector has been both written into its result register and forwarded to the second functional unit. The results of this second vector operation may themselves be chained into a third operation, and so on, as shown in the example in figure 7.5.

Figure 7.5 shows a schematic representation of the execution of the instruction sequence

$$\begin{aligned} V0 &\leftarrow \mathit{Memory} \\ V1 &\leftarrow V0 * S1 \\ V3 &\leftarrow V1 + V2 \end{aligned}$$

The first instruction causes 64 operands from a designated area in memory to be read out and copied in sequence into the 64 element positions in V0. Store requests are pipelined in such a way that the store appears to the processor as a pseudo functional unit. Thus after a start-up delay of seven clock periods, the first element of the vector from store becomes available for delivery to V0, and successive elements follow in successive clock periods.

In the clock period following the issue of the first instruction, the second instruction in the sequence is copied into CIP, but the reservation on V0 prevents it from being issued immediately. This reservation is lifted, however, allowing the instruction to issue, during the clock period in which the first vector element arrives from store ready for delivery to V0. This clock period is known as ***chain slot time***. Chaining allows the vector elements being copied into V0 to flow directly from the memory read pipeline into the Floating-point Multiply Unit pipeline, where each element is multiplied by the value taken from S1 at the start of the operation, to produce the vector V1.

The third instruction in the sequence becomes ready for issue in the clock period following issue of the second instruction, and it too is held up by a reservation on one of its input operands, this time V1. When the first element of V1 appears from the Floating-point Multiply Unit, the reservation on V1 is lifted, allowing this third instruction to issue. Now the elements emanating from the Floating-point Multiply Unit can flow directly into the Floating-point Add Unit pipeline as well as into the result register V1. Thus the memory read pipeline, and the Floating-point Multiply and

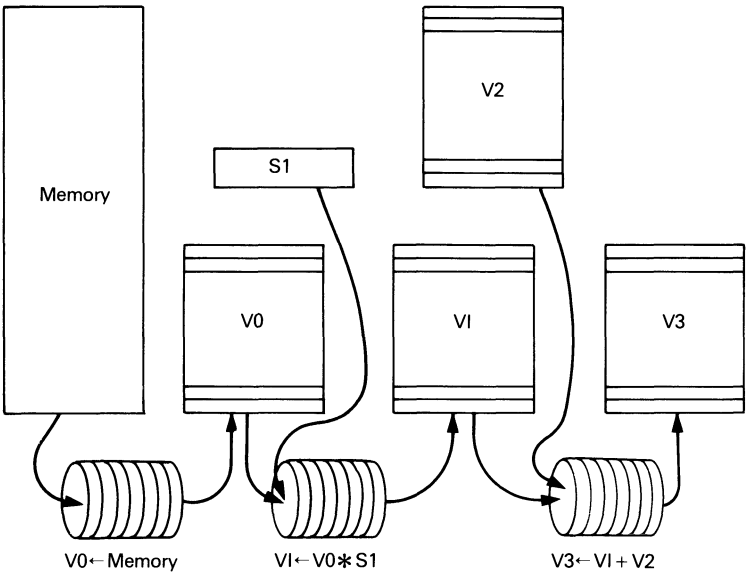

Figure 7.5 Chained vector operations in CRAY-1

Floating-point Add Unit pipelines are all chained together to produce the elements of V3, and the need for all pipelines to have the same segment time now becomes very apparent; pipelines such as those in the CDC 7600, which have different segment times, could not be chained together in this way.

An alternative representation of the sequence of events illustrated by figure 7.5 is shown in the timing diagrams of figure 7.6. Figure 7.6(a) shows in detail the activities involved in producing the first element of V3. The memory read instruction issues in clock period 0 and the first memory word arrives at V0 in clock period 8. During clock period 9 (chain slot time, marked *) this element is transmitted to the Floating-point Multiply Unit (together with the value in S1 for this first element). In clock period 17 the first result element from the Multiply Unit is transmitted to V1, and in clock period 18 (chain slot time) this element is transmitted to the Floating-point Add Unit, together with the first element if V2. In clock period 25 the first result element of the composite operation (V3 = V0*S1 + V2) becomes available and is transmitted from the Floating-point Add Unit to V3. Figure 7.6(b) shows how successive elements of V3 are produced, each one staggered one clock period behind the element preceding it.

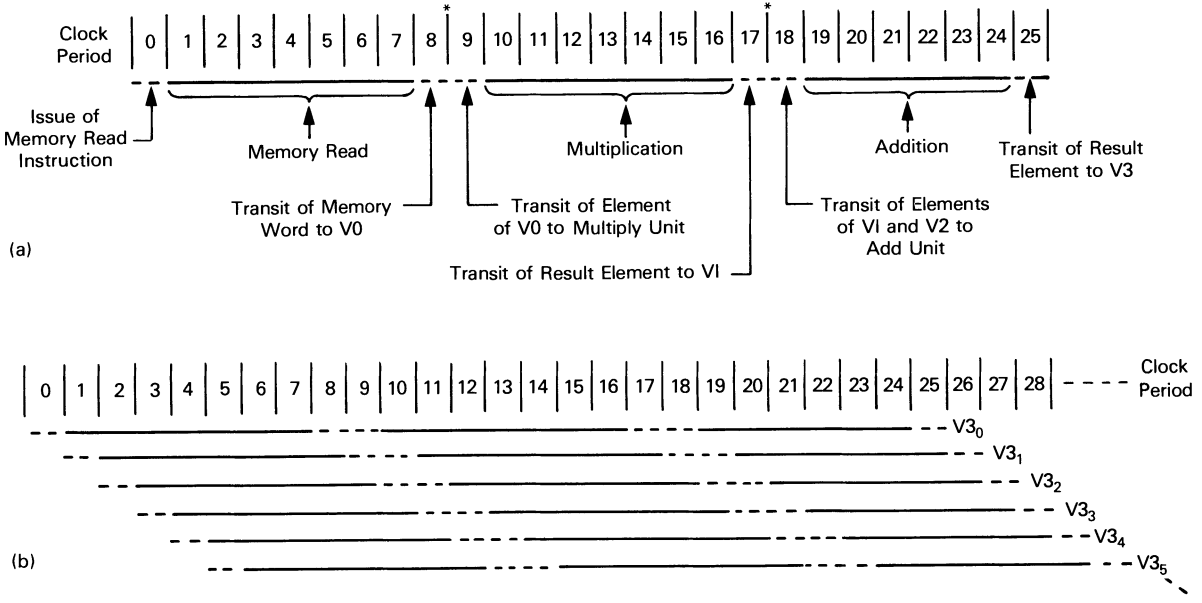

Figure 7.6 CRAY-1 chained vector operation timing diagrams

7.1.4 Performance

Vector processing is only of any value firstly if users' numerical problems involve vectors to a sufficient extent, in terms of both number and length, and secondly if these vectors can be identified during the process of mapping the numerical problem on to the hardware. For CRAY-1 users the mapping problem is largely the responsibility of the FORTRAN compiler, which *vectorises* DO loops and splits long vectors into 64-element segments. CRAY Research Inc. have produced results of performance studies [Joh78] which show that the vector subroutines in their FORTRAN library of mathematical functions out-perform repetitive use of the equivalent scalar subroutines for vectors having as few as only two elements. These results are reproduced in figure 7.7, in which the cost, in clock periods per result, is plotted against vector length. For scalar subroutines the cost per result remains constant, since the subroutine must be called each time. For vector subroutines the cost per result drops very rapidly as the vector length increases from one and approaches a lower limit equal to about one-seventh of the scalar cost. A more detailed analysis of the performance characteristics of vector processors in general is presented in chapter 10.

A second set of results, shown in figure 7.8, relates to matrix multiplication. This involves repeated scalar product operations, and as we saw in chapter 4, this operation lends itself particularly well to vector processing. In the CRAY-1 a vector loop is used to multiply together up to 64 pairs of vector elements to form the same number of sub-products. These sub-products must then be added together in what is essentially a scalar loop to form the scalar product. However, by executing a vector add instruction in which one of the operand registers is the result register of the multiplication, and the other is identical with the result register of the addition itself, a recursive effect is achieved in which 64 sub-products can be reduced to 8 in an operation chained with the multiplication. This recursive effect occurs because the element selection pointer of the result/operand vector does not move on from the first element until receipt of the first result. The value contained in this first element (initially set to zero) is therefore sent repeatedly until the first result element (with a value equal to the first multiplication result) is received in the eighth clock period of the operation. In the ninth clock period the value of this first result is returned as an operand to the Floating-point Add Unit to be added to the ninth result, with subsequent elements following on similarly. The result of this addition is then returned as an operand in the 17th clock period to be added to the 17th result, and so on, until at the end of the add operation elements 56 to 63 of the result vector contain the eight partially summed sub-products.

In figure 7.8 the execution rate in MFLOPS is plotted against increasing matrix dimension for matrix multiplication operations on square matrices.

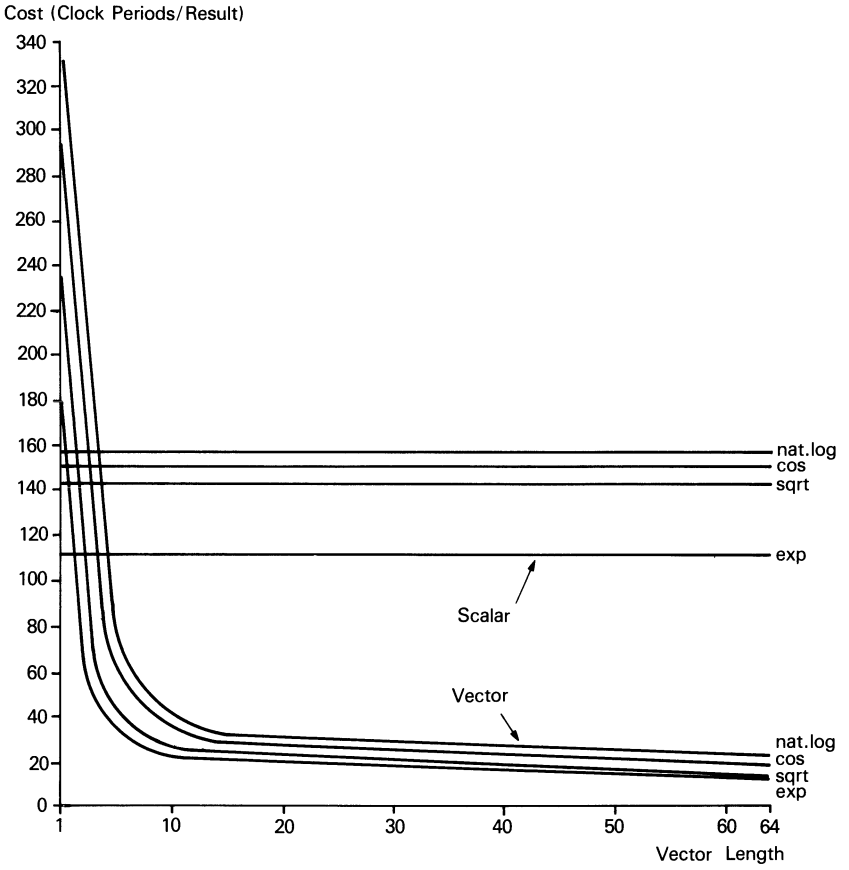

Figure 7.7 Comparison of vector and scalar performance

Matrix sizes which are multiples of 64 give the best results, as would be expected, and execution rates of up to 140 MFLOPS are clearly shown to be possible. This is a best case example, of course, and execution rates such as this cannot be sustained over a more typical mix of user programs. Average rates in the range 20 to 50 MFLOPS have been reported by Hockney and Jesshope [HJ81]. One of the reasons why these rates are considerably lower than the 160 MFLOPS peak performance can be seen in the following analysis.

In order to sustain an execution rate of 2 FLOPS/CLOCK it must be possible to transfer four operands per clock from the vector registers into the floating-point units and two operands per clock from the units back into the vector registers. Since there are eight vector registers, each capable

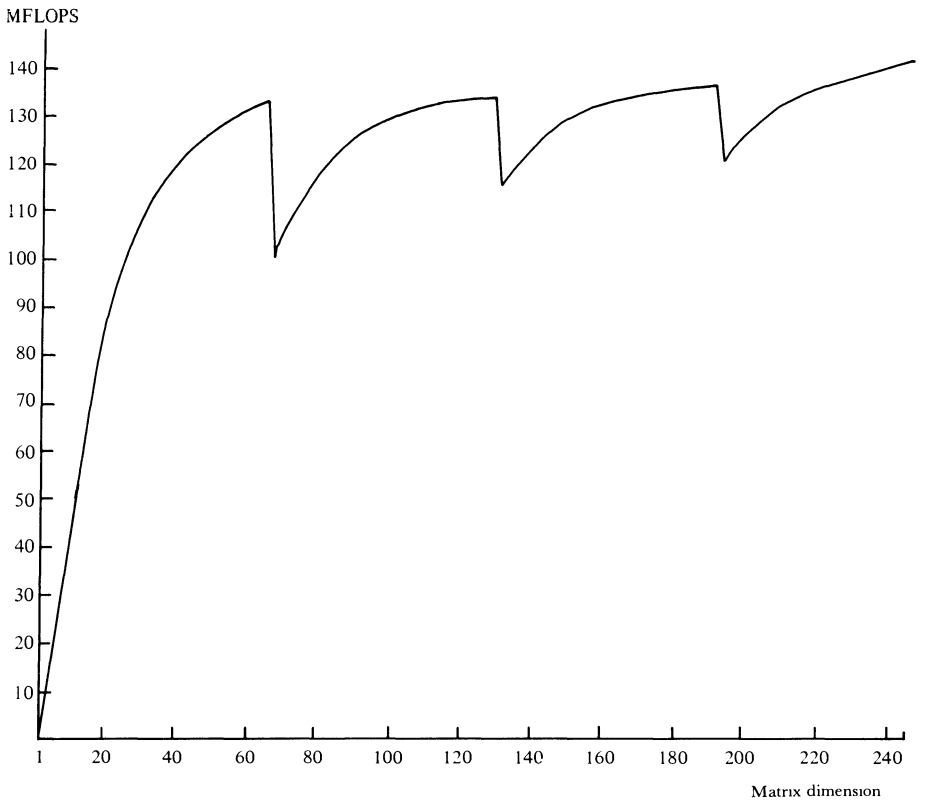

Figure 7.8 Matrix multiplication performance

of transferring operands in or out at a rate of one operand per clock, 2 FLOPS/CLOCK can easily be supported. However, because there are *only* eight vector registers, transfers between these registers and the store are required at regular intervals. For these transfers the data rate is also one operand per clock, but here there is only one data path, so that the bandwidth between the store and the registers is one-sixth of that used between the registers and the floating-point units to support 2 FLOPS/CLOCK.

In the case of matrix multiplication, where a sequence of scalar product operations may be carried out using the same column vector of one matrix repeatedly, it is in fact only necessary to transfer row vectors of the other matrix from store at this rate in order to support 2 FLOPS/CLOCK. In many other situations, however, the restricted bandwidth between the store and the registers is a performance bottleneck. As we shall see in the following section, this bottleneck is overcome in the CRAY X-MP by the use of

multiple paths between the store and the vector registers.

7.2 The CRAY X-MP

The first CRAY-1 was delivered in 1976 and shortly after that Seymour Cray began a development effort aimed at producing a significantly more powerful system, the CRAY-2 (section 7.3). In mid 1979, however, a separate group within Cray Research started work on the design of a system which would have an essentially identical instruction set to the CRAY-1 but considerably more power. The possibility of obtaining this power by adding more vector units was considered, but although improved vector performance was obviously important, improved scalar performance was seen to be more important. A deliberate decision was therefore made to pursue a multiprocessor design rather than one involving multiple vector units. The prototype two-processor version of this system, the CRAY X-MP/2, [Che83,CHLS,Tho86] became operational early in 1982. Following this a four-processor version and a single processor version appeared in rapid succession. The multiprocessor versions not only offer improved overall throughput for single-process jobs in a time-sharing environment but the FORTRAN compiler has an *auto partitioning* capability which allows it to generate object code which utilises multiple processors within a single job whenever possible.

The X-MP range of computer systems now includes models with one, two or four processors and memory options of 1, 2, 4, 8 or 16 million (64-bit) words. The memory is organised as 16, 32 or 64 interleaved banks, according to size, and all banks can be accessed independently and in parallel in each clock cycle (section 7.2.1). This Central Memory is shared by all processors, as shown in figure 7.9, which shows the overall organisation and major data paths of the X-MP/4.

The processors (CPU0 - CPU3) communicate with each other through shared access to clusters of communication registers (CR), and with peripheral devices through shared input-output (I/O) ports. I/O can be initiated by any processor and any processor can in principle deal with any interrupt. In practice I/O interrupts are handled by whichever processor happens to be running the operating system at the time of an interrupt, or, if no processor is running the operating system, by the initiating processor.

The input-output subsystem (IOS) itself contains two or four peripheral processors. The IOS communicates with central memory through one (or in some models two) channels, each having a burst transfer rate of 100 Mbytes/s, with a (possible) variety of front-end computer systems and with peripherals such as disc units and magnetic tape drives. The (optional) Solid-state Storage Device (SSD) is a random access secondary storage device which can be treated as a disc for job swapping and for storing non-

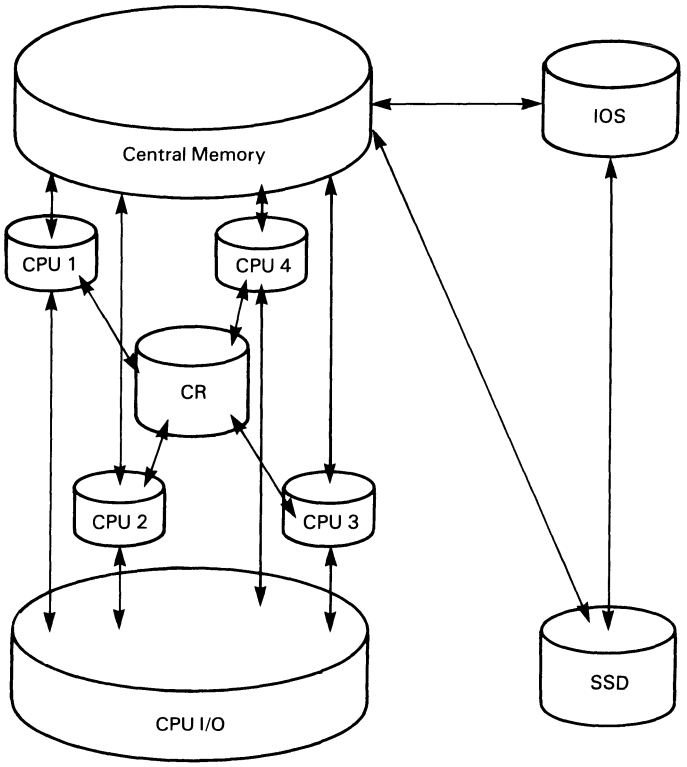

Figure 7.9 Organisation of the CRAY X-MP/4

permanent files. It has a storage capacity of between 256 Mbytes and 4 Gbyte and a 0.4 ms access time. It is connected to central memory by one (or two) channels, each of which has a transfer rate capability of 1 Gbyte/s, one hundred times that of the fastest disc available. There is also an (optional) data path (made up of between one and four 100 Mbyte/s channels) between the IOS and SSD.

7.2.1 Central memory organisation

Central memory is organised as four sections each containing 16 banks, and each processor has an independent access path into each of the four sections, as shown for the X-MP/4 configuration of figure 7.10. The least significant six address bits select the required bank and hence also select the appropriate path to memory. The banks are distributed among the sections in such a way that Section 0, for example, contains (octal) banks 00 - 03, 20 -

23, 40 - 43 and 60 - 63. The CPU Ports contain the logic which generates sequential memory addresses for vector and block transfers. CPU Port A is used by vector block read instructions and for transfers between memory and the B registers. Port B is also used by vector block read instructions and for block transfers between memory and the T registers. Port C is used by vector block store instructions and for block transfers between the B or T registers and memory, and by scalar instructions.

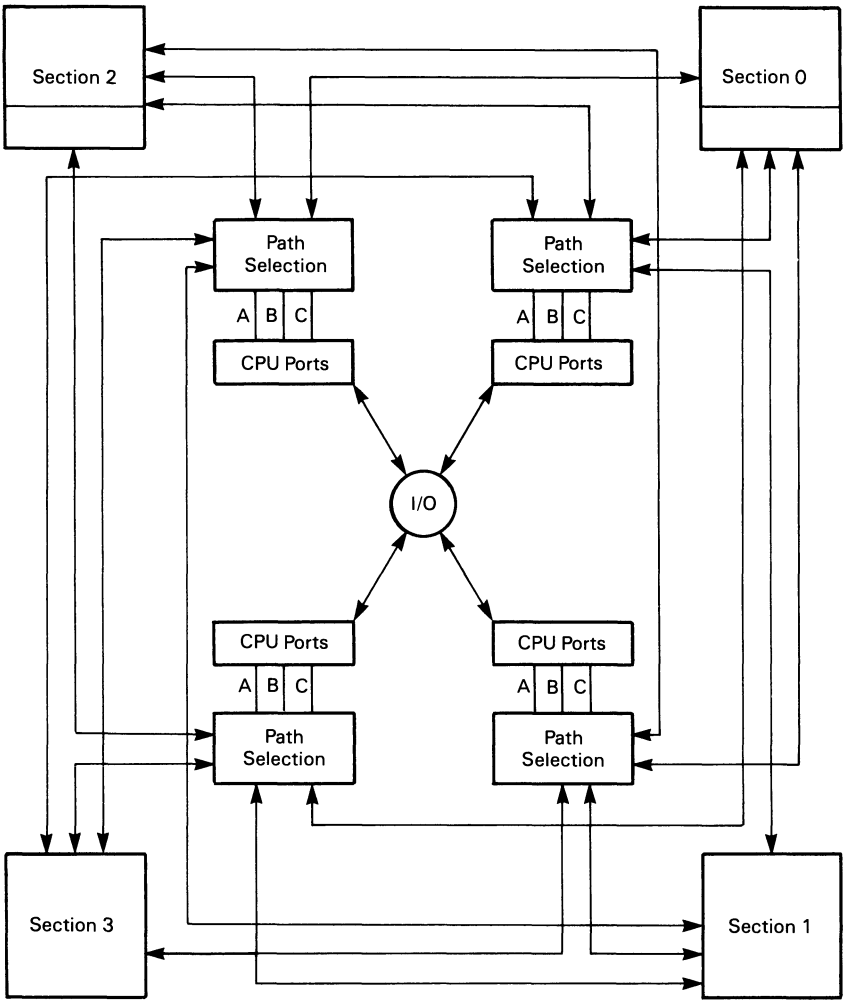

Figure 7.10 Memory organisation in the CRAY X-MP/4

Once issued to a port, the port is reserved until all references in the

sequence have been completed. The flow of data through a port may not be continuous, however, since in each clock period all references to all memory banks are examined for memory access conflicts. If a conflict occurs, then the reference is held up and no further references can be issued from that port until the conflict is resolved.

Three types of conflict can occur: Bank Busy, Simultaneous Bank and Section Access. The memory cycle time is four clock periods and so a Bank Busy conflict occurs if a CPU memory port attempts to access a bank which has been accessed within the last three clock periods. The conflict is resolved when the bank cycle is complete, after one, two or three clock periods. A Simultaneous Bank conflict occurs when two or more ports in different CPUs request the same bank. All ports in the CPU are held up for one clock period while the conflict is resolved, and the loser then suffers a Bank Busy conflict. A Simultaneous Bank conflict is resolved by assigning priorities to the CPUs and allowing the request from the CPU with the highest priority to proceed. To ensure that all CPUs have equal priority on average, the individual priorities are rotated among the CPUs every four clock periods.

A Section Access conflict occurs when two or more ports in the same CPU request any bank in the same section. Resolution of the conflict is again based on a priority mechanism which incurs an extra clock period delay. Among the ports in a single CPU priority is assigned according to the order in which requests were issued to the ports, except that a port with an odd memory address increment always has a higher priority than a port with an even increment, regardless of the issued sequence, since an even increment could well be modulo four and thus be to the same bank as an immediately preceding access.

I/O channels can also reference memory through the I/O port of a specific CPU and the I/O port can be active regardless of the activities on Ports A, B or C. In the case of a conflict, I/O ports have lowest priority.

Figure 7.11 shows the effect of having multiple memory paths on the execution of the following vector computation.

```
C = A + S * B
```

This is common in linear algebra, and compiles to the following sequence of instructions.

```
load A
load B
* S
+
store C
```

A and B are vectors (assumed not to be in the vector registers at the start of this sequence) and S is a value in a scalar register. On the CRAY-1 this sequence of instructions takes three chained operation times (*chimes*) as shown in figure 7.11(a). The chimes consist of (load), (load, *, +) and (store). The single port to memory prevents any further instruction overlap. On the CRAY X-MP this computation takes only one chime, as shown in figure 7.11(b) because two vectors can be read from store, and a third vector written to store, simultaneously via the multiple ports.

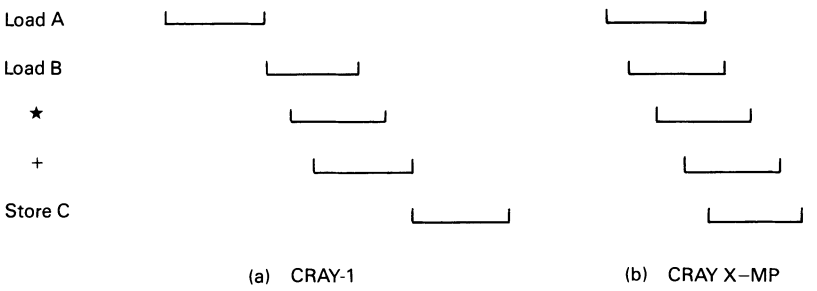

Figure 7.11 Timing of chained operations

7.2.2 Chaining

Apart from the use of multiple memory ports and a doubling of the instruction buffer size, the architecture of each processor in a CRAY X-MP system is basically the same as that of the CRAY-1 shown in figure 7.1. The processor design is nevertheless quite different at the implementation level. The use of the same 16-gate ECL gate arrays as those used in the CRAY-2, and improved packaging density over the CRAY-1, mean that four X-MP processors occupy the same physical space as a single CRAY-1. Increased packaging density implies shorter distances between components, of course, leading to the possibility of a reduced clock period. In early versions of the X-MP the common clock, which provides synchronous control for all the processors in an X-MP system, had a period of 9.5 ns (compared with the 12.5 ns of the CRAY-1) and now has an 8.5 ns period. The performance of the X-MP processor is also improved by an important change in the detail of the implementation of chaining.

In the CRAY-1 two instructions can be chained if the second requires the result of the first as one of its inputs and if it is ready to start at the moment the first element of the first instruction appears out of the relevant arithmetic pipeline, the *chain slot time* (section 7.1.3). If chain slot time is missed, the second instruction must wait until all the elements of the first

instruction have been written into the result vector register. This is because only one vector element can be written into, or read from, a vector register in one clock period. In the CRAY X-MP one element can be written into, and another element read from the same vector register simultaneously. Thus if the second instruction is not ready to start at chain slot time, but becomes ready to start shortly thereafter, it does not have to wait for the first instruction to complete, but can start straightaway.

7.2.3 Interprocessor communication

Communication among the processors in an X-MP system is accomplished through the clusters of communication registers (figure 7.9), of which there are three in an X-MP/2 and five in an X-MP/4. Each cluster consists of eight 24-bit shared B registers, eight 64-bit shared T registers and 32 1-bit semaphore registers. One cluster is typically reserved for operating system use and the others are available for user jobs. The task of assigning clusters to processors is a function of the operating system. At any one time a processor may have none or one cluster assigned to it. Two or more processors may be assigned to the same cluster, however, in which case they are able to communicate. When there is access contention for a cluster, hardware arbitration is used to rotate the accesses among the contending processors in successive memory cycles.

The CRAY-1 instruction set is expanded in the X-MP to allow 24-bit transfers between a processor's A registers and the shared B registers, 64-bit transfers between a processor's S registers and the shared T registers, and between the most significant half of an S register and the 32 semaphore bits. A processor may also set or clear individual semaphores and may also be instructed to wait for a single semaphore bit to become clear before setting the bit itself. The hardware which implements this *wait and set* operation includes an interlock which prevents *wait and set* operations on the same semaphore register being executed by more than one processor simultaneously when the semaphore is cleared.

These operations on the semaphore registers provide the basic facilities for ensuring proper synchronisation between two processes running on separate processors. Further discussion of synchronisation can be found in Volume II of this book, but it is worth observing at this point that the provision in hardware of special semaphore registers avoids the need to use memory locations for this purpose. Wait and set operations *can* be performed on any shared memory multiprocessor system using memory locations, but as we explain in Volume II this can be very wasteful of resources. The hardware controlling the semaphore registers in the X-MP can detect when a processor is in a waiting state and, as a consequence, that processor can be re-assigned to a different task (such as I/O processing) at an inter-

rupt. In addition, the hardware can detect a primary deadlock condition. This arises when all processors assigned to a particular cluster are waiting on semaphores. In this case a *deadlock interrupt* is sent to the waiting processors so that software action can be taken to resolve the deadlock.

7.2.4 The CRAY Y-MP

The CRAY Y-MP, introduced in 1988, represents the next stage of the X-MP line of development. Architecturally the Y-MP is upward compatible with the X-MP but has a 32-bit addressing capability, uses newer technology and offers greater performance. The first model in the range, the Y-MP/832, contains eight processors with a 6 ns clock, 32 Mwords of ECL central memory (in 256 banks) and an SSD memory containing 128 Mwords as standard or 256 or 512 Mwords as options.

7.3 The CRAY-2

Like the Cray X-MP/4, the CRAY-2 [CRA87,Tho86] is a multiprocessor system, the first model containing four processors. These four *Background Processors* are controlled by a single *Foreground Processor* and share a 256 Mword Common (Dynamic MOS) Memory. Other models introduced subsequently include a four-processor version with 128 Mwords of Common (Static MOS) Memory, and two two-processor versions with 64 Mwords and 128 Mwords of Common Memory respectively. All have a common clock period of 4.1 ns.

The organisation of a four-processor CRAY-2 is shown in figure 7.12. The architecture of each of the (identical) Background Processors (section 7.3.1) is derived from that of the CRAY-1, but with a number of differences, the most significant of which is the replacement of the B and T registers by 16 Kwords of high-speed Local Memory. These Background Processors are supervised by the single Foreground Processor, which can, for example, load the Program Address register in each of the Background Processors with the address required to initiate a computational sequence. Thus the Foreground Processor runs UNICOS, an operating system based on AT & T Unix System V, and acts as the system and I/O manager while the Background Processors carry out the computational tasks. The Foreground Processor has its own instruction and data memory, a 32-bit integer arithmetic unit and a real-time clock.

Connecting the various components of the system together are two sets of communication channels. The Foreground Processor has four ports which connect it to the four Background Processors, enabling it to control their operation. In addition there are four 4 Gbyte/s communication channels, each of which connects the Foreground Processor to a Background Proces-

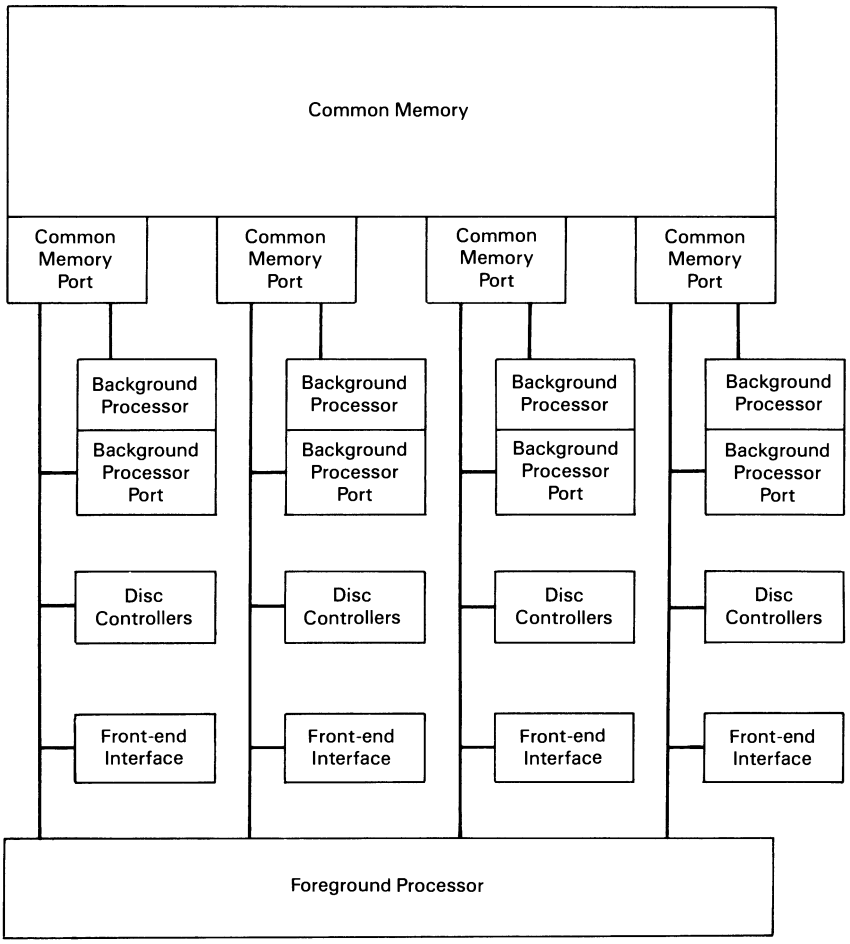

Figure 7.12 CRAY-2 organisation

sor, to one group of disc controllers and front-end interfaces, and to one Common Memory port. The Common Memory ports accept requests on a priority basis, with Background Processor operand requests having highest priority, Background Processor instruction requests second priority and Foreground channel transfer requests lowest priority.

In the largest CRAY-2 system the Common Memory consists of 256 Mwords (1 word = 64 bits) of Dynamic MOS memory arranged in four quadrants, each containing 32 banks. Each 2 Mword bank is connected to four common data paths, one for each Common Memory port. Each

memory access requires exclusive use of a common data path for four clock periods while data is transferred to the port but, because the data is buffered at the port, data words taken from successive quadrants can pass through the port at a rate of one per clock period. Memory addressing is organised to take advantage of this fact, as shown in figure 7.13. When a Background Processor makes a request for a 64-element vector (say), held in sequential addresses, successive requests cycle through the quadrants and access successive banks within each quadrant in successive cycles and thus proceed at a rate of one per clock period.

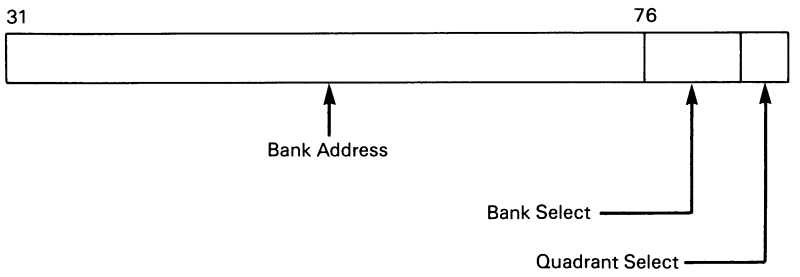

Figure 7.13 CRAY-2 Memory Addressing

If successive vector elements all happened to be in the same quadrant (because the vector stride was a multiple of four, for example), then the accesses would proceed more slowly. Any one Background Processor can only access a given quadrant in every fourth clock period, although if all four Background Processors make successive accesses to the same quadrant, then provided they are to different banks, accesses from the four processors can be interleaved to give an average access rate of one word per clock period through the port.

If all four Background Processors are accessing vectors with elements in sequential addresses, then their accesses become locked into a phased access scheme in which Processor A (say) accesses quadrant 0 in phase 0 of a four clock period sequence, quadrant 1 in phase 1, and so on, while Processor B (say) accesses quadrant 0 in phase 1, quadrant 1 in phase 2, and so on. If Processor A were to attempt to access quadrant 0 in phase 1 of the cycle, then its request would be held up until the next phase 0 of the cycle. Hold-ups are also incurred if Processor B (say) is accessing a vector with elements held in the same bank(s) as a vector being accessed by Processor A.

The large amount of memory provided on the CRAY-2 means that several user jobs can be co-resident in Common Memory. Users familiar with the benefits of virtual memory in an interactive environment have criticised CRAY machines in the past for relatively poor interactive performance.

The provision of a very large amount of real memory largely overcomes this problem, and the CRAY-2 also goes further by providing extremely high transfer rates between Common Memory and its discs. Each of the four disc controllers can accommodate up to nine disc storage units, and all 36 can be operated in parallel at a transfer rate of 10 Mbytes/s each.

7.3.1 Background Processor architecture

The architecture of a Background Processor (figure 7.14) is similar to that of the CRAY-1. As in the CRAY-1, there are eight 64-element, 64-bit V (vector) registers, eight S (scalar) registers and eight A (address) registers. However, whereas these A registers hold 24-bit addresses in the CRAY-1, in the CRAY-2 they hold 32-bit addresses, giving a maximum addressing capability of 4 Gwords. Like the CRAY-1, the CRAY-2 has functional units for address, scalar, vector and floating-point operations, capable of carrying out essentially the same set of functions. There are only nine units in the CRAY-2, however, compared with 13 in the CRAY-1, and the distribution of functional capability among them is therefore somewhat different. A further difference is that there is no chaining in the CRAY-2, a feature that was in fact introduced rather late on in the design of the CRAY-1, and which was subsequently improved upon in the design of the CRAY X-MP (section 7.2.2). Chaining could re-appear in future versions of the CRAY-2.

There is one major difference between the architecture of the CRAY-2 Background Processor and the earlier CRAY processors. The B and T registers are replaced by a Local Memory of 16 Kwords. This is not a cache memory, and indeed there is no mechanism for effecting direct transfers between a Local Memory and the Common Memory. All transfers into and out of the Local Memory are to or from the processor registers, and even context switching must be effected by programming transfers through V or S registers. The large Common Memory size, coupled with the high bandwidth transfer capability between Common Memory and the discs mean that context switching occurs relatively infrequently, however.

The instruction format of the CRAY-2, though basically similar to that of the CRAY-1 (figure 7.2), has to be extended to allow addressing of the Local Memory. Thus in the CRAY-2 a basic 16-bit instruction can be extended by one, two or four 16-bit parcels of constant data. Single-parcel constants are generally used to address Local Memory, two-parcel constants to address Common Memory and four-parcel constants to enter 64-bit values into the S registers. Instruction buffering in the CRAY-2 operates in a similar fashion to that in earlier CRAY processors. Here there are eight buffers each holding 64 16-bit parcels, as compared with four 64-parcel buffers in the CRAY-1 and four 128-parcel buffers in the X-MP.

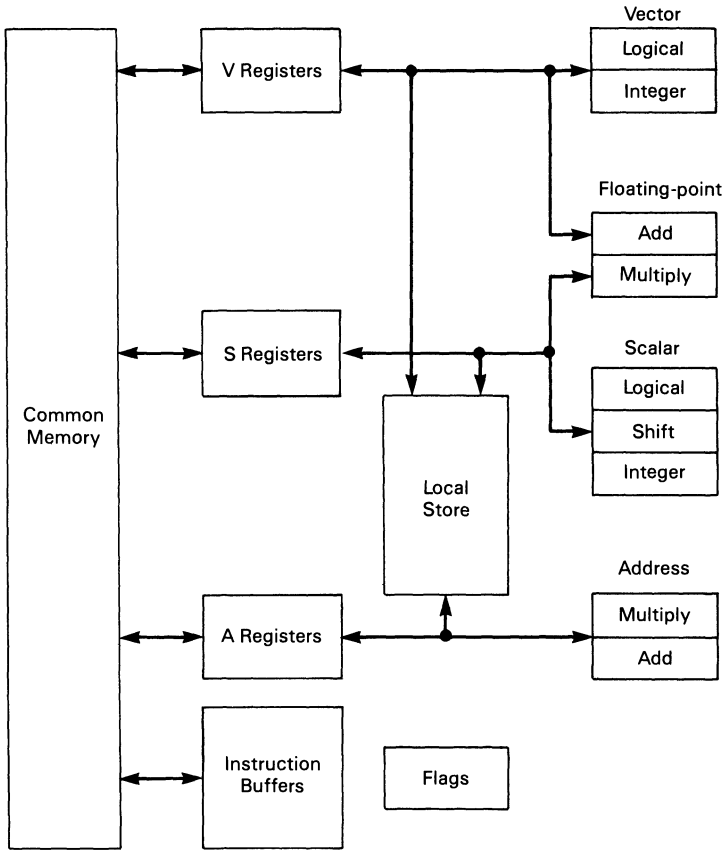

Figure 7.14 The CRAY-2 Background Processor

7.3.2 Interprocessor communication

The mechanism of interprocessor communication in the CRAY-2 is different from that in the CRAY X-MP (section 7.2.3). Here there are no communication registers; instead all data is communicated via Common Memory. In each Background Processor all accesses to Common Memory are made relative to a base register and checked against a limit register, so that each processor has its own protected area of Common Memory. This is essential if each processor is running a separate job.

Where multiple processors are cooperating on the execution of a single job they need to access common areas of memory. To ensure that no two processors attempt to access the same common area of memory simultaneously, eight semaphore flags are provided, accessible from each of the Background Processors. One semaphore is assigned by the system to each

unique group of Background Processors which is cooperating on a multiprocessing task. A Background Port Status register indicates which of the flags is relevant at any time, and instructions are provided which can set or clear the flag and which can jump on the state of the flag. When a processor wishes to access an area of Common Memory which is shared with another processor, it must first check that the flag is clear and, when it is clear, set it. When it has finished operating on that area of memory it must then clear the flag to allow other processors to gain access.

7.3.3 Technology

The circuit technology used in CRAY machines has always been relatively conservative. The technology used in their construction, on the other hand, has always been spectacular. The CRAY-1 uses only three basic (16-pin flat-pack) ECL IC types: a 16 x 4 register IC (cycle time 6 ns), a 1024 x 1 memory IC (cycle time 50 ns), and a logic IC containing two NAND gates, one 5-input and one 4-input. The ICs are mounted on 5-layer 6" by 8" printed circuit boards, and two such boards are mounted back-to-back against a central copper plate to form a module. The copper plates carry the heat from the ICs to the cold bars which form the vertical walls of columns and into which up to 72 modules can be slotted. The cold bars are aluminium blocks containing stainless steel tubes through which flows the Freon coolant. The complete machine consists of twelve columns arranged in a 270° arc, thus giving the machine its now famous C-shaped horizontal cross-section.

The X-MP uses the same constructional technology as the CRAY-1, but the 16-gate array integrated circuits used to implement the logic are not only faster but also denser than the circuitry used in the CRAY-1. Thus an X-MP/4 occupies the same space as a CRAY-1, an X-MP/2 occupies eight columns in a 180° arc and an X-MP/1 six columns in a 135° arc. Memory in the X-MP/4 is constructed of ECL circuits (cycle time 38 ns); in the X-MP/2 and X-MP/1 MOS circuits are used (cycle time 76 ns). The Y-MP moves away from the C-shape and is in fact Y-shaped with the processors occupying the stem and the SSD and Input-Output system occupying arcs which form the arms of the Y.

The CRAY-2 uses the same logic circuits as the X-MP, but mounted in three-dimensional pluggable modules. Each module contains eight printed circuit boards, each holding an 8 x 12 array of integrated circuits. (The largest CRAY-2 contains approximately 240,000 ICs, nearly 75,000 of which are memory.) Up to 24 modules can be contained in a column and the complete system contains 14 columns in a 300° arc. The power supplies are integrated into the bottom of each column, rather than forming a 'seat' around the base as they do on the CRAY-1 and X-MP machines, and the cooling technology is also different.

Cray has always maintained that the most important consideration in building a supercomputer is to use the fastest available component technology, and this inevitably means that heat dissipation is a problem. In fact, building the first cold bar was one of the most significant problems encountered in developing the CRAY-1. In the CRAY-2 the move to bring the heat-generating circuits and the coolant into ever closer proximity is taken to its logical extreme through the use of liquid immersion cooling. Here the coolant flows through the module circuit boards (at a velocity of one inch per second), and is in direct contact with the integrated circuits and the power supplies.

7.4 Beyond the CRAY-2

The CRAY-3 project was started in 1984, the goal being to produce a machine with ten times the performance of the first CRAY-2. Cray himself is quoted as saying "...what we want to do this time is concentrate on speed." A CRAY-3 system contains 16 processors with a 2 ns clock period, half that of the CRAY-2. Because the logic gate delays are about four times smaller, however, there is more logic per pipeline stage, so the overall speed of a CRAY-3 processor is about three times that of a CRAY-2 processor. Much of the performance improvement derives from the packaging. In order to make a system capable of operating with a 2 ns clock period, the maximum allowable wire length is 16". Thus the modules in a CRAY-3 are much smaller than the modules in a CRAY-2. A CRAY-2 Background Processor comprises about 30 modules, each 4" x 8" x 1". A complete CRAY-3 Background Processor contains just four modules, occupying in total about half as much space as a single CRAY-2 module.

The CRAY-4 project is scheduled to start in 1988. The plans include the use of LSI gallium-arsenide technology to produce a 1 ns clock period and a system containing 64 processors.

8 *Vector Facilities in MU5*

In the CRAY series of computers vector processing is used principally as a means of improving vector performance, rather than as a means of providing support in hardware for the management of data structures. Thus a data structure as seen by the programmer is broken up into small sections and mapped into the vector registers by software. In machines which process the programmer's data structures more directly, a number of quite different problems arise from those encountered in the CRAY designs.

As an example of a machine offering sophisticated facilities of this sort we turn first to MU5 and, by implication, its commercial derivatives at the top of the ICL 2900 series. Although these machines do not provide SIMD vector orders of the type found in the CRAY machines, they do include a number of interesting vector accessing mechanisms, and do include SIMD string processing orders which demonstrate many of the features and problems relevant to processing whole vectors via a single order. Some of these problems arose from the use of virtual memory, and in chapter 9 we shall see the same problems arising in the design of the CDC CYBER 205.

8.1 Introduction

As we have seen in earlier chapters, the MU5 instruction set enables information on the nature of operands to be transmitted to the hardware to allow optimal operand buffering. In particular this allows scalar variables, of which only a small number are in frequent use at any one time, to be buffered separately from array elements, which are generally selected sequentially from a large group. Furthermore, hardware assistance is provided for accessing elements of arrays and other data structures, particularly with regard to bound checking. This is achieved through the use of descriptors; an access to a data structure element involves a primary access to the data structure name (the result of which is the descriptor of that data structure), followed by a secondary access involving the descriptor and possibly a modifier. As we saw in section 2.5, descriptors are of two main types

Vector Descriptors	Tv	BOUND	ORIGIN
	8	24	32
String Descriptors	Ts	LENGTH	ORIGIN
	8	24	32

Vector elements may be of size 1, 4, 8, 16, 32 or 64 as defined by bits within the type field (Tv), whereas string elements are constrained to be of size 8 bits, and all are close-packed in store. A typical access to a vector element would involve the following instructions

```
B   =  i
ACC =  X[B]
```

The first of these instructions loads the variable i into the B register for use by the second instruction as a modifier. This instruction loads the named variable X into a descriptor register (DR) for interpretation. The hardware associated with this register first forms the address of the element, by scaling the modifier according to the element size and adding the result to the origin address contained within the descriptor. It also checks that the modifier is greater than or equal to zero, and less than the bound; if not an interrupt is generated. This check incurs virtually no time penalty and removes the need for the costly software bound check which is often required on more conventional machines, particularly during program development. The address is then used to access the store, and the required vector element is selected from within the corresponding store word, right justified, and zero-filled for presentation to the designated arithmetic unit. This mechanism is equally applicable to dynamic arrays whose starting address and bounds are known only at run time, as in Algol-like languages, or to arrays where this information is available at load time, as in FORTRAN.

Operations on whole vectors are organised by program loops such as that in the example already given in section 4.3, which forms the scalar product of two vectors X and Y each of length LIMIT

```
      B   =     0
     ACC  =     0
L1:  ACC  *=    X[B]     ::  Stack ACC, load with element of X
     ACC  *     Y[B]     ::  Multiply by element of Y
      B   CINC  LIMIT    ::  Compare B against LIMIT, inc. B
     ACC  +     STACK    ::  Add Stack content to ACC
      IF  /=,   -> L1    ::  Continue until B = LIMIT
```

The arguments for processing vectors in this way, rather than by specific vector orders, are that in MU5 the hardware can distinguish control and address calculation instructions from those operating on the vector elements, and can execute the former at the peak rate of the Primary Operand Unit pipeline (section 4.2), while the latter are queued in the Secondary Operand Unit pipeline awaiting operand fetches (figure 8.1). Furthermore, the compilers are only required to generate conventional code in a straightforward way without using intricate algorithms to *vectorize* the operations. There

are, however, other features of vector processing which would make vector orders attractive in an MU5-like machine. The scalar product loop is a particular example; as we saw in section 4.3, a running total in MU5 has to be maintained either in the top-of-stack location (as in the coding given above), or in some named variable location. In either case this requires that a new value be written into a buffer store (the Name Store) each time the loop is obeyed, and then read out again by the next but one ACC order. In the absence of complex data forwarding techniques such as those used in the IBM System/360 Model 91, this situation creates problems in a pipeline environment. A vector processing mechanism such as that in the TI ASC, which takes pairs of operands, multiplies them together and adds the result to a running total held within the arithmetic unit offers obvious performance advantages.

By contrast, operations on strings in MU5 are carried out by string processing (store-to-store) orders which provide facilities for the manipulation of data records in languages such as COBOL, PL/1 and Algol 68 [CIP74], and which operate on complete strings of byte elements defined by String Descriptors. We shall deal with these in some detail in section 8.2. Before doing so, however, we shall first examine the hardware used in the MU5 Secondary Operand Unit, and some of the more complex descriptor mechanisms which it implements.

8.1.1 The MU5 secondary instruction pipeline

The MU5 Secondary Instruction Pipeline (figure 8.1) is mainly concerned with accessing and processing the data structure elements specified as secondary operands by means of descriptors. It consists of three major sections: the D-unit, the Operand Buffer System and the A-unit (containing the ACC register). The D-unit is itself made up of two units: the Descriptor Addressing Unit (Dr) and the Descriptor Operand Processing Unit (Dop). The D-unit and the Operand Buffer System together constitute the Secondary Operand Unit (SEOP) of the processor (see figure 3.5).

The Secondary Operand Unit receives instructions from the Primary Operand Unit (PROP). For an instruction specifying an array element as its operand, the descriptor supplied by PROP is loaded into the DR Register, and Dr proceeds to generate the required address using the content of the B-Register as a modifier. This address is then sent to OBS which, like the Name Store in PROP, is invisible to the programmer and exists simply to improve the instruction execution rate. OBS makes the necessary store request and sends the required 64-bit word to Dop, which contains masking and shifting circuitry to select the array element from the appropriate position in the word. Dop is controlled by a combination of a control field generated by Dr (at the same time as the address) and the original function

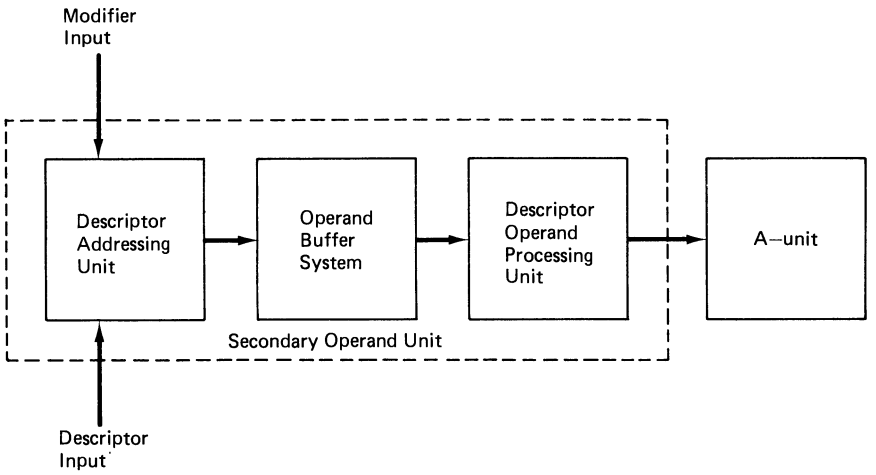

Figure 8.1 The MU5 secondary instruction pipeline

code, which accompanies the instruction as it passes through the pipeline. Instructions leaving Dop are normally sent direct to the A-unit, but may also be sent to PROP, the B-unit or to Dr. In the case of the store-to-store functions, the operand processing takes place in Dop.

8.1.2 Descriptor processing hardware

The basic tasks of the D-unit are comparatively simple. They involve the formation of the address of an array element, by the addition of the modifier to the origin address contained within the descriptor, and the subsequent selection of that element from within the corresponding store word. These tasks are complicated, however, by a number of factors, such as the existence of different descriptor types and different operand sizes, the possibility of an operand straddling across two store words, and the need for bound checking. In addition, the two 64-bit descriptor registers, DR and XDR, can be manipulated in whole or in part by various functions. Also the origin and length fields of one or both of these registers must be incremented and decremented, respectively, during each cycle of the execution of the store-to-store orders (section 8.2.1). For both this task and the bound checking operation, it would clearly be desirable to use a subtractor, separate from the adder used either to increment the origin or to add the modifier and origin, since the appropriate additions and subtractions could then proceed in parallel. However, the MU5 SEOP actually uses one adder to carry out these tasks sequentially, since this arrangement involves less hardware.

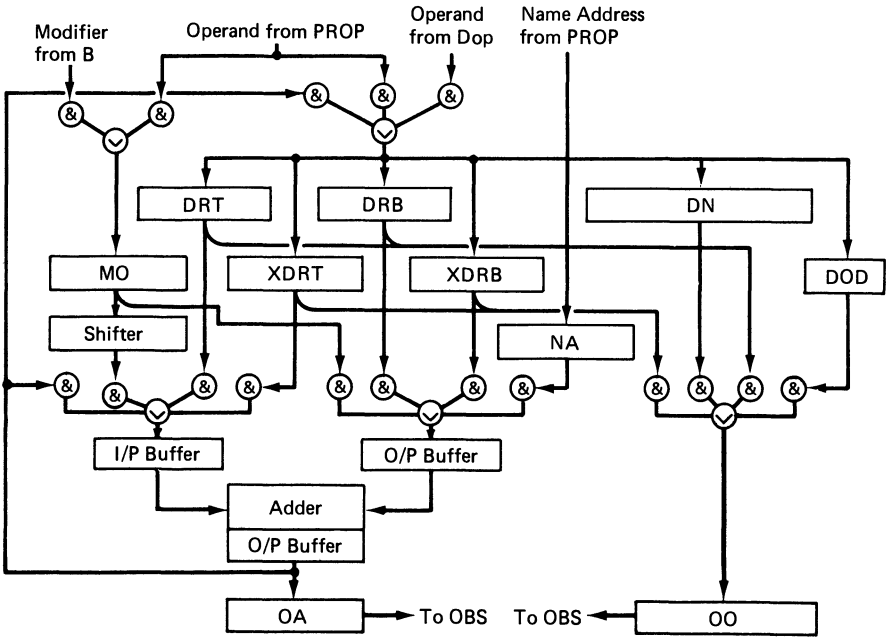

Figure 8.2 The MU5 descriptor addressing unit

Each of the two descriptor registers consists of a top and bottom half (DRT and DRB, XDRT and XDRB). DR is the main descriptor register, used for array accesses and the destination address in store-to-store orders. DRB contains the origin and DRT the type and bound or length fields. XDR is similarly divided and is used to hold the source address required by some of the store-to-store orders. DR and XDR are normally loaded from PROP, but may in some circumstances be loaded from Dop.

The 32-bit modifier is held in register MO, which is loaded from the B-unit during an array access or from PROP during the execution of functions which manipulate DR or XDR. The true output of MO is connected to one side of the 32-bit adder via a shift network, and the complementary output directly to the other side. The modifier normally refers to the position of an element within an array, regardless of its size, and must therefore be shifted relative to the origin, which always refers to a byte address. The complementary output is used in the bound checking operation, where the subtraction of the modifier from the bound (equal to the number of elements in the array) is performed by adding the complement of the modifier to the bound and forcing a carry into the least significant digit position of the

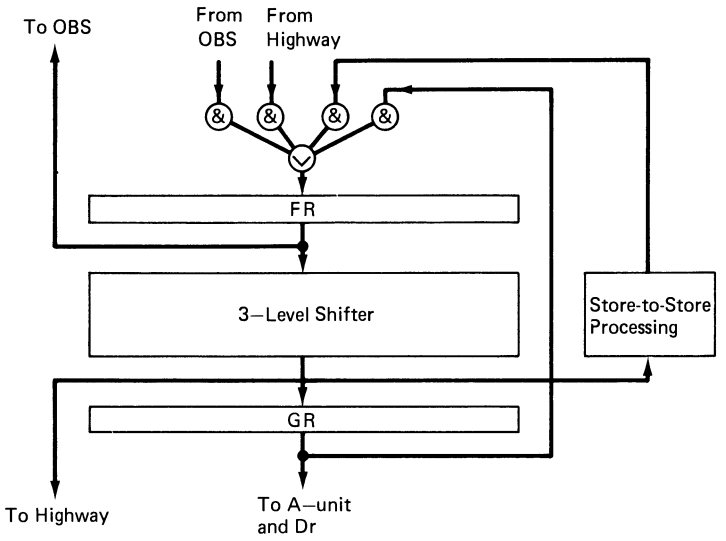

Figure 8.3 The MU5 descriptor operand processing unit

adder. Following the formation of the address, the output of the adder is passed on to OBS, via the OBS Address buffer (OA). This address may also be used internally for the updating action of the store-to-store orders and for array accesses which involve elements straddling store-word boundaries.

Register DN is a 64-bit temporary buffer used to hold primary operands (names, literals or internal register values) associated with those ACC functions which do not require a secondary operand access. These functions pass through the Secondary Instruction Pipeline en route to the A-unit in order that the correct program sequence be preserved, but must not disturb the contents of the descriptor registers in Dr. The contents of the descriptor registers themselves also pass through the primary operand route to OBS (via the OBS Operand Buffer, OO) when required for orders such as *DR* =>. A route through Dr is also required for addresses of names used with ACC functions and held in the OBS Name Store (section 3.4). This is provided via register NA, which is loaded from PROP and can be gated as an input to the Dr address adder. Register DOD is concerned with D-unit interrupts.

Orders leaving Dr are forwarded to OBS, which obtains the store word containing the operand and sends it to Dop. Control information generated by Dr during the address formation is buffered with the function in OBS, and is supplied to Dop with the store word. Using this information, Dop

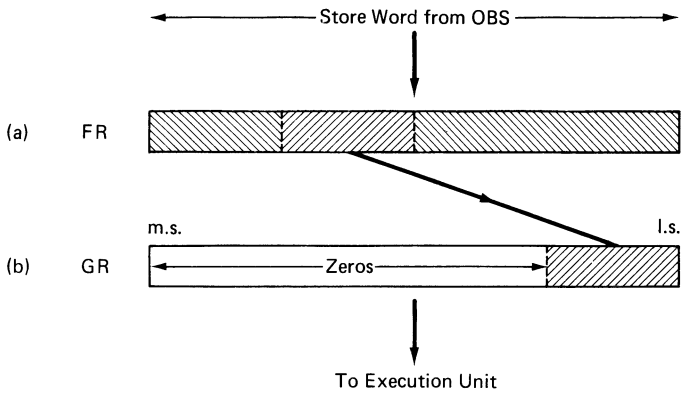

Figure 8.4 Dop actions for a load order

selects the required array element from within the store word, and in the case of a load order, routes it to the least significant end of its output highway. In the case of a store order, Dop updates the appropriate part of the store word and returns the updated version to OBS.

Dop consists of two sections (figure 8.3), one with a 64-bit data path used for operands associated with computational orders, and one with an 8-bit data path used for operands associated with the store-to-store orders. The main input register (FR) has masking facilities which permit the selection of left or right hand masks to any bit position over the full 64-bit width, while the shift mechanism allows any circular shift from 0 to 63 bits in single bit increments. This is achieved by three levels of logic, the first allowing shifts of 0, 16, 32 or 48 bits, the second shifts of 0, 4, 8, or 12 bits and the third shifts of 0, 1, 2 or 3 bits. The output of the shifter is copied into the main output register (GR) if the order is destined for the A-unit or the Dr unit, or gated on to the processor Highway if the order is destined for PROP or the B-unit. The route from GR back into FR is used during the execution of store orders.

Figure 8.4 illustrates the actions taken for a load order requiring a 16-bit operand. The store word containing the operand is loaded into FR (a), and the shifter routes the required 16-bit operand to the least significant end of the input to GR. At the same time, the most significant 48 bits of the input to GR are set to zero by inhibiting the appropriate section of the data path in the third level of the shifter (b). GR is strobed when sufficient time has elapsed for the shifter outputs to settle, following the receipt of a store word in FR, and when the previous order in GR has been accepted by another unit.

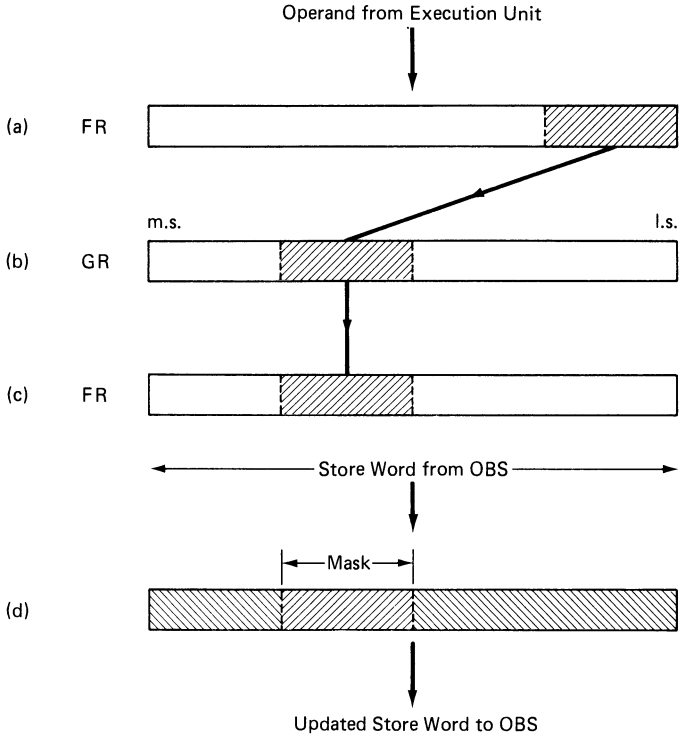

Figure 8.5 Dop actions for a store order

The actions for a store order are more complex. The order first passes through Dop as if it were a load order, en route to the appropriate execution unit, and is only processed by Dop when the execution unit indicates that the required operand is available. The operand is received from the processor Highway (figure 8.3) and is copied into the least significant end of register FR (figure 8.5(a)). It is then shifted to its eventual alignment in the store word, copied into GR (b) and copied back from GR into FR (c). The store word of which it is to form part is then sent again from OBS and selectively copied into FR around the operand, using the masking facility (d). The updated version of the store word is then returned from FR back to OBS.

8.1.3 Complex descriptor mechanisms

The MU5 instruction set has provision for a number of descriptor mechanisms more complex than those described so far. Thus one of the special

descriptor types (Indirect) may be used to describe a structure containing other descriptors. Hence

```
  B  =  i
 DR  =  X[B]
  B  =  j
ACC  =  D[j]
```

where `D[j]` implies use of the existing descriptor in DR, could be used to access an element X[i,j] of a two-dimensional array using an addressing vector technique. This mechanism has the disadvantage in MU5 of requiring a descriptor to be accessed through the Secondary Operand Unit and returned to the front of it before the element access can begin. This destroys the overlap in the secondary pipeline and for simple languages an alternative technique involving the use of the multiplication facilities in the B-unit has the advantage that the subscript calculations take place independently of operations queued in the secondary pipeline and can be substantially overlapped with them. The previous sequence becomes

```
  B  =  i
  B  *  n
  B  +  j
ACC  =  X[B]
```

with a bound check on the overall limits of the array, rather than on each dimension. For languages where the lower bound of arrays is 1, a load and decrement B order is provided. A further, more elaborate, *dope vector* mechanism is convenient for arrays with dynamic lower bounds or for slices of arrays. For each dimension the dope vector contains three 32-bit elements, a lower bound which is subtracted from the subscript, an upper bound against which the subscript is checked, and a stride by which it is multiplied. The SUB1 and SUB2 instructions are used to implement this facility, so X[i,j] becomes

```
  B  =  i
SUB1 X1
  B  =  j
SUB2
  B  =  DO
ACC  =  X[B]
```

where `SUB1` processes the first subscript using dope vector X1, `SUB2` processes the second subscript, and `B = DO` moves the composite subscript in DO (the Origin field of DR) to the B register. From a performance point of view these orders are worse than the Indirect descriptor mechanism, since

several of the actions implied by SUB1 and SUB2 require accesses to quantities which are to be returned to the front of the secondary pipeline. Some quite different structure from that of MU5 would be needed to obtain good performance using these orders.

8.1.4 The Operand Buffer System

The need for an Operand Buffer System (OBS) in the MU5 Processor arises from the disparity between the data rate and access time of the Local Store. In principle the MU5 Local Store can supply 128-bit words at intervals of 65 ns, but the time gap between the generation of a secondary operand address, and the receipt of the corresponding store word, is over 600 ns. When MU5 was designed it was expected that floating-point addition and subtraction would take only around 100 ns, and so the store access time was unacceptably high. The difference between the access patterns for named variables (of which a small group is generally used repeatedly) and data structure elements (which are generally selected sequentially from among a large group) precluded the use of a system corresponding directly to the Name Store for secondary operands. Instead a first-in/first-out *Function Queue* is used; as each address is generated by Dr, the corresponding function, together with control information, is entered into the Queue. A function leaves the Queue when the word containing its operand is received from store. Since the store accessing system is itself a form of pipeline, the effective access time is reduced by a factor corresponding to the number of positions in the Queue.

In a synchronous system, no extra operand buffering would be required, but because the MU5 Processor operates asynchronously, it is essential that as many buffers as Queue positions be available to receive the returning store words. Thus the Operand Buffer System contains eight 128-bit buffers, for which the corresponding virtual addresses are held in an associative store. Fetching a 128-bit store word whenever an operand is accessed, and retaining each of these words in associatively addressed buffers means that many operands are automatically pre-fetched, and the corresponding store requests avoided.

An additional problem arises as a result of page fault interrupts; a significant fraction of the access time for secondary operands is taken up with generating a virtual address and obtaining either a real address or a page fault interrupt. When such an interrupt arises, the Control Register at the end of the Primary Operand Unit will have been incremented on beyond the address of the corresponding instruction and that instruction will therefore have become irrecoverable as far as the program is concerned. Thus in order to be able to preserve this and subsequent instructions and re-execute them after the interrupt has been serviced, the Queue is designed to contain all

functions for which store requests are outstanding. In the event of a page fault, the Page Registers can then be manipulated by the Operating System using orders which bypass the Queue or, if a process change is required, the whole of the Queue and its associated buffer registers can be preserved (for subsequent restoration) in the store, thereby unblocking the Queue and allowing other processes to be run.

8.2 String operations in MU5

The orders provided in MU5 for the string processing functions which occur in languages such as COBOL and PL/1 may be divided into two groups: string-to-string and byte-to-string. The string-to-string orders operate on two fields, or strings, each described by a descriptor. The descriptor of the destination string is held in the DR register while that for the source string is held in XDR. As the operation of the instruction proceeds, the descriptors in DR and XDR move along the strings. No visible register is used by the strings themselves. The operand of the order is an 8-bit mask which determines what bits within each byte are to be operated on, together with an 8-bit filler and in some cases four ***function*** digits used as described below. Provision is made in the hardware for these orders to be interrupted (section 8.2.2)

BMVE Use the filler byte as source and copy to all bytes of the destination string.

BMVB Copy the filler byte to the first byte of the destination string.

SMVB Move one byte from the source to the destination string, or use the filler byte if the source is exhausted.

SMVF Move the whole source string to the destination string followed by filler bytes if the source is shorter than the destination.

SCMP Compare the source and destination strings byte by byte ending when inequality is found, or the destination string is exhausted.

SLGC Logically combine the source and destination strings into the destination. The form of combination (logical OR, for example) is selected by the ***function*** bits in the operand.

An example of the use of these orders is in the implementation of the MOVE verb in COBOL. Suppose that two fields C and D are specified

```
02  C  PIC  X(7)
02  D  PIC  X(7)
```

In MU5 descriptors would be created at compile time for C and D, each describing a 7-byte field starting at the required byte address. The COBOL sentence

```
MOVE C TO D
```

would then become in MU5 instructions

```
XDR   =  C
DR    =  D
SMVF
```

If D is specified as

```
02  D  PIC  X(9)
```

then the final two bytes of D must be spaces. The filler option of SMVF allows this to be carried out automatically. The sequence becomes

```
XDR   =  C
DR    =  D
SMVF     space
```

If the source field is too long, then the SMVF order terminates when the destination field is full, and an optional interrupt enables this condition to be monitored if required.

An extension of the above technique to vector operations of a mathematical form would be fairly straightforward. For example, a vector add of the form

```
F:= F + E
```

would become

```
DR    =  F
XDR   =  E
VECTOR ADD
```

However, as we have already seen, operations of this type are programmed out into loops in MU5. Thus the idea of including vector orders in the MU5 instruction set was dropped in favour of a pipeline approach which would lead to execution rates for such loops approaching the peak rate at which the store could deliver the vector operands.

8.2.1 Operation of the store-to-store orders

The store-to-store orders are controlled by logic in the Dr unit which, by incrementing the origin field(s) of the descriptor(s) at each access, generates the appropriate series of addresses, and by decrementing the length field(s), until zero is reached, determines the point at which to terminate the order. The descriptors themselves are loaded into DR and XDR by preceding orders, while the filler/mask, which is the operand associated with the actual store-to-store order, is automatically transferred to Dop by a preliminary access during the execution of the store-to-store order itself. A *Dop bit* accompanying this access indicates its special nature and causes the operand to be loaded into the filler/mask register within the store-to-store processing logic in Dop (figure 8.3). This logic allows source bytes to be copied to the destination, source and destination bytes to be logically combined or compared, and so on, in accordance with the requirements of the various store-to-store orders.

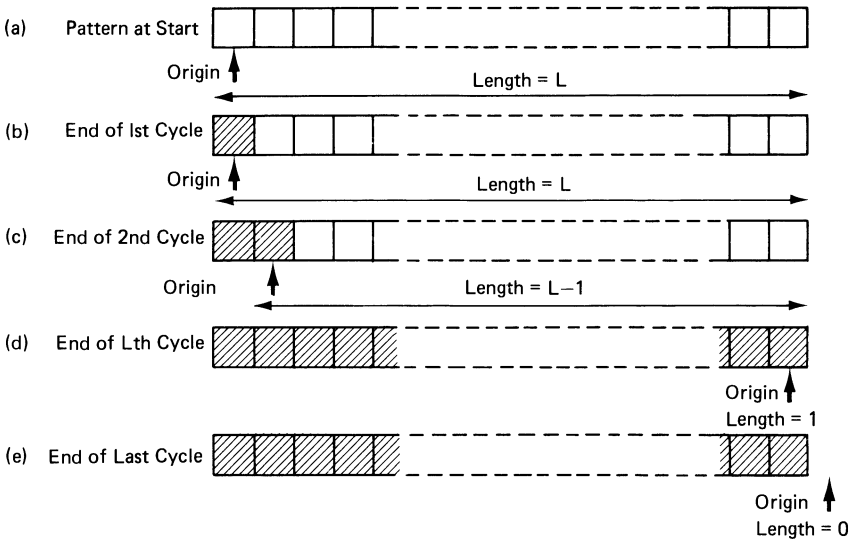

Figure 8.6 String patterns in a byte-string order

During the execution of these orders, the main data path is used to extract bytes from the source and destination store words, and to return them to the destination store words as appropriate. Thus, for the SMVB and SMVF orders, for example, Dop performs *load* operations to extract bytes from the source string, and for the SMVB, SMVF, BMVB and BMVE orders, performs *store* operations to return bytes to the destination. For the

logic orders, SLGC and BLGC, bytes are also extracted from the destination string before being logically manipulated and returned to the destination string. For the comparison orders, only *load* operations are required since the destination string is not updated.

The sequence of operations in a store-to-store order is best illustrated by reference to the patterns of bytes in the strings as the order proceeds. Figure 8.6(a) shows the pattern of bytes in the destination string at the start of a BMVE order. The origin of the descriptor initially points to the first byte of the string, and the length is set to the total number of bytes L. At the end of the first cycle, the position is as shown in (b). The (masked) filler has been moved into the first byte, but the origin and length are unchanged; the only route to the OBS address highway is through the Dr adder, and since the first address to be sent must be that contained in the descriptor origin, the updated value is not available for loading into the descriptor. For the second and subsequent cycles, the address sent to OBS is the origin plus 1. The origin is then updated with this new value, and finally the length field is decremented by 1. The string patterns are as shown in (c) and (d).

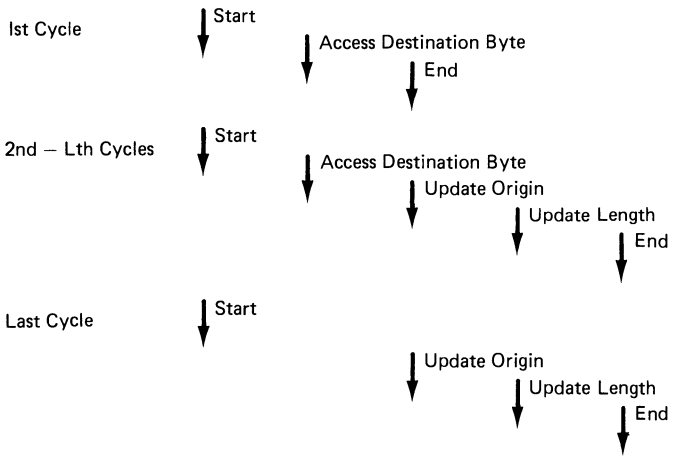

Figure 8.7 Event sequence in byte-string order cycles

The end of the move sequence is reached when the length becomes equal to 1. This indicates that the last byte has been filled, as shown in (d). Since the descriptor must point after the end of the string when the order terminates, however, an extra cycle is needed (without a store access) to increment the origin by 1 and decrement the length by 1. The final string

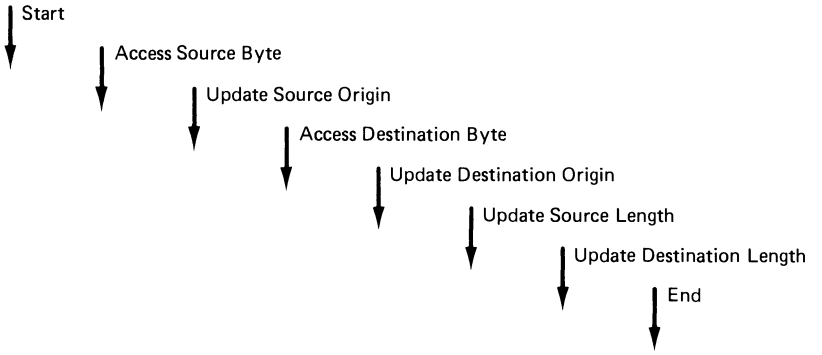

Figure 8.8 Event sequence in a string-string normal cycle

pattern is shown in (e) and figure 8.7 shows the event sequences in the various cycles.

The event sequence for a *normal* cycle of a string-string order is shown in figure 8.8. The source byte address is generated first and sent to OBS, and then the source origin is updated. The destination byte is then accessed in the same way and the destination origin is updated. Finally, the two length fields are decremented by 1. The order is normally terminated when the destination length reaches zero, or when equality is found in SCMP. If the source length reaches zero before the destination length, the action taken depends on the order involved. SLGC is terminated immediately, and an interrupt is generated, whereas SMVF and SCMP are converted to the corresponding byte orders. This is illustrated in the complete event sequences for SMVF (figure 8.9).

In the first cycle, the origin fields are sent out directly as addresses and the two descriptors are left unchanged at the end of the cycle. *Normal* cycles are then executed until the source string runs out. For subsequent cycles only the destination address is generated, and the function code associated with these accesses is changed to BMVE, so that the rest of the destination string is filled with the filler byte held in Dop. The first of the BMVE cycles is also a *last* cycle for the source descriptor, in which the origin and length fields are updated so that the descriptor finally points after the end of the string and has zero length.

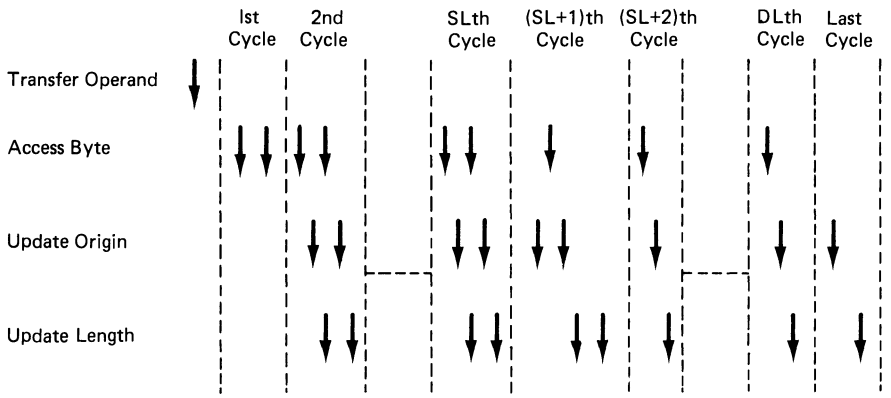

Figure 8.9 Complete event sequence for a string-string order

8.2.2 Special store management problems

Since the addresses generated by the store-to-store orders are virtual addresses, the execution of a store-to-store order will generally involve the crossing of one or more page boundaries, and hence require access to a page of data not currently available. A system which attempted to overcome this problem by ensuring the availability of all the necessary pages of data before the start of the order would involve different hardware constraints on the length of strings which could be handled in different implementations of the instruction set, and was therefore not considered. The alternative approach adopted in MU5 is to allow the store-to-store orders to be interruptable, so that only parts of each string need be resident in the Local Store at any one time. Thus Dr samples the interrupt signal at the start of each cycle, and when an interrupt occurs it converts the new cycle to a *last* cycle in order to terminate the order normally with the origin(s) pointing to the byte(s) immediately following those for which requests have already been sent to OBS. Requests trapped in the OBS Queue as a result of the interrupt are dealt with as for any page fault interrupt (section 8.1.4) and are automatically re-issued after the interrupt has been serviced. When an interrupted store-to-store order is re-started, the first cycle generates the address(es) of the next byte(s) in sequence. Thus it is possible to interrupt execution of a store-to-store order at a partially completed stage and re-start it from the same point when a new page of data has been made available.

9 *The CDC Series*

In chapters 6 and 7 we followed the development of the CDC 6600 design as it evolved first into the 7600 and then into the CRAY-1. In this chapter we follow the progress of a separate line of development started by CDC in 1965 in response to a requirement of the Lawrence Livermore Laboratory for a vector processor capable of executing 100 MFLOPS. The machine which resulted was the STAR-100 [HT72]. A great deal of controversy raged about this machine in its early years, and many of the essential design issues and performance goals have been obscured [Lin78]. Despite the many difficulties which arose in the course of the STAR-100 programme, CDC remained convinced that the underlying architectural concepts of the STAR-100 were sound, and went on to produce a second version, the STAR-100A, which appeared commercially as the CYBER 203, and a completely re-engineered version, the STAR-100C, which is now produced commercially as the CYBER 205. In 1983 CDC formed a spin-off company, ETA Systems Inc., with the goal of producing a multiprocessor system (the ETA^{10}), based on the CYBER 205 architecture and having a performance capability of 10 GigaFLOPS.

9.1 The CDC STAR-100

The fundamental premise on which the STAR-100 project was based was the provision of as much immediate access storage as was possible within engineering constraints. In the mid-1960s this meant using ferrite core technology; semiconductor storage technology, while holding out a great deal of promise for the future, was not sufficiently mature for its use in large quantities to be a realistic venture. The limitations on the amount of storage which could be provided using cores were physical space requirements, the physical interconnection problem, and overall reliability. These limitations led to a maximum memory size of one-million 64-bit words using the 2D core memory developed for use as Extended Core Storage in the CDC 6600. These stores had a 280 ns access time and a 1280 ns cycle time[1].

[1]The limitations on store size have not gone away with the improvements in storage density and access time provided by semiconductor technology, because logic densities and processing speeds have also improved and have led to higher user expectations; transmission delays still account for roughly the same percentage of the total system delay in the CYBER 205 as they did in the original STAR-100.

Coupled with the provision of large amounts of storage in the STAR-100 was a high bandwidth for the transfer of data between the store and the central processing unit. Because the access time to an individual store word was relatively long, however, it was clear from the outset that it would only be possible to use this bandwidth effectively if operand pre-fetching could be used in the same way as instruction pre-fetching is used in conventional machines. Instruction pre-fetching relies on the sequential nature of instruction execution; operand pre-fetching is similarly only effective if the processor can generate long sequences of addresses, a situation which arises quite naturally in vector processing. Thus the STAR-100 project was naturally geared to the provision of computing power for highly vectorisable problems, and vectors with up to 65,536 elements could be processed by a single vector instruction.

Memory bandwidth is obtained in the STAR-100 by a combination of eight-way interleaving of memory banks with wide store words (512 bits accessed per store cycle) and the pipelined transmission of each of these *superwords* to the processor in four sequential groups of 128 bits. In order to handle this data rate in the processor, however, the designers were faced with choices ranging, in the extreme, between using a multiplicity of 6600-like arithmetic units and a single pipelined arithmetic unit capable of executing 32-bit operations at a rate of one every 10 ns. In practice the solution chosen for the STAR-100 uses two pipelined arithmetic units as shown in figure 9.1, each of which can act as a single 64-bit or a twin 32-bit unit. Floating-point Pipe 1 performs floating-point operations such as add, subtract, multiply and compare, and also fixed-point address operations. Floating-point Pipe 2 is similar, except that it includes a divide unit and a multi-purpose unit which is used for special operations such as square root. For operations such as floating-point addition the two pipes act in parallel and can therefore accept the full 128 bits available from store in each clock period. For 32-bit operations this leads to an execution rate of four operations per 40 ns clock period, equivalent to 100 MFLOPS.

The store is made up of eight sections, each containing four banks of 16K 64-bit words. 512 bits are read from a bank in one store cycle, and the outputs of the four banks in any one section are multiplexed at 40 ns intervals on to a 128-bit data trunk. The data trunks from each section are then multiplexed together in the Store Access Control Unit which provides a stream of operand store words to the processor. The outputs from the Store Access Control Unit cannot be fed directly into the arithmetic units, however, since in practice vectors cannot necessarily be expected to start on 512-bit word boundaries and the two operands making up a pair of source operands are normally taken from separate superwords. Thus the Stream Unit contains input shift networks which align the input operands correctly

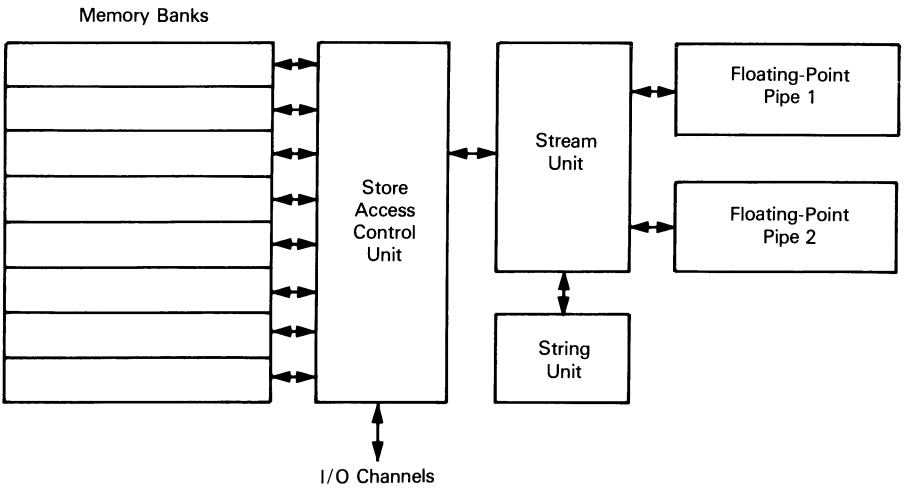

Figure 9.1 STAR-100 processor organisation

and a *front-end pre-count* mechanism which discards any unwanted elements from the first superword received at the start of a vector order[2].

The Stream Unit must also align the result vector correctly, using an output shift network, and at the start and end of a result vector must ensure that only the appropriate parts of the store word are overwritten. This problem also occurs in exactly the same way in MU5 and is dealt with by the Descriptor Operand Processing Unit (section 8.1.2) which has facilities for updating store words down to the bit level. In the STAR-100 such facilities exist in the String Unit, but for arithmetic operations the smallest unit is a 32-bit word, and selective updating of the 512-bit superword in a given memory bank is carried out, on a 32-bit word basis, at the store interface. Thus the Stream Unit sends a four-bit *write enable* control field with each 128-bit result, as illustrated in figure 9.2 which shows the effect for eight 32-bit results positioned arbitrarily within a 512-bit superword.

Any read or write operation on a core store involves a read-write cycle and in the STAR-100 a word read out from a complete memory bank is held in a 512-bit holding register before being written back. The write enables control the updating of the contents of this register by the results sent from the processor, giving the effect of a read-modify-write operation on the selected superword in the memory bank. Once the use of these

[2]The parallels between operand accessing in the STAR-100 and techniques used elsewhere in instruction accessing can also be seen in this mechanism. A similar system is used in the MU5 Instruction Buffer Unit to allow control transfers to jump to any 16-bit parcel within a 128-bit store word.

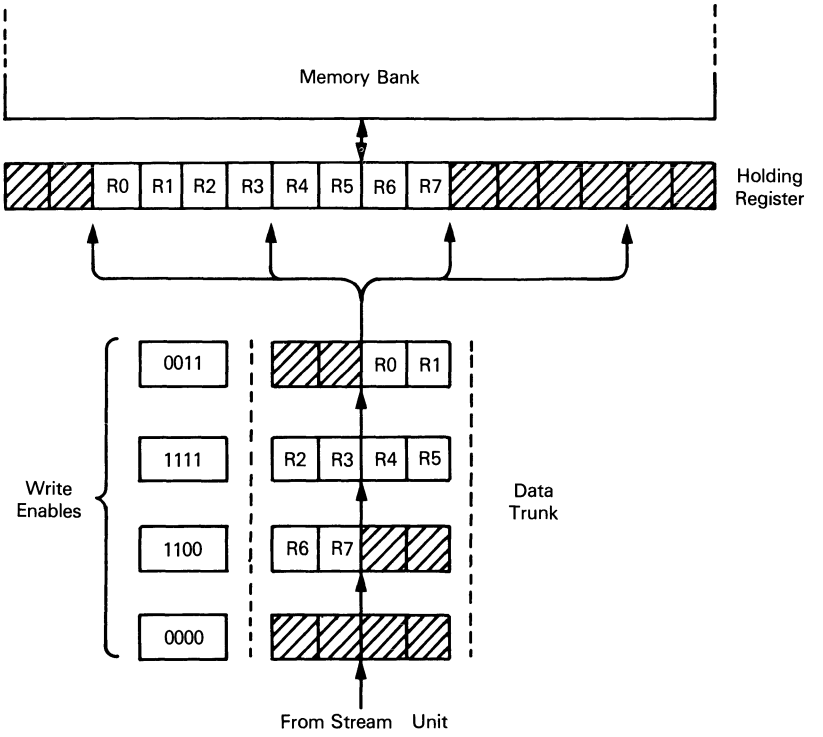

Figure 9.2 Result operand alignment in a STAR-100 superword

write enables had been established as part of the design of the STAR-100, the concept of a control vector was an obvious extension. Control vectors became an important feature of the STAR-100 design and are used extensively, not only to define write enables, but also to control vector restructuring operations such as MASK, COMPRESS, and MERGE, which derive directly from the programming language APL [3]. These operations will be discussed in more detail in section 9.2; there is still another problem to be discussed in connection with the writing of results back to store.

The STAR-100 uses a virtual memory organisation based on that used in the Atlas computer, and in executing a long vector operation inevitably encounters page faults in exactly the same way as the store-to-store orders in MU5. The solution to this problem is relatively simple in the MU5 case because destination addresses are generated before any action is taken to produce the corresponding results (section 8.2.2). Updating of the store

[3] Iverson's book on APL [Ive62], which describes these operations and the notion of a control vector, was a significant influence on the designers of the STAR-100.

words takes place within the processor on a byte-by-byte basis, so that whole words are fetched from store before any new values can be written into them. In the STAR-100 vector orders are normally three-address, and destination addresses are only generated after the results have been produced. Thus in order to achieve the full vector rate, while still responding to page fault interrupts, a one-page look ahead is required on the output stream to ensure that operands progressing through the long processor pipeline can be returned to store before the instruction is interrupted.

9.2 The CDC CYBER 205

The STAR-100 was criticised on a number of grounds by users who wished to apply it to more general computing problems than those for which it was designed. The grounds for criticism were mainly the long vector start-up time and poor performance on scalar arithmetic, both of which were inevitable consequences of the design. These problems are largely overcome in the CYBER 205 [Lin82] by the use of a very much faster (80 ns access time) semiconductor memory and by the inclusion of a high performance scalar unit. The overall performance of the CYBER 205 is further enhanced by its implementation in specially developed ECL LSI Uncommitted Logic Array technology, allowing a reduction of the clock period from the 40 ns used in the STAR-100 to 20 ns.

Figure 9.3 shows the overall design of the CYBER 205 computer system [CDC81], the major components of which are the central processing unit (containing the vector processor, scalar processor and eight or 16 I/O ports), the central memory, and the maintenance control unit (linked to a standard I/O port via a system channel adaptor). A basic system has one-million words of central memory, expandable to two or four-million words (and with a capability to expand to eight-million words as memory technology progresses), with each million words being connected to the memory interface via a 512-bit data highway. Within the vector processor there can be one, two or four floating-point pipelines.

Instruction execution is controlled by a number of independent high-speed microcode memories, each containing 256, 512 or 1024 words of microcode with between 48 and 120 bits per word. During instruction execution each memory operates as a read-only memory, having been loaded at start-up time by the maintenance control unit.

The scalar processor receives instructions from central memory, decodes them, and (according to type) either executes them itself or sends them to the vector processor. A typical vector instruction has the format shown in figure 9.4(a), and causes the two source vectors defined by A and B to be combined and copied to a result vector defined by C. Each address field

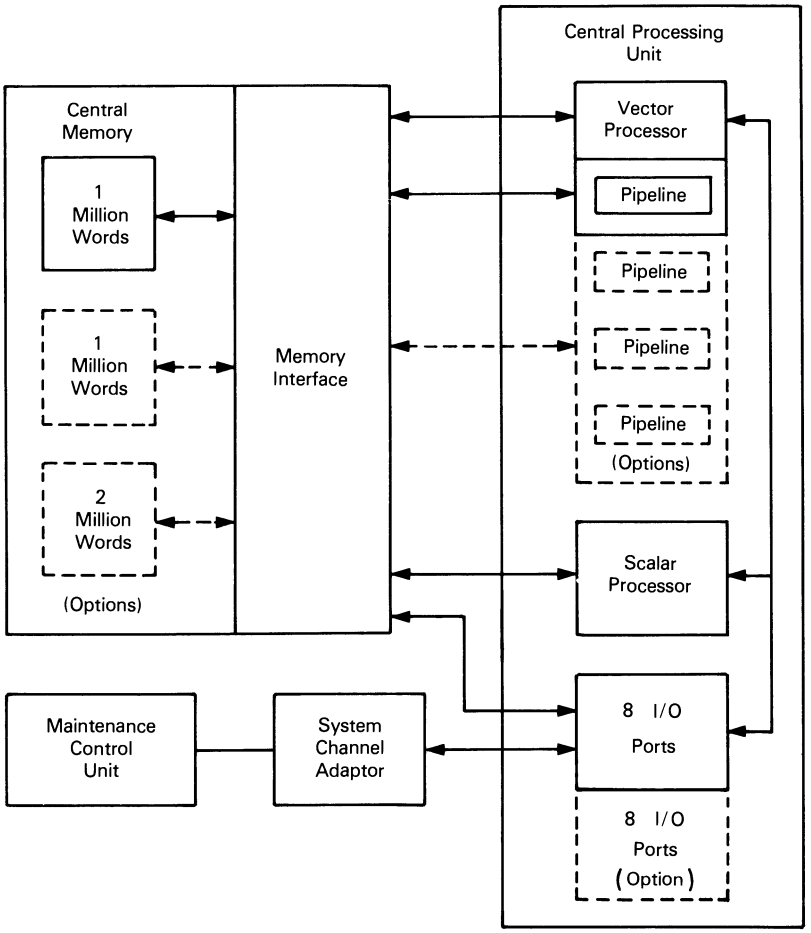

Figure 9.3 The CYBER 205 computer system

within the instruction is an eight-bit designator referring to one of 256 64-bit locations within a Register File in the scalar processor.

Whenever the scalar processor issues an instruction to the vector processor it also sends, via two 64-bit highways, the contents of the required Register File locations[4]. This information is then used by the vector pro-

[4]The chips used in the construction of the Register File have separate read and write access, and the entire file is duplicated in hardware. Operands are always written to both copies of the Register File simultaneously (to maintain consistency) but may be read separately, so that two source operands can be read, and one result operand returned to the File, in one clock period.

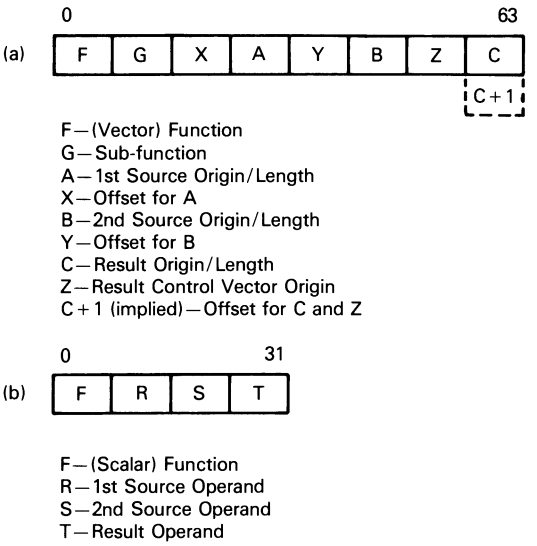

Figure 9.4 Typical CYBER 205 instruction formats

cessor to generate the appropriate sequences of addresses required to access the individual elements of the source, control and result vectors. The address of the first element of the source vector defined by A, for example, is formed by adding the off-set obtained from the register defined by X to the 48-bit origin address obtained from the register defined by A. Subsequent addresses are formed by incrementing the first address, and the number of elements accessed is determined by the value in the 16-bit length field contained in the same register as the origin.

9.2.1 The scalar processor

The design of the CYBER 205 scalar processor (figure 9.5) is largely derived from that of the CDC 7600. The Instruction Stack is filled from central memory (under control of the Branch/Pre-fetch Unit) two superwords ahead of the instruction currently being executed, and can contain up to eight associatively addressed superwords. Instructions are taken from the Stack by the Instruction Issue unit, which decodes each instruction in turn and sends it to the appropriate functional unit for execution. Vector and string instructions are sent to the vector processor (section 9.2.3) while scalar instructions are sent to one of five arithmetic sub-units within the Scalar Floating-point Unit. The Add/Subtract, Multiply, Logical and Single Cycle Units are all pipelined and can accept a new pair of input operands at

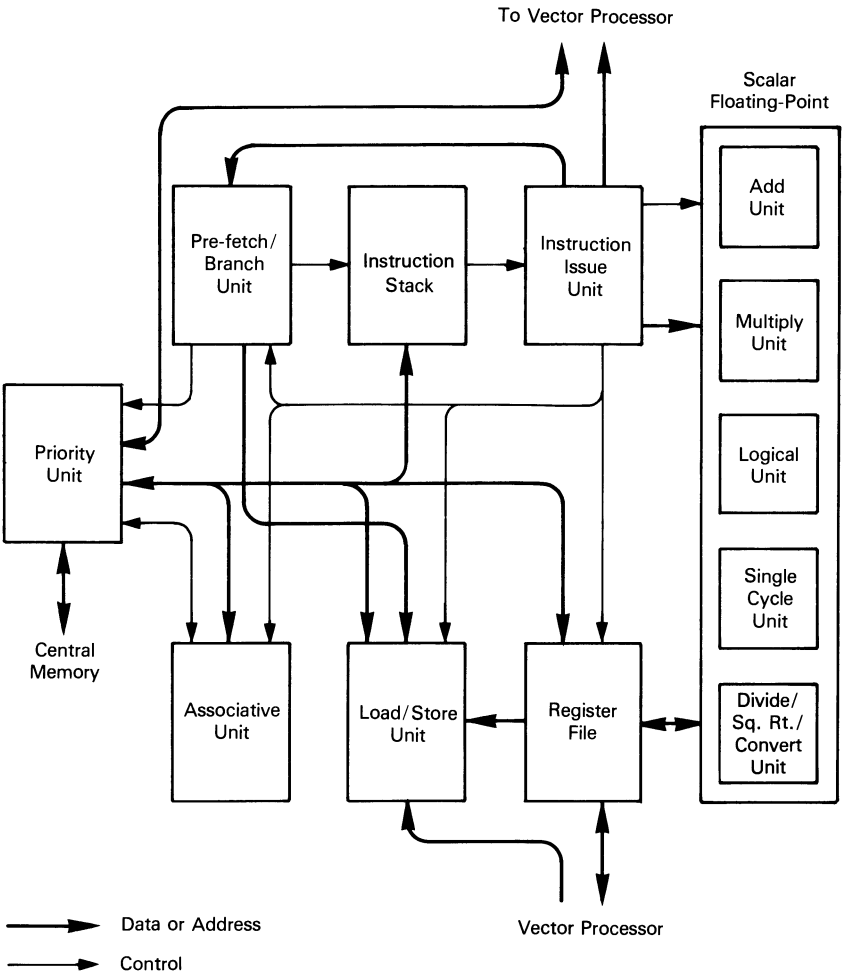

Figure 9.5 The CYBER 205 scalar processor

every clock cycle. Scalar instructions are of the form shown in figure 9.4(b). Operands are taken from locations in the Register File defined by R and S, and the result is returned to a location in the Register File defined by T. Transfers between the Register File and central memory are controlled by instructions executed in the Load/Store Unit. In these instructions R and S typically specify an indexed central memory address and T the location of the operand in the Register File.

The Add/Subtract and Multiply Units each take five clock periods to produce a result and return it to the input of another unit. The Logical Unit

takes three, and the Single Cycle Unit, which is used, for example, to load a literal operand into a Register File location, takes one. The Divide/Square Root/Convert Unit is not pipelined and takes between 21 and 54 clock periods to produce its result, depending on the function. These times are all *shortstop* times. Shortstopping is a technique which allows a result required as a source operand by a succeeding instruction to be returned directly to the input of an arithmetic unit. This is the scalar equivalent of chaining in the CRAY-1, and although similar in some respects to data forwarding in the IBM System/360 Model 91, it is different in that a shortstopped result in the CYBER 205 is sent as an input operand to an arithmetic unit at the same time as being written into the Register File. In the Model 91 a result is not written into a register at all if it is forwarded.

Shortstopping is controlled by the Instruction Issue Unit, which is implemented as a three-stage pipeline and issues one instruction per clock period unless held up by an instruction dependency. These dependencies are similar to those found in the 7600 (section 6.2.1). Thus an instruction may not issue if it specifies as its result operand or as one of its source operands a location in the Register File which is awaiting the result of some other previously issued but as yet uncompleted instruction. Source operand conflicts are resolved in the CYBER 205 by the use of 16 Result Address Registers, which hold the Register File addresses of the result operands of uncompleted instructions. Before an instruction is issued its source operand addresses are checked, simultaneously, against all 16 Result Address Registers, and if a match is found with a valid address, issuing of the instruction is held up until the conflict is resolved.

Shortstopping allows these conflicts to be resolved more quickly in the CYBER 205 than in the 7600, since the shortstop route allows a result to be returned as a source operand before it has been written into the Register File. Only one shortstopped result can appear in any one clock period, since as in the case of the 7600, only one result can be written into the Register File in any one clock period. Thus the issuing of an instruction must be delayed if its result would arrive at the Register File at the same time as the result from a previously issued, but slower, instruction. This type of conflict, and result operand conflicts, are resolved by means of a result position timing chain, similar in principle to the X Register Access Control Logic in the 7600 (section 6.2.1). Before an instruction is issued its result operand address is checked against result addresses in the timing chain, and if a match is found, or if the required Register File access reservation register is occupied, issuing of the instruction is held up until the conflict is resolved.

Vector and scalar operations can be executed in parallel, provided there are no Register File reference conflicts, and provided that no central mem-

ory references are required by scalar instructions during the execution of a previously issued vector instruction. When the Instruction Issue Unit decodes a vector instruction, and the vector processor is not busy, it supplies the vector processor with the function and with the contents of all the Register File locations specified in the instruction. The Instruction Issue Unit reserves any Register File locations which the vector instruction may update during the course of its operation, and once the vector processor has started execution of the vector instruction, the Instruction Issue Unit is free to issue subsequent (scalar) instructions.

9.2.2 Virtual address translation

As in the case of the STAR-100, the CYBER 205 uses a virtual addressing scheme derived from that used on Atlas. Thus addresses generated by the scalar and vector processors are normally logical (virtual) addresses which must be translated into absolute (real) storage addresses before being sent to central memory. This translation is effected by the Associative Unit (figure 9.5) which contains 16 associative address registers. In the Atlas scheme (section 3.2) each associative page register bore a one-to-one correspondence with a real page in the core store. The CYBER 205 Associative Unit is more closely related to the MU5 Store Access Control Unit (section 3.5), however, in that the number of associative registers is considerably smaller than the number of pages in real store, and the associative registers are therefore current page registers containing the most recently used virtual pages. The remainder of the page address translation table (the *space table*) is held in fixed locations in central memory in the CYBER 205 and is searched sequentially under microprogram control if a particular page address is not found in the associative registers (the *page table*).

Entries in the page table registers (AR00 - 15) are held as an ordered list of most recent use. Whenever a virtual address is presented for association and a match occurs, the contents of the register containing the matching address are moved to the top of the list (register AR00) and the contents of all those above the matching register move down one position. If no match is found, the contents of AR15 are copied into a buffer register and the contents of registers AR00 to AR14 move down one position leaving a null entry in AR00. The contents of all the associative registers are then copied into central memory and the contents of the space table are examined in sequence using the associative facilities of the page table registers. Thus the first entry in the space table is read and its location in central memory replaced by the previous contents of AR15, held in the buffer register. After this first word has been examined it is itself copied into the buffer register. If no match occurs this cycle of events is repeated, with the second space table entry being read and replaced in central memory by the first word

(now held in the buffer register), and so on. When a match occurs the page table registers are re-loaded from central memory and the content of the matching address, held in the buffer register, is copied into AR00. Thus the ordering of entries in the page table also extends into the space table.

Since different programs running on the CYBER 205 require differing numbers of pages, a mechanism is required to prevent the hardware searching invalid page or space table entries. An *end-of-table* marker is used for this purpose. If this marker is found in a page table register when there is a page fault, an interrupt is generated immediately and the space table is not searched. If the end-of-table marker is in the space table, however, then it will only be read, and an interrupt generated, after all valid entries in the space table have been examined.

The real addresses generated by the Associative Unit are passed to the Priority Unit (figure 9.5), which co-ordinates central memory requests from different sources. In the event of two requests arriving simultaneously, the one with higher priority is serviced first and the other is delayed. I/O requests, for example, have the highest priority and are always accepted immediately. Requests may also be delayed because of bank busy conflicts in central memory. In the case of vector requests being delayed, the inertia of the address generation pipeline is such that a Re-try Unit is required to buffer unsatisfied requests. When the Re-try Unit is activated, address generation in the vector processor is halted, but as many as three additional requests may arrive before the flow ceases, and the Re-try Unit therefore has buffering capacity for four requests. During the fourth clock period after activation the Re-try Unit re-transmits the buffered requests to the Priority Unit and, if the initial request is accepted, releases the address generation hold-up in the vector processor. The first new request then arrives at the Priority Unit just after the last buffered request is transmitted from the Re-try Unit.

9.2.3 The vector processor

Figure 9.6 shows the overall structure of the CYBER 205 vector processor. The Vector Control Unit receives vector functions sent from the scalar processor, together with the appropriate Register File contents, and controls the setting up and execution of each instruction within the vector processor. During the execution of a typical vector instruction, addresses are generated by the Stream Addressing Pipeline and sent via the Associative Unit in the scalar processor to central memory. The corresponding store words accessed from central memory are received by the Input Stream Unit which aligns the various input operand streams correctly for transmission to the Floating-point Pipeline or String Unit. Results from the Floating-point Pipeline or String Unit flow into the Output Stream Unit, and thence back

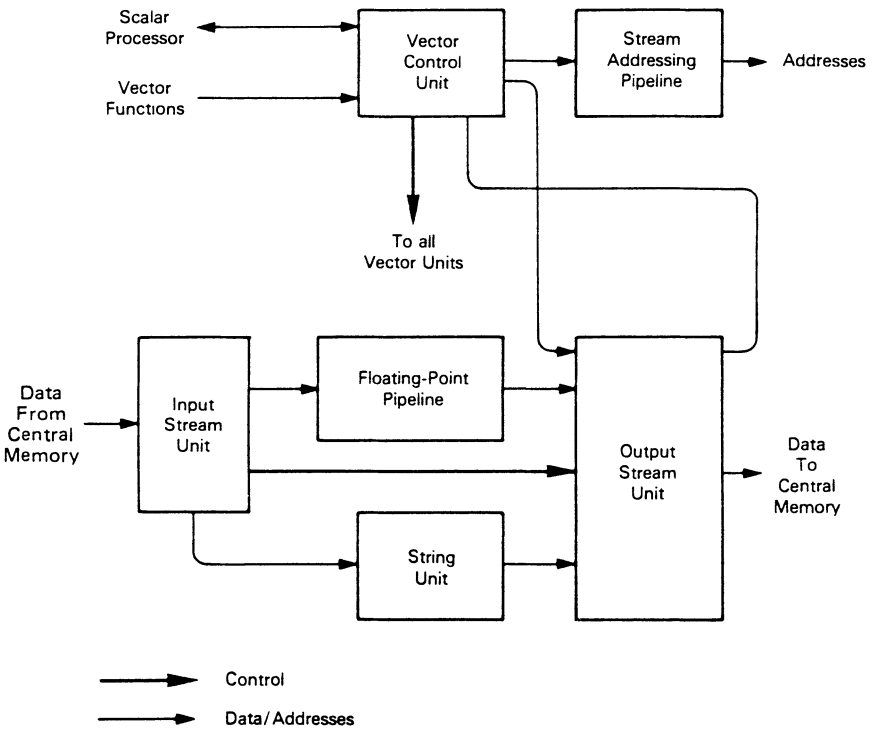

Figure 9.6 The CYBER 205 vector processor

to central memory. The Stream Units act in a similar way to the Stream Unit in the STAR-100 (section 9.1), not only in aligning the input streams correctly, but also in organising proper starting and finishing points for vectors which are not aligned on 512-bit store word boundaries.

Most vector instructions process data using two data input streams, one control vector input stream and one data output stream. These streams are specified by the origin addresses, field lengths and off-sets or indices selected from locations in the Register File by the instruction before it leaves the scalar processor, and sent to the Vector Control Unit when the instruction is issued. During the setting up of a vector instruction within the vector processor, off-set values are added to the corresponding origin addresses and subtracted from the field lengths before being sent to the Stream Addressing Unit. For each data stream the Stream Addressing Unit generates sequential store addresses (normally to 512-bit words) by

incrementing the origin address until the field length, which is decremented correspondingly, reaches zero.

The generation of addresses for each input and output stream is treated as an independent operation within the Stream Addressing Unit. This allows each operation to be stopped and started independently, and allows the Stream Addressing Unit to avoid bank busy conflicts in central memory. Since the banks have an 80 ns cycle time, a request to any one bank causes a four clock-period busy condition, and a subsequent request to the same bank is not allowed to proceed until a further three clock periods have elapsed. In addition the priority of requests must be taken into account; within the vector processor input operand requests have highest priority, vector write requests have second priority, string write requests third and control vector read requests fourth. A request of low priority must therefore be checked against higher priority requests to the same bank up to three clock periods ahead and up to three clock periods behind the clock period in which it will itself access central memory. A timing chain mechanism similar to that used to control entry of results into the Register File (section 9.2.1) is used for this purpose.

The Stream Addressing Unit also controls the flow of data through the Input and Output Stream Units. The Input Stream Unit contains a buffer for each input stream large enough to hold all data requested but not used. Associated with each buffer is a counter in the Stream Addressing Unit which is incremented each time a 128-bit word is requested from central memory, and decremented each time a 128-bit word is removed from the buffer for use. If a counter indicates that the corresponding buffer is full, then further requests for that input stream are held up. For the Output Stream Unit a counter in the Stream Addressing Unit is incremented each time a result is entered into the output buffer, and decremented each time a write request is sent to central memory.

A vector order terminates once all the input data has passed through the Input Stream Unit. For each of the two data input streams a field length register in the Input Stream Unit is loaded at start-up time. These registers are decremented whenever data is sent to a functional unit and, when they reach zero, an *empty* signal is sent to the Vector Control Unit. Processing of the next instruction can then begin, even before the last few operands of the current instruction have been processed and the results written back to central memory. If an interrupt occurs before an instruction terminates (as a result of a page fault, for example) the current values of operand addresses and field lengths are preserved in central memory. Once the interrupt has been serviced, these values are copied back into the vector processor, and the interrupted instruction is re-started.

9.2.4 The vector floating-point pipeline

The vector Floating-point Pipeline (figure 9.7) provides logical and arithmetic operand processing for all vector instructions. It is made up of five pipelined operand processing units, a Data Interchange and associated control logic. The typical CYBER 205 system has a two-pipeline configuration in which two 64-bit or four 32-bit operands presented on each of the A and B inputs can be processed simultaneously. Except in the case of divide and square root operations, these pipelines can accept a new pair of inputs in every clock period. A one-pipeline configuration processes the two halves of the 128-bit words supplied to it sequentially and therefore runs at half the speed of the two-pipeline version. The four-pipeline version is essentially two two-pipeline processors, giving a total data path width of 256 bits. Floating-point numbers use a binary exponent and have the formats shown in figure 9.8, where all exponents and mantissae are represented by 2's complement integers.

In a normal two-pipeline configuration the Add Unit receives operands from the Data Interchange over two 128-bit data highways and returns results to the Data Interchange over a single 128-bit highway. In a straightforward add or subtract operation successive elements of vector B are added to or subtracted from successive elements of vector A, and the results written to successive elements of vector C. Feedback paths within the Add Unit allow other types of add operation to be implemented, however, so that a single result element C may be obtained by summing all the elements of vector A, for example, or a vector C may be produced by repeated addition of a scalar element B to the initial value of scalar element A.

The Multiply Unit is similarly connected to the Data Interchange by two 128-bit input operand highways, and one 128-bit result highway. In a typical multiply operation successive elements of vector A are multiplied by successive elements of vector B and the results written to successive elements of vector C. An internal feedback path allows a single result to be formed by multiplying together all the elements of vector A, and the Multiply Unit also contains the logic for divide and square root operations.

The Shift Unit has one 128-bit data input highway (A) and one 14-bit input highway (B) which supplies a 7-bit shift count for each half of the pipeline. Each 64-bit element of vector A is shifted left or right according to the most significant bit of the corresponding 7-bit element of vector B, with the number of bit position shifts being determined by the six least significant bits of B.

The Logical Unit carries out bit-by-bit logical operations between pairs of A and B elements supplied via 128-bit input highways and returns its results to the Data Interchange via a 128-bit result highway. It also carries out pack and unpack operations on floating-point numbers (similar to those

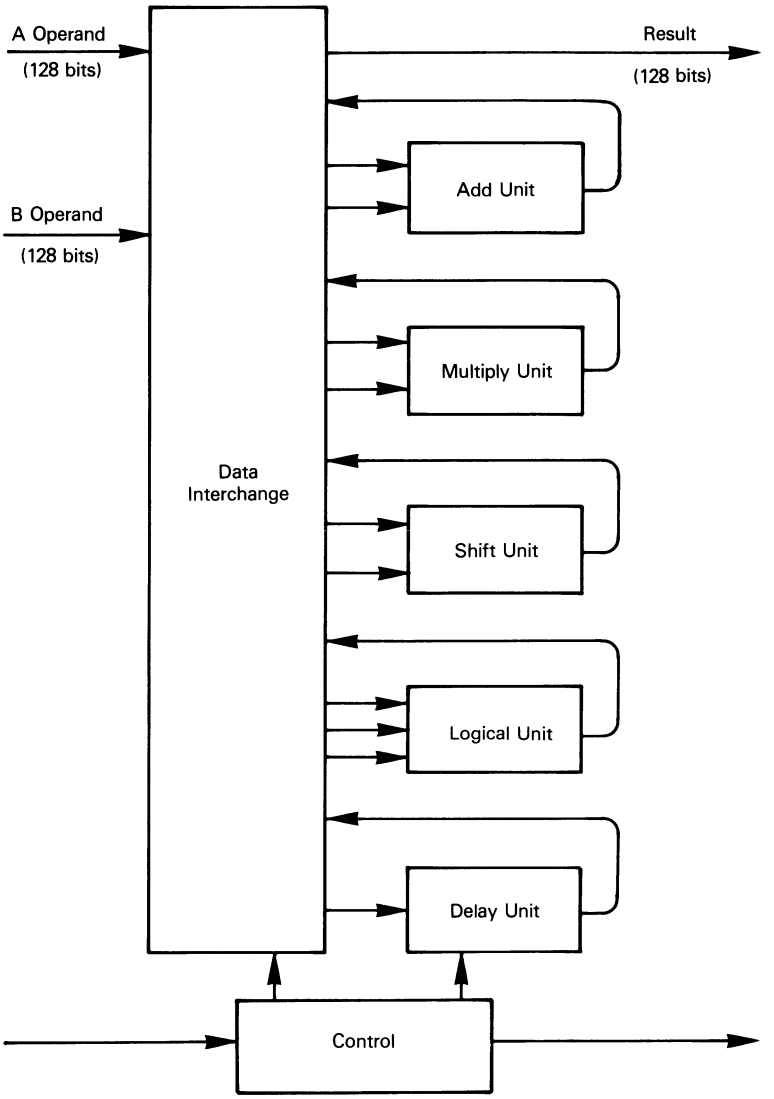

Figure 9.7 The CYBER 205 vector floating-point pipeline

implemented in the CDC 6600 and 7600) and a masked compare instruction for which a third input is required, containing the 128-bit mask. This instruction searches elements of vector A in sequence for a bit-by-bit match with the single element B; bit positions for which the bit in the mask is zero are assumed to match. As each word is examined, the Register File location

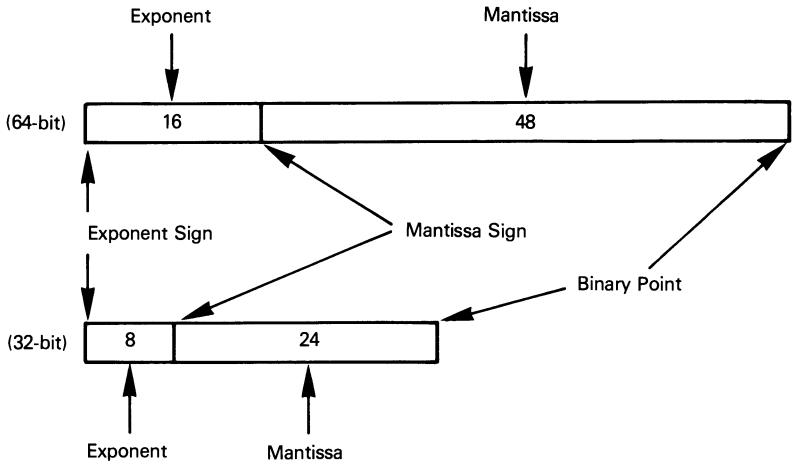

Figure 9.8 CYBER 205 floating-point formats

containing the index of A is updated, so that, if a match is found, the index provides a means of locating the position of the matching element. If no match is found the index is left pointing to the end of the vector. When the instruction terminates a condition code is set to indicate the result.

For simple vector instructions the Data Interchange is configured to connect the input and output highways to the appropriate processing unit. The *Select Link* instruction, however, causes the Data Interchange to be configured such that the succeeding two instructions in the code sequence become chained together. In this case the output of the unit used by the first instruction of the pair is routed to the input of the unit used by the second. Only two vector streams may be used in total, but this does allow commonly occurring triadic operations such as

```
Vector C = Vector A * Constant - Vector B
```

to be implemented in this manner. Not only does this allow the Multiply and Add Units to operate in parallel, but it also avoids the need to write the intermediate result vector into central memory and then read it out again.

Chaining of this sort occurs automatically during the execution of vector macro instructions such as the scalar product instruction, for example, in which pairs of input operands are multiplied together in the Multiply Unit and their results then summed in the Add Unit. This summation also involves the use of the Delay Unit, which contains a 16-word temporary buffer store. Successive 128-bit words sent to the Delay Unit are written into successive locations selected on a cyclic basis by a write counter. These same words are then read out again, and returned to the Data Interchange,

under the control of a read counter. The delay function is implemented by off-setting the read and write counters by the required number of clock cycles of delay.

In the summation of results from the Multiply Unit in the scalar product operation (and similarly in the summation of all the elements of a single vector), the addition is performed by feeding values into input A of the Add Unit, and routing the output of the Add Unit back into input B. This produces a recursive effect similar to that found in the CRAY-1 (section 7.1.4). Because of the delay through the Add Unit pipeline, the zeroth input value, having passed through the Add Unit, is returned to input B in time to be added to the eighth value, the first in time to be added to the ninth, and so on. After a further pass through the Add Unit the sum of the zeroth and eighth values is ready to be added to the sixteenth value, and so on, until the input stream is exhausted and eight partial sums are left circulating in the Add Unit. Adding these partial sums together involves the use of the Delay Unit. The first four partial sums are delayed by four clock periods so that they become aligned with the last four at the input to the Add Unit. The four results obtained from this operation are then further added in pairs using a two-clock period delay, and a final add involving a single clock period delay produces the desired result ready to be written into the Register File.

9.2.5 Sparse vector operations

One of the distinctive features of the CYBER 205 architecture is the provision in hardware of facilities for handling sparse vectors. In many computational problems vectors with large numbers of zero or near-zero elements occur, and the ability to store and process these vectors in sparse form allows considerable savings to be made in both storage space and processing time. The key to the handling of sparsity in the CYBER 205 is the control or *order vector* which, as we saw in section 9.1, was introduced in the STAR-100 to overcome the problem of writing data to only part of a word in store. Thus a sparse vector is made up of two parts, a data vector and an order vector. The order vector contains one bit for each element of the full vector indicating whether or not a (non-zero) value exists for that element. The data vector is a sequence of 32-bit or 64-bit floating-point numbers constituting the non-zero values.

Figure 9.9 shows how a sparse vector might typically be formed. A `COMPARE` instruction is first used in which each element of the initial vector A is compared against the limiting small value below which elements are to be treated as zero. For each element below this limit a zero is generated in the result control vector Z, while for each element above the limit a one is generated in Z. A `COMPRESS` instruction is then used in which the

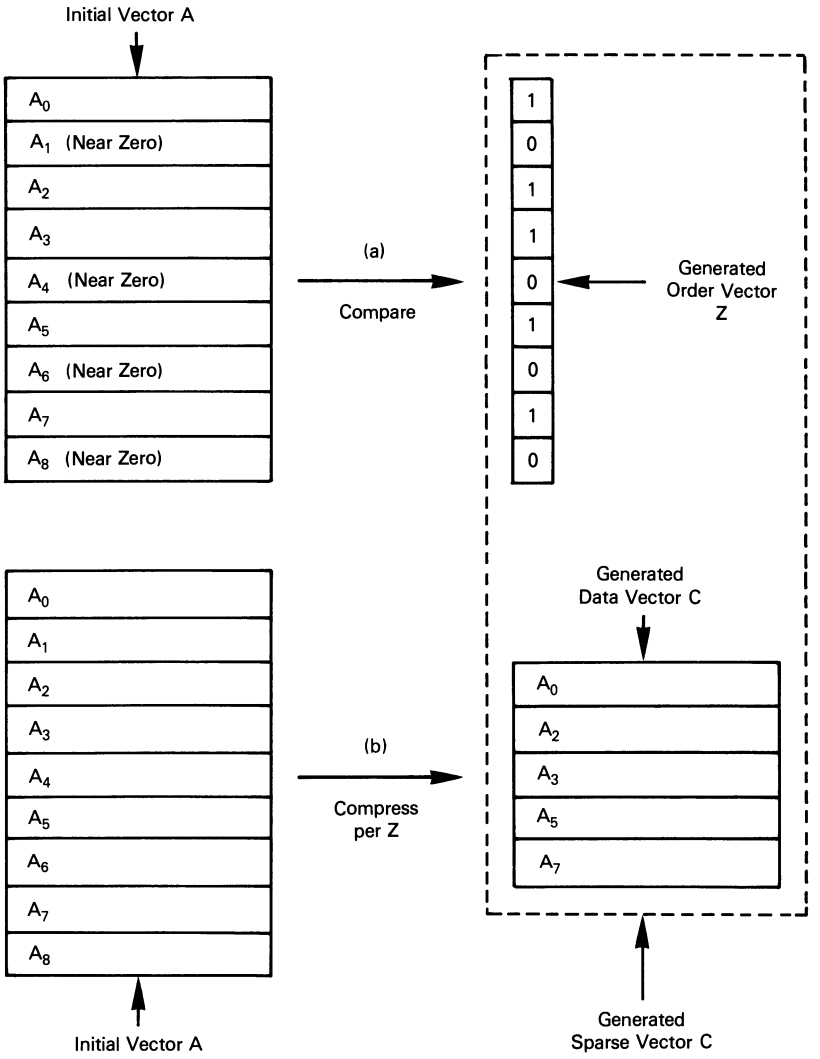

Figure 9.9 Formation of a sparse vector

order vector Z generated by the COMPARE instruction is used to determine whether or not values in the initial vector A are to be copied into the data vector C. Only values for which the corresponding bit in the order vector is a one are copied to vector C, and these are written into sequential store locations. Thus there is no longer any direct correspondence between the position of an element in the vector and its position in store relative to the vector origin, and in sparse vector operations the order vector must be used

to identify the positional significance of each element in the data vector. These order vectors are specified in sparse vector instructions by the X, Y and Z fields. For the source vectors this involves interpreting the contents of the locations in the Register File specified by the X and Y designators (figure 9.4(a)) as origin + field length pairs rather than as off-sets.

When two sparse vectors are combined in arithmetic operations, their order vectors are used to align corresponding elements correctly and to determine the outcome of the operation. When two sparse vectors are combined by multiplication or division, for example, a result element is normally produced, and a one written into the result order vector, only when corresponding elements in the input vector are both non-zero, as indicated by the presence of a one in each of the source order vectors. When two sparse vectors are added or subtracted, a result element is normally produced when there is a non-zero element in corresponding positions in either source vector.

The actual logical combination of source order bit pairs used to produce the bit to be written into the result order vector, and hence to determine whether or not a value is to be written into the result data vector, is determined by bits in the sub-function field of the instruction (G in figure 9.4(a)). For multiplication or division the AND operation is normally used, while for addition or subtraction the OR operation is normal. However, the OR operation may be specified for the former, the AND operation for the latter, and Exclusive OR or Implication for either. In a multiplication instruction where Exclusive OR is specified, for example, a non-zero element in one source vector is implicitly multiplied by 1 when no corresponding non-zero element exists in the other source vector, and no result is produced if neither or both source elements exist.

9.2.6 Indexed list operations

The memory of the CYBER 205 is organised such that within each one-million-word unit there are 16 interleaved 32-bit stacks allowing eight sequentially addressed 64-bit words (512 bits) to be accessed in parallel. Each stack is divided into eight independent banks, each holding 16 Kwords. A new bank read access can be initiated in every clock period, provided it is to a bank which is not already busy, and the banks can share a single stack data path since this path is only required during the last clock period of the four clock-period bank cycle. One of the criticisms levelled at the CYBER 205 is that vectors may only be accessed on the basis of a unit increment between successive elements, so that matrices stored by row, for example, cannot easily be accessed by column. The high memory bandwidth of the CYBER 205 is only obtainable because many adjacent elements can be read out in parallel in each memory cycle, however, and to allow different

accessing patterns in standard vector orders would severely degrade performance. Provision of increments different from unity would also require changes to the instruction format, and alternative general mechanisms for the re-ordering of operands are provided instead. These mechanisms are invoked by Indexed List Operations which allow for the gathering or scattering of periodic or random data within a data structure.

During execution of a scatter operation (`Transmit List` → `Indexed C`), for example, two source vectors (A and B) are normally read from memory and a result vector (C) is written to memory. Source vector A is a list of indices referring to positions in the result vector, while source vector B is the sequential list of 32-bit or 64-bit operands to be written. During each cycle of operation, the index taken from vector A is shifted appropriately and added to the result vector origin to form a memory address. This address is then sent to memory, accompanied by appropriate write control bits and the corresponding operand taken from vector B. The index list may be formed either by geometrical considerations or by data dependent considerations. Sparse vectors with a population density of the order of 0.1 per cent, for example, can be dealt with more efficiently by this mechanism than by the order vector technique described in section 9.2.5, which is suitable for population densities in the range 1 to 10 per cent.

In a gather operation (`Transmit Indexed List` → `C`), source vector A is again a list of indices, but these indices refer to positions in source vector B rather than the result vector. Addresses formed by adding each index from vector A in turn to the origin of vector B are used to access operands from vector B, and these operands are then written to sequential result vector locations.

A number of variations on the general form of the gather and scatter operations may be invoked by appropriate settings of bits in the G field of the instruction (figure 9.4(a)). In particular, one option allows source vector A to be replaced by a fixed increment (a) taken from the Register File location specified by A. In this case source vector addresses in a gather operation are of the form b, b+a, b+2a, b+3a, etc. (where b is the origin address of B), allowing access by column to a matrix stored by row, for example. An alternative variation allows groups of sequential B operands to be selected for each element of vector A.

As with ordinary vector operations (section 9.2.3), the Stream Units must avoid bank busy conflicts in memory during the execution of gather and scatter operations. A request to any one bank causes a four-clock busy condition, and a subsequent request to the same bank is not allowed to proceed until a further three clock periods have elapsed. In scatter operations these conflicts can occur on the write accesses, while in gather operations they can occur on read accesses to source vector B. These conflicts might be

expected to affect performance quite severely in the case of non-sequential accesses. In practice, however, the number of memory banks provided is so large that, except when a specific sequence is presented in which each address has the same value modulo 64, few are found to be busy.

9.2.7 Performance

In the typical two-pipeline CYBER 205 system two 64-bit operands can be transferred into the Floating-point Pipeline on each of its two input highways, and two 64-bit results transferred out, in one 20 ns clock period. This corresponds to an execution rate for 64-bit floating-point numbers of 100 MFLOPS, or 200 MFLOPS for 32-bit numbers. For linked triadic operations, or during the execution of the scalar product operation, this rate is doubled to 200 MFLOPS for 64-bit numbers or 400 MFLOPS for 32-bit numbers. The memory bandwidth required to support this execution rate is six 64-bit words per 20 ns interval, or 300 Mwords/s. Since the data highway between each one-million words of central memory and the memory interface is 512 bits wide, and a read or write transfer can occur every 20 ns, memory bandwidth is actually 400 Mwords/s, and a one-million word memory configuration can therefore support a two-pipeline system without difficulty. For the four-pipeline system at least two million words of central memory are required, with each one million words being connected to the memory interface via its own data highway. In this case the maximum execution rate is 800 MFLOPS for 32-bit linked triadic or scalar product operations.

Where accessing patterns involving non-unary increments are required, performance is reduced because the vector arithmetic instruction must be preceded by a gather instruction or succeeded by a scatter instruction. These instructions themselves run at less than the full arithmetic rate and have been measured to proceed at an average rate of 40 million operations per second. This performance is an average of many operations running at a rate of 50 million per second with occasional periods of 12.5 million per second.

Perhaps a more serious criticism of the CYBER 205 from the point of view of performance on general problems is the long vector start-up time. Thus whereas an execution rate of 100 MFLOPS for 64-bit numbers can in principle be obtained with a two-pipeline configuration, this can only be achieved in practice when very long vectors are used. As the length of the vectors being used becomes shorter, the start-up time has a progressively more serious effect, and Hockney and Jesshope [HJ81] have used the vector length at which performance is halved as a measure of vector efficiency Equations for this and other parameters of vector processor performance are derived in chapter 10. For the CYBER 205 the nominal start-up time

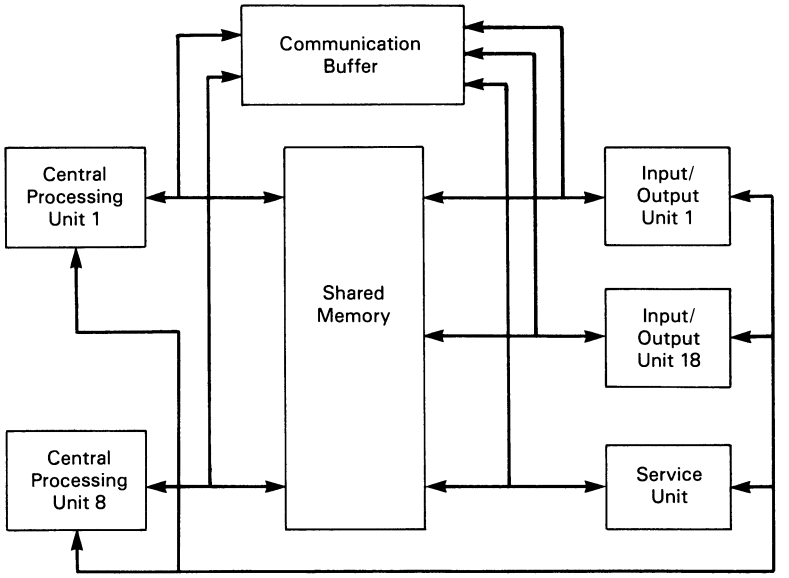

Figure 9.10 Organisation of an ETA^{10} System

is 1 μs, and for a nominal result rate of 100 MFLOPS this length is clearly 100. For the CRAY-1 Hockney and Jesshope quote values in the range 10 to 20, though even lower values are possible. However, the CYBER 205 was intended to be used for problems involving long vectors, and as we observed in section 9.2.3, part of the start-up time can be eliminated for successive vector operations since the processing of one instruction can often begin before the last few elements of the previous instruction have been fully processed. Any attempt at a generalised comparison between the CYBER 205 and the CRAY-1 is largely irrelevant since the performance of each is critically dependent both on the application being run and on the way it is mapped on to the hardware.

9.3 The ETA^{10}

As in the case of the CRAY-1, the evolution of the CYBER 205 into a system with significantly greater performance has involved its development into a multiprocessor system. The result is the ETA^{10} system; an ETA^{10} system (figure 9.10) has between two and eight Central Processing Units (CPUs), each of which has the same functional capability as a CYBER 205. Each CPU has its own 4 Mword Central Processor Memory (CPM) and is connected to both a Shared Memory (of between 64 and 256 Mwords) and 1 Mword Communication Buffer, the latter being used to hold small

amounts of shared data and synchronisation locks. Additional instructions are provided to access these shared memories. The shared memories are also connected to between two and eighteen Input/Output units which provide connections to, and control of, peripheral devices and network interfaces. The Service Unit is connected to all parts of the system and provides an operator's console and maintenance connections.

Each processor has the same 48-bit virtual addressing capability as a CYBER 205, but in the ETA^{10} this virtual address space maps on to a four-level real memory hierarchy as shown in figure 9.11. The Register File is managed by the compilers which use local and store instructions to transfer data between the registers and the CPM, as in the CYBER 205. Mapping of virtual addresses on to the CPM is also carried out using hardware and software support similar to that used on the CYBER 205, but here a demanded page which is not in the CPM may either be in the Shared Memory or on a disc, and is transferred from whichever is appropriate. A page which must be swapped out of a CPM is transferred to the Shared Memory, which contains buffers for CPM paging files and shared data files that are being read and written by processes running on the processors. When a page resident in Shared Memory is to be replaced by other data, that page is written to the processor's paging file on disc. For a single processor the hierarchy is thus organised in a manner very similar to one we have already encountered, in MU5 (section 3.5).

The multi-processor capability of the ETA^{10} can be used to provide high system throughput on sets of independent jobs running concurrently or can be invoked by calls to a multi-tasking library to provide increased performance on a single job. This library provides several sets of calls, corresponding to different models of parallel computation. Further discussion

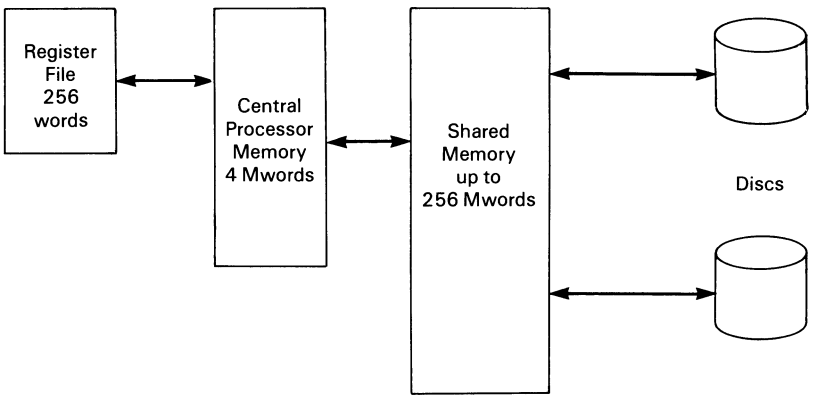

Figure 9.11 ETA^{10} memory organisation

of parallelism can be found in Volume II of this book.

9.3.1 Technology

The CYBER 205 is built in ECL gate array technology and dissipates over 100 kW of power. Eight times this amount of power would be prohibitive, and the alternative technology used in the ETA^{10} is high density CMOS. Each IC contains up to 20,000 circuits, giving reduced inter-gate transmission delays and allowing a complete processor to be built on a single printed circuit board containing about 250 ICs. The power dissipation of each of these processors is around 600 watts. Like the CRAY-2 (section 7.3), the ETA^{10} uses liquid immersion cooling, but here the liquid is nitrogen, at 77 K. Cooling CMOS to this temperature has the advantageous effect of doubling its switching speed relative to room temperature, another factor contributing to the design clock speed of 5 ns. In practice the best clock speed achieved so far has been 10 ns, but ETA Systems plan to produce machines with clock speeds of 7 ns, 14 ns and 21 ns and thus to offer a range of price-performance alternatives.

10 *Performance of Vector Machines*

In the previous two chapters the architectures of two distinct families of vector processor were described. One, the CDC CYBER 205 uses operands stored in memory, whereas the other, the CRAY family, uses operands held in vector registers. Knowledge about such aspects of their design, together with knowledge about the clock periods of these machines, provides a limited picture of their performance potential. For example, before spending several millions of pounds (or dollars) on a high performance vector processor it is worth knowing just how well the machine is likely to perform on the types of problem for which it is intended, rather than knowing only the peak performance of the machine. The performance of all vector computers is the result of a combination of advanced architecture and advanced technology, both of which play an important rôle. Thus, when the performance of such machines is analysed it is useful to separate the architectural measurements from the technological measurements since this permits a comparison of the *architectural quality* of machines constructed from different technologies. This chapter therefore considers the ways in which the performance potential of vector processors can be quantified both from a technological and an architectural viewpoint, and begins by examining the raw hardware performance which derives from the use of pipelined vector instructions.

10.1 Hardware models

Consider again the equation for the time required to evaluate a function on n pairs of operands in a k-stage pipeline (equation 4.1). The time for such a pipelined operation is $T_k = \tau(k + n - 1)$, and hence, the number of clock periods required to perform this pipelined operation is given by C_k

$$C_k = \frac{T_k}{\tau} = k + (n - 1) = \text{startup length} + (\text{vectorlength} - 1)$$

Since the startup time in a vector processor is the time taken from the *issue* of a vector instruction to the production of the first result, it is determined not only by the pipeline length (which we denote l), but also by the latency associated with *decoding* the instruction and *assembling* the first pair of input operands. For example in the CYBER 205, where vector instructions are 64-bits long and reading the first pair of operands requires a full memory cycle, startup times are particularly lengthy, varying between 50 and 125

clock periods. Conversely in the CRAY-1, where vector instructions are just 16-bits long and all operands are held in fast registers close to the vector pipelines, startup times can be as low as two clock periods for chained instructions. The total startup time is therefore defined as $s + l$, where s is the startup overhead (instruction decode and operand assembly) and l is the pipeline length.

10.1.1 Efficiency of vector operations

From the standpoint of vector processing it is instructive to characterise the efficiency of vector instructions in terms of the time required to set up an operation and the peak processing rate which then follows. The peak processing rate is determined solely by the pipeline clock period (assuming operands can be supplied at the required rate), whereas the startup time for an instruction depends on a variety of architectural features. From our definition of the startup time above, and following the notation adopted by Hockney and Jesshope [HJ81], the time to execute a single vector instruction, t_v, is hence

$$t_v = \tau\left[s + l + (n-1)\right] \tag{10.1}$$

This results in an *average* rate of processing, over the duration of the vector instruction, of r_v operations per second

$$r_v = \frac{n}{\tau\left[s + l + (n-1)\right]} = \frac{r_{v_\infty} n}{n + (s + l - 1)} \tag{10.2}$$

where r_{v_∞} is equal to τ^{-1}, which is the steady-state throughput of the pipeline, or equivalently the asymptotic limit of r_v that is approached when a vector of infinite length is processed. This leads to a similar equation for the efficiency of vector processing to equation 4.3 in chapter 4, and hence the vector processing efficiency denoted by η_v is defined as

$$\eta_v = \frac{r_v}{r_{v_\infty}} = \frac{n}{(s + l - 1) + n} \tag{10.3}$$

Figure 10.1 shows how the vector efficiency η_v varies with the vector length n. A characteristic reference point on this curve, which has been suggested by Hockney [Hoc77], is the value of n at which $\eta_v = 0.5$. This value of n is known as $n_{1/2}$ since it is the value of n at which exactly half peak vector performance is achieved. A value for $n_{1/2}$ can be derived trivially from equation 10.3, thus

$$n_{\frac{1}{2}} = s + l - 1 \tag{10.4}$$

It then follows that, given $\alpha = n_{1/2}/n$

$$\eta_v = [1 + \alpha]^{-1}$$

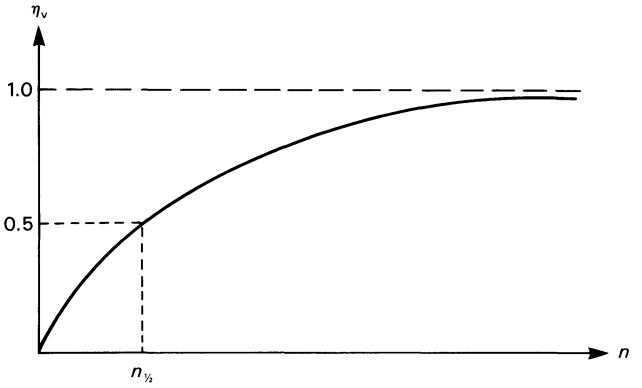

Figure 10.1 Variation of vector efficiency η_v with vector length n

10.2 Measuring vector efficiency

The equations for vector efficiency and vector processing throughput (equations 10.3 and 10.2) are somewhat idealistic since they only provide an indication of how well a single vector operation will perform in isolation. However, real programs consist of *sequences* of vector instructions, and these are occasionally overlapped, for example as a result of chaining (in the CRAY family) or linked-triadic operations (in the CYBER 205). It would be reasonable to expect a reduction in $n_{1/2}$ and an increase in r_{v_∞} under these circumstances. This overlap can make an analytical approach to performance evaluation in terms of $n_{1/2}$ and r_{v_∞} rather complex, and since experimental values for r_{v_∞} and $n_{1/2}$ are not difficult to obtain they are normally used.

Experimental values for r_{v_∞} and $n_{1/2}$ can be derived by timing the execution of a number of vector operations, each of varying length. From these measurements a graph depicting the execution time, t_v, for different values of n can be constructed, and this will look similar to the graph in figure 10.2.

Since we know that $t_v = r_{v_\infty}^{-1}[n + n_{1/2}]$, the graph should be a straight line. Furthermore, the first-order differential of t_v with respect to n is equal to the gradient of the straight-line, and this is also equal to

$$\frac{dt_v}{dn} = r_{v_\infty}^{-1}$$

The gradient of the straight-line graph therefore provides an immediate indication of the peak vector processing rate. In addition, setting t_v to zero in equation 10.1 produces $n = -n_{1/2}$. Therefore, by extrapolating the straight-line representing t_v to where it meets the n-axis, yields a point of intercept at which $n = -n_{1/2}$. In the graph of figure 10.2 the position on the t_v axis at which the performance line crosses it is labelled t_0. This

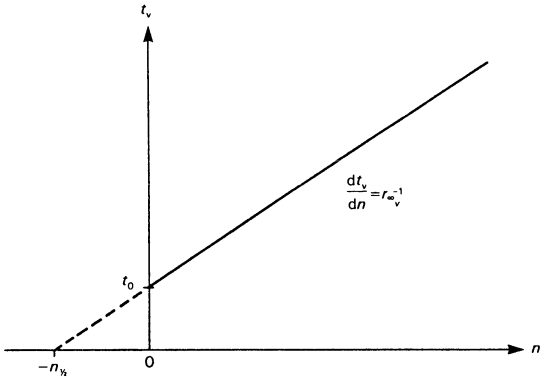

Figure 10.2 Graph showing vector instruction execution time, t_v, against n

represents the *asymptotic minimum* time overhead incurred due to pipelined vector processing. Since the slope of the line is equal to $r_{v_\infty}^{-1}$, the value of t_0 is $n_{1/2}/r_{v_\infty}$ and this is therefore an absolute indicator of the minimum startup time for a vector operation.

10.2.1 The effect of technology

If the technology from which a pipelined machine is constructed is improved such that the basic delay through each gate is reduced by a constant factor x, then the improved clock period of the machine, τ', will decrease in proportion to x, and hence $\tau' = \tau/x$. As a direct result of this $r'_{v_\infty} = x r_{v_\infty}$, and the peak processing rate is increased by a factor of x. Naturally, the *actual* minimum startup time t_0 will also be decreased by a factor x, and so

$$t'_0 = \frac{t_0}{x} = \frac{\tau}{x} n_{\frac{1}{2}}$$

However, since we know that $\tau' = \tau/x$, then it must be the case that $n'_{1/2} = n_{1/2}$. The performance metric $n_{1/2}$ is therefore a purely architectural metric, and can be used as a comparative figure of merit between machines constructed from different technologies.

10.2.2 Optimal pipelining

In chapter 4 it was shown that the asymptotic maximum speedup that can be achieved with a pipelined structure is exactly equal to the number of pipeline stages. In fact this was a slight over-simplification, since it did not take into account the delays introduced by the inter-stage latches. The latches that are introduced into a pipeline also increase the cost of the system, measured simply in terms of the number of gates. Given that the delay through each pipeline stage, and hence τ, is bounded below by

the delay through the latch (denoted here by ϕ) the cost-effectiveness of a pipelined system cannot increase indefinitely.

We can assess the cost-effectiveness of pipelining an arbitrary segment of logic by defining a performance cost function in terms of the depth of pipelining. If the delay through a non-pipelined implementation of the logic in question is T_s, then assuming the logic can be partitioned into k stages of exactly equal delay, the clock period of the pipeline, τ, will be

$$\tau \geq \frac{T_s}{k} + \phi \tag{10.5}$$

It follows that in $n\tau$ clock periods at most n results can be produced, and hence the peak throughput must be τ^{-1}.

The cost of such an arrangement can be assessed by letting c be the total cost of the logic in all stages of the pipeline, and letting d be the cost of a single pipeline latch. The total cost of the pipelined implementation is $c + kd$, and hence, we can define the performance cost ratio, P, to be

$$P = \frac{1}{\left(\frac{T_s}{k} + \phi\right)(c + kd)} \tag{10.6}$$

If $k = k_0$ when P is at a maximum, then

$$k_0 = \left(\frac{T_s\, c}{\phi\, d}\right)^{\frac{1}{2}} \tag{10.7}$$

For almost all pipelines the optimum number of stages is greater than one, but very high levels of pipeline parallelism require cheap, fast latches, if they are to be cost-effective.

Increasing the level of pipelining indefinitely is neither possible nor desirable since r_{v_∞} is effectively defined by

$$r_{v_\infty} = \frac{1}{\frac{T_s}{k} + \phi} \tag{10.8}$$

and therefore

$$\lim_{k \to \infty} r_{\infty_v} = \phi^{-1} \tag{10.9}$$

To summarise, every pipelined function has an optimum level of pipelining determined by the latency and cost of both the logic and the inter-stage latches. Furthermore, there is a diminishing return from increased levels of pipelining, due to the fundamental limitations imposed by the delay through a single latch.

10.2.3 Mixed-mode performance

This discussion of vector processor performance has so far considered only the efficiency with which vector pipelines can be implemented and utilised by vector instructions of varying lengths. However, real programs contain a mixture of vector instructions and non-vector (scalar) instructions. Therefore, to obtain a complete picture of the behaviour of vector machines the net effect of executing a mix of scalar and vector instructions must be analysed.

We have seen that the efficiency of vector processing is dependent on the length of the vector being processed, and consequently when vectors are very short it may be the case that programming the operations as a sequence of scalar operations might actually achieve a higher throughput. This effect is particularly noticeable in machines such as the STAR-100 and the CYBER 205 which have a long startup time. It is therefore important to be aware of the minimum vector length required to make the use of vector operations worthwhile.

Since we know that the time to execute n scalar operations is $t_s = nr_{s_\infty}^{-1}$, and the time to execute n elemental operations within a vector instruction is $t_v = r_{v_\infty}^{-1}[n + n_{1/2}]$, the value for n at which $t_s = t_v$ defines a *break-even* vector length, denoted n_b. When $n < n_b$, processing the n operations as scalar instructions is faster than executing a vector operation of length n. When $n > n_b$ the reverse is true. Using the equations for t_s and t_v presented above, produces

$$\frac{n_b}{r_{s_\infty}} = \frac{n_b + n_{1/2}}{r_{v_\infty}} \tag{10.10}$$

which can be re-arranged to produced

$$n_b = \frac{n_{1/2}}{\left(\frac{r_{v_\infty}}{r_{s_\infty}} - 1\right)} = \frac{n_{1/2}}{R_\infty - 1} \tag{10.11}$$

For convenience R_∞ is defined as the ratio of the maximum vector processing rate r_{v_∞} to the maximum scalar processing rate r_{s_∞}. In fact, the scalar processing rate r_s is constant, and therefore $r_{\infty_s} = r_s$.

As one might expect, the use of vector processing becomes appropriate at low values of n when either $n_{1/2}$ is small (indicating small vector startup times) or when R_∞ is large (indicating very high performance on vector instructions once they are started up).

This still leaves us with no knowledge about how well-used the vector processing capability is, nor how close to the peak numerical processing rate each individual application code will be.

Vectorisation, speedup and Amdahl's Law

A vector processor is a parallel machine, and certain operations are able to exploit the parallelism offered by such a machine while others are not. To be a candidate for parallel processing on a vector computer an elemental operation must be vectorisable; that is it must be contained within a vector instruction. The proportion of all elemental operations that can be vectorised is known as the *level of vectorisation*, and is denoted here by v. The proportion of elemental operations that cannot be placed within a vector instruction $(1-v)$ includes unstructured scalar arithmetic and house-keeping operations, such as outer-loop address calculations and control transfer operations. In many vector machines the processing of such scalar operations can sometimes proceed in parallel with vector operations.

Since we are interested in analysing the performance of a vector processor an obvious performance metric is the *speedup* which results from exploiting a given level of vectorisation on hardware with a given degree of pipeline-parallelism. In fact it is more useful to consider the actual speedup relative to the maximum speedup (or efficiency), and this is defined as

$$E_v \stackrel{\text{def}}{=} \frac{r}{r_v} \tag{10.12}$$

where r is defined as the reciprocal of the average time to process a single elemental operation, and r_v is the actual vector processing rate. Hence, r is defined as

$$r = \frac{1}{vt_v + (1-v)t_s} \tag{10.13}$$

Using $R = t_s/t_v$ produces the relationship

$$E_v = \frac{r}{r_v} = \frac{1}{vt_v r_v + (1-v)t_s r_v} \tag{10.14}$$

and knowing that $r_v t_v = r_s t_s = 1$, we obtain

$$E_v = \frac{1}{R + v(1-R)} \tag{10.15}$$

Since R represents the ratio of actual vector processing rate to scalar processing rate we can say that $R = \eta_v R_\infty$.

Therefore, through a combination of η_v and E_v it is possible to model the performance of a vector processing system in terms of technology (implementation), architecture and application parameters. In addition, just as $n_{1/2}$ provides a technology independent indicator of how long each vector needs to be to obtain a vector efficiency of 0.5, we can define a level of

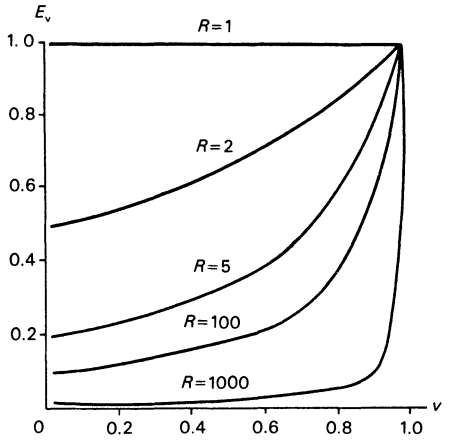

Figure 10.3 Graph showing speedup E_v versus level of vectorisation v.

vectorisation for which a relative speedup of 0.5 is observed. From equation 10.15, we obtain

$$v_{\frac{1}{2}} = \frac{R-2}{R-1} \tag{10.16}$$

A graph showing relative speedup E_v plotted against the level of vectorisation for several values of R is depicted in figure 10.3. This graph seems to indicate a rather alarming relationship between R and $v_{1/2}$, since to obtain a level performance which approaches the actual vector rate requires levels of vectorisation which grow very rapidly in relation to R.

This represents something of a contradiction. If one accepts that scalar processing rates have essentially reached (or are very close to) the limitations imposed by the speed of light, then the only hope for improvement in the performance of vector processors is through increased pipelining and hence increased values for R. However, increasing R means that higher levels of vectorisation are required. It is instructive to examine what happens to the average processing rate r as the vector:scalar processing rate ratio increases, hence

$$\lim_{R\to\infty} \quad r \to \frac{r_s}{(1-v)} \tag{10.17}$$

This indicates that as pipeline parallelism becomes large, the mixed-mode processing rate is ultimately determined by a combination of the scalar processing rate and the proportion of all operations which cannot be vectorised.

The effect described above is often referred to as *Amdahl's Law* [Amd67], and can be summarised quite simply. A computer capable of parallel processing can be viewed as a *two-state* machine; capable of processing either in its parallel state or in its sequential state. Let us assume that there is an effective degree of parallel hardware of f, which yields a processing rate

which is at most f times faster when it is in the parallel mode than when it is in the sequential mode. Any application run on such a machine will contain some number of operations capable of being performed in parallel, and hence processed at the higher rate, with the remainder having to be processed sequentially because of the inherent data-dependencies within the code. Let us assume that a proportion α of all operations can be performed at the higher rate, and that a proportion $(1-\alpha)$ has to be processed at the lower rate.

If the sequential processing rate is s, and a total of n operations must be performed, then the execution time for a totally sequential computation, T_s, must satisfy

$$T_s = s\,n$$

Furthermore, the parallel execution time T_p for the same computation, but with $\alpha\,n$ operations processed at a rate of sf operations/second and $(1-\alpha)n$ operations processed at a rate of s operations per second must satisfy the following inequality.

$$T_p \geq (1-\alpha)s\,n + \frac{\alpha\,s\,n}{f}$$

Since the speedup due to parallel processing, S, is defined as

$$S \stackrel{\text{def}}{=} \frac{\text{sequential execution time}}{\text{parallel execution time}}$$

we can assert that

$$S \leq \frac{1}{(1-\alpha)+\alpha/f} \tag{10.18}$$

Amdahl's Law therefore states that however great the parallelism in the hardware and software, it is the proportion of sequential operations which determines the asymptotic maximum speedup in a parallel system. This can be observed quite dramatically by assuming that the system operates infinitely faster in the parallel mode than in the sequential mode. This defines the speedup for an infinitely parallel hardware configuration, S_∞, in terms of the following inequality.

$$S_\infty \leq \frac{1}{(1-\alpha)}$$

In vector processors f is equivalent to the ratio of vector to scalar processing rates R, and α is equivalent to the proportion of elemental operations which can be expressed within vector instructions v, and hence Amdahl's Law confirms equation 10.17 .

10.3 Comparing vector machines

In recent years several manufacturers of high performance computers, most notably from Japan, have entered the high-risk vector processor market with machines which could be looked upon as architectural derivatives of the vector machines described in earlier chapters. For example, in 1982 Fujitsu announced two models of the FACOM Vector Processor known as the VP-100 and VP-200 [Miu86]. Around this time Hitachi introduced their S-810 range of machines [ONK86] (S-810/5, S-810/10, S-810/20), and NEC also introduced their SX Series of vector computers [WKI86] (SX-1 and SX-2).

The architecture of these systems owes a great deal to the pioneering research work which was carried out during the development of the early machines such as the STAR-100, the CRAY-1, the CYBER 205 and Texas Instruments' Advanced Scientific Computer. The recent Japanese machines all incorporate vector registers, although much enlarged, with the Fujitsu VP Series also possessing dynamically configurable vector registers. This means that it is possible to configure the total 64 Kbyte register file as 256 $\times$ 32-element vector registers, or 128 $\times$ 64-element vector registers, or any such combination up to 8 $\times$ 1024-element vector registers.

This second generation of vector computers is generally provided with sophisticated vectorising FORTRAN compilers capable of vectorising DO loops containing IF statements, providing cost analyses of individual statements, and suggesting reasons for why certain statements could not be vectorised.

This diversity of systems presents an ideal opportunity for using the performance metrics defined earlier in this chapter to compare the effectiveness of a number of different architectures. For example, at the simplest level we could compare their peak vector performance (r_{v_∞}). Of course, such comparisons do not mean very much in isolation, since r_{v_∞} only indicates the performance level which each machine cannot exceed. However, for completeness some comparative figures are shown in table 10.1. NEC claim a peak performance of 1.3 GFLOPS for the SX-2, but this relies on all sixteen vector functional units operating at their maximum burst rate, and in practice this is unlikely to be a common occurrence. A more realistic method of comparing vector machines is that of benchmarking, effectively the *timed execution* of carefully chosen segments of code or even whole programs. However, this method of comparison can be extremely subjective, since it depends heavily on the choice of workload presented to the machines under test. In general, different machines in the same class (such as vector machines) are good at different types of workload. For example, the CRAY-1 is particularly efficient on relatively short vectors, whereas the CYBER 205 is not very good at processing short vectors but exceeds the performance of

Table 10.1 Peak vector processing rates

Machine	r_{v_∞} (MFLOPS)	τ (ns)
Cray-1	160	12.5
Cray X-MP (per CPU)	235	8.5
Cyber 205 (2-pipe, 64-bit)	100	20
Fujitsu VP-200	533	7.5
Hitachi S-810/20	630	14
NEC SX-2	1300	12

the CRAY-1 when processing vectors with more than around 300 elements.

One particular benchmark comparison of the CRAY X-MP, the Fujitsu VP-200 and the Hitachi S-810/20 has been performed [LMM85] by timing the execution of a variety of repeated DO loops (some vectorisable, others not), as well as a variety of complete Fortran programs, on each machine. By repeating each loop a number of times, the overheads of initiating a vector operation, typically involving a small number of scalar instructions is included in the resultant figures. In a vector-register architecture, which all three machines are, the loading and storing of values between main memory and vector registers also affects the measurement of startup times. Therefore, machines which can overlap a number of load operations, such as the CRAY X-MP, may appear to perform better. In table 10.2 some of the results of these experiments are displayed in the form of the net startup times and peak floating-point rates for a number of repeated statements, on each machine in question. The code executed for each benchmarked statement had the following basic structure.

```
      CALL CLOCK
      DO 100 N = 1, NLOOP
      DO 100 I = 1, VECLENGTH
100   <statement>
      CALL CLOCK
```

The six statements in the first column of table 10.2 represent the <statement> code which was executed within this harness to obtain the results in the associated rows. The expectation was that the inner loop would be vectorised, and that the outer loop would cause a number of vector instructions to be issued. The first column for each machine indicates the asymptotic minimum time to execute the code (t_0), and thus represents the startup time for these code segments. The second column for each machine indicates the number of floating point operations taking place during each clock

Table 10.2 Comparison of startup times and processing rates

Code segment (Fortran)	CRAY X-MP		Fujitsu VP-200		Hitachi S-810/20	
	t_0	F	t_0	F	t_0	F
[1] A(I) = B(I)+S	444	1	655	2	1065	2
[2] A(I) = B(I)*C(I)	468	1	729	1	1158	2
[3] A(I) = B(I)*C(I)+D(I)*E(I)	558	0.5	966	0.75	1268	1
[4] S = S + A(I)*A(I)	3723	0.5	1931	2	2421	2
[5] A(I) = B(J(I))+S	1142	0.037	1261	0.5	1608	1
[6] A(J(I)) = A(I)*B(I)	1191	0.043	1202	0.25	1281	0.5

Notes:

1. t_0 is the asymptotic minimum execution time, measured in nanoseconds, and occurs when the vector length is zero.
2. The value of F indicates the number of floating-point results produced per clock period for the specified operations.
3. The data is taken from Lubeck *et al.* [LMM85]

cycle, when the pipelines are operating at their peak rate for the operations listed. In all cases the stride (the increment for I) is unity[1]. One further point of note is that in benchmarks [5] and [6] the CRAY X-MP performs very badly. This is because in this benchmark the *scatter* and *gather* operations were not vectorised, and were therefore executed as loops of scalar instructions. The performance of these scalar operations suggests an operational (average) vector:scalar processing rate ratio in the CRAY X-MP of approximately 25.

These benchmarks indicate that operations on long vectors can be as much as 2 to 3 times as faster in the VP-200 than in the CRAY X-MP, but that their scalar performance is roughly equivalent. From equation 10.17 we know that as the differential between scalar and vector performance becomes large, the perceived net rate of processing becomes heavily dependent on the speed of scalar processing. In addition, we know that a low value for $n_{1/2}$ is a desirable attribute in a vector processor, and we note from equation 10.4 that $n_{1/2}$ is equal to the startup time of a vector operation (or a sequence

[1]In benchmark tests where the stride was not unity the CRAY X-MP was able to maintain its full processing rate, whereas both the VP-200 and the S-810/20 processing rates were somewhat reduced.

of vector operations) divided by the number of clock periods per floating-point operation minus one. Consequently, the task of comparing vector processors where R is large is very much centered around a comparison of *scalar* processing rates, and vector startup times, rather than a comparison of asymptotic maximum vector processing rates.

Bibliography

[ABB64] G.M. Amdahl, G.A. Blaauw, and F.P. Brooks. Architecture of the IBM System/360. *IBM Journal of R & D*, 8:87–101, 1964.

[Amd67] G.M. Amdahl. The Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *AFIPS Conf. Proc.*, 1967.

[AST71] D.W. Anderson, F.J. Sparacio, and R.M. Tomasulo. The IBM System/360 Model 91: Machine Philosophy and Instruction Handling. *IBM Journal of R & D*, 11:8–24, 1971.

[Bel71] C.G. Bell. *Computer Structures: Readings and Examples.* McGraw-Hill, New York, 1971.

[BKHH78] C.G. Bell, A. Kotek, T.N. Hastings, and R. Hill. The Evolution of the DEC System 10. *Communications of the ACM*, 21:44–62, 1978.

[BS76] C.G. Bell and W.D. Strecker. Computer Structures: What we have learned from the PDP-11. In *Proc. 3rd Annual Symposium on Computer Architecture*, pages 1–14, 1976.

[BSG*77] M.R. Barbacci, D. Siewiorek, R. Gordon, R. Howbrigg, and S. Zuckerman. An Architectural Research Facility — ISP Descriptions, Simulation, Data Collection. *Proc. AFIPS NCC*, 46:161–173, 1977.

[Buc62] W. Buchholz, editor. *Planning a Computer System.* McGraw-Hill, New York, 1962.

[Buc78] J.K. Buckle. *The ICL 2900 Series.* Macmillan, London, 1978.

[CDC77] *Control Data 7600 / CYBER 70 Model 76 Computer Systems — Hardware Reference Manual.* Control Data Corporation, St Paul, Minnesota, 1977.

[CDC81] *Control Data CYBER 200 Model 205 Computer System — Hardware Reference Manual.* Control Data Corporation, St Paul, Minnesota, 1981.

[Che83] S.C. Chen. Large-scale and high-speed multiprocessor system for scientific applications — CRAY X-MP. In J. Kowalik, editor, *Proc. NATO Advanced Workshop on High Speed Computation*, Springer-Verlag, New York, 1983.

[CHJ*85] R.P. Colwell, C. Hitchcock, E. Jensen, H. Brinkley-Sprunt, and C. Kollar. Computers, Complexity and Controversy. *Computer*, September, 1985.

[CHLS] S.C. Chen, C.C. Hsiung, J.L. Larson, and E.R. Somdahl. CRAY X-MP: A Multiprocessor Supercomputer. In M. Ginsberg, editor, *Vector and Parallel Processors: Architecture, Applications, and Performance Evaluation*, North-Holland, Amsterdam, to be published.

[CIP74] P.C. Capon, R.N. Ibbett, and C.R.C.B. Parker. The Implementation of Record Processing in MU5. *IEE Conference Proceedings*, 121, 1974.

[CP78] R.P. Case and A. Padegs. Architecture of the IBM System/370. *Communications of the ACM*, 21:73–96, 1978.

[CRA87] *The CRAY-2 Series of Computer Systems.* Cray Research Inc., Minneapolis, Minnesota, 1987.

[CW76] H.J. Curnow and B.A. Wichmann. A Synthetic Benchmark. *Computer Journal*, 19:43–49, 1976.

[DIS80] J. Djordjevic, R.N. Ibbett, and F.H. Sumner. Evaluation of Some Proposed Name-space Architectures Using ISPS. *IEE Proceedings*, 127E:120–125, 1980.

[FBSL77] S.H. Fuller, W.E. Burr, P. Shaman, and D.A. Lamb. Evaluation of Computer Architectures via Test Programs. *Proc. AFIPS NCC*, 46:147–160, 1977.

[FJW85] M.J. Flynn, J.D. Johnson, and S.P. Wakefield. On Instruction Sets and their Formats. *IEEE Transactions on Computers*, C-34:242–254, 1985.

[FL71] M.J. Flynn and P.R. Low. The IBM System/360 Model 91: Some Remarks on System Development. *IBM Journal of R & D*, 11:2–7, 1971.

[Fly72] M.J. Flynn. Some Computer Organisations and their Effectiveness. *IEEE Transactions on Computers*, C-21:948–960, 1972.

[Fly74] M.J. Flynn. *Information Processing*, chapter Trends and Problems in Computer Organisation. North-Holland, Amsterdam, 1974.

[Gos80] J.B. Gosling. *Design of Arithmetic Units for Digital Computers.* Macmillan, London, 1980.

[HI80] R.W. Holgate and R.N. Ibbett. An Analysis of Instruction Fetching Strategies in Pipelined Computers. *IEEE Transactions on Computers*, C-29:325–329, 1980.

[HJ81] R.W. Hockney and C.R. Jesshope. *Parallel Computers.* Adam Hilger, Bristol, 1981.

[Hoc77] R.W. Hockney. *Supercomputer Architecture*, pages 277–305. Infotech Intl Ltd, Maidenhead, 1977.

[HT72] R.G. Hintz and D.P. Tate. Control Data STAR-100 Processor Design. *COMPCON '72 Digest*, 1–4, 1972.

[IBM] *IBM System/360 Principles of Operation.*

[IH77] R.N. Ibbett and M.A. Husband. The MU5 Name Store. *Computer Journal*, 20:227–231, 1977.

[Ive62] K.E. Iverson. *A Programming Language.* Wiley, New York, 1962.

[Joh78] P.M. Johnson. An Introduction to Vector Processing. *Computer Design*, 89–97, Feb. 1978.

[KELS62] T. Kilburn, D.B.G. Edwards, M.J. Lanigan, and F.H. Sumner. One-level Storage System. *IRE Transactions*, EC-11:223–234, 1962.

[KW76] D.J. Kinniment and J.V. Woods. Synchronisation and Arbitration Circuits in Digital Systems. *IEE Proceedings*, 123:961–966, 1976.

[Lav75] S.H. Lavington. *History of Manchester Computers.* NCC, Manchester, 1975.

[Lin78] N. Lincoln. A Safari through the Control Data STAR-100 with Gun and Camera. *Proc. AFIPS NCC*, 47, 1978.

[Lin82] N. Lincoln. Technology and Design Tradeoffs in the Creation of a Modern Supercomputer. *IEEE Transactions on Computers*, C-31:349–362, 1982.

[Lip68] J.S. Liptay. Structural Aspects of the System/360 Model 85 – The Cache. *IBM Systems Journal*, 7:15–21, 1968.

[Lis84] A.M. Lister. *Fundamentals of Operating Systems.* Macmillan, London, third edition, 1984.

[LMM85] O. Lubeck, J. Moore, and R. Mendez. A Benchmark Comparison of Three Supercomputers: Fujitsu VP-200, Hitachi S-810/20, and Cray X-MP/2. *IEEE Computer*, 10–23, December, 1985.

[LTE77] S.H. Lavington, G. Thomas, and D.G.B. Edwards. The MU5 Multicomputer Communication System. *IEEE Transactions on Computers*, C-26:19–28, 1977.

[MI79] D. Morris and R.N. Ibbett. *The MU5 Computer System.* Macmillan, London, 1979.

[Miu86] K. Miura. Fujitsu's Supercomputer: Facom Vector Processor System. In S. Fernbach, editor, *Supercomputers: Class VI Systems, Hardware and Software*, pages 137–152, North-Holland, Amsterdam, 1986.

[MW70] J.O. Murphy and R.M. Wade. The IBM 360/195. *Datamation*, 72–79, April, 1970.

[ONK86] T. Odaka, S. Nagashima, and S. Kawabe. Hitachi Supercomputer S-810 Array Processor System. In S. Fernbach, editor, *Supercomputers: Class VI Systems, Hardware and Software*, pages 113–136, North-Holland, Amsterdam, 1986.

[Org72] E.I. Organick. *The Multics System.* MIT Press, Cambridge, Mass., 1972.

[Org73] E.I. Organick. *Computer System Organisation – The B5700/B6700.* Academic Press, New York, 1973.

[Pat85] D.A. Patterson. Reduced Instruction Set Computers. *Communications of the ACM*, 28(1), January, 1985.

[RL77] C.V. Ramamoorthy and H.F. Li. Pipeline Architecture. *Computer Surveys*, 9:61–102, 1977.

[Rus78] R.M. Russell. The CRAY-1 Computer System. *Communications of the ACM*, 21:63–72, 1978.

[Sto75] H.S. Stone. *Introduction to Computer Architecture.* Science Research Associates, Chicago, 1975.

[Tho70] J.E. Thornton. *Design of a Computer: The Control Data 6600.* Scott Foresman & Co, Glenview, Ill, 1970.

[Tho86] J.R. Thompson. The CRAY-1, the CRAY-XMP, the CRAY-2 and Beyond: The Supercomputers of Cray Research. In S. Fernbach, editor, *Supercomputers: Class VI Systems, Hardware and Software*, pages 169–82, North-Holland, Amsterdam, 1986.

[Tom71] R.M. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of R & D*, 11:25–33, 1971.

[Tuc86] S.G. Tucker. The IBM 3090 system: An overview. *IBM Systems Journal*, 25:4–19, 1986.

[Wat72] W.J. Watson. The TI ASC – A Highly Modular and Flexible Super Computer Architecture. *AFIPSFJCC*, 41:221–228, 1972.

[WKI86] T. Watanabe, H. Katayama, and A. Iwaya. Introduction of the NEC Supercomputer SX System. In S. Fernbach, editor, *Supercomputers: Class VI Systems, Hardware and Software*, pages 153–168, North-Holland, Amsterdam, 1986.

[YIH77] N.A. Yannacopoulos, R.N. Ibbett, and R.W. Holgate. Performance Measurements of the MU5 Primary Instruction Pipeline. In *Information Processing 77*, North-Holland, Amsterdam, 1977.

Index

Primary references are in boldface, obscure references are in italics.