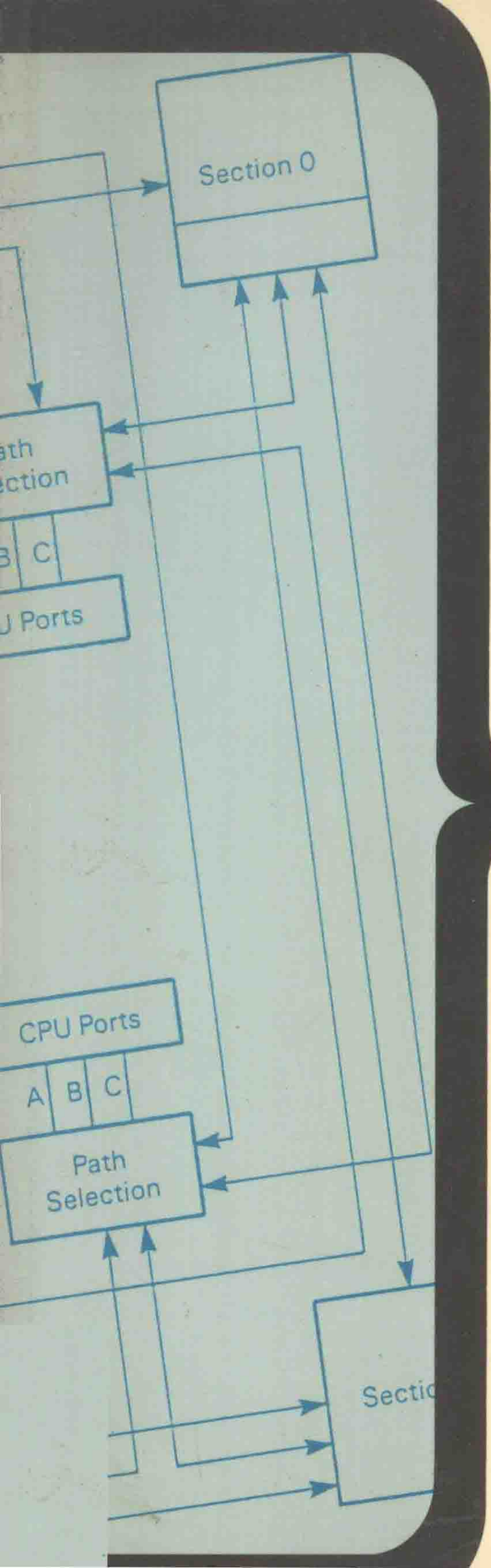

Architecture of High Performance Computers
Volume I

R.N. Ibbett and
N.P. Topham

Architecture of High Performance Computers
Volume I

Uniprocessors and vector processors

R. N. Ibbett and N. P. Topham

Department of Computer Science
University of Edinburgh

M
MACMILLAN

First published 1989

Published by
MACMILLAN EDUCATION LTD
Houndmills, Basingstoke, Hampshire RG21 2XS
and London
Companies and representatives
throughout the world

Printed and bound in Great Britain at
The Camelot Press Ltd, Southampton

British Library Cataloguing in Publication Data
Ibbett, Roland N.
Architecture of high performance computers.
——(Macmillan computer science series).
1. High performance computer systems.
Design
I. Title II. Topham, Nigel P.
001.64

ISBN 0–333–46362–5 V.1 Pbk
ISBN 0–333–48988–8 V.2 Pbk

(continued overleaf)

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

Contents

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-

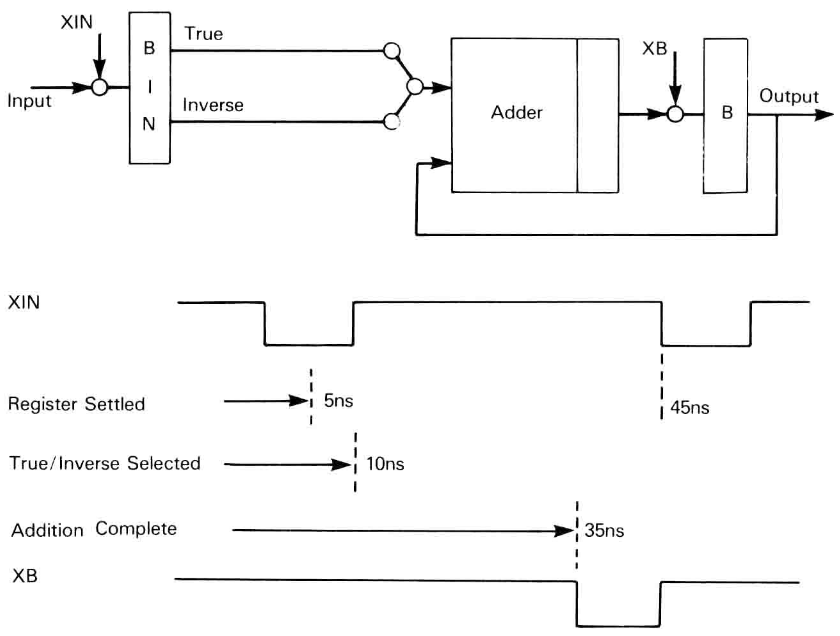

Figure 1.1 An index arithmetic unit

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

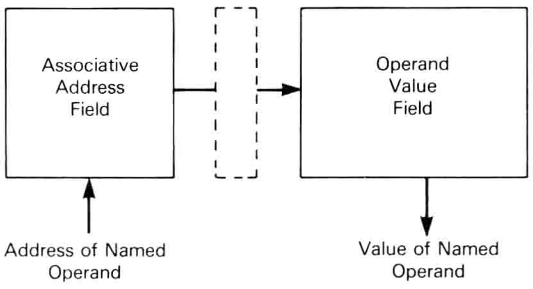

Figure 1.2 The MU5 name store

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

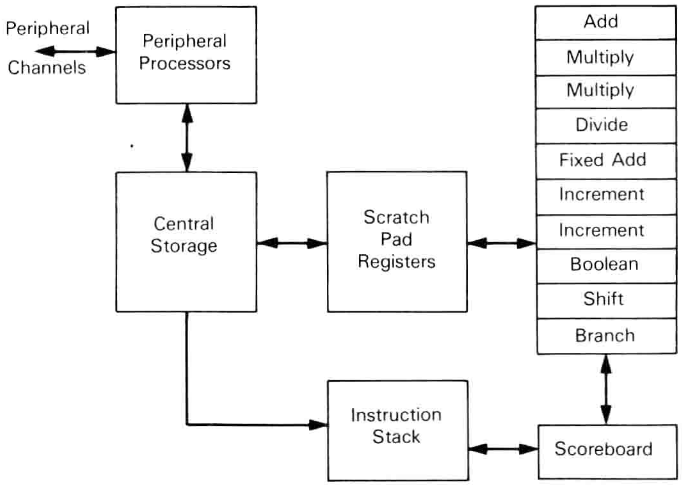

Figure 2.1 CDC 6600 processor organisation

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.