

Computer Architecture

Design and performance

Barry Wilkinson

*Department of Computer Science
University of North Carolina, Charlotte*



Prentice Hall

New York London Toronto Sydney Tokyo Singapore

08
2 3. JUNI 1992



First published 1991 by
Prentice Hall International (UK) Ltd
66 Wood Lane End, Hemel Hempstead
Hertfordshire HP2 4RG
A division of
Simon & Schuster International Group

© Prentice Hall International (UK) Ltd, 1991

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording or otherwise, without prior permission, in writing, from the publisher.

For permission within the United States of America contact Prentice Hall Inc., Englewood Cliffs, NJ 07632.

Typeset in 10/12pt Times with Courier

Printed in Great Britain at
the University Press, Cambridge

Library of Congress Cataloging-in-Publishing Data

Wilkinson, Barry.

Computer architecture: design and performance/by Barry Wilkinson

p. cm.

Includes bibliographical references and index.

ISBN 0-13-173899-2. — ISBN 0-13-173907-7 (pbk.)

1. Computer architecture. I. Title.

QA76.9.A73W54 1991

004.2'2—dc20

90-7953

CIP

British Library Cataloguing in Publication Data

Wilkinson, Barry 1947—

Computer architecture: design and performance.

1. High performance computer systems. Design

I. Title


004.22

ISBN 0-13-173899-2

ISBN 0-13-173907-7 pbk

1 2 3 4 5 94 93 92 91 90

0892 81228



*To my wife, Wendy
and my daughter, Johanna*

Contents

Preface	xiii
Part I Computer design techniques	1
1 Computer systems	3
1.1 The stored program computer	3
1.1.1 Concept	3
1.1.2 Improvements in performance	10
1.2 Microprocessor systems	12
1.2.1 Development	12
1.2.2 Microprocessor architecture	14
1.3 Architectural developments	16
1.3.1 General	16
1.3.2 Processor functions	16
1.3.3 Memory hierarchy	18
1.3.4 Processor–memory interface	19
1.3.5 Multiple processor systems	22
1.3.6 Performance and cost	24
2 Memory management	25
2.1 Memory management schemes	25
2.2 Paging	27
2.2.1 General	27
2.2.2 Address translation	32
2.2.3 Translation look-aside buffers	36
2.2.4 Page size	38
2.2.5 Multilevel page mapping	39
2.3 Replacement algorithms	41
2.3.1 General	41

2.3.2	Random replacement algorithm	43
2.3.3	First-in first-out replacement algorithm	44
2.3.4	Clock replacement algorithm	45
2.3.5	Least recently used replacement algorithm	45
2.3.6	Working set replacement algorithm	47
2.3.7	Performance and cost	49
2.4	Segmentation	51
2.4.1	General	51
2.4.2	Paged segmentation	55
2.4.3	8086/286/386 segmentation	57
	Problems	61
3	Cache memory systems	64
3.1	Cache memory	64
3.1.1	Operation	64
3.1.2	Hit ratio	67
3.2	Cache memory organizations	68
3.2.1	Direct mapping	68
3.2.2	Fully associative mapping	71
3.2.3	Set-associative mapping	73
3.2.4	Sector mapping	74
3.3	Fetch and write mechanisms	75
3.3.1	Fetch policy	75
3.3.2	Write operations	76
3.3.3	Write-through mechanism	77
3.3.4	Write-back mechanism	80
3.4	Replacement policy	81
3.4.1	Objectives and constraints	81
3.4.2	Random replacement algorithm	82
3.4.3	First-in first-out replacement algorithm	82
3.4.4	Least recently used algorithm for a cache	82
3.5	Cache performance	86
3.6	Virtual memory systems with cache memory	90
3.6.1	Addressing cache with real addresses	90
3.6.2	Addressing cache with virtual addresses	91
3.6.3	Access time	93
3.7	Disk caches	94
3.8	Caches in multiprocessor systems	95
	Problems	99

4 Pipelined systems	102
4.1 Overlap and pipelining	102
4.1.1 Technique	102
4.1.2 Pipeline data transfer	103
4.1.3 Performance and cost	105
4.2 Instruction overlap and pipelines	107
4.2.1 Instruction fetch/execute overlap	107
4.2.2 Branch instructions	111
4.2.3 Data dependencies	117
4.2.4 Internal forwarding	121
4.2.5 Multistreaming	122
4.3 Arithmetic processing pipelines	123
4.3.1 General	123
4.3.2 Fixed point arithmetic pipelines	124
4.3.3 Floating point arithmetic pipelines	127
4.4 Logical design of pipelines	130
4.4.1 Reservation tables	130
4.4.2 Pipeline scheduling and control	133
4.5 Pipelining in vector computers	138
Problems	140
5 Reduced instruction set computers	144
5.1 Complex instruction set computers (CISCs)	144
5.1.1 Characteristics	144
5.1.2 Instruction usage and encoding	146
5.2 Reduced instruction set computers (RISCs)	148
5.2.1 Design philosophy	148
5.2.2 RISC characteristics	150
5.3 RISC examples	153
5.3.1 IBM 801	153
5.3.2 Early university research prototypes – RISC I/II and MIPS	156
5.3.3 A commercial RISC – MC88100	160
5.3.4 The Inmos transputer	165
5.4 Concluding comments on RISCs	166
Problems	167

Part II Shared memory multiprocessor systems	169
6 Multiprocessor systems and programming	171
6.1 General	171
6.2 Multiprocessor classification	173
6.2.1 Flynn's classification	173
6.2.2 Other classifications	175
6.3 Array computers	175
6.3.1 General architecture	175
6.3.2 Features of some array computers	177
6.3.3 Bit-organized array computers	180
6.4 General purpose (MIMD) multiprocessor systems	182
6.4.1 Architectures	182
6.4.2 Potential for increased speed	188
6.5 Programming multiprocessor systems	193
6.5.1 Concurrent processes	193
6.5.2 Explicit parallelism	194
6.5.3 Implicit parallelism	199
6.6 Mechanisms for handling concurrent processes	203
6.6.1 Critical sections	203
6.6.2 Locks	203
6.6.3 Semaphores	207
Problems	210
7 Single bus multiprocessor systems	213
7.1 Sharing a bus	213
7.1.1 General	213
7.1.2 Bus request and grant signals	215
7.1.3 Multiple bus requests	216
7.2 Priority schemes	218
7.2.1 Parallel priority schemes	218
7.2.2 Serial priority schemes	227
7.2.3 Additional mechanisms in serial and parallel priority schemes	234
7.2.4 Polling schemes	235
7.3 Performance analysis	237
7.3.1 Bandwidth and execution time	237
7.3.2 Access time	240
7.4 System and local buses	241
7.5 Coprocessors	243
7.5.1 Arithmetic coprocessors	243
7.5.2 Input/output and other coprocessors	247
Problems	248

8	Interconnection networks	250
8.1	Multiple bus multiprocessor systems	250
8.2	Cross-bar switch multiprocessor systems	252
8.2.1	Architecture	252
8.2.2	Modes of operation and examples	253
8.3	Bandwidth analysis	256
8.3.1	Methods and assumptions	256
8.3.2	Bandwidth of cross-bar switch	257
8.3.3	Bandwidth of multiple bus systems	260
8.4	Dynamic interconnection networks	262
8.4.1	General	262
8.4.2	Single stage networks	263
8.4.3	Multistage networks	263
8.4.4	Bandwidth of multistage networks	270
8.4.5	Hot spots	273
8.5	Overlapping connectivity networks	275
8.5.1	Overlapping cross-bar switch networks	276
8.5.2	Overlapping multiple bus networks	279
8.6	Static interconnection networks	282
8.6.1	General	282
8.6.2	Exhaustive static interconnections	282
8.6.3	Limited static interconnections	282
8.6.4	Evaluation of static networks	287
	Problems	290
Part III	Multiprocessor systems without shared memory	293
9	Message-passing multiprocessor systems	295
9.1	General	295
9.1.1	Architecture	295
9.1.2	Communication paths	298
9.2	Programming	301
9.2.1	Message-passing constructs and routines	301
9.2.2	Synchronization and process structure	304
9.3	Message-passing system examples	308
9.3.1	Cosmic Cube	308
9.3.2	Intel iPSC system	309
9.4	Transputer	311
9.4.1	Philosophy	311
9.4.2	Processor architecture	312
9.5	Occam	314
9.5.1	Structure	314

xii Contents

9.5.2	Data types	315
9.5.3	Data transfer statements	316
9.5.4	Sequential, parallel and alternative processes	317
9.5.5	Repetitive processes	320
9.5.6	Conditional processes	321
9.5.7	Replicators	323
9.5.8	Other features	324
	Problems	325
10	Multiprocessor systems using the dataflow mechanism	329
10.1	General	329
10.2	Dataflow computational model	330
10.3	Dataflow systems	334
10.3.1	Static dataflow	334
10.3.2	Dynamic dataflow	337
10.3.3	VLSI dataflow structures	342
10.3.4	Dataflow languages	344
10.4	Macrodataflow	349
10.4.1	General	349
10.4.2	Macrodataflow architectures	350
10.5	Summary and other directions	353
	Problems	354
	References and further reading	357
	Index	366

Preface

Although computer systems employ a range of performance-improving techniques, intense effort to improve present performance and to develop completely new types of computer systems with this improved performance continues. Many design techniques involve the use of *parallelism*, in which more than one operation is performed simultaneously. Parallelism can be achieved by using multiple functional units at various levels within the computer system. This book is concerned with design techniques to improve the performance of computer systems, and mostly with those techniques involving the use of parallelism.

The book is divided into three parts. In Part I, the fundamental methods to improve the performance of computer systems are discussed; in Part II, multiprocessor systems using shared memory are examined in detail and in Part III, computer systems not using shared memory are examined; these are often suitable for VLSI fabrication. Dividing the book into parts consisting of closely related groups of chapters helps delineate the subject matter.

Chapter 1 begins with an introduction to computer systems, microprocessor systems and the scope for improved performance. The chapter introduces the topics dealt with in detail in the subsequent chapters, in particular, parallelism within the processor, parallelism in the memory system, management of the memory for improved performance and multiprocessor systems. Chapters 2 and 3 concentrate upon memory management – Chapter 2 on main memory/secondary memory management and Chapter 3 on processor/high speed buffer (cache) memory management. The importance of cache memory has resulted in a full chapter on the subject, rather than a small section combined with main memory/secondary memory as almost always found elsewhere. Similarly, Chapter 4 deals exclusively with pipelining as applied within a processor, this being the basic technique for parallelism within a processor. Scope for overall improved performance exists when choosing the actual instructions to implement in the instruction set. In Chapter 5, the concept of the so-called *reduced instruction set computer* (RISC), which has a very limited number of instructions and is used predominantly for register-to-register operations, is discussed.

Chapter 6, the first chapter in Part II, introduces the design of shared memory

multiprocessor systems, including a section on programming shared memory multiprocessor systems. Chapter 7 concentrates upon the design of a single bus multiprocessor system and its variant (system/local bus systems); the bus arbitration logic is given substantial treatment. Chapter 8 considers single stage and multistage interconnection networks for linking together processors and memory in a shared memory multiprocessor system. This chapter presents bandwidth analysis of cross-bar switch, multiple bus and multistage networks, including overlapping connectivity networks.

Chapter 9, the first chapter in Part III, presents multiprocessor systems having local memory only. Message-passing concepts and architectures are described and the transputer is outlined, together with its language, Occam. Chapter 10 is devoted to the dataflow technique, used in a variety of applications. Dataflow languages are presented and a short summary is given at the end of the chapter.

The text can serve as a course text for senior level/graduate computer science, computer engineering or electrical engineering courses in computer architecture and multiprocessor system design. The text should also appeal to design engineers working on 16-/32-bit microprocessor and multiprocessor applications. The material presented is a natural extension to material in introductory computer organization/computer architecture courses, and the book can be used in a variety of ways. Material from Chapters 1 to 6 could be used for a senior computer architecture course, whereas for a course on multiprocessor systems, Chapters 6 to 10 could be studied in detail. Alternatively, for a computer architecture course with greater scope, material could be selected from all or most chapters, though generally from the first parts of sections. It is assumed that the reader has a basic knowledge of logic design, computer organization and computer architecture. Exposure to computer programming languages, both high level programming languages and low level microprocessor assembly languages, is also assumed.

I would like to record my appreciation to Andrew Binnie of Prentice Hall, who helped me start the project, and to Helen Martin, also of Prentice Hall, for her support throughout the preparation of the manuscript. Special thanks are extended to my students in the graduate courses CPGR 6182, CSCI 5041 and CSCI 5080, at the University of North Carolina, Charlotte, who, between 1988 and 1990, helped me "classroom-test" the material; this process substantially improved the manuscript. I should also like to thank two anonymous reviewers who made constructive and helpful comments.

Barry Wilkinson
University of North Carolina
Charlotte

PART

I

*Computer
design
techniques*

In this chapter, the basic operation of the traditional stored program digital computer and microprocessor implementation are reviewed. The limitations of the single processor computer system are outlined and methods to improve the performance are suggested. A general introduction to one of the fundamental techniques of increasing performance – the introduction of separate functional units operating concurrently within the system – is also given.

1.1 The stored program computer

1.1.1 Concept

The computer system in which operations are encoded in binary, stored in a memory and performed in a defined sequence is known as a *stored program computer*. Most computer systems presently available are stored program computers. The concept of a computer which executes a sequence of steps to perform a particular computation can be traced back over 100 years to the mechanical decimal computing machines proposed and partially constructed by Charles Babbage. Babbage's Analytical Engine of 1834 contained program and data input (punched cards), memory (mechanical), a central processing unit (mechanical with decimal arithmetic) and output devices (printed output or punched cards) – all the key features of a modern computer system. However, a complete, large scale working machine could not be finished with the available mechanical technology and Babbage's work seems to have been largely ignored for 100 years, until electronic circuits, which were developed in the mid-1940s, made the concept viable.

The true binary programmable electronic computers began to be developed by several groups in the mid-1940s, notably von Neumann and his colleagues in the United States; stored program computers are often called *von Neumann computers*, after his work. (Some pioneering work was done by Zuse in Germany during the 1930s and 1940s, but this work was not widely known at the time.) During the

4 Computer design techniques

1940s, immense development of the stored program computer took place and the basis of complex modern computing systems was created. However, there are alternative computing structures with stored instructions which are *not* executed in a sequence related to the stored sequence (e.g. dataflow computers, which are described in Chapter 10) or which may not even have instructions stored in memory at all (e.g. neural computers).

The basic von Neumann stored program computer has:

1. A memory used for holding both instructions and the data required by those instructions.
2. A control unit for fetching the instructions from memory.
3. An arithmetic processor for performing the specified operations.
4. Input/output mechanisms and peripheral devices for transferring data to and from the system.

The control unit and the arithmetic processor of a stored program computer are normally combined into a *central processing unit* (CPU), which results in the general arrangement shown in Figure 1.1. Binary representation is used throughout for the number representation and arithmetic, and corresponding Boolean values are used for logical operations and devices. Thus, only two voltages or states are needed to represent each digit (0 or 1). Multiple valued representation and logic have been, and are still being, investigated.

The instructions being executed (or about to be executed) and their associated data are held in the *main memory*. This is organized such that each binary word is stored in a location identified by a number called an *address*. Memory addresses are allocated in strict sequence, with consecutive memory locations given consecutive

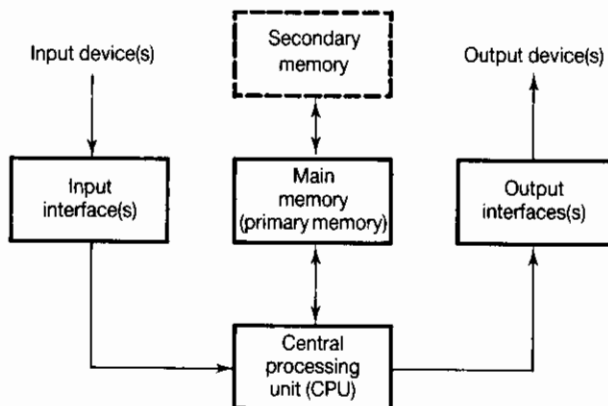


Figure 1.1 Stored program digital computer

addresses. Main memory must access individual storage locations in any order and at very high speed; such memory is known as *random access memory* (RAM) and is essential for the main memory of the system.

There is usually additional memory, known as *secondary memory* or *backing store*, provided to extend the capacity of the memory system more economically than when main memory alone is used. Main memory usually consists of semiconductor memory and is more expensive per bit than secondary memory, which usually consists of magnetic memory. However, magnetic secondary memory is not capable of providing the required high speed of data transfer, nor can it locate individual storage locations in a random order at high speed (i.e. it is not truly random access memory).

Using the same memory for data and instructions is a key feature of the von Neumann stored program computer. However, having data memory and program memory separated, with separate transfer paths between the memory and the processor, is possible. This scheme is occasionally called the *Harvard architecture*. The Harvard architecture may simplify memory read/write mechanisms (see Chapter 3), particularly as programs are normally only read during execution, while data might be read or altered. Also, data and unrelated instructions can be brought into the processor simultaneously with separate memories. However, using one memory to hold both the program and the associated data gives more efficient use of memory, and it is usual for the bulk of the main memory in a computer system to hold both. The early idea that stored instructions could be altered during execution was quickly abandoned with the introduction of other methods of modifying instruction execution.

The (central) processor has a number of internal registers for holding specific operands used in the computation, other numbers, addresses and control information. The exact allocation of registers is dependent upon the design of the processor. However, certain registers are always present. The *program counter* (PC), also called the *instruction pointer* (IP), is an internal register holding the address of the next instruction to be executed. The contents of the PC are usually incremented each time an instruction word has been read from memory in preparation for the next instruction word, which is often in the next location. A *stack pointer* register holds the address of the “top” location of the *stack*. The stack is a set of locations, reserved in memory, which holds return addresses and other parameters of subroutines.

A set of general purpose registers or sets of data registers and address registers are usually provided (registers holding data operands and addresses pointing to memory locations). In many instances these registers can be accessed more quickly than main memory locations and hence can achieve a higher computational speed.

The binary encoded instructions are known as *machine instructions*. The operations specified in the machine instructions are normally reduced to simple operations, such as arithmetic operations, to provide the greatest flexibility. Arithmetic and other simple operations operate on one or two operands, and produce a numeric result. More complex operations are created from a sequence of simple instructions by the user. From a fixed set of machine instructions available in the computer (the *instruction set*) the user selects instructions to perform a particular computation.

6 Computer design techniques

The list of instructions selected is called a *computer program*. The selection is done by a *programmer*. The program is stored in the memory and, when the system is ready, each machine instruction is read from (main) memory and executed.

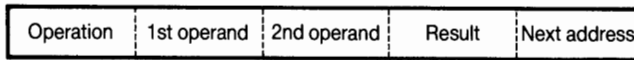
Each machine instruction needs to specify the operation to be performed, e.g. addition, subtraction, etc. The operands also need to be specified, either explicitly in the instruction or implicitly by the operation. Often, each operand is specified in the instruction by giving the address of the location holding it. This results in a general instruction format having three addresses:

1. Address of the first operand.
2. Address of the second operand.
3. Storage address for the result of the operation.

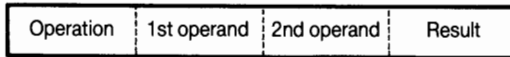
A further address could be included, that of the next instruction to be executed. This is the *four-address instruction format*. The EDVAC computer, which was developed in the 1940s, used a four-address instruction format (Hayes, 1988) and this format has been retained in some microprogrammed control units, but the fourth address is always eliminated for machine instructions. This results in a *three-address instruction format* by arranging that the next instruction to be executed is immediately following the current instruction in memory. It is then necessary to provide an alternative method of specifying non-sequential instructions, normally by including instructions in the instruction set which alter the subsequent execution sequence, sometimes under specific conditions.

The third address can be eliminated to obtain the *two-address instruction format* by always placing the result of arithmetic or logic operations in the location where the first operand was held; this overwrites the first operand. The second address can be eliminated to obtain the *one-address instruction format* by having only one place for the first operand and result. This location, which would be within the processor itself rather than in the memory, is known as an *accumulator*, because it accumulates results. However, having only one location for one of the operands and for the subsequent result is rather limiting, and a small group of registers within the processor can be provided, as selected by a small field in the instruction; the corresponding instruction format is the *one-and-a-half-address instruction format* or register type. All the addresses can be eliminated to obtain the *zero-address instruction format*, by using two known locations for the operands. These locations are specified as the first and second locations of a group of locations known as a *stack*. The various formats are shown in Figure 1.2. The one-and-a-half- or two-address formats are mostly used, though there are examples of three-address processors, e.g. the AT&T WE3210 processor.

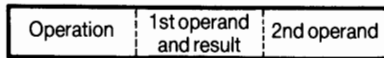
Various methods (*addressing modes*) can be used to identify the locations of the operands. Five different methods are commonly incorporated into the instruction set:



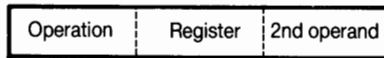
(a) Four-address format



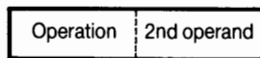
(b) Three-address format



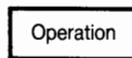
(c) Two-address format



(d) One-and-a-half address format



(e) One-address format



(f) Zero-address format

Figure 1.2 Instruction formats (a) Four-address format (b) Three-address format (c) Two-address format (d) One-and-a-half-address format (e) One-address format (f) Zero-address format

1. *Immediate addressing* – when the operand is part of the instruction.
2. *Absolute addressing* – when the address of an operand is held in the instruction.
3. *Register direct addressing* – when the operand is held in an addressed register.
4. *Register indirect addressing* – when the address of the operand location is held in a register.
5. Various forms of *relative addressing* – when the address of the operand is computed by adding an offset held in the instruction to the contents of specific registers.

The operation of the processor can be divided into two distinct steps, as shown in Figure 1.3. First, an instruction is obtained from the memory and the program counter is incremented – this step is known as the *fetch cycle*. Then the operation is performed – this step is known as the *execute cycle* and includes fetching any operands and storing the result. Sometimes, more than one memory location is necessary to hold an instruction (depending upon the design of the instructions).

8 Computer design techniques

When this occurs the program counter is incremented by one as each location is accessed to extract a part of the instruction. The contents of the program counter can be purposely altered by the execution of “jump” instructions, used to change the execution sequence. This facility is essential to create significant computations and different computations which depend upon previous computations.

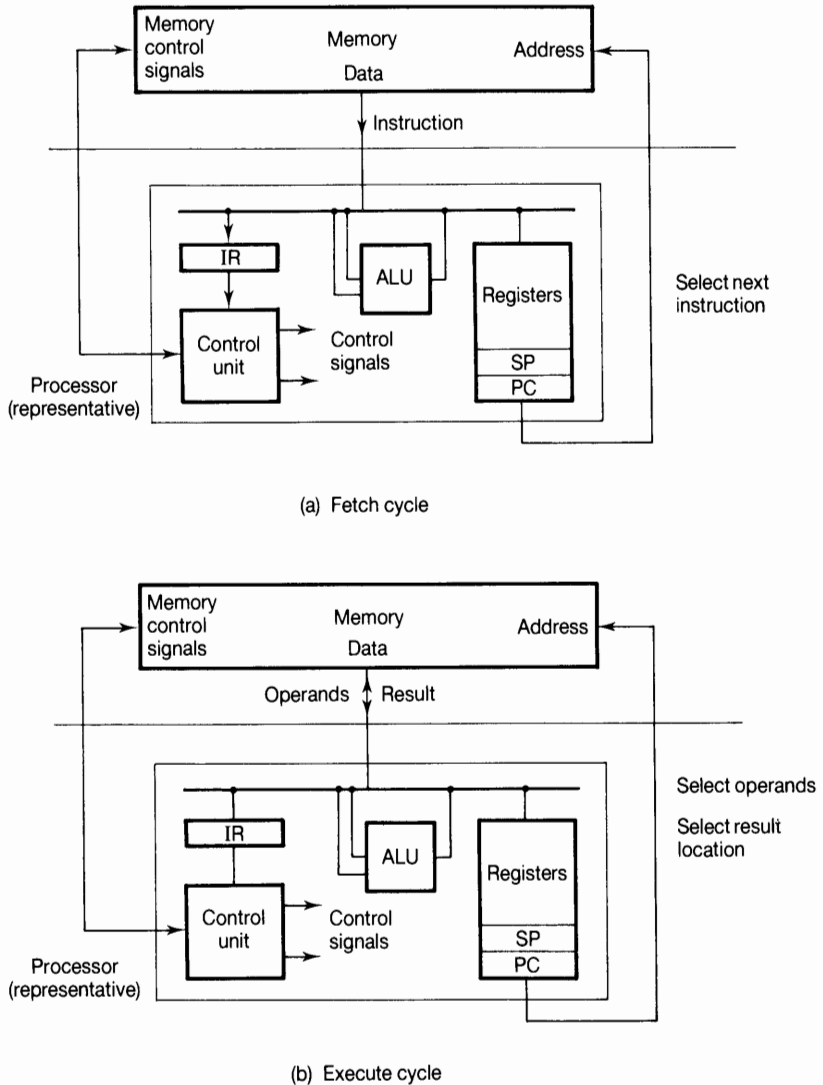


Figure 1.3 CPU mode of operation (a) Fetch cycle (b) Execute cycle (SP, stack pointer; PC, program counter; IR, instruction register; ALU, arithmetic and logic unit)

The operations required to execute (and fetch) an instruction can be divided into a number of sequential steps performed by the control unit of the processor. The control unit can be designed using interconnected logic gates and counters to generate the required signals (a random logic approach). Alternatively, each step could be binary-encoded into a *microinstruction*. A sequence of these microinstructions is formed for each machine instruction and is then stored in a control memory within the internal control unit of the processor. The sequence of microinstructions is known as a *microprogram* (or *microcode*) and one sequence must be executed for each machine instruction read from the main memory. This technique was first suggested by Wilkes in the early 1950s (Wilkes, 1951) but was not put into practice in the design of computers until the 1960s, mainly because the performance was limited by the control memory, which needs to operate much faster than the main memory. Given a control memory with alterable contents, it is possible to alter the machine instruction set by rewriting the microprograms; this leads to the concept of *emulation*. In emulation, a computer is microprogrammed to have exactly the same instruction set as another computer, and to behave in exactly the same manner, so that machine instruction programs written for the emulated computer will run on the microprogrammed computer.

The general arrangement of a microprogrammed control unit is shown in Figure 1.4. An instruction is fetched into an instruction register by a standard instruction fetch microprogram. The machine instruction “points” to the first microinstruction of the microprogram for that machine instruction. This microinstruction is executed, together with subsequent microinstructions for the machine instruction. The sequence can be altered by conditions occurring within or outside the processor. In particular, microprogram sequences of conditional jump instructions may be altered by conditions indicated in a processor *condition code register*. Also, subroutine microinstructions can be provided to reduce the size of the microprogram. Just as a stack is used to hold the return address of machine instruction subroutines, a control memory stack

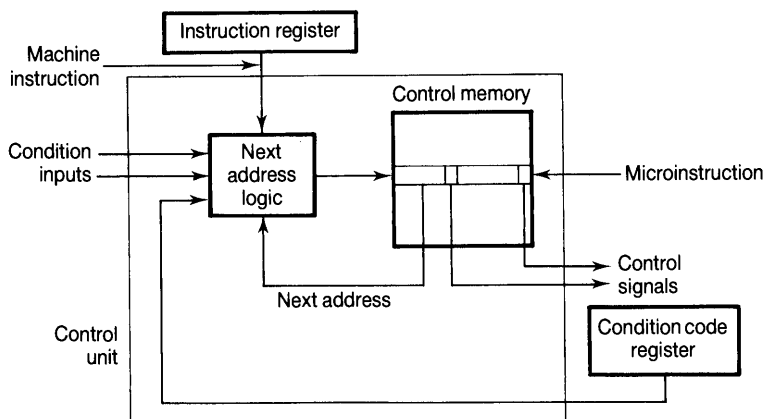


Figure 1.4 Microprogrammed control unit

10 Computer design techniques

can be provided to hold the return address of a microinstruction subroutine return. The microinstructions can have one bit for each signal to be generated, binary-encoded fields, or a combination. A two-level approach is also possible, in which a short microinstruction points to a set of much longer nanoinstructions held in another control memory.

To summarize, we can identify the main operating characteristics of the stored program computer as follows:

1. Only elementary operations are performed (e.g. arithmetic addition, logical operations).
2. The user (programmer) selects operations to perform the required computation.
3. Encoded operations are stored in a memory.
4. Strict sequential execution of stored instructions occurs (unless otherwise directed).
5. Data may also be stored in the same memory.

The reader will find a full treatment of basic computer architecture and organization in Stallings (1987) and Mano (1982).

1.1.2 Improvements in performance

Since the 1940s the development of stored program computer systems has concentrated upon three general areas:

1. Improvements in technology.
2. Software development.
3. Architectural enhancements.

Improvements in technology, i.e. in the type of components used and in fabrication techniques, have led to dramatic increases in speed. Component speeds have typically doubled every few years during the period. Such improvements are unlikely to continue for electronic components because switching times now approach the limit set by the velocity of electrical signals (about $2/3$ speed of light 0.2 m ns^{-1}) and the delay through interconnecting paths will begin to dominate. In fact, this limit has been recognized for some time and has led some researchers to look at alternative technologies, such as optical technology (optical computers).

After the overall design specification has been laid down and cost constraints are made, one of the first decisions made at the design stage of a computer is in the choice of technology. This is normally between TTL/CMOS (transistor-transistor logic/complementary metal oxide semiconductor) and ECL (emitter-coupled logic) for high performance systems. Factors to be taken into account include the availability

of very large scale integration (VLSI) components and the consequences of the much higher power consumption of ECL. ECL has a very low level of integration compared to CMOS but has still been chosen for the highest performance systems because, historically, it is much faster than MOS (metal oxide semiconductor). Predictions need to be made as to the expected developments in technology, especially those developments that can be incorporated during the design phase of the system. For example, it might be possible to manufacture a chip with improved performance, if certain design tolerances are met (see Maytal *et al.*, 1989).

A computer system can be characterized by its instruction execution speed, the internal processor cycle time or clock period, the capacity and cycle time of its memory, the number of bits in each stored word and by features provided within its instruction set among other characteristics. The performance of a high performance computer system is often characterized by the basic speed of machine operations, e.g. millions of operations per second, MOPS (or sometimes millions of instructions per second, MIPS). These operations are further specified as millions of floating point operations per second, MFLOPS, or even thousands of MFLOPS, called gigaflops, GFLOPS, especially for large, high performance computer systems. A computer is considered to be a *supercomputer* if it can perform hundreds of millions of floating point operations per second (100 MFLOPS) with a word length of approximately 64 bits and a main memory capacity of millions of words (Hwang, 1985). However, as technology improves, these figures need to be revised upwards. A Cray X-MP computer system, one of the fastest computer systems developed in the early 1980s, has a peak speed of about 2 GFLOPS. This great speed has only been achieved through the use of the fastest electronic components available, the most careful physical design (with the smallest possible distances between components), very high speed pipelined units with vector processing capability (see discussion, page 138 and Chapter 4), a very high speed memory system and, finally, multiple processors, which were introduced in the Cray X-MP and the Cray 2 after the single processor Cray 1.

The internal *cycle time (clock period)* specifies the period allotted to each basic internal operation of the processor. In some systems, notably microprocessor systems (see page 12), the clock frequency is a fundamental figure of merit, especially for otherwise similar processors. A clock frequency of 10 MHz would correspond to a clock period of 100 ns. If one instruction is completed after every 100 ns clock period, the instruction rate would be 10 MOPs. This would be the peak rate. One or more periods may be necessary to fetch an instruction and execute it, but very high speed systems can generate results at the end of each period by using pipelining and multiple unit techniques. The Cray X-MP computer had a 9.5 ns clock period in 1980 and finally achieved its original design objective of an 8.5 ns clock period in 1986, by using faster components (August *et al.*, 1989). Each subsequent design has called for a shorter clock period, e.g. 4 ns and 1 ns for the Cray 2 and Cray 3, respectively. Other large “mainframe” computer systems have had cycle times/clock periods in the range 10–30 ns. For example, the IBM 308X, first delivered in 1981, had a cycle time of 26 ns (later reduced to 24 ns) using TTL circuits mounted on

12 Computer design techniques

ceramic thermal conduction modules. The IBM 3090, a development of the 3080 with faster components, first introduced in 1985, had a cycle time of 18.5 ns (Tucker, 1986).

Software development, i.e. the development of programming techniques and the support environment, have included various high level languages such as PASCAL and FORTRAN and complex multitasking operating systems for controlling more than one user on the system. Some developments in software have led to variations in the internal design of the computer. For example, computers have been designed for the efficient handling of common features of high level languages by providing special registers or operating system operations in hardware. Most computer systems now have some hardware support for system software.

In this text we are concerned with architectural developments, i.e. developments in the internal structure of the computer system to achieve improved performance. Such developments will be considered further in the next section. First though, let us examine the most striking technological development in recent years – the development of the microprocessor – as this device is central to the future development of multiprocessor systems, particularly those systems with large numbers of processors.

1.2 Microprocessor systems

1.2.1 Development

Since the late 1960s, logic components in computer systems have been fabricated on integrated circuits (chips) to achieve high component densities. Technological developments in integrated circuits have produced more logic components in a given area, allowing more complex systems to be fabricated on the integrated circuit, first in small scale integration (SSI, 1 to 12 gates) then medium scale integration (MSI, 12 to 100 gates), large scale integration (LSI, 100 to 1000 gates), through to very large scale integration (VLSI, usually much greater than 1000 gates). This process led directly to the microprocessor, a complete processor on an integrated circuit. The early microprocessors required the equivalent of large scale integration.

Later integration methods are often characterized by the applied integrated circuit design rules specifying the minimum features, e.g. 1.25 μm and then 0.8 μm line widths. Smaller line widths increase the maximum number of transistors fabricated on one integrated circuit and reduce the gate propagation delay time. The number of transistors that can be reasonably fabricated on one chip with acceptable yield and 1.25 μm design rules is in excess of one million, but this number is dependent upon the circuit complexity. Repetitive cells, as in memory devices, can be fabricated at higher density than irregular designs.

Microprocessors are often manufactured with different guaranteed clock frequencies, e.g. 10 MHz, 15 MHz or 20 MHz. There is a continual improvement in the

clock frequencies due to an improved level of component density and the attendant reduced gate propagation delay times. By increasing the clock frequency the processor immediately operates more quickly, and in direct proportion to the increase in clock frequency, assuming that the main memory can also operate at the higher speed. The choice of clock frequency is often closely related to the speed of available memory.

Microprocessors are designated 4-bit, 8-bit, 16-bit, 32-bit or 64-bit depending upon the basic unit of data processed internally. For example, a 32-bit microprocessor will usually be able to add, subtract, multiply or divide two 32-bit integer numbers directly. A processor can usually operate upon smaller integer sizes in addition to their basic integer size. A 32-bit microprocessor can perform arithmetic operations upon 8-bit and 16-bit integers directly. Specific machine instructions operate upon specific word sizes. An interesting computer architecture not taken up in microprocessors (or in most other computer systems), called a *tagged architecture*, uses the same instruction to specify an operation upon all allowable sizes of integers. The size is specified by bits (a tag) attached to each stored number.

The first microprocessor, the Intel 4004, introduced in 1971, was extremely primitive by present standards, operating upon 4-bit numbers and with limited external memory, but it was a milestone in integrated circuits. Four-bit microprocessors are now limited to small system applications involving decimal arithmetic, such as pocket calculators, where 4 bits (a *nibble*) can conveniently represent one decimal digit. The 4004 was designed for such applications and in the ensuing period, more complex 8-bit, 16-bit and 32-bit microprocessors have been developed, in that order, mostly using MOS integrated circuit technology. Binary-coded decimal (BCD) arithmetic is incorporated into these more advanced processors as it is not subject to rounding, and is convenient for financial applications.

Eight-bit microprocessors became the standard type of microprocessor in the mid-1970s, typified by the Intel 8080, Motorola MC6800 and Zilog Z-80. At about this time, the microprocessor operating system CP/M, used in the 8080 and the Z-80, became widely accepted and marked the beginning of the modern microprocessor system as a computer system capable of being used in complex applications.

Sixteen-bit microprocessors started to emerge as a natural development of the increasing capabilities of integrated circuit fabrication techniques towards the end of the 1970s, e.g. the Intel 8086 and Motorola MC68000, both introduced in 1978. Subsequent versions of these processors were enhanced to include further instructions, circuits and, in particular, memory management capabilities and on-chip cache memory (see pages 18–20 and Chapters 2 and 3). In the Intel 8086 family, the 80186 included additional on-chip circuits and instructions and the 80286 included memory management. In the Motorola family, the MC68010 included memory management. Thirty-two bit versions also appeared in the 1980s (e.g. the Intel 80386 with paged memory management, the Motorola MC68020 with cache memory and the MC68030 with instruction/data cache memories and paged memory management). In 1989 the 64-bit Intel 80486 microprocessor was introduced.

Floating point numbers can be processed in more advanced microprocessors by additional special processors intricately attached to the basic microprocessor,

though a floating point unit can also be integrated into the processor chip. Floating point numbers correspond to real numbers in high level languages and are numbers represented by two parts, a mantissa and an exponent, such that the number = mantissa \times base^{exponent}, where the base is normally two for binary representation. For further details see Mano (1982).

1.2.2 Microprocessor architecture

The basic architecture of a microprocessor system is shown in Figure 1.5, and consists of a microprocessor, a semiconductor memory and input/output interface components all connected through a common set of lines called the *bus*. The memory holds the program currently being executed, those to be executed and the associated data. There would normally be additional secondary memory, usually disk memory and input/output interfaces are provided for external communication. The bus-based architecture is employed in all microprocessor systems, but microprocessor systems were not the first or only computer systems to use buses; the PDP 8E minicomputer, introduced in 1971, used a bus called the Omnibus and the PDP 11, first introduced in 1970, used a bus called Unibus. The expansibility of a bus structure has kept the technique common to most small and medium size computer systems.

The bus is the communication channel between the various parts of the system, and can be divided into three parts:

1. Data lines.
2. Address lines.
3. Control lines.

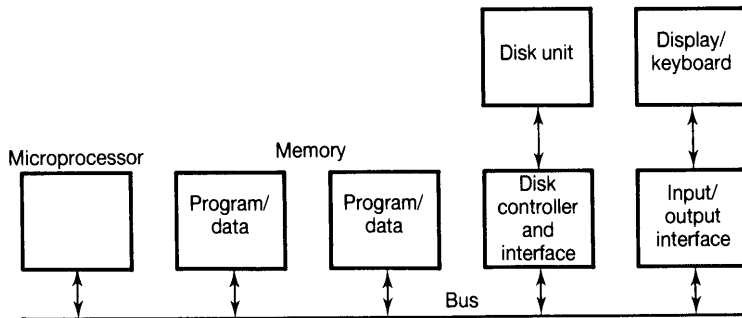


Figure 1.5 Fundamental parts of a microprocessor system

The data lines carry (1) the instructions from the memory to the processor during each instruction fetch cycle, and (2) data between the processor and memory or input/output interfaces during instruction execute cycles, dependent upon the instruction being executed. Eight-bit microprocessors have eight data lines, 16-bit microprocessors have sixteen data lines (unless eight lines are used twice for each 16-bit data transfer, as in some 16-bit microprocessors). Similarly, 32-bit microprocessors have thirty-two data lines, unless reduced by the same technique. Notice that the microprocessor bit size – 8-bit, 16-bit, 32-bit or whatever – does not specify the number of data lines. It specifies the basic size of the data being processed internally and the size of the internal arithmetic and logic unit (ALU).

The instructions fetched from memory to the processor comprise one or more 8-bit words (*bytes*), or one or more 16-bit words, depending upon the design of the microprocessor. The instructions of all 8-bit microprocessors have one or more bytes, typically up to five bytes. One byte is provided for the operation including information on the number of subsequent bytes, and two bytes each for each operand address when required. Sixteen/32-bit microprocessors can have their instructions in multiples of bytes or in multiples of 16-bit words, generally up to 6 bytes or three words.

When the data bus cannot carry the whole instruction in one bus cycle, additional cycles are performed to fetch the remaining parts of the instruction. Hence, the basic instruction fetch cycle can consist of several data bus transfers, and the timing of microprocessors is usually given in terms of bus cycles. Similarly, the operands (if any) transferred during the basic execute cycle may require several bus cycles. In all, the operation of the microprocessor is given in read and write bus transfer cycles, whether these fetch instructions or transfer operands/results.

During a bus cycle, the bus transfer might be *to* the processor, when an instruction or data operand is fetched from memory or a data operand is read from an input/output interface, or *from* the processor, to a location in the memory or an output interface to transfer a result. Hence, the data lines are bidirectional, though simultaneous transfers in both directions are impossible and the direction of transfer must be controlled by signals within the control section of the bus.

The address lines carry addresses of memory locations and input/output locations to be accessed. A sufficient number of lines must be available to address a large number of memory locations. Typically, 8-bit microprocessors in the 1970s provided for sixteen address lines, enabling 2^{16} (65 536) locations to be specified uniquely. More recent microprocessors have more address lines, e.g. the 16-bit 8086 has twenty address lines (capable of addressing 1 048 576 bytes, i.e. 1 megabyte), the 16-bit 80286 and MC68000 have twenty-four (capable of addressing 16 megabytes) and the 32-bit MC68020, MC68030 and 80386 have thirty-two (capable of addressing 4294 megabytes, i.e. 4 gigabytes).

The control lines carry signals to activate the data/instruction transfers and other events within the system; there are usually twelve or more control lines. The control signals, as a group, indicate the time and type of a transfer. The types of transfer include transfers to or from the processor (i.e. read or write) and involve memory and input/output interfaces which may be differentiated.

1.3 Architectural developments

1.3.1 General

There have been many developments in the basic architecture of the stored program computer to increase its speed of operation. Most of these developments can be reduced to applying parallelism, i.e. causing more than one operation to be performed simultaneously, but significant architectural developments have also come about to satisfy requirements of the software or to assist the application areas. A range of architectural developments has been incorporated into the basic stored program computer without altering the overall stored program concept. In general, important architectural developments can be identified in the following areas:

1. Those concerned with the processor functions.
2. Those concerned with the memory system hierarchy.
3. Those around the processor–memory interface.
4. Those involving use of multiple processor systems.

Let us briefly review some of these developments, which will be presented in detail in the subsequent chapters.

1.3.2 Processor functions

As we have noted, the operation of the processor is centered on two composite operations:

1. Fetching an instruction.
2. Executing the fetched instruction.

First, an instruction is read from memory using the program counter as a pointer to the memory location. Next, the instruction is decoded, that is, the specified operations are recognized. In the fetch/execute partition, the instruction decode occurs during the latter part of the fetch cycle and once the operation has been recognized, the instruction can be executed. The operands need to be obtained from registers or memory at the beginning of the execute cycle and the specified operation is then performed on the operands. The results are usually placed in a register or memory location at the end of the execute cycle.

The execution of an instruction and the fetching of the next instruction can be performed simultaneously in certain circumstances; this is known as instruction *fetch/execute overlap*. The principal condition for success of the instruction fetch/execute overlap is that the particular instruction fetched can be identified before the previous instruction has been executed. (This is the case in sequentially executed instruc-

tions. However, some instructions will not be executed sequentially, or may only be executed sequentially after certain results have been obtained.)

The two basic cycles, fetch and execute, can be broken down further into the following three steps which, in some cases, can be overlapped.

1. Fetch instruction.
2. Decode instruction and fetch operands.
3. Execute operation.

The execute operation can be broken into individual steps dependent upon the instruction being executed. Simple arithmetic operations operating upon integers may only need one step while more complex operations, such as floating point multiplication or division, may require several steps.

In high speed processors the sequence of operations to fetch and decode, and the steps to execute an instruction, are performed in a *pipeline*. In general, a pipeline consists of a number of stages, as shown in Figure 1.6, with each stage performing one sequential step of the overall task. Where necessary, the output of one stage is passed to the input of the next stage. Information required to start the sequence enters the first stage and results are produced by the final (and sometimes intermediate) stage.

The time taken to process one complete task in the pipeline will be at least as long as the time taken when one complex homogeneous functional unit, designed to achieve the same result as the multistage pipeline, is used. However, if a sequence of identical operations is required, the pipeline approach will generate results at the rate at which the inputs enter the pipeline, though each result is delayed by the processing time within the pipeline. For sequential identical operations, the pipeline could be substantially faster than one homogeneous unit.

Clearly, instruction operations are not necessarily identical, nor always sequential and predictable, and pipelines need to be designed to cope with non-sequential, dissimilar operations. Also, it is not always possible to divide a complex operation into a series of sequential steps, especially into steps which all take the same length of time. Each stage need not take the same time, but if the times are different, the pipeline must wait for the slowest stage to complete before processing the next set of inputs. However, substantial speed-up can be achieved using the pipeline technique and virtually all computer systems, even modern microprocessors, have a

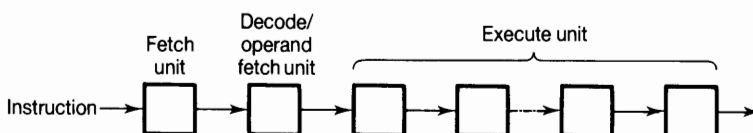


Figure 1.6 Processor pipeline

18 Computer design techniques

pipeline structure (Chapter 4 deals with pipelining and pipelined processors in detail).

The design of the processor involves the implementation of a defined instruction set which, over the years, has become more and more complex as additional instructions have been added to match the requirements of the software more closely. It is assumed that complex operations can be performed more quickly in hardware than in software, which is certainly true. However, the introduction of complex instructions often impacts on simpler ones, and could slow down the operation of these simpler instructions. Hence, the choice of instructions is a major design decision and one school of thought believes that having a limited number of instructions available in the instruction set leads to increased overall system performance, as the processor can then be designed to operate faster. It is found that better use of the instructions can be made by an optimizing compiler. Systems with a limited number of instructions, perhaps less than 128 instructions, are known as *reduced instruction set computers* or RISCs (Chapter 5 is devoted to such computers). One of the aspects of reduced instruction set computers, particularly the early prototype RISCs, is their use of a simple pipeline. Other features include the use, where possible, of registers, for greatest speed, and a very limited number of memory addressing modes. Those systems which attempt to provide as many operations as possible in the hardware often have 100–300 instructions and are called *complex instruction set computers* (CISCs).

1.3.3 Memory hierarchy

The external memory system so far described consists of the main, random access, memory supported by non-random access secondary memory, the latter usually being based upon magnetic technology. Various types of magnetic memory may be present, including exchangeable magnetic floppy disk memory, Winchester magnetic disk memory and magnetic tape memory. Optical disk technology offers vast capacity for large amounts of data in one unit, and becomes another level in the memory hierarchy. A substantial part of architectural enhancements is concerned with making this memory hierarchy easy and efficient to use, and assisting multi-programming.

Multiprogramming is the term used to describe system programming when more than one user program is executed, in effect concurrently, by executing parts of each program in sequence. In the presence of a memory hierarchy it is necessary to transfer programs from the secondary memory into the main memory before the program code can be executed. Similarly, data must be transferred into the main memory before being read or altered. Since only a limited amount of space is available in the main memory, programs or data not immediately required may need to be transferred out of the main memory and stored in the secondary memory until required. Moving programs or data into and out of the main memory requires a memory management scheme, preferably one which is hidden from the user and

activated automatically.

The principal memory management method is known as a *virtual memory scheme*. This creates an automatic mechanism for arranging that data or program code is in the main memory, ready for execution, without the programmer having to program the main memory/secondary memory transfers. The stored information is divided into fixed (or variable) sized blocks which are moved between the main and secondary memories. The operand addresses used within the programs are not altered, but a hardware translation mechanism is in place to translate the addresses as they are generated by the processor so that they refer to the actual memory locations. The scheme is a significant and widely used architectural development (Chapter 2 deals with memory management in detail).

1.3.4 Processor–memory interface

The processor–memory interface is concerned with:

1. Carrying instructions from the memory to the processor during instruction fetch.
2. Carrying data between the processor and memory during instruction execution.

Naturally, it is important that the transfer of instructions/data takes place at least as fast as the information can be digested by the destination (processor or memory). It is not necessary to exceed this requirement and, when systems are being designed, attempts are made to match the processor and memory data rates. If the maximum processor and memory information transfer rates are different, the system speed may be constrained by the slower device. In general, a first-in first-out buffer can be used to link two units operating with transfer rates that can vary.

The processor can normally be designed to accept and produce information faster than the main memory system can produce or accept information. A technique for matching the speeds of the two parts is to introduce a small, very high speed memory, called a *cache*, between the processor and main memory, as shown in Figure 1.7. Program instructions and data are first loaded into the cache and then accessed by the processor. Assuming that the information is required more than once, which is usually true of program instructions, substantial improvements in overall speed can be achieved (Chapter 3 considers the design of cache memory systems).

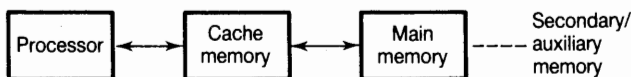


Figure 1.7 Cache memory

20 Computer design techniques

A general technique for increasing the effective rate of a slow unit is to duplicate the unit and make more than one transfer simultaneously. For example, if there is a factor of eight in access time between the source and destination speeds, such that the destination can accept information at eight times the rate that the source can generate information, then the source could be replicated so that there are eight units, and eight transfers could be made to the destination simultaneously. Generally an eight-stage buffer would be used to hold the information before it could be accessed in sequence (or out of sequence) by the destination.

Typically, this technique would be used to match the speed of memory to the speed of the processor in those systems in which the memory operates at a substantially slower speed than the processor. For example, suppose instructions are fetched from memory one at a time. If two memory modules are provided, with separate data paths to the processor, two instructions could be fetched from the memory simultaneously, increasing the transfer rate by a factor of two. The same double word address is sent to both memory modules. If a factor of n increase in transfer rate is required, then n memory modules could be provided. There can be any number of memory modules, each with a separate data path, though in most applications a number which is a power of two would be used. Linear memory addressing can be maintained with part of the full address specifying the n -word address. However, this particular scheme constrains transfers to be blocks of sequential locations and only works effectively if all (or most) items transferred are actually required; in essence, we are increasing the memory word length. The technique can be applied in cache systems to reduce the main memory/cache memory transfer times by matching the speed of the cache and main memory.

It is possible to send different addresses to each memory module so that n unrelated locations in different memory modules can be accessed simultaneously. This scheme is known as *memory interleaving*. Memory interleaving is a fundamental architectural technique applicable to various systems described in Chapters 2, 3 and 4. It is important to differentiate between wide word length memory transfers and true memory interleaving. In the former, a block of consecutive locations can be accessed simultaneously and in the latter, locations not in order can be accessed simultaneously, if these locations are in different memory modules.

In memory interleaving, we divide the memory address field into two parts, one to select the memory module and the other to select the location within the memory module. The memory module can be selected by either the least or the most significant bits; the latter is known as *low order interleaving* and the former as *high order interleaving*. These two alternative address formats are shown in Figure 1.8. Low order interleaving is suitable for single processor interleaved memory, so that consecutive memory locations are in different memory modules and can be accessed simultaneously. With, for instance, four modules and low order interleaving, the first module addresses would be 0, 4, 8, 12, 16, ..., the addresses in the second module would be 1, 5, 9, 13, 17, ..., those in the third module, 2, 6, 10, 14, 18, ..., and in the fourth module, 3, 7, 11, 15, 19, In general, with n -way low order interleaving, the first module addresses would be 0, n , $2n$, $3n$, ..., and the same stride

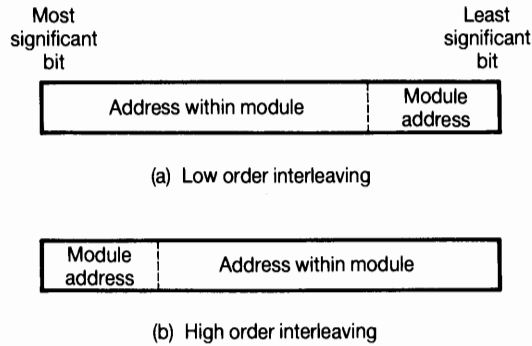


Figure 1.8 Address formats for interleaving (a) Low order interleaving (b) High order interleaving

in the other modules. (Normal memory systems divided into memory modules use the high order format, but only one module is addressed at a time and there is no overlap between memory operations.)

Figure 1.9 shows an interleaved memory organization for a single processor system using a separate data bus for each memory module to the processor. The memory module addresses select the modules and the module word addresses generated are loaded into address buffers in succession. As each address is loaded, the module can proceed to identify the location and provide read or write access. For read, data appear on each of the data buses in succession after the memory access time has elapsed. For write, the data is produced by the processor on the data

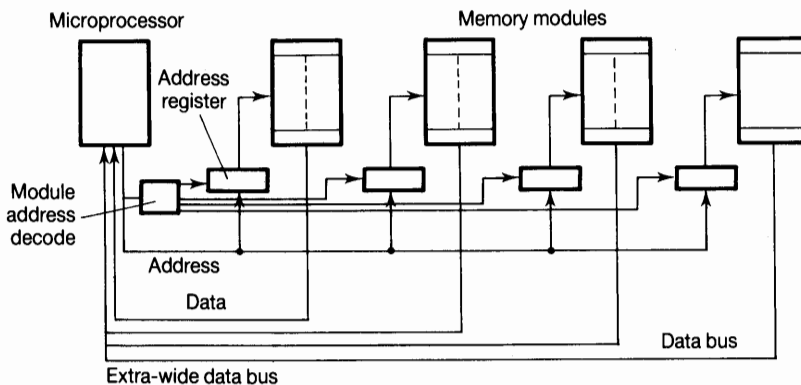


Figure 1.9 Memory interleaving with wide data bus

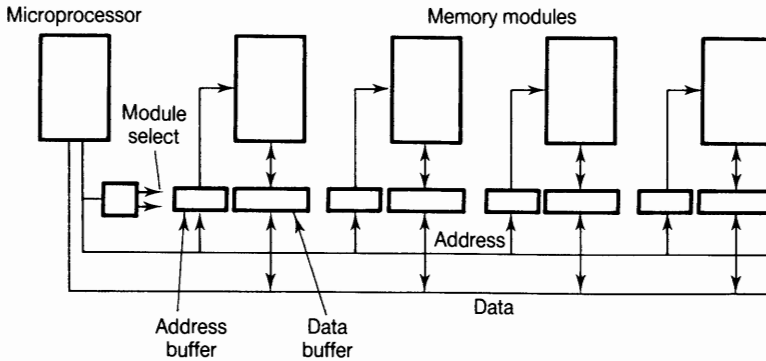


Figure 1.10 Memory interleaving with single bus

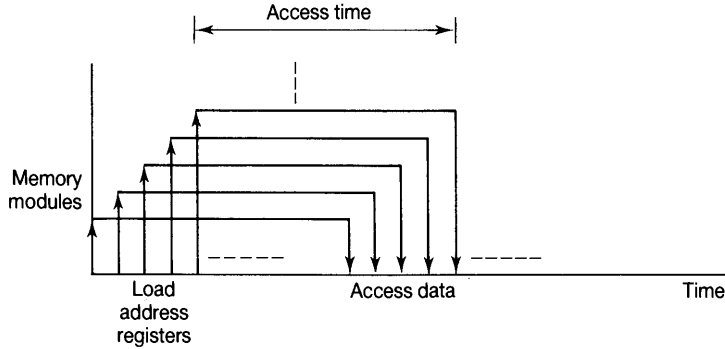
bus and taken in succession by the memory modules. This organization might be suitable for a cache system which is also divided into modules.

A single data bus can be used, as shown in Figure 1.10. It is necessary to provide each memory module with a data buffer register to hold the data to be written into the module or read out. The timing of this system is shown in Figure 1.11(a). It is possible to supply the same address to all modules simultaneously so that consecutive words can be accessed (for the low order interleaved address format). The timing of this is shown in Figure 1.11(b).

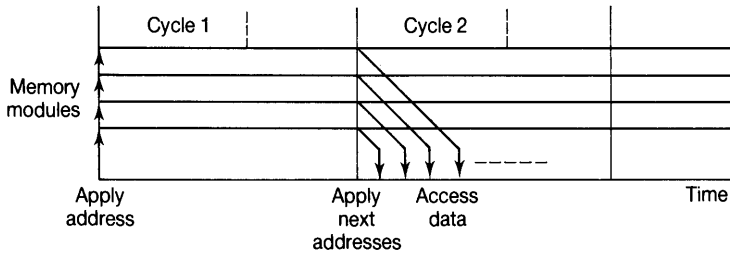
Memory interleaving can be used in a pipelined system, as described in Chapter 4, to fetch more than one instruction simultaneously. After these instructions are fetched, each one is executed in sequence. The interleaving releases the processor–memory interface for subsequent operand accesses.

1.3.5 Multiple processor systems

The application of more than one processor working in a coherent manner on a single task within a single computer system has been studied since at least 1960 (Conway, 1963) and is an obvious method of increasing the speed of the system. One would expect that if n processors worked continuously and simultaneously on a single problem, the results would be obtained n times faster than if one processor was applied to the problem. However, it is not always easy to partition a problem so that n processors can operate simultaneously, and even if this were possible, an interprocessor communication overhead generally exists in the multiprocessor version. But multiprocessor solutions are necessary to achieve substantial increases in speed over existing high speed single processor systems, and a large part of this text is devoted to the design of multiprocessor systems, particularly the possible architectural arrangements that could be employed for coupling the processors.



(a) Different addresses to each memory module



(b) Same address to each memory module

Figure 1.11 Interleaved memory timing (a) Different addresses to each memory module (b) Same address to each memory module

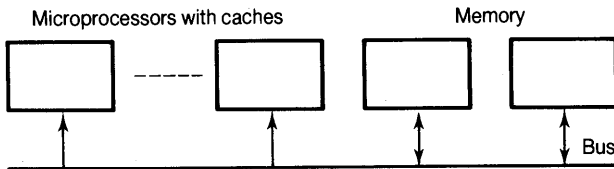


Figure 1.12 Single bus multiple microprocessor system

Multiprocessor systems can be developed from a single processor system by simply adding one or more processors, all of which share the same memory, resulting in the so-called *shared memory multiprocessor system*. Each processor might have local memory, but would use the shared memory to pass information

between processors and to obtain shared programs/information – Part II of this text is largely devoted to such multiprocessor systems. In a bus-based system, additional processors can be added to the bus, as shown in Figure 1.12 – this approach has been taken by microprocessor system designers and will be discussed in Chapter 7. Generally, the approach is only suitable for small numbers of processors because of bus and memory contention, though multiple buses and a hierarchy of buses can be used to extend the system.

Alternatively, multiprocessor systems can use direct connections between processing elements. This approach leads to multiple processors which operate independently, passing information to other processing elements perhaps through a message-passing protocol, rather than through a shared memory system. Message-passing multiprocessor systems are easier to expand to large numbers of processors and are more suitable for VLSI fabrication with large numbers of processors. They do not suffer from the problems of maintaining consistency between the shared and local memories or from the problems of controlling access to shared information. Such multiprocessor systems are considered in Part III, together with dataflow computer systems.

1.3.6 Performance and cost

Cost is a major factor in design decisions and there are trade-offs between increased architectural (and technological) complexities and increased cost. There are often diminishing returns for added complexity and a point is reached when the added costs cannot be justified. For example, a cache memory inserted between the processor and main memory can substantially improve the performance and the larger the cache, the greater the improvement. However, a point is reached when the increased cost of further cache memory does not materially improve the performance, or if there *is* a significant improvement, the cost is not justified. There are various ways of organizing a cache (see Chapter 3) and each has different cost and performance implications. Unfortunately, it may not be immediately clear which organization should be chosen, as performance is highly dependent upon the programs being executed, and the choice has to be made after considering likely applications of the system and general program characteristics. In a multiprocessor design (Part II and Part III), the use of more than one processor has to be justified in terms of performance and cost when compared to high speed single processor systems. In a multiprocessor system, a significant factor is the method of interconnecting the processors. It is theoretically possible to connect all the processors together and allow them to communicate simultaneously, but such exhaustive interconnections incur heavy cost penalties and limited interconnection networks (see Chapter 8) need to be evaluated in terms of effect on performance and the cost.

This chapter studies the methods of managing the main and secondary (auxiliary) memory hierarchy in a computer system. These “memory management schemes” relieve the programmer of the problems of ensuring that the required programs are in the main memory for execution. A memory management scheme has been present in virtually all larger computers since the early 1970s. We will consider the two principal memory management schemes – paging and segmentation (and their combination) – together with the hardware requirements for these schemes.

2.1 Memory management schemes

The total memory in a computer system is composed of various memory types, in particular a main random access memory and a secondary, usually non-random access memory (disk memory being called direct access memory).

The main memory must have high speed, random access quality and programs and data must reside here for the processor to access the information (whether instructions or data). Another level of memory – cache memory – may be inserted between the main memory and the processor. (Cache memory is considered separately in Chapter 3.)

The secondary memory usually consists of magnetic disk memory, including exchangeable and non-exchangeable disk systems; other types of secondary memory include magnetic tape systems (sometimes called mass memory). Optical disk systems might be present to hold vast amounts of possibly read-only information.

With several types of memory present, information will reside in the slowest memory when it is not in use, and be brought to the faster secondary memories as its use becomes more imminent. Exchangeable media such as floppy disks would be used as appropriate, for example when programs must be moved from one computer system to another, unattached, computer system.

The memory hierarchy needs a scheme to arrange that the required information is in the main memory when it is to be read or altered by the processor; such schemes are called *memory management* schemes. We will concentrate upon the main

memory–secondary memory interface rather than any strategy for transferring information between different secondary/mass memory devices. When necessary, we will assume that the secondary memory is disk memory.

The simplest memory management scheme is *overlaying* – when programs or sections of programs are transferred into main memory, as required, under program control (by explicit program routines) and overwrite existing programs. This method places a heavy burden on the programmer but early computers in the 1950s and 1960s used it, as do many single-user microprocessor systems, including the operating system MS-DOS (see Duncan, 1986). Overlaying has been automated to some extent in more recent versions of microprocessor operating systems and utilities. For example, the MS-DOS operating system program linker utility, LINK (Microsoft, 1987), provides for semiautomated overlays. LINK can create overlaid programs, specified by the user, in which parts of the program, which are specified as needed, will be loaded during run time and will occupy the same memory space as previously executed programs. Such techniques conserve memory space at the expense of much slower execution.

We note that the magnetic disk memory (usually the first level of secondary memory) operates much more slowly than the semiconductor main memory – at least three orders of magnitude slower than main memory. Data can be accessed in a semiconductor main memory in the range 100–300 ns, while the latency before the required data is even reached on a disk might be in the range 10–30 ms. The gap widens as integrated circuit technology improves and, given that disk memory latency time (time to locate one sector on the disk) and associated data rate are limited by mechanical factors, the transfer rate between the main memory and the disk memory is dictated by the slower device. The processor often cannot continue with the current program while transfers are in progress between the main memory and the disk memory. The processor in a single-user system will be idle and waiting for the transfer to be completed, even though this may be done by a separate direct memory access (DMA) device. Hence, reducing the number of transfers to a minimum is very important to achieve the highest performance.

One apparent solution recognized and suggested at a very early stage in the development of computers (for example by a group at MIT during the period 1957–61) was to provide a very large amount of main memory, sufficient to hold all the programs currently being executed or about to be executed. Though a brute force method, and not really a good technological solution at a time when main memory was a very expensive and valuable resource, the provision of a large amount of main memory has recently become routine and inexpensive and extremely large amounts of main memory may be common in the future. Certain applications, for example some graphics applications, find that a large amount of main memory is better than a smaller amount plus a memory management scheme to transfer information between the main and secondary memories. However, in any application, large main memory usually requires even larger secondary memory and the memory management problem reappears on a larger scale. (The complete programs are more likely to be held in their entirety in the main memory and the memory management mechanism

is active mainly during program load time.)

Given that we have main memory and secondary memory for economic reasons (and also to give media exchangeability), a truly automatic method of transferring blocks of words into and out of the main memory is highly desirable to relieve the burden of programming transfers. The method should take into account the blocks likely to be required in the near future, to reduce disk transfers. Computer systems, particularly in a multiprogramming environment in which many programs, or parts of many programs, reside in the main memory, require an efficient mechanism for handling the storage of various information and also require memory protection mechanisms. These involve preventing specified memory operations on specified parts of the memory, notably the memory holding the operating system and other user programs. Memory management schemes normally incorporate features for memory protection. Hence, we can identify two separate issues for a memory management scheme:

1. Handling the main and secondary memory hierarchy.
2. Providing memory protection.

Memory protection will be considered later in the chapter. First, we will consider the original memory management method – *paging* – which was introduced to handle the memory hierarchy.

2.2 Paging

2.2.1 General

At about the same time as the MIT group was proposing very large main memory, a group at Manchester University (Kilburn *et al.*, 1962) developed a method, originally called a *one-level scheme* and now called *paging*, which has become the standard method of managing the memory hierarchy. The objective of the original one-level scheme was to make the main core memory and secondary drum memories seem as though all the memory was main random access memory, hence the term one-level. The term *virtual memory* is now normally used, as the user is given the impression of a very large main memory space (a virtual memory space) which hides the actual memory space (the real memory space). Separate addresses are used for the virtual memory space and the real memory space. The actual memory addresses are called *real addresses* and the program generated addresses are called *virtual addresses*.

The real and virtual memory spaces are both divided into blocks of words called *pages*. All pages are the same size, which might be between 64 bytes to 4 Kbytes, depending upon the design. The virtual and real memory addresses are each divided into a *page field* and a word within the page field called a *line field*. The processor generates program dependent virtual addresses which assume that all the memory

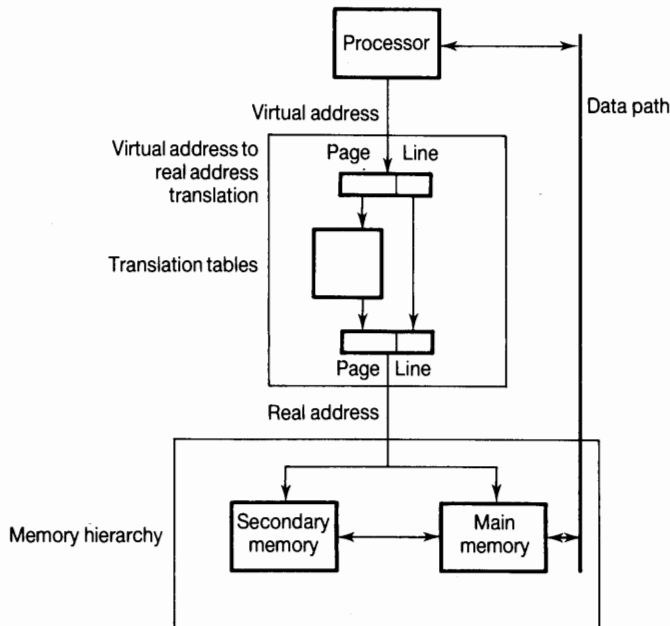


Figure 2.1 Virtual memory system

can be addressed directly. At any instant, each virtual address has a corresponding real address in the physical memory – either in the main memory or in the secondary memory – and *page tables* are maintained to record the correspondence between the virtual and real pages. Each virtual address generated by the processor is translated into the actual address in main memory by reference to a hardware page table which holds all the addresses of the pages currently in the main memory. If the page is not currently in the main memory, a software routine is activated to bring it in automatically, updating the page tables accordingly. Figure 2.1 shows an overall view of a virtual memory system.

Figure 2.2 shows a snap-shot of a system with 32 pages in the main memory and 192 pages in secondary memory (as implemented in the original Atlas computer; nowadays there would be many more pages, but the concept is the same). Nine-bits are allocated to specify the line and 11-bits are used to specify the page in main or secondary memory. Secondary memory page addresses are shown starting at 32. The page table is shown, with a possible distribution of pages in the system.

In the assignment of virtual pages to real pages shown in Figure 2.2, some virtual pages are unassigned (not used). Real page 31, the last page in the main memory, is currently free. Suppose virtual page 3 is requested by the processor. First hardware is activated to check whether the page is in the memory. The hardware finds that the

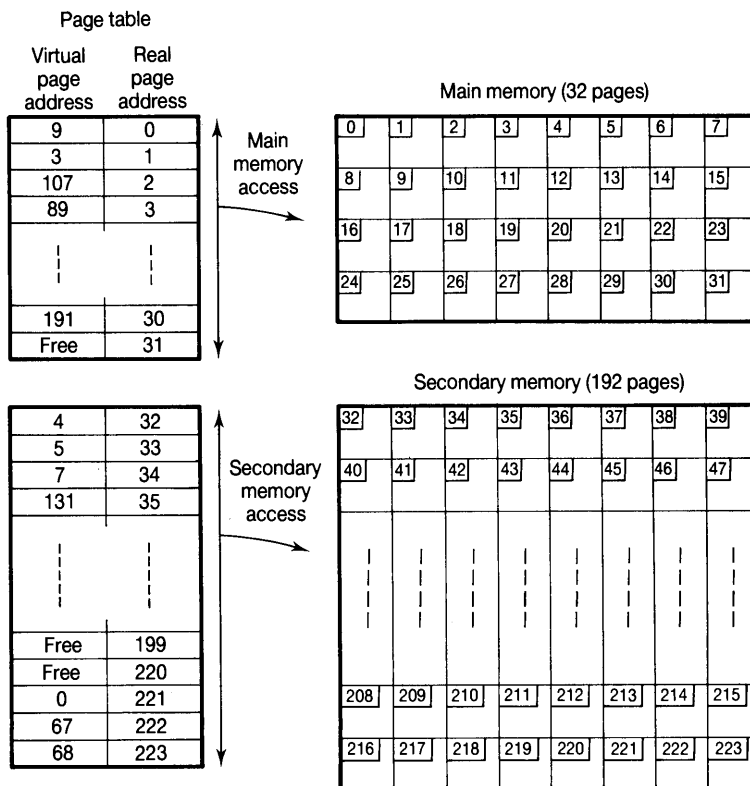


Figure 2.2 Page addresses in a very small virtual memory system

page is currently residing in real page 1 and the page can be referenced. However, suppose virtual page 7 is now requested by the processor. The check shows that the page is not currently residing in the main memory and a software routine is activated to search for its location in the secondary memory. In this case, the page is found in real page 34 in the secondary memory. The page could be transferred into the free (real) page 31 in the main memory, and is subsequently referenced by the processor; now the main memory is full. If a reference is made to another page in the secondary memory, an existing page in the main memory must be returned to the secondary memory before the new page is transferred into the main memory, unless a valid copy is held in the secondary memory. Then the main memory copy can be simply overwritten. Kilburn suggests keeping a vacant page in main memory to allow the transfer to the main memory to take place first, and any writing back to the secondary memory can be done while the processor continues with normal processing.

There are various possible ways of translating the virtual page number into the corresponding real page number which we will discuss later, but they must operate on every memory reference. We hope that the page will usually be found in the main memory and that the high speed hardware translation can be used successfully on main memory pages. We will look at the situation when there are too many pages in the main memory for hardware translation to be used for all main memory pages. Software address translation is used for pages currently in the secondary memory. In addition, if necessary, a *page replacement algorithm* selects a page to be removed from the main memory to make room for the incoming page from secondary memory prior to use.

Though introduced simply to make all the memory look as one, it was also known at the time (Kilburn *et al.*, 1962) that paging allowed an operating system to relocate programs and parts of programs between the main and the secondary memories without altering any of the program addresses. Such relocation is essential and forms the basis of all operating system activities of moving user programs.

Main memory sizes have increased since 1962 and are now often greater than a million bytes. A main memory of 1 048 576 (2^{20}) bytes, with a page size of 512 bytes, would give 2048 pages in the main memory. The secondary memory is normally several orders of magnitude greater than the main memory, and consequently will have a very large number of pages. Page addresses are numbered from zero onwards in the main memory and could continue through to the pages on the secondary memory as shown in Figure 2.2, which was done in the original one-level store. However, in practice, secondary memory normally has its own addressing scheme. For example, a disk memory stores its information arranged on concentric tracks; each track is divided into a number of sectors. One page might be stored on one sector, or on more than one sector, depending upon the size of the page and sector. The disk is usually organized so that one page can be located on the disk surface as one unit addressed in terms of track and sector.

The number of bits in the virtual address is normally specified by the addressing capability of the processor, i.e. by the number of address bits generated by the processor. The number of bits in the virtual and real addresses need not be related except that the number of bits provided to address the words within a page must be the same. The virtual address space is usually much larger than the real address space, in keeping with the original motive of giving the programmer the illusion of a very large main memory. However, it is possible for the two to be the same size, or to have a smaller virtual than real address space. This has been used to expand the addressing capability on computers with a limited addressing capability. For example, a processor with only 16-bit byte addressing (as was the case with early microprocessors) would be limited to using 64 Kbytes of main memory without a virtual memory system or another address translation mechanism. With a virtual address translation with a larger real address, the addressing space could be expanded. The same virtual address might then translate to different real addresses, depending upon the context and program. Whether this mechanism could actually be called virtual memory is debatable. Having the same size for both the virtual and real main

memory address spaces would be quite natural for processors which can generate large virtual addresses.

Instrumental to the success of paging are certain characteristics of programs. If programs were executed purely in sequence, from one memory address onwards, the same instructions never re-executed and the sequence fixed and known, then semiautomatic overlaying might be just as good as paging, especially as some overlays could be brought into main memory before they were needed. However, though code is generally executed sequentially, virtually all programs repeat sections of code and repeatedly access the same or nearby data. This characteristic is embodied in the *principle of locality*, which has been found empirically to be obeyed by most programs and which tends to apply to both instruction references and data references, though it is more likely in instruction references. It has two main aspects:

1. Temporal locality (locality in time) – individual locations, once referenced, are likely to be referenced again in the near future.
2. Spatial locality (locality in space) – references, including the next location, are likely to be near the last reference. (This last characteristic is sometimes separated into a third aspect, known as sequential locality.)

Temporal locality is found in instruction loops, data stacks and variable accesses. Spatial locality describes the characteristic that programs access a number of distinct regions. Sequential locality describes sequential locations being referenced and is a main attribute of program construction. It can also be seen in data accesses, as data items are often stored in sequential locations.

Figure 2.3 shows a histogram of typical page references. References are grouped into particular regions and many, if not all, locations are referenced several times. One region will commonly be for the stack holding procedure return addresses and

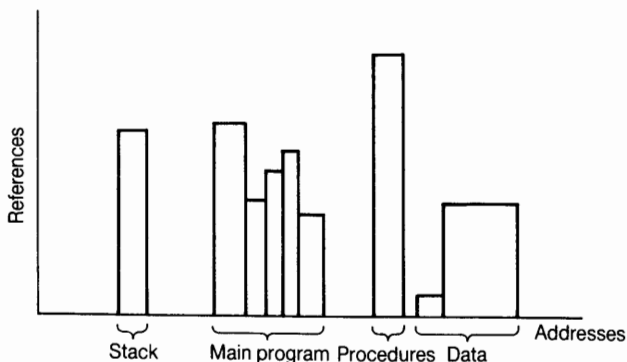


Figure 2.3 Program page references

procedure parameters. Another memory area will be for local variables, which are often stored together. A major area will be for the main program and called procedures might be in other, separate, areas, especially if shared with other programs.

Let us now consider methods of performing the virtual-real address translation. The chosen method must be implemented in hardware as a translation has to be performed on every memory reference, and the translation time directly adds to the overall instruction execution time.

2.2.2 Address translation

There are three basic hardware techniques to translate the virtual page address into a real page address:

1. Direct mapping.
2. Associative mapping.
3. Set-associative mapping.

In all cases, after translation the real page address is concatenated with the line number to form the complete virtual address.

Direct mapping

The *direct mapping* approach is shown in Figure 2.4. All real page addresses are stored in a high speed random access memory page table, in locations whose addresses are the virtual page addresses of the stored real page addresses. Consequently, a real page address can be found directly from the memory by using the virtual page address to address the page table. For example, the virtual page address 34 selects location 34 in the page table, and location 34 holds the corresponding real page address for virtual page address 34. Unfortunately, the direct mapping technique can only be used if the number of pages is relatively small, as there has to be one entry in the page table for each virtual page, even if the page is not in the main memory (in which case the entry must indicate that the page is not present).

The hardware page table need only hold the main memory address if it is present in main memory, with an additional bit to indicate whether the page is in main memory. (In fact, several additional bits are included in the page table, as we shall see later.) If a page is not in the main memory, the page entry in the page table is invalid. The number of bits provided for the page entry need only be sufficient for the number of main memory pages, though the number of entries is still very large. It is, of course, possible to have a page table which holds both the primary and secondary memory page addresses in different fields. The page table for the secondary memory pages is kept in main memory or paged in from the secondary memory. The secondary memory page table is only referenced when the required page is not found in the main memory table.

Though it is possible, the main memory is not usually used to hold the main

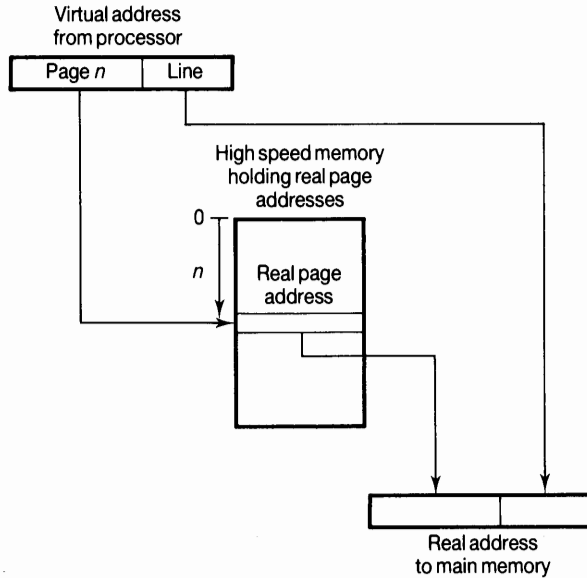


Figure 2.4 Direct mapping address translation

memory page table, as this would double the number of main memory references, i.e. one to access the real page address from the page table and another to reference the actual required location. Instead, as described, dedicated very high speed memory is necessary.

Unfortunately, with a large number of pages, the direct mapped table would be very large, and is generally too expensive. The direct method is also wasteful of high speed memory, as there needs to be one location for each possible virtual address, whether or not the virtual address refers to a page in the main memory. Typically, most of the possible virtual addresses will refer to real addresses in the secondary memory, and hence the entries in the memory table will often show a main memory miss condition using, say, an extra bit associated with each entry.

Associative mapping

The *associative mapping* approach is shown in Figure 2.5. Here both the real page address and the corresponding virtual page address are stored together in a high speed memory. The incoming virtual page is compared with all the stored virtual pages simultaneously and, if a match is found, the real page is read out. Each virtual page entry requires a comparator. The original Kilburn one-level store employed associative mapping with 32 page registers and 32 comparators.

A special type of memory – an *associative* (or *content addressable* (CAM) *memory*) – which incorporates comparators, can be used to store the virtual page

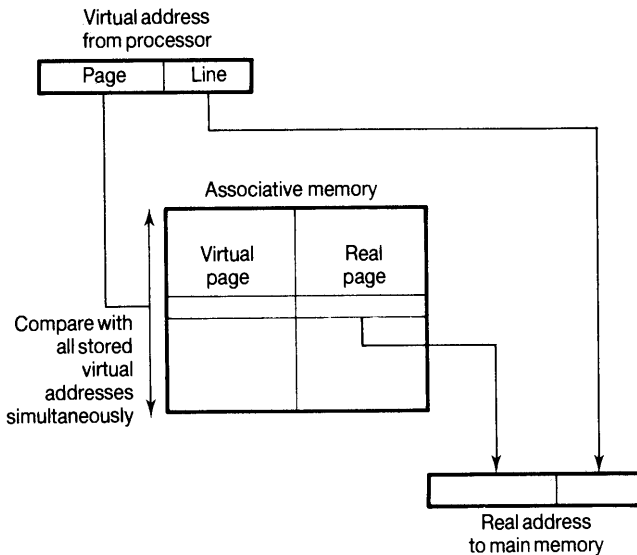


Figure 2.5 Associative mapping address translation

addresses. In such a memory a location is identified by its contents, rather than by an assigned address. CAMs are random access memories with a comparator associated with each stored location. They can be designed to operate at high speed – approaching the speed of high speed random access memory – but they are relatively expensive. When used for associative mapping, the content addressable memory is coupled to normal random access memory, giving two parts to the memory. The CAM section holds the virtual page addresses and the RAM section the real page addresses. When a virtual address is generated by the processor, the virtual page address is compared with all virtual page addresses stored in the CAM simultaneously, using comparison logic within the associative memory. If a match is found, the corresponding real page address held in the RAM part is read out.

In associative mapping, the page table look-up is a two-stage process. First, a comparison is done between the submitted page address and each of the stored page addresses. This process is indivisible; all the comparisons are performed simultaneously in hardware – sequential comparisons would be too slow. The next step depends upon whether a match is found. If a match was found, the real address is obtained and a main memory access occurs. If a match is *not* found, a *page fault* occurs and a page replacement routine is activated.

Set-associative mapping

Set-associative mapping is a combination of direct and associative mapping. In the set-associative method, the virtual addresses and real addresses are divided into a most significant tag field, an index (row) field and a least significant word (offset) field. The corresponding page field often consists of the tag and index fields together. High speed random access memory is organized in blocks, each of which contains 2^i locations, where there are i index bits, as shown in Figure 2.6. Each location holds a virtual tag/real page address pair and the blocks are arranged such that one pair from each block is accessed via the index simultaneously. The virtual tags read are compared with the virtual tag presented by the processor and if a match is found, the corresponding real page is taken and concatenated with the word (offset).

With only one tag/real address at each location, as shown in Figure 2.6, all the virtual page addresses to be translated must have different indices, but there is a fair probability that more than one virtual address will have the same index. A set-associative table can be designed so that there is more than one tag/real page entry at each index, and all the tags can be compared simultaneously. Figure 2.7 shows a set-associative table with two tag/real address entries for each index. The number of entries that can be compared simultaneously is called the *set size* or *associativity*. The associativity is sometimes given as s -way for a set size of s .

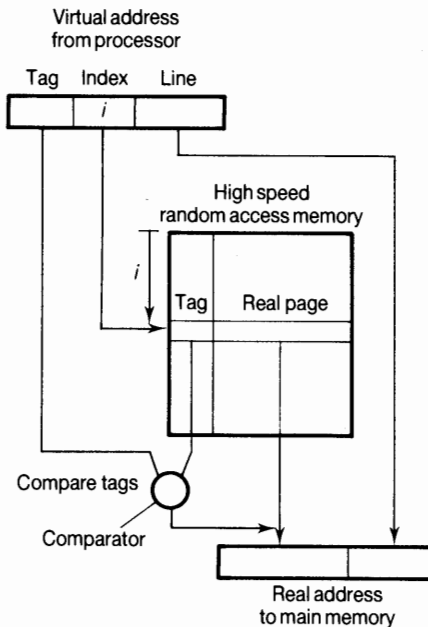


Figure 2.6 Set-associative mapping address translation (one-way)

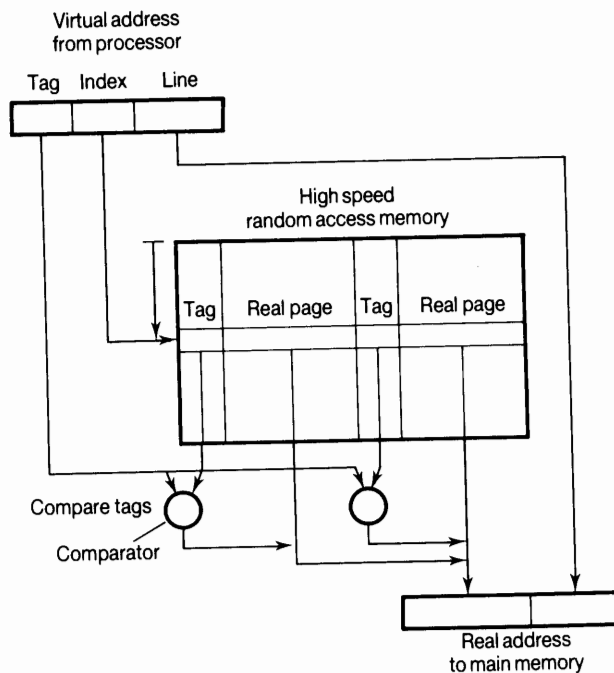


Figure 2.7 Set-associative mapping address translation (two-way)

2.2.3 Translation look-aside buffers

In practice, the number of pages in a modern computer system is too large to employ either the direct or an associative method totally in hardware. Given the program characteristics embodied in the principle of locality, we expect that one particular set of pages (the so-called *working set*) will be referenced until a change of context occurs. Hence, to reduce the hardware requirements without unduly reducing performance, only those page addresses predicted as most likely to be used could be translated in hardware. The rest of the page references are initially handled by reading a main memory page look-up table, and subsequently the high speed hardware page look-up table is updated. The high speed page address translation memory holding the most likely referenced page entries is sometimes known as a *translation look-aside buffer* (TLB) (also called a *translation buffer* (TB), a *directory look-aside table* (DLT) or an *address translation cache*). The predicted most likely pages can be translated in the TLB using either the associative or the set-associative method. The TLB acts very much like a data cache by holding those items most likely to be referenced – hence the term address translation cache. Figure 2.8 shows a system with a TLB.

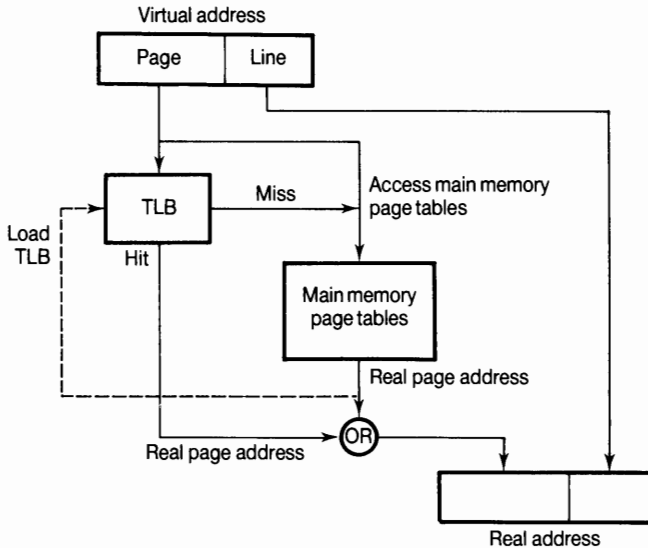


Figure 2.8 Translation look-aside buffer

An example of a set-associative TLB is the translation buffer in the VAX-11/780 which has a two-way 128 word buffer; the VAX-8600 uses a one-way 512 word buffer. The fully associative method is used in some TLBs within microprocessor integrated circuits and specialized integrated circuit TLBs, an example being the MC88200 Cache/Memory Management Unit for the MC88100 reduced instruction set processor (Motorola, 1988b). The MC88200 has two fully associative TLBs. One, the *Block Address Translation Cache*, provides for translating addresses of ten 512 Kbyte blocks used principally for the operating system and other “high-use software”. The other, the *Page Address Translation Cache*, provides for translating addresses of fifty-six 4 Kbyte pages used principally by the user. The virtual address space is divided into equal system and user spaces.

Because the TLBs do not translate all addresses into real addresses, even though the location may be in the main memory, a high speed translation and TLB entry replacement algorithm are necessary for those virtual addresses not translated immediately by the TLB. These actions can be performed in microcode (as in the VAX-11/780) but integrated circuit TLBs have special logic to perform these actions automatically.

The set-associative TLB with an index directly addressing the TLB has the major disadvantage that only n pages with virtual addresses having the same lower page bits (index bits) can be translated with a set size of n . The set size is often only one or two. The chance of virtual addresses having the same lower page bits is quite

high, and the low order TLB entries would be heavily used, especially as virtual pages are likely to be assigned from zero onwards. To counteract this, higher page bits could be used instead of the lower page bits or, alternatively, a mixture of some lower and some higher bits could be used. In a system with user and supervisor address spaces separated by the most significant address bit, it may be advantageous to use the most significant bit in the index, so that the TLB is evenly divided between system and user addresses. This technique is used in the VAX-11/780 translation buffer. In this buffer, the system pages remain when user pages are purged on a task switch.

On some large computer systems (for example IBM 3033 and Amdahl 470), a hardware hashing technique is used to “randomize” the virtual page address before accessing the TLB. Hashing is a general computer technique for converting one number into another (usually one with fewer bits) such that any expected sequence of input numbers will generate different and unique hashed numbers. Various hashing functions are known and some can reasonably be implemented in hardware. These hashing functions are mostly based upon logically exclusive ORing bits in the input number. Figure 2.9 shows the simple hashing function used on the IBM 3033. In this hashing function, two pages with the same initial index only hash into pages with the same index when the upper five page bits used in the hashing are the same as the lower five index bits, and generally, in any page, sequential indices are made non-sequential. (It is left as an exercise to determine when the sequential nature continues through the hashing.)

2.2.4 Page size

Various page sizes are used in paging schemes, from small pages of 64 bytes through to very large pages of 512 Kbytes. Some systems provide for different page

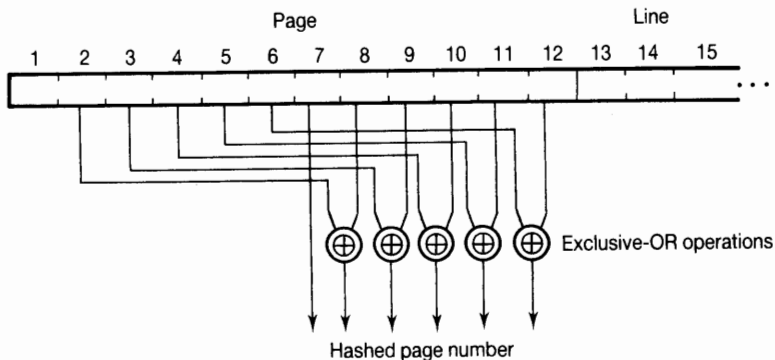


Figure 2.9 IBM 3033 page hashing function

sizes for flexibility, selected by context or by setting bits in registers, and separate page tables might be provided for each page size. For example, there could be different page sizes for data and for code. A small page of 64 bytes might be suitable for code while a larger page of 512 bytes might be suitable for data. Different page sizes might be appropriate for system and user pages. A larger page size might be better for system software, which resides in the main memory. For example, the Motorola MC88200 has 4 Kbyte pages for the users and 512 Kbyte pages for the system software.

If a small page size is chosen, the time taken in transferring a page between the main memory and the secondary memory is short, and a large selection of pages from various programs can reside in the main memory. A small page also reduces the storing of *superfluous* code which is never referenced (e.g. an error routine which is never selected). However, a small page size necessitates a large page table, and *table fragmentation* increases. This is the term used to describe the effect of memory being occupied by mapping tables and hence being unavailable for code/ data.

Conversely, a large page size requires a small page table but the transfer time is generally longer. Unused space at the end of each page is likely to increase – an effect known as *internal fragmentation*; on average, the last page of a program is likely to be 50 per cent full. The magnetic disk secondary memory also constrains the page size to that of a disk sector, or to a multiple of a sector, unless additional sector buffer storage is provided to enable one page from several in a sector to be selected. Making the sector small increases the proportion of recorded information given over to sector identification on the disk. Overall, the number of words in each page has to be chosen as a compromise between the various factors.

2.2.5 Multilevel page mapping

The full page table giving all the virtual/real page associations for the main memory requires considerable memory when the main memory address has, say, 32 bits (a standard for 32-bit microprocessors). In two-level page mapping the virtual address is divided into three fields; a page directory field, a page within a page directory field and a line (or offset) within a page. The page directory field points to an entry in a directory table which gives the start address of the page table for that directory. The page field then selects the start address of the real page which, concatenated to the line, gives the complete real address. Two-level mapping requires more table entries in total than when the directory and page fields are combined into one page field and it has the disadvantage that two table references are required to extract the page address. However, it has the distinct advantages that the individual tables are much smaller and that only some of these tables need reside in the main translation memory simultaneously. It would be reasonable to place the most recently used tables in high speed translation look-aside buffers.

The system can be extended to more than two levels. A tree structure can be formed by a hierarchy of pointers, as shown in Figure 2.10, though two levels of

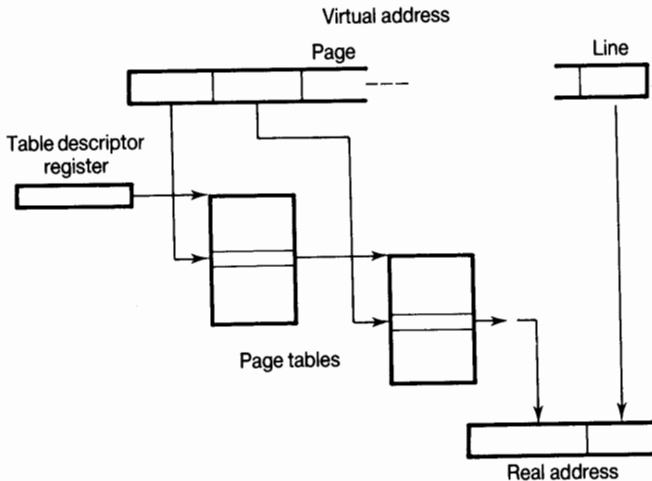


Figure 2.10 Page table organization

tables are often sufficient. Such structures enable easy manipulation of the tables and reduction of the full table by eliminating the page tables of those pages which are not used. The first table in the memory is usually located by a dedicated register, sometimes called a *table descriptor register*.

Paging, as described, does not allow pages to be shared between processes, and yet sharing system software between processes is a common requirement. A paging solution is to divide the virtual address space into two or more distinct spaces, one or more for user programs and one or more for the system software. The most significant bit(s) of the virtual address can be used to select the region, and the rest can be divided into a directory, page and line fields. The system space can use a one-level translation, using both the directory and page fields. The user space can use a large page field, to select an entry in a single page table which holds the real page addresses and which is available to all users. The user space could use two-level translations using different page tables selected by different directory pointers and only available to the associated process.

The VAX-11/780 computer system designers chose this solution partly because it represented an evolution of the earlier PDP-11/70 (the two alternatives considered were true segmentation and *capabilities* (see Strecker, 1978). In the VAX-11/780, the virtual address consists of thirty-two bits. A page size of 512 bytes is used and the least significant nine bits of the virtual address select the location within the page. The two most significant bits of the virtual address are allocated to select one of four regions: the program region (P0), the control region (P1), or one of two system space regions (S0, and one region originally reserved for future use). P0 contains the user program and data. P1 is used to hold user and system stacks and to

process specific code and data. S0 contains procedures common to all processes and page tables. A 128 entry TLB is used, half for the system and half for the user, for high speed translation of active page addresses. When the context changes, i.e. a new process is started, the user TLB has to be flushed of information and reloaded with new real addresses corresponding to the new process, so that virtual addresses of the new process are properly translated and are not translated according to the old process.

An example of multilevel page mapping is the MC68030, which has the ability to specify between 0 and 4 levels by software. The 80386 also has a two-level page mapping mode, which is described in Section 2.4.3. The term *linear addressing* is used to describe page addressing, especially with multilevel mapping. Two-level and multilevel page mapping (also called *linear segmentation*) should not be confused with true (symbolic or named) segmentation, which will be described in Section 2.4.

2.3 Replacement algorithms

2.3.1 General

A *page fault* occurs whenever the page referenced is not already in the main memory, i.e. when a valid page entry cannot be found in the address translator (which includes the main memory page tables in the TLB method). When this occurs the required page must be located in the secondary memory using the secondary memory page tables, and a page in the main memory must be identified for removal if there is no free space in the main memory. Secondary memory/main memory transfers are relatively slow and are performed by a separate direct memory access (DMA) controller or input/output processor; thus, in a multiprogramming environment the processor can select another process for execution while these transfers are being done. The DMA controller needs to be started by the processor but then proceeds without further intervention, leaving the processor free for other activities.

There are various replacement algorithms that can be used to select the page to be removed from the main memory to make room for the incoming page. Algorithms can be classified as:

1. Usage-based algorithms.
2. Non-usage-based algorithms.

In a usage-based algorithm the choice of page to replace is dependent upon how many times each page in the main memory has been referenced. Non-usage-based algorithms use some other criteria for replacement. To implement usage-based algorithms, hardware is necessary to record when pages are referenced.

42 Computer design techniques

The simplest and most common hardware is to incorporate a *use* (or *accessed*) bit with each page entry in the page tables. The use bits are set if the corresponding page is referenced and are automatically reset when read. They are examined under program control to determine whether the pages have been used. To catch every increase in usage, use bits need to be scanned after each reference. Clearly this produces an unacceptable overhead, and the use bits are usually scanned at a much reduced rate to obtain an approximation of the usage, perhaps after 1 ms of process time. (Easton and Franaszek (1979) made a study of the use bit scanning technique in usage-based replacement algorithms.) To obtain a true value for usage, hardware counters could be introduced to record each reference, but this is not normally done.

Apart from a use bit, each entry in a page table has other bits to assist or improve the replacement algorithm, including a *modified* (or *written*, *changed* or *dirty*) bit. The modified bit is set if a write operation is performed on any location within the page. It is not necessary to write an unaltered page back to the secondary memory if a copy has been maintained there, and this increases the speed of operation. Very occasionally, there is an *unused* bit associated with each page, which is set to 1 when the page is loaded into main memory and reset to 0 when subsequently referenced. This bit may be helpful to make sure that a page demanded is not removed before being used. Protection bits concerned with controlling access to pages are also present; these will be discussed later.

Paging replacement (usage-based or non-usage-based) algorithms can be classified as suitable as:

1. A global algorithm.
2. A local algorithm.

They may be classified as both. *Global replacement algorithms* make their selection of main memory page among all those existing in the main memory, irrespective of the programs associated with the pages. *Local replacement algorithms* make a selection only from those pages related to the working set of the “paged-faulted” program, and do not consider those pages in the main memory associated with other programs. In general, local algorithms should be better than global algorithms in a multiprogramming environment because, for one reason, global policies do not take into account the fact that different programs may have different working set sizes and may take a page out of the working set of a program which is executed next, which would lead to *thrashing*. Denning (1970) used the term thrashing to describe the phenomenon of excessive page transfers that can occur in a multiprogramming environment when the memory is overcommitted; he attributed the term to Saltzer (Denning, 1980).

A process or program has a group of pages. In a multiprogramming environment all groups of pages might be the same size (known as a *fixed partition*) or different sizes (known as a *variable partition*). Early algorithms naturally have a fixed partition but can be extended to variable partitions. Some later algorithms naturally have a variable partition. A fixed partition is easier to implement than a variable

partition, but the latter is more flexible and reduces the memory requirements, typically by 30 per cent.

There are three policies to consider when handling page faults in a virtual memory system:

1. Replacement policy – to determine which page in the main memory to remove or overwrite.
2. Fetch policy – to determine when pages are loaded into the main memory.
3. Placement policy – to determine where the pages are to be placed in the main memory.

The normal fetch policy is called *demand paging*, which is the term used to describe the fetch policy of waiting until a page fault occurs and then loading the required page from the secondary memory. There has been a debate on the possibility of fetching pages before they are required in some prescribed prefetch policy. However, most paging systems employ demand paging. It appears that demand paging will result in the same or fewer page faults than are incurred by a prefetch paging policy (see Denning, 1970). A general metric for evaluating replacement algorithms is the number of page faults created.

With regard to the placement policy, we have assumed that when a page fault occurs, a page is removed from the main memory to make room for the incoming page and that the required page is brought into the same main memory location. Apart from using the same locations for the outgoing and incoming pages, alternative placement policies are possible if free space is maintained in the main memory. A placement policy might be created to maintain a certain amount of free main memory in the presence of variable memory usage.

Let us now consider the main page replacement policies and their implementation.

2.3.2 Random replacement algorithm

In the *random replacement algorithm*, a page is chosen randomly at page fault time; there is no relationship between the pages or their use. This algorithm does not take the principle of locality of programs into account and hence would not be expected to work very well. (It is generally believed that replacement algorithms which take account of the characteristics of programs expounded in the principle of locality will work better.) The generation of page numbers for random replacement can be done using a numerical pseudorandom number generator, or by counting the occurrences of some event. The random replacement algorithm is simple to implement but is not widely used; although the VAX 11/780 translation buffer (TLB) uses random replacement policy.

2.3.3 First-in first-out replacement algorithm

In the *first-in first-out replacement algorithm*, the page existing in the main memory for the longest time is chosen at page fault time. This algorithm is naturally a fixed global policy but could be modified to operate locally. The algorithm can be described by a first-in first-out queue, which holds the list of all pages currently in the main memory. As a new entry is inserted, all the entries move down one place and the last entry is taken out and specifies the page to be removed. Each page in the queue may be referenced many times before the next page is referenced, and the number of references to one page between page changes does not affect the algorithm. Initially, when the main memory is empty, page faults occur when pages are first referenced. Each time a new page is referenced the page entries are moved one place to the right (conceptually, not the actual page entries). When the memory partition is full and a page fault occurs, the page deleted (and if necessary returned to the secondary memory) is given by the entry at the rightmost end of the queue.

The algorithm requires no extra hardware to record memory references (as some other algorithms do) because the queue is maintained and updated only at page fault time with page fault information, and can be maintained in software. It does not matter how many times a page is referenced between page faults, though it is expected that it will be referenced many times (and of course at least once).

The algorithm can be implemented using a circular list holding the page entries, as shown in Figure 2.11. A pointer indicates the current rightmost end of the queue and the leftmost entry of the queue is immediately before the pointed entry. Upon a page fault, the page deleted and replaced is that indicated by the pointer. The pointer is then incremented to point to the next entry.

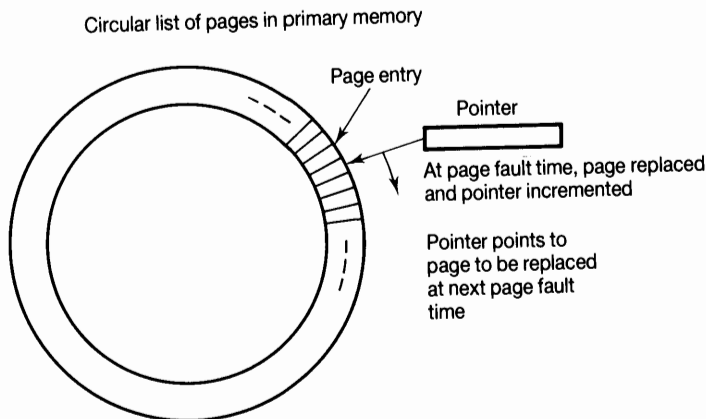


Figure 2.11 First-in first-out replacement algorithm using a circular list

The first-in first-out algorithm anticipates that the program will move from one page to the next in a linear, sequential fashion, but it is not at all certain that such characteristics are found in practice. More often, programs reference a group of pages repeatedly, but in various patterns, as different procedures are called. The first-in first-out algorithm performs particularly badly when the partition consists only of a loop of pages, sequentially and repeatedly executed, because every time execution moves from the last page back to the first page, a page fault will occur. For example, for the sequence of changes in page references:

1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, ...

and four pages in the main memory partition, the first-in first-out algorithm will generate a fault on every page change. In fact, virtually all fixed partition algorithms give bad results on this sequence and the best strategy here would be a *last-in first-out replacement algorithm* (a rarely used algorithm) which replaces the page just left, giving a page fault on every fourth page change on the above sequence.

Loop characteristics are, of course, a common characteristic of programs. However, large loops do not often occur; small loops with one or two pages are more likely, and all pages can be kept in memory simultaneously.

2.3.4 Clock replacement algorithm

The first-in first-out algorithm can be modified to avoid frequent transfers by moving over pages in the queue which have been referenced (and hence are likely to be accessed again). This algorithm is known as the *clock algorithm* because a pointing movement is used like that of the hand of a clock. The algorithm is also known as the *first-in-not-used first-out* algorithm. It requires the addition of a *use* bit set by the hardware when the page is referenced. The algorithm can be described as a circular list of page entries, corresponding to the pages in main memory, and a pointer which identifies the next page to be replaced, as shown in Figure 2.12. When a page replacement is necessary, the use bit of the page entry identified by the pointer is examined. If the use bit is set to 1, the bit is reset to 0 and the pointer advanced to the next page entry. This process is repeated until a use bit is already reset to 0. Then, the corresponding page is replaced and the pointer advanced to the next page entry. Whenever a page is referenced subsequently, the associated use bit is set to 1. Various modifications can be made to the clock algorithm (see Easton and Franaszek (1979) for further details).

2.3.5 Least recently used replacement algorithm

In the *least recently used (LRU) replacement algorithm*, the page which has not been referenced for the longest time is transferred out at page fault time. The least

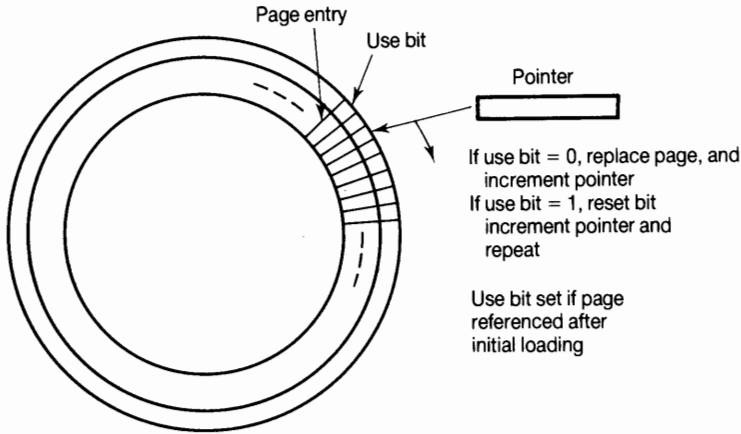


Figure 2.12 First-in-not-used first-out replacement algorithm

recently used algorithm can be described as a “stack” holding the list of pages in the main memory in the order in which they have been referenced. Whenever a reference is made, the order of the list has to be updated. This means that the page entry is placed at the top of the list and all other page entries are moved down one place. As with the first-in first-out algorithm, the LRU algorithm fails badly on a single loop of pages. It produces identical results to the first-in first-out algorithm on a sequence of pages in which pages in memory are not re-entered, because the page which has been longest in the memory is also the page referenced the longest time ago. However, the LRU algorithm would seem to match program characteristics.

The LRU algorithm poses some practical problems for a true implementation if there are a lot of pages in the main memory, as would be the case in a virtual memory system, because a record has to be made of references to each page (whether or not a page fault has occurred) during the execution of the programs. An obvious implementation of the LRU algorithm is the use of counters recording the number of references to each page. For example, a counter could be associated with each page and incremented at regular intervals. Every time a page is referenced, the associated counter is reset. Hence the counter holding the largest number identifies the least recently used page. (A counter solution is also possible for the first-in first-out algorithm, with counters associated with all existing pages incremented and the counter of the new page reset to 0 when a page fault occurs, see Tanenbaum, 1984). Counter solutions are not practical for a true virtual memory LRU algorithm because of the number of hardware counters and the substantial logic which would be required. In any event, counter methods give more information than is required; the only information required to identify the least recently used page after a page fault is the actual ordering, not how many times each page has

been referenced. A true alternative LRU implementation suitable for a small virtual memory system is the reference matrix method (see Section 3.4.4, page 83).

A true LRU algorithm can be implemented on the pages of a small TLB, but otherwise, in a virtual memory system, an approximation is made. A common approximation to the LRU algorithm is to employ the use bits. At intervals, say after every 1 ms as recorded by the system interrupt timer, all of the use bits are examined by the operating system and automatically reset when read. A record of the number of times the bits are found set to 1 would give an approximation of the usage in units of the interval selected. The approximation becomes closer to a true LRU algorithm as the interval is decreased. This method can also be implemented by having separate queues for different activity pages, a high activity queue, a medium activity queue and a low activity queue. Page entries are moved from one queue to the lower queue if the page was not referenced during the last time interval, and moved to the higher activity queue if it was referenced. Pages in the lowest activity queue move to a replacement queue for those pages which can be replaced.

2.3.6 Working set replacement algorithm

The *working set*, $w(t, T)$, at time t is defined as the collection of pages referenced by the process during the process time interval $(t-T, t)$. The working set function, $w(t, T)$, as a function of T and fixed t , increases monotonically, because the working set with an increased interval, $w(t, T+d)$, must include those pages of the original interval, $w(t, T)$, for a specified t . The working set must include sufficient pages for the program to run. The working set as a function of time, t , is not expected to change radically when only one process is being executed. Abrupt changes would occur when a new process is started, as in a multiprogramming environment.

The working set algorithm, which follows directly from the concept of programs having working sets (Denning, 1968), replaces the page which has not been referenced during the immediately preceding interval, T , given in terms of process time (or number of page references). The set of pages maintained are those which have been referenced during this interval. As time passes, a "window" moves along, capturing the working set of pages, as shown in Figure 2.13. It is possible for the memory allocation not to be full. For example, given a memory allocation of four pages, if the last five references were pages 5, 2, 6, 8 and 2, the pages in the set would be the three pages 2, 6 and 8.

Pages are added to the working set when a page fault occurs. Theoretically, pages are removed from the working set as soon as they have not been referenced during the preceding interval, though in practice such removal is only done at page fault time. The number of pages could grow very large, limited only by the number of references possible to different pages in the time interval, though the possibility of every single reference being to a different page is very remote. If the window is measured in terms of page references, then the maximum number of pages is given

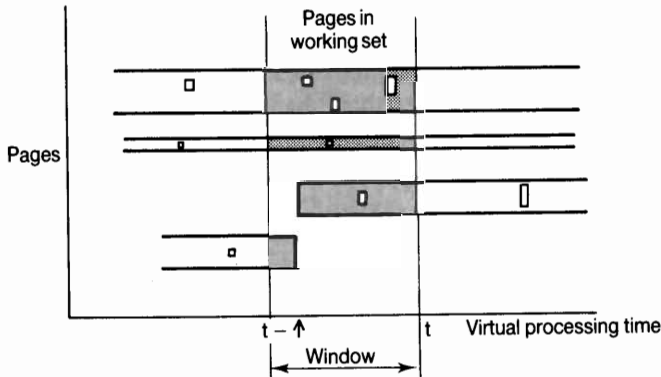


Figure 2.13 Working set reference patterns

by the window size. In a pure working set policy, pages could be released at times other than at page fault time, though most implementations wait until a page fault. If pages are only taken from the working set at the time of a page fault, there may be more than one page which was not referenced during the preceding interval. The least recently used algorithm can be employed to choose and remove only one or two pages.

The partition for this algorithm is naturally variable and uses a local policy, whereas the previous algorithms are naturally fixed and global. (The previous algorithms can be modified to operate locally using process stacks/queues.) An interval needs to be chosen for the window; this interval is normally kept constant. The strength of the working set algorithm is that it only keeps those pages likely to be required in memory, and does not use unnecessary memory.

A pure implementation would record the window interval in process time. The interval is never measured in actual seconds as there may be times when the process is suspended or delayed (such as during interrupt processing). One "counter" implementation would be to assign a hardware counter to each page, together with an identifier register, to indicate whether the page is part of the current window. When a page is referenced in the current window, the associated counter is reset to 0. All of the counters of the current window are incremented at regular intervals. If a counter overflows, the associated page is taken from the current window and is a candidate for removal at page fault time. If each counter has b bits and clock pulses are generated every p process seconds, then $T = p2^b$. The value of T is chosen by the system to give the best performance (i.e. the system is "tuned").

A major disadvantage of the working set algorithm is the necessity to record references between page faults and the need for hardware counters for a pure implementation. An approximate implementation could use the use bit scanning technique, as in approximate LRU algorithm implementations, by reading use bits in

the page table at intervals using the system interrupt timer and record the pages referenced since the last scan. Baer (1980) describes a working set implementation using the scanning routine.

A replacement algorithm called the *page fault frequency algorithm* (Chu and Opderbeck, 1976) resembles the working set algorithm but has a dynamically variable window. This algorithm maintains a set of most recently used pages. The set varies in size, depending upon the frequency of page faults, given threshold values. With more page faults, the set grows to attempt to reduce the frequency of the page faults. However, it is known that the algorithm can exhibit erratic behavior.

2.3.7 Performance and cost

The question of the selection of the replacement algorithm now arises – there are several possible algorithms to choose from and the choice rests upon performance and cost. The optimal replacement algorithm could be defined as one which creates the minimum number of page faults. We would expect that the minimum number of page faults is generated when the pages discarded from the memory are those which are not wanted again for the longest time in the future, which is known as the *principle of optimality*. This algorithm was described by Belady (1966) and is known as MIN (minimum page fault algorithm) or the optimal replacement policy (OPT). It operates like the least recently used algorithm, but on page references in the future. Stone presents an argument that MIN/OPT should be the optimum replacement algorithm (Stone, 1987) and some mathematical proofs exist for specific assumptions. However, the principle of optimality does not always hold (see Denning, 1970).

The MIN/OPT algorithm extended to variable partitioning is called the *variable space page replacement algorithm*, VMIN (Prieve and Fabry, 1976), or an optimal variable replacement algorithm, and it operates in a similar way to the working set algorithm on page references in the future. In VMIN, the window is an interval from the present to a point in the future, i.e. an interval $(t, t+\theta)$. At each page reference (time, t), the page is kept if the next reference to the page is in the interval $(t, t+\theta)$, otherwise the page is removed. VMIN generates the same sequence of page faults as the working set (Denning and Slutz, 1978) and has the interesting effect of anticipating transitions between disjointed working sets. Of course MIN and VMIN cannot be implemented in practice, but serve as benchmarks for comparison with practical algorithms.

In any selected replacement algorithm we would hope that, if the memory allocation is increased, the number of page faults decreases or at least stays the same. This would always be true if an allocation of m pages includes the pages in an allocation of $m-1$ pages. Algorithms with this characteristic are known as *stack algorithms*. The stack characteristic is useful for studying different replacement algorithms using a known reference string and different memory partitions.

Particular practical algorithms will perform well under certain conditions. For example, the FIFO algorithm (not a stack algorithm) works well for programs in

which pages are referenced in a long sequence, but otherwise can perform poorly. LRU works well for programs which repeatedly reference a set of pages. Global algorithms seem to perform worse than local algorithms in a multiprogramming environment and the working set algorithm appears to produce close to optimal results. Denning makes a strong case for his working set algorithm, arguing that the cost/complexity of implementation (its major disadvantage) should not be a deterrent.

The memory hierarchy in a system has mainly come about due to cost considerations. The cost of a memory system is normally characterized by the cost per bit of the memory. At each level of memory hierarchy, the cost per bit reduces, sometimes substantially, and the memory capacity (the number of locations in the memory system) increases. The access time also increases significantly, and there is a trade-off between cost and speed. The average cost of a memory system per bit having main and secondary memories in the hierarchy is given by:

$$C_{av} = \frac{c_{main}m_{main} + c_{sec}m_{sec}}{m_{main} + m_{sec}}$$

where c_{main} and c_{sec} are the costs per bit of the main and secondary memories and m_{main} and m_{sec} are the capacities of the memories. The average access time will depend upon how often the required information is in the highest speed memory (connected to the processor). The goal of any memory management scheme is to ensure that when it is required, information is in the highest level of memory as often as possible. The probability that an item is found immediately in the highest level of memory considered is known as the *hit ratio* (h) for this memory. The *access time* is the time required between a memory request being made and the location being read or written. Read and write access times often have the same value. The average access time is given by:

$$t_{av} = ht_{main} + (1 - h)t_{sec}$$

or, if all requests must first be made to the main memory, then:

$$t_{av} = t_{main} + (1 - h)t_{sec}$$

where t_{main} is the total access time of accessing the main memory, including the address translation, and t_{sec} is the additional access time for accessing the secondary memory.

A criterion which embodies the memory allocation and overall speed of execution is the *space-time product* (ST). This is the product of the memory used by a program and the amount of time that it is used. Since these are directly related to cost, the space-time product is regarded as an indication of the cost of executing the program. As memory requirements change over time, the space-time product becomes the integral of the set of resident memory pages over time while the program is being executed including times for waiting for a missing page, i.e.:

$$ST(t_1, t_2) = \int_{t_1}^{t_2} M(t) dt$$

over the time interval t_1, t_2 and $M(t)$ pages at time t . For a fixed memory allocation, M , the space–time product reduces to:

$$ST = M(n + D \cdot f)$$

where n = number of references; D = average time to transfer page from secondary memory to main memory and f = number of page faults.

For a variable memory allocation, we take into account that the memory allocation can be different with different memory references (in particular after a page fault) to obtain:

$$\begin{aligned} ST &= \sum_{t=1}^T M(t) + D \sum_{i=1}^f M(t_i) \\ &= M_{av}T + D \sum_{i=1}^f M(t_i) \end{aligned}$$

where T = total program execution period; t_i = time of the i th page fault; M_{av} = the average memory allocation over the execution period. The space–time product should normally be minimized to reduce costs, although no optimal policy always does this.

Though outside the scope of this book, theoretical studies have been performed using mathematical models for program behavior to predict memory references, but most models cannot easily incorporate the transitions that occur between processes in a multiprogramming environment.

2.4 Segmentation

2.4.1 General

In a *segmented* system, the memory space is not divided into equal sized pages but into blocks of contiguous locations called *segments*; these may be of different sizes. This approach suits programs and data which are naturally generated in various sizes. Each address is composed of a *segment number* and a *displacement* within the segment. The displacement is also called an *offset* and the segment number is sometimes called the *base*. Rather than concatenate the segment and offset numbers, in a segmented system the segment number and the offset are added together to form the

real address, because segments do not necessarily start at fixed boundaries. Segments are usually restricted to start at, say, 16 word boundaries so that the least significant bits of the segment address can be assumed to be 0. (Least significant 4 bits are 0 for 16 word boundaries.) The term *logical address* is used to describe the virtual address and *physical address* describes the real address. Segments can be shared between programs or may partially overlap if required.

An important aspect of the (symbolic) segmentation described here is that the segment number and offset are separate entities and any alteration to the offset by the program cannot affect the segment number. Once the maximum offset is reached (assuming that the segment grows with increasing addresses) adding one to the offset should create an error condition. A simple impure implementation of segmentation might cause the real address to wrap around to the beginning of the same segment. It would be unforgivable to implement a segmentation system that entered another segment when one was added to the maximum offset, though this effect is known in linear segmentation. It should not be possible to enter segments from other segments unless the access has been specifically allowed (as with shared system segments). In particular, data segments and code segments are separated, so that trying to execute data in data segments as code should generate an error condition. Similarly, trying to alter information in code segment should generate an error condition because code is normally assumed to be read-only and accessed only during a fetch cycle.

Segmentation, as a method of separating sections of program and data, dates from about the same time as the paging concept and was a main aspect of early Burroughs computers; it was first used in the B5000 and subsequently in other Burroughs systems. Since the early 1970s, segmentation has been combined with paging as a main memory management technique on most larger computer systems and more recently developed microprocessors.

Figure 2.14 shows the usual method of translating logical addresses into physical addresses. The logical address has two parts, a segment number and an offset. The segment number specifies the logical segment and the offset specifies the number of locations from the beginning of the segment. The segment and offset fields are physically separate and are not obtained by simply dividing the address from the processor into two fields. The processor has to be designed to generate the segment and offset separately.

The translation mechanism usually employs direct mapping, as shown. The starting addresses of the physical segments are held in segment tables, and there is a different segment table for each active process. The starting address of the appropriate segment table is contained in a segment table pointer register and this is added to the segment number to locate the physical segment base address in the table. The base address read from the segment table is added to the offset to form the required physical address.

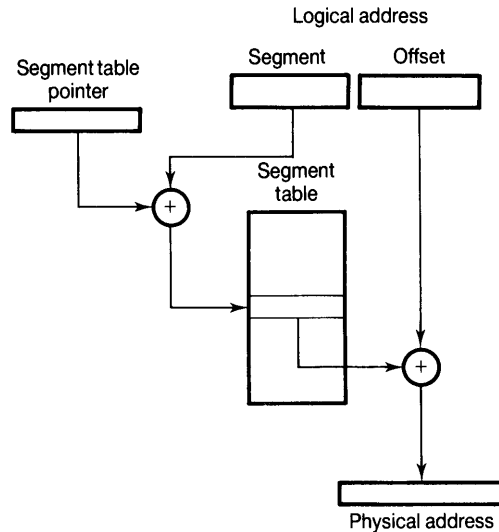


Figure 2.14 Segmentation address translation

The segment table incorporates additional information, usually including:

1. Segment length.
2. Memory protection bits.
3. Bits for the replacement algorithm.

Segment length

Different segments can have different lengths. The length of each segment is stored in the length fields of the segment table entry to prevent programs referencing a location beyond the end of a particular segment. If the offset in the virtual address is greater than the stored length (limit) field, i.e. an attempt is being made to reference beyond the end of the segment, an error signal is generated, usually in the form of a system interrupt. A system is assigned a maximum segment length which will specify the number of bits in the length field. Maximum segment lengths range from 64 Kbytes for systems with small address spaces through to 4 gigabytes for 32-bit address spaces.

As an added facility for segments used to hold stacks that grow downwards (which is how most stacks grow), it is useful to know when one of the first 256 words (say) is being accessed, so that a warning that the end of the stack space is being reached can be given. A separate flag, which is set when such accesses are made, can be provided.

Memory protection

Memory protection involves preventing specified types of access to the addressed location and discarding or stopping the address translation occurring. The protection applies to all of the locations in the segment and not to particular locations. Note that segments should be produced for unified purposes, i.e. for data, for a procedure, etc. and the protection applied to the whole segment.

Typically, by setting bits in the segment tables, any segment can be assigned as:

1. Read-only.
2. Execute only.
3. System only.

Assigning a segment as read-only allows data to be protected from alteration. Assigning a segment as execute-only means that the segment can only be referenced during a fetch cycle, which prevents unauthorized copying of programs since execute-only code cannot be read as data. Segments that are shared by different processes could have different access rights for each process.

It is necessary for the processor to have two operating modes for the system-only assignment, a normal mode, which is for ordinary users, and a system mode dedicated to the operating system. Generally, when in the normal mode, there will be certain instructions (including input/output instructions) which cannot be executed. The only way to enter the system mode is through a system call to the operating system, either intentionally or due to an error condition. Hence, functions such as input/output can be totally controlled by the operating system without interference from the user programs. Though some memory management schemes do not have the full selection of protection bits, a system-only bit is regarded as the minimum protection that must be present.

Rather than having an “only” assignment, it is possible to have an “excluded” assignment, especially in a microprocessor-based multiprocessor system with a separate *memory management unit* (MMU), for example:

1. CPU excluded.
2. DMA excluded.

In CPU excluded, the segment cannot be accessed by the central processor, however, this leaves all other possible “bus masters”, such as DMA input/output controllers. In DMA excluded, the DMA controllers are excluded, leaving the central processor and other bus masters, i.e. the other processors in a multiprocessor system. Any attempted violation of the assignments would cause the MMU to set appropriate error flags in an MMU error condition register, and to signal the processor with a special segment trap interrupt signal. The current instruction and status information will be saved. Multiple violations must be handled.

Replacement algorithm

The replacement algorithm in a segmented system can be similar to the replacement algorithm in a paged system, except that it needs to take the varying size of the segments into account when allocating space for new segments. Typically, as in a paged system, replacement algorithm flags are associated with each logical/physical address entry in the segment table, in particular with the use (accessed) and modified (written) flags. As we have seen, the use flag is usually sufficient to implement a replacement algorithm or approximations to a replacement algorithm.

Placement algorithm

The variable size of segments causes some additional problems in main memory allocation. During operation, with segments returned to the secondary memory, the main memory will become a “checkerboard”, as shown in Figure 2.15, with *holes* between segments. Clearly, an incoming segment must be smaller than the main memory space available (hole) for the segment to be overwritten. However, leaving small spaces which cannot be used subsequently should be avoided, and is known as *external fragmentation*. Several placement algorithms for finding a suitable place in the main memory to hold an incoming segment have been proposed, including *first fit*, *best fit* and *worst fit*. In first fit, a table of memory allocation, in particular the available holes, is scanned from the beginning until a space which is big enough is found, and the segment is entered there. This algorithm can be modified to skip over spaces which would leave a very small, unusable space had the segment been placed there. The best fit scans the complete list of hole sizes to select the memory space which would leave the smallest hole and the worst fit selects the space which would leave the biggest hole, in the hope that this hole will be big enough for another segment.

To help fitting in segments, it is usually necessary to compact the memory by moving segments together and eliminating the holes between them, which is a very time consuming process. These problems have led to the incorporation of paging in most large systems that use segmentation.

2.4.2 Paged segmentation

Segmentation and paging can be combined, and usually are combined, to gain the advantages of both systems, i.e. the logical structure of segmentation and the

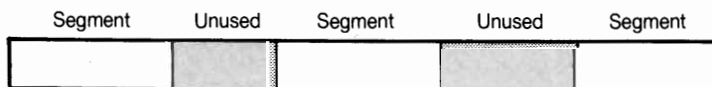


Figure 2.15 Checkerboard effect

hardware mapping between main and secondary memory of paging. The paging aspect simplifies the memory allocation problem of a pure segmented system. When segmentation and paging are combined, but the concept of segments as logical units is kept, the segmentation is regarded as *symbolic segmentation* or *segmented name space*. Each segment is divided into a number of equal sized pages and the basic unit of transfer between main and secondary memory is the page. It is not necessary to transfer the complete segment into the main memory as in the pure segmentation method; only those pages required need be transferred. Hence, the main memory might consist of pages from various segments, as shown in Figure 2.16, and pages of a new segment can be easily fitted into the memory.

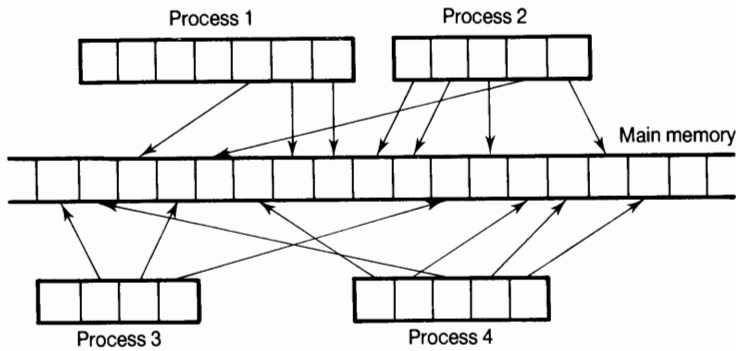


Figure 2.16 Paged segments in memory

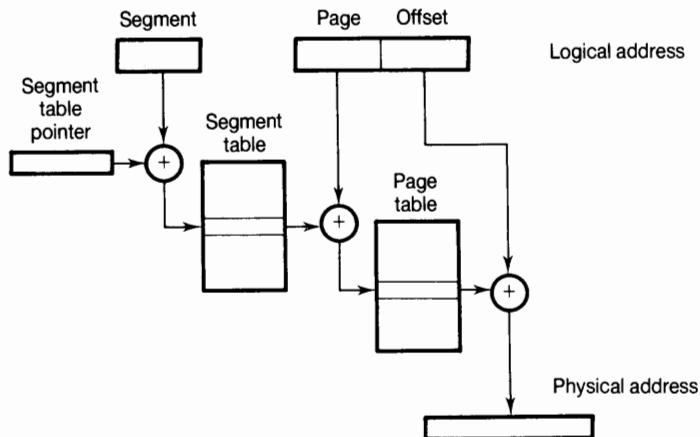


Figure 2.17 Paged segmentation address translation

The virtual address is divided into a segment number, a page number and displacement identifying the word within a page. The translation mechanism is shown in Figure 2.17. A segment table pointer selects a set of segment tables and the segment number selects a page table entry to select a page table. The page selects a real address which is concatenated with the displacement to obtain the full real address. In symbolic segmentation, length limit and other protection is naturally applied at the segment table level and replacement bits applied at the page table level.

The segment can be specified in the instruction in several different ways other than as a single number as shown. For example, as in the Multics system (Baer, 1980), one bit in the instruction can select either the current process segment, whose address can be held in a dedicated processor register which eliminates any segment table memory accesses, or one of a set of external segments whose addresses are held in other processor registers (if the segment is in memory). The external segment entries need to be loaded, and this could be done by *static binding* or by *dynamic binding*. In static binding, the segment entries are loaded before the program is executed by examining the program requirements during a linking/loading process. In dynamic binding (as in Multics), the segment entries are loaded on demand during the program execution by the operating system.

2.4.3 8086/286/386 segmentation

The 16-bit 8086 microprocessor, introduced in 1978, is perhaps the first example of a microprocessor to incorporate a very restricted form of segmentation within the device (Intel, 1985a), though this nevertheless enables code and three forms of data to be separated in one program. However, it does not allow the facilities of true segmentation such as sharing and complete protection. The microprocessor contains four segment registers called the *code segment register* (CS), the *data segment register* (DS), the *stack segment register* (SS) and the *extra segment register* (ES) respectively. The address generated by the program is a 16-bit offset, without a segment number. A 16-bit offset allows segments up to 64 Kbytes. The particular segment is selected by context. Instruction fetch cycles always use the code segment with the offset provided by the program counter (called the instruction pointer, IP, in the 8086). Most data operations normally assume the use of the data segment register, though any segment register can be selected using an additional prefix instruction. Stack instructions always use the stack segment register. The extra segment is used for results of string operations. The segment registers have 16-bits. Four least significant 0s are added, giving a 20-bit base address and a 20-bit physical address.

The Intel 16-bit 80286 microprocessor in the “protected virtual address” mode (Intel, 1987a) extended the memory management scheme of the 8086 to give up to 2^{13} (8192) separately addressed segments, though only four, designated as CS, DS, SS and ES, can be used at a time and these are then used as in the 8086. Physical

memory addressing has been extended to 24 bits (16 Mbytes). The original segment registers within the device are now called *segment selectors* by Intel and are used as pointers to within main memory segment (descriptor) tables holding the segment information, in the form of a descriptor. A descriptor consists of a 24-bit segment base address, protection bits (access rights) and a length field.

In effect, four separate internal 24-bit segment registers are provided as part of what are called *descriptor cache registers*, which hold the descriptors of four actual segments being used, including 24-bit segment base addresses. The descriptor cache registers cannot be accessed directly by the programmer. The formats of the descriptor cache registers and segment selector registers are shown in Figure 2.18. The trans-

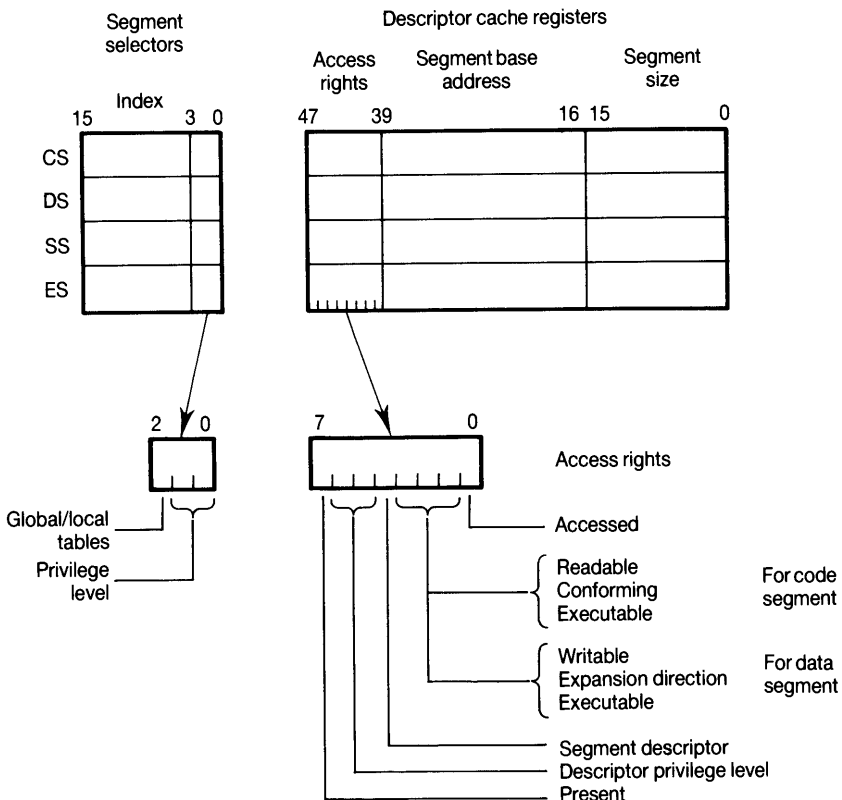


Figure 2.18 80286 Segment selectors and descriptors

lation mechanism is shown in Figure 2.19. To change to another segment not currently being used, a 13-bit number is loaded into the segment selector to identify the entry in the descriptor table, together with 3 bits giving privilege level and whether global or local tables are to be used. The processor then automatically loads the descriptor cache register from the main memory table, and addressing now uses the new base address. Segment selectors are loaded using normal move instructions for data selectors or branch instructions (CALL, JMP, RET, IRET) for a code selector, under certain privilege rules, and with a protection mechanism to ensure that proper information is selected.

Two descriptor tables can be accessed by a task (process) at a time – a global table for shared segments and a local segment table for the currently active task (which is only accessible by this process). (A further type of descriptor table exists, called an interrupt descriptor table, for interrupts.) In our discussion on memory management, two types of access are identified, namely, system and user. In the 80286, this concept is taken further by introducing four levels of privilege, the highest (PL0) for the operating system kernel through to PL3 for applications. Further information on the protection mechanisms provided by the 80286 can be found in Intel literature (see, for example, Intel, 1987a).

In a system with very large segments, segments can be paged using a two-level translation, as described in Section 2.2.5, to reduce the number of page tables necessary for one task. There are potentially three levels of translation, one for the

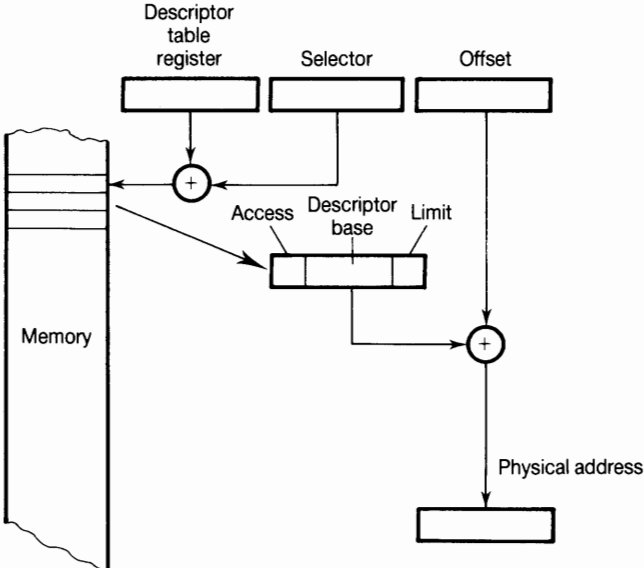


Figure 2.19 80286 segment address formation (protected virtual address mode)

segment and two for the page. This type of scheme is used on the Intel 80386 32-bit microprocessor (Intel, 1985b), a 32-bit development of the 80286 which includes demand paging. The processor generates, as instruction pointer or data addressing, a 32-bit offset of a segment. The segment is specified and identified separately in a similar fashion to the 80286 using a selector to select a descriptor, and produces a 32-bit “linear address”. Paging can be enabled or disabled by setting or resetting a flag in a control register (CR0). If paging is disabled, the linear address formed is used to access memory. If paging is enabled, the linear address passes through a two-level page translation mechanism, as shown in Figure 2.20, to produce a 32-bit physical address used to access memory.

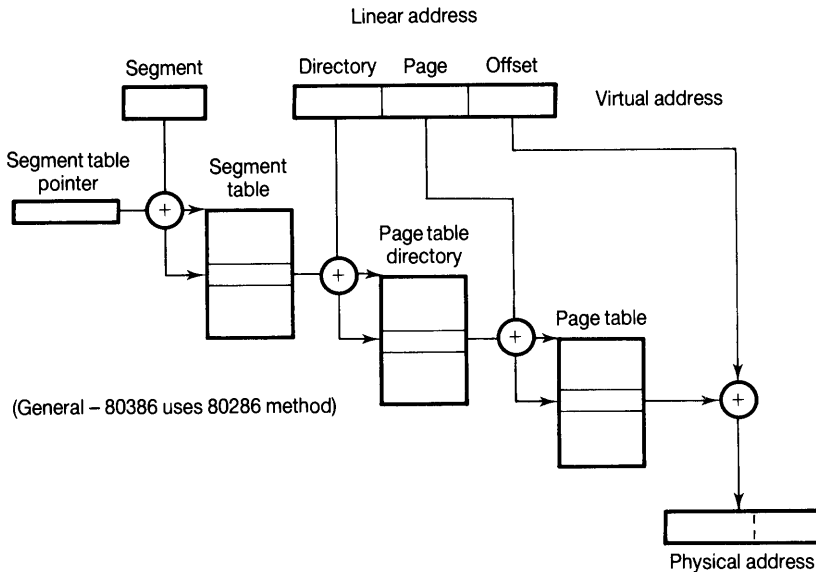


Figure 2.20 Two-level segmentation with paging

PROBLEMS

2.1 In a paged system, suppose the following pages are requested in the order shown:

12, 14, 2, 34, 56, 23, 14, 56, 12, 12

and the main memory partition can only hold four pages at any instant (in practice usually many more pages can be held). List the pages in the main memory after each page is transferred using each of the following replacement algorithms:

1. First-in first-out replacement algorithm.
2. Least recently used replacement algorithm.
3. Clock replacement algorithm.

Indicate when page faults occur.

2.2 Identify which of the following replacement algorithms are stack algorithms:

1. Random replacement algorithm.
2. First-in first-out replacement algorithm.
3. Clock replacement algorithm.
4. Least recently used replacement algorithm.
5. Working set replacement algorithm.
6. Page fault frequency replacement algorithm.

2.3 Suggest how the optimal replacement policy could be implemented given that the memory reference string is known.

2.4 Deduce a sequence of page references for a paging system with eight main memory pages, using the least recently used replacement algorithm, which produces each of the following characteristics:

1. The largest number of page faults.
2. The smallest number of page faults.

2.5 Using one use bit with each page entry, list the pages recorded in the following sequence and hence determine the pages removed using the one use bit approximation to the least recently used algorithm:

13, 47, 13, 99, 47, 35, 13, 67, 47, 13, 34, 35, 99

given that there are four pages in the memory partition. The use bits are only read at page fault time.

Suppose two use bits are provided. The first use bit is set when the page is first referenced. The second use bit is set when the page is referenced again. Deduce an algorithm to remove pages from the partition, and list the pages.

2.6 If the cost of a semiconductor main memory is four times the cost of disk memory per bit, and the total amount of memory required is 25 Mbytes, determine the amount of each type of memory to achieve a total cost per bit of twice that of the semiconductor memory.

2.7 A microprocessor generates a 20-bit byte address A19, A18, A17, A16, A15, A14, A13, A12, A11, A10, A9, A8, A7, A6, A5, A4, A3, A2, A1, A0 (A0 being the least significant bit). Design a translation look-aside buffer for the system giving details of the address translation and numbers of bits in address fields, if the following applies. The page size is to be 512 bytes. 256 page addresses are to be handled by the TLB. The groups of virtual addresses below are likely to appear frequently:

- Five addresses with A19 the same.
- Two addresses with A18 through A0 the same.
- Three addresses with A16 through A0 the same.
- Four addresses with A15 through A0 the same.
- Two addresses with A13 through A0 the same.
- Two addresses with A10 through A0 the same.
- Two addresses with A9 the same.

Clearly indicate your reasoning. Design for minimum cost.

2.8 In a paging system, the page size is p and a program requires P pages. The last page in the program is 50 per cent full. Each page requires t locations in the main memory page table. Obtain an equation for the total amount of main memory required for the program and page table entry combined. Find the page size to give minimum memory requirements by differentiating the total memory requirement equation with respect to p , and equating the result to zero. Determine a suitable page size for a program of 128 Kbytes given four bytes per page entry.

2.9 Determine the number of locations required in the page tables for a three-level page mapping given that the virtual address has 32 bits divided into a 12-bit directory field, a 10-bit page field and a 10-bit line (offset) field. How many bits are there in the table entries for addresses?

2.10 Apply the first fit, best fit and worst fit placement algorithm to insert a 290 byte segment in a 10 Kbyte memory partition which contains

segments at locations given in Table 2.1. Show the location of the incoming segment. Repeat, taking into account that no incoming segment will be less than 11 bytes.

Table 2.1 Segment locations and sizes for Problem 2.10

Address	Segment size (bytes)
0	3400
3800	230
4500	630
5590	100
7000	200
7500	550
10000	120

2.11 As a designer of a new paged segmentation memory management system, you are to develop the formats of information stored in the page table and segment tables given that the page size is 512 words, the maximum segment size is 65 536 words and the processor generates a 32-bit address. Making appropriate design decisions, choose and list the sizes of each field in the segment and page table. Memory can be made read-only or execute-only, and four other levels of privileged access are provided (one for the user and three for routines within the operating system).

This chapter studies the use of a relatively small capacity but high speed memory called a *cache*, which is inserted between the processor and main memory. The cache is introduced into the system to decrease the effective memory access time and hence increase the operational speed of the system.

3.1 Cache memory

3.1.1 Operation

The speed at which locations can be accessed in a memory is a critical factor in the system design. Semiconductor memory speeds are characterized by the memory access time and memory cycle time. We have already mentioned in Chapter 2 that the *memory access time* is the time between the submission of a memory request and the completion of transfer of information into or from the addressed location. Normally, the access time for a read and for a write operation is the same, and we will assume read and write access times to be the same in one memory. The *memory cycle time* is the minimum time that must elapse between two successive operations to access locations in the memory (read or write). Sometimes the access time and cycle time are almost the same, i.e. immediately a location has been accessed, another memory operation can be initiated, but often a short period must elapse after the access for the internal circuits to settle and be ready for the next read/write operation (i.e. a precharge period in some semiconductor designs). In high speed memory systems the cycle time is almost double the access time.

Secondary memory is usually several orders of magnitude slower than the main memory. There is also a mismatch between the speeds of operation of the processor and the main memory; processors (except early microprocessors) are generally able to perform operations on operands faster (perhaps one order of magnitude faster) than the access time of large capacity main memory. Though semiconductor memory which can operate at speeds comparable with the operation of the processor

exists, it is not economical to provide all the main memory with very high speed semiconductor memory.

The problem can be alleviated by introducing a small block of high speed memory called a *cache* between the main memory and the processor. A cache consists of very high speed random access memory operating at the speed required by the processor. Programs and data are transferred to the cache, which is then accessed by the processor. Any data items to be changed are first written to the cache and either written to the main memory at the same time, or subsequently, when the locations are replaced with new information from the main memory. A cache was first used in a commercial computer system by IBM in the IBM System/360 Model 85. The IBM 360 Model 85 cache was described and the term *cache memory* used by Conti in 1968 (Conti *et al.*, 1968).

A cache is generally successful because of the principle of locality of reference exhibited by programs and data (see Chapter 2). For example, a purely sequential list of instructions which is executed only once is rare; instructions are more often formed into loops, which are executed many times. The length of a loop is usually quite small. Therefore once a cache is loaded with loops of instructions from the main memory, the instructions are used more than once before new instructions are required from the main memory. The same situation applies to data.

The principle of locality allows a cache system to improve system speed, just as it allows virtual memory systems to operate efficiently. If every memory reference to a cache required a transfer of one word between the main memory and the cache, no increase in speed would be achieved; in fact the speed would drop because apart from the main memory access, an additional access to the cache would be required. However, suppose the reference is repeated n times in all during a program loop, and, after the first reference, the location is always found in the cache, then the average access time would be:

$$\text{Average access time} = \frac{(nt_c + t_m)}{n} = t_c + \frac{t_m}{n}$$

where t_c = cache access time; t_m = main memory access time and n = number of references. If $t_c = 25$ ns, $t_m = 200$ ns and $n = 10$, the average access time would be 45 ns, as opposed to 200 ns without the cache, i.e. a substantial increase in speed using a cache operating at eight times the speed of the main memory. It is assumed that there are no additional timing factors with the introduction of the cache and the processor must be able to handle the increased speed (ten 25 ns accesses and one 200 ns access). We note that as n increases, the average time approaches the access time of the cache. The increase in speed will, of course, depend upon the program. Some programs might have a large amount of temporal locality, while others have less. Also, the average access time gives only an indication of the system improvement. The actual improvement will be different because instruction execution speeds have components other than instruction fetch and operand fetch times.

In large computer systems the main memory is often interleaved or interfaced to

wide word length main memories (see Section 1.3.4) to match the speed of transfer of the main memory with the cache. The number of modules, m , is chosen to produce a suitable match in the speed of operation of the main and cache memories. For a perfect match, m would be chosen such that $mt_c = t_m$. The cache words could subsequently be accessed by the processor in sequential order in another mt_c seconds. Hence the average access time of these words when first referenced would be $2mt_c/m = 2t_c$. Should the words be referenced n times in all, the average access time would be:

$$\text{Average access time} = \frac{2t_c + (n-1)t_c}{n} = \frac{(n+1)t_c}{n}$$

For example, if a cache has an access time of 25 ns and the main memory has an access time of 200 ns, eight main memory modules would allow eight words to be transferred to the cache in 200 ns. With ten references in all, we have:

$$\text{Average access time} = \frac{(50 + 9 \times 25)}{10} = 27.5 \text{ ns}$$

The average access time is approximately t_c for large n , making the same rather broad assumptions as before. However, it does indicate that substantial speed improvements can be achieved by using the cache.

Notice that if the locality in programs was only instruction sequential locality, and if we could always rely on instructions being sequential, wide word length memories would be sufficient to keep the processor content with instructions, with perhaps a single wide word length buffer. However, this is not the case and variations in sequential instruction fetches and data references need to be taken into account.

We have assumed that it is necessary to reference the cache before a reference is made to the main memory to fetch a word, and it is usual to look into the cache first to see if the information is held there. The advantage of the cache comes from information it holds and, although it may be necessary to make a second reference to the cache after the word has been fetched from it for a read operation, it is likely that the word can be sent to the cache and the processor simultaneously. However, for write operations through the cache, the cache location will be altered. Write operations require an additional scheme to deal with the main memory; this is described on page 76. Any word altered in the cache must be transferred back to the main memory eventually, and this transfer will reduce the average time. We will develop formulae to compute the average access time including the main memory write mechanism, but first let us assume read-only operations. In general, all formulae are applicable to both access time and cycle time.

Though most high performance computers use cache memory, there is a notable exception; the Cray vector computers use files of register buffer storage instead of cache memory.

3.1.2 Hit ratio

The probability that the required word is already in the cache depends upon the program and on the size and organization of the cache; typically 80–90 per cent of references will find their words in the cache. A *hit* occurs when a location in the cache is found immediately, otherwise a *miss* occurs and a reference to the main memory is necessary. The cache *hit ratio*, h , is mentioned in Chapter 2 for main memory–secondary memory systems and is also applicable to cache–main memory systems. For a cache system, it is defined as:

$$h = \frac{\text{Number of times required word found in cache}}{\text{Total number of references}}$$

The cache hit ratio is also the probability that a word will be found in the cache. The *miss ratio* is given by $1-h$. In cache studies, the miss ratio is quoted rather than the hit ratio. The average access time, t_a , is given by:

$$t_a = t_c + (1 - h)t_m$$

assuming again that the first access must be to cache before an access is made to the main memory. Accesses made to the main memory add the time $(1 - h)t_m$ to the access time. For example, if the hit ratio is 0.85, the main memory access time is 200 ns and the cache access time is 25 ns, then the average access time is $25 + 0.15 \times 200 = 55$ ns.

The average access time can be computed for virtual memory systems in the same manner. In these systems there is a much greater gap between the access time of two memories and the miss ratio has a much greater effect on the overall access time. The average access time in either system ignoring additional time involved in write operations can be given as:

$$t_a = t_{\text{mem1}} + (1 - h)t_{\text{mem2}}$$

where t_{mem1} = access time of the higher speed memory (cache in a cache system, main memory in a virtual memory system); t_{mem2} = access time of the lower speed memory (main memory in a cache system, secondary memory in a virtual memory system) and $(1-h)$ = miss ratio.

Therefore:

$$t_a = t_{\text{mem2}}(1/k + (1 - h))$$

where $k = t_{\text{mem2}}/t_{\text{mem1}}$ (i.e. the ratio of lower speed memory access time to higher speed memory access time). We can see that the average access time will be dominated by the miss ratio if the ratio of the memory access times (k) is large, as in the case of a virtual memory system. For example, with a 20 ms access time disk

and a 200 ns access time high capacity semiconductor random access memory, $k = 100\,000$. A miss ratio of 1 per cent creates an average access time of 200.2 μs , which is much larger than the main memory access time. Clearly, for a virtual memory system the miss ratio ought to be very low indeed to approach the access time of the main memory. (Of course, in practice, a disk will not be accessed directly.)

Conversely, the average access time will be dominated by the ratio of the access times of the memories when this difference is small, rather than by the miss ratio. For a cache system, the ratio of main memory access time to cache access time (k) is in the region of 3–10. For example, with a 200 ns access time main memory and a 25 ns access time cache, $k = 8$. A miss ratio of 1 per cent creates an average access time of 27 ns, which is close to the cache access time. The miss ratio for a cache system need not be as low as for a virtual memory system for the average access time to approach the access time of the cache.

3.2 Cache memory organizations

The problem of mapping the information held in the main memory into the cache is similar to virtual memory systems, though any cache mapping scheme must be totally implemented in hardware to achieve improvements in the system operation. Various strategies are possible:

3.2.1 Direct mapping

In cache *direct mapping*, the least significant bits of the memory address in the main memory and the cache are the same. The most significant bits of the address are stored in the cache and read after the least significant bits have been used to access the cache word.

First, consider the example shown in Figure 3.1. The address from the processor is divided into two fields, a *tag* and an *index*. The tag identifies a page in the main memory and the index identifies the word within the page. When the memory is referenced the index is first used to access a word in the cache. Then the tag stored in the accessed word is read and compared with the tag in the address. If the two tags are the same, indicating that the word is the one required, access is made to the addressed cache word. However, if the tags are not the same, indicating that the required word is not in the cache, reference is made to the main memory to find it. For a memory read operation, the word is then transferred into the cache where it is accessed. It is possible to pass the information to the cache and the processor simultaneously, i.e. to *read-through* the cache. The cache location is altered for a write operation. The main memory may be altered at the same time (*write-through*) or later. Write operations will be discussed in Section 3.3.2.

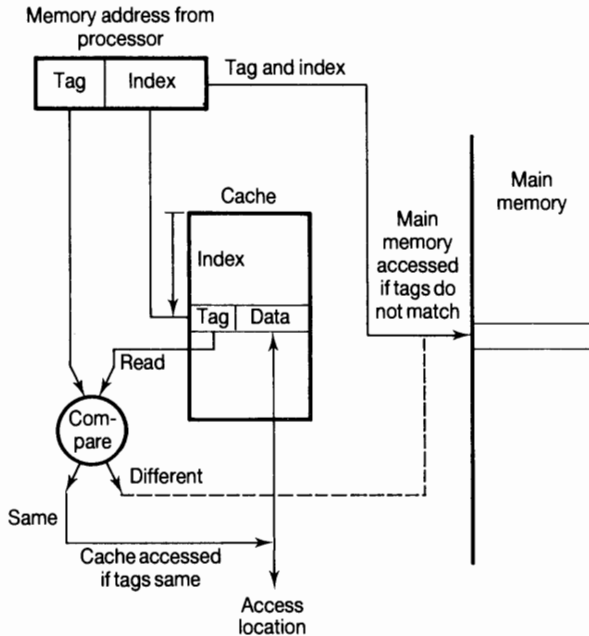


Figure 3.1 Cache with direct mapping

In the direct mapping scheme described, one word is transferred to the cache at a time. In Figure 3.2, several words are transferred together by interleaving or using wide word length main memories. (In fact, the latter is sufficient in this application because the words are sequential.) We shall call the words transferred a *block* (sometimes a block is called a *line*). The main memory address is composed of a tag, a block and a word within a block. All the words within a block in the cache have the same stored tag. The block/word part of the address is used to access the cache and the stored tag is compared with the required tag address. For a read operation, if the tags are the same, the word within the block is selected for transfer to the processor. If the tags are not the same, the block containing the required word is first transferred to the cache.

In direct mapping, the corresponding blocks with the same index in the main memory will map into the same block in the cache, and hence only blocks with different indexes can be in the cache at the same time. A replacement algorithm is unnecessary, since there is only one allowable location for each incoming block. Efficient replacement relies on the low probability of blocks with the same index being required. However, there are such occurrences, for example, when two data vectors are stored starting at the same index and pairs of elements need to be processed together. To gain the greatest performance, data arrays and vectors need

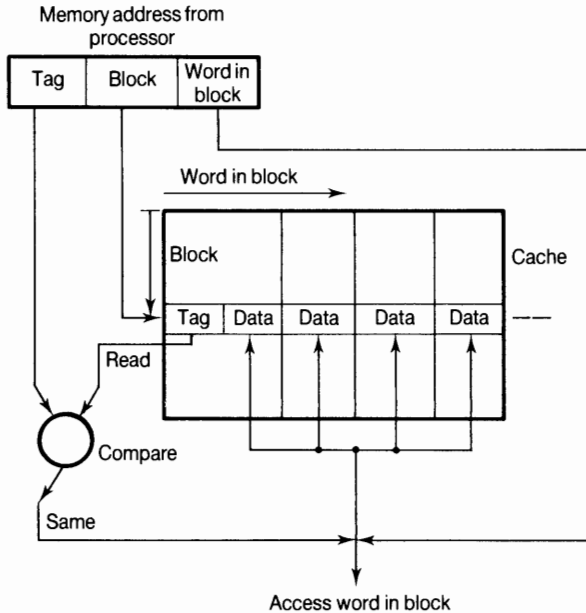


Figure 3.2 Direct mapped cache with block organization

to be stored in a manner which minimizes the conflicts in processing pairs of elements. Figure 3.1 shows the lower bits of the processor address used to address the cache location directly. It is possible to introduce a mapping function between the address index and the cache index so that they are not the same.

The direct mapping cache has been used on the PDP-11/60 with a one word block and in the IBM System/370 Model 158. The VAX-8800 uses a 64 Kbyte direct mapped cache. The following advantages can be identified for the direct mapped cache:

1. No replacement algorithm necessary.
2. Simple hardware and low cost.
3. High speed of operation.

Disadvantages include:

1. Hit ratio lower than associative mapping methods (Sections 3.2.2 and 3.2.3).
2. Direct mapping is unsuitable for parallel virtual address translation (Section 3.6.1).
3. Performance drops significantly if accesses are made to locations with the same index.

However, as the size of cache increases, the difference in the hit ratios of the direct and associative caches reduces and becomes insignificant. The trend is for larger direct caches, which suit direct mapped caches. Hill (1988) presents a detailed case for the direct mapped cache.

3.2.2 Fully associative mapping

Fully associative mapping requires the cache to be composed of associative memory (content addressable memory, CAM) as used in the main–secondary memory associative mapping scheme (Chapter 2). The incoming memory address is simultaneously compared with all stored addresses using the internal logic of the associative memory, as shown in Figure 3.3. If a match is found, the corresponding data is read out. Single words from anywhere within the main memory could be held in the cache, if the associative part of the cache is capable of holding a full address.

As with the other schemes, the data can be more than one word, i.e. a block of consecutive locations. The whole block can be transferred to and from the cache in one transaction if there are sufficient data paths between the main memory and cache. With only one data word path, the words of the block have to be transferred in separate transactions, and then an additional bit must be stored with each byte/word

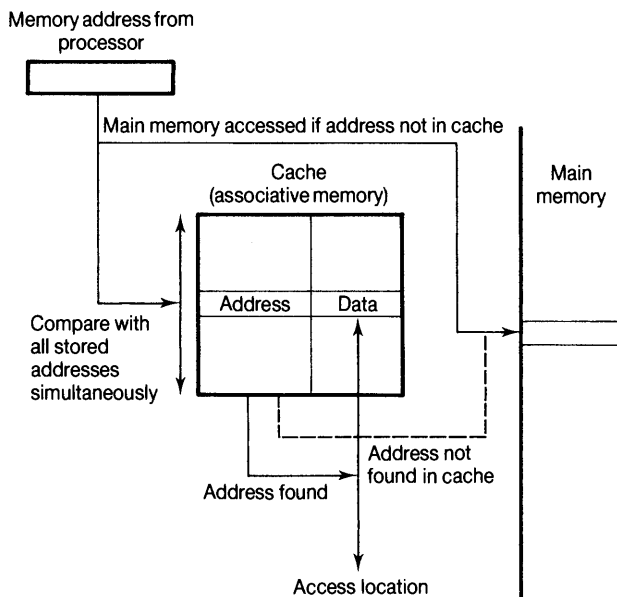


Figure 3.3 Cache with fully associative mapping

in each block to indicate whether valid data is present in the cache.

The fully associative mapping cache gives the greatest flexibility of holding combinations of blocks in the cache and minimum conflict for a given sized cache, but is also the most expensive, due to the cost of the associative memory. It requires a replacement algorithm to select a block to discard upon a miss (as do the following cache organizations) and the algorithm must be implemented in hardware to maintain a high speed of operation.

The fully associative cache can only be formed economically with a moderate size capacity. Microprocessors with small caches often employ the fully associative mechanism, incorporating valid bits within the cache, as shown in Figure 3.4. The single bus of the microprocessor constrains main memory transfers to one byte/word at a time and the cache is loaded one byte/word at a time (which for instructions may be in a burst mode). Each byte in the cache has a valid bit which is set when the byte forms part of a block having the same address as the stored address. When a byte is loaded, the corresponding valid bit is set. Each address tag in the cache identifies a block, though not all the bytes within the block may be part of the current block; they may be parts of previous blocks. When the processor accesses the cache, the address tag from the processor is compared with all the address tags stored in the cache. If a match is found, the valid bit associated with the required byte is checked to see whether the stored byte is a valid part of the block. If it is valid, the byte is accessed, otherwise the main memory is accessed and the cache loaded with the byte. A suitable location is found in the cache, using, say, the least recently used algorithm implemented in hardware. The address tag is updated and the valid bit is set. All valid bits in the block, except those associated with the newly loaded byte, are reset. Subsequent bytes of the block are loaded when they are requested and associated valid bits set then.

Examples include the Z-280 (Zilog, 1986) which has a fully associative cache consisting of 16 blocks. Each block has 16 bytes and there is a 20-bit stored address (called a tag) with each block. Sixteen valid bits are also stored with each block. The full physical address used to locate the byte in the cache has 24 bits. The Z80000 (Zilog, 1984) has a similar fully associative cache of 16 blocks, each of 16 bytes.

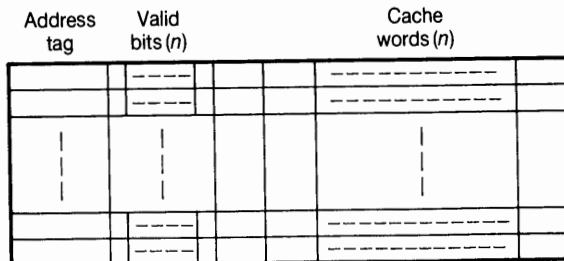


Figure 3.4 Cache with valid bits

3.2.3 Set-associative mapping

In the direct scheme described, all words stored in the cache must have different indices. The tags may be the same or different. In the fully associative scheme, blocks can displace any other block and can be placed anywhere, but the cost of the fully associative scheme becomes prohibitive for large caches and large associative memories operate relatively slowly.

Set-associative mapping allows a limited number of words or blocks, with the same index and different tags, in the cache and can therefore be considered as a compromise between a fully associative cache and a direct mapped cache. The organization is shown in Figure 3.5. The cache is divided into “sets” of blocks. Each set has one or more blocks depending upon the design of the cache. A four-way set-associative cache would have four blocks in each set. The number of blocks in a set is known as the *associativity* or *set size*. Each block in each set has a stored tag which, together with the index (set number), completes the identification of the block. First, the index of the address from the processor is used to access the set. Then, comparators are used to compare all tags of the selected set with the incoming tag.

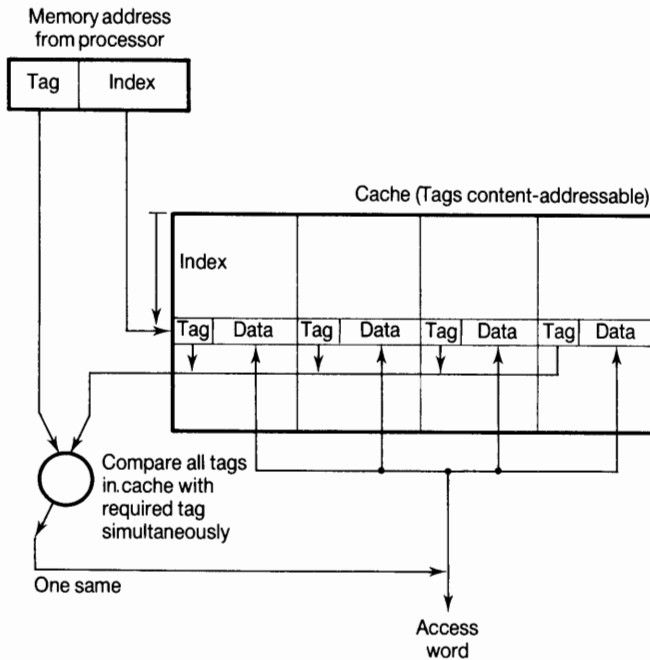


Figure 3.5 Cache with set-associative mapping

If a match is found, the corresponding location is accessed, otherwise, as before, an access to the main memory is made.

The tag address bits are always chosen to be the most significant bits of the full address, the block address bits are the next significant bits and the word within the block is the least significant bits as this spreads out consecutive main memory blocks throughout consecutive sets in the cache. This addressing format is known as *bit selection* and is used by all known systems. (The direct mapped system in Section 3.2.1 also uses bit selection.) In a set-associative cache it would be possible to have the set address bits as the most significant bits of the address and the block address bits as the next significant, with the word within the block as the least significant bits, or with the block address bits as the least significant bits and the word within the block as the middle bits. We will see later (Section 3.6) that the number of bits in each field for a system with both cache and virtual memory is often arranged so that both cache and virtual memory address translations can be done at least partially concurrently.

Notice that the association between the stored tags and the incoming tag is done using comparators and can be shared for each associative search, and all the information, tags and data, can be stored in ordinary random access memory. The number of comparators required in the set-associative cache is given by the number of blocks in a set, not the number of blocks in all, as in a fully associative memory. In a fully associative cache, each tag requires its own comparator within the content addressable memory element. In the set-associative cache, the set can be selected quickly and all the blocks of the set can be read out simultaneously with the tags before waiting for the tag comparisons to be made. After a tag has been identified (assuming that a hit has occurred), the corresponding block can be selected. (Writing cannot be done before the tag has been identified, and it generally requires additional mechanisms, see page 76.)

The replacement algorithm for set-associative mapping need only consider the blocks in one set, as the choice of sets is predetermined by the index (set number) in the address. Hence, with two blocks in each set, for example, only one additional bit is necessary in each set to identify the block to replace. Typically, the set size is 2, 4, 8 or 16. A set size of one block reduces the organization to that of direct mapping and an organization with one set becomes fully associative mapping. For a given number of blocks, there is a design choice between increasing the number of sets or increasing the number of blocks in each set, as the cache size = (number of sets) \times (number of blocks in each set).

3.2.4 Sector mapping

In sector mapping, the main memory and the cache are both divided into sectors, each sector composed of a number of blocks. Any sector in the main memory can map into any sector in the cache and a tag is stored with each sector in the cache to identify the main memory sector address. However, a complete sector is not

transferred to the cache or back to the main memory as one unit. Instead, individual blocks are transferred as required. On cache sector miss, the required block of the sector is transferred into a specific location within one sector. The sector location in the cache is selected by the replacement algorithm and all the other existing blocks in the sector in the cache are from a previous sector. To differentiate between blocks of the sector given by the stored tag and old blocks, a valid bit is associated with each block in each sector. When a new block of a new sector is read into the cache, the valid bit of the block is set, and all the other blocks are marked as invalid. Subsequent accesses to the same sector but to invalid blocks cause the required blocks to be read in and the corresponding valid bits set.

We notice that though sectors can be placed anywhere, once the position of the sector is selected the block must be placed at the appropriate location within the sector, i.e. block i is placed in the i th location from the beginning of the sector, and the block bits act as an index. Hence the replacement algorithm need only consider sector addresses in its replacement algorithm.

Sector mapping was used on the first commercial cache system, IBM System/360 Model 85. In this computer system, there were sixteen sectors, each sector with sixteen blocks. Each block consisted of 64 bytes, giving a total of 1024 bytes in each sector and 16 Kbytes in all. On a miss, 4 bytes were sent to the cache and also to the processor, using four-way interleaved memory; the remaining 60 bytes of the block being transferred subsequently. A true least recently used replacement algorithm was implemented in hardware.

Sector mapping might be regarded as a fully associative mapping scheme with valid bits, as in some microprocessor caches. Each block in the fully associative mapped cache corresponds to a sector, and each byte corresponds to a “sector block”. Hence, though the sector mapping generally lost favor after the System/360 Model 85 (around 1968), perhaps because the hit ratio of the sector mapping was said to have been less than comparable to set-associative mapping, a form of it has reappeared in microprocessor systems. We note that the limited bus width (8, 16 or 32 bits) of microprocessor systems prevents large numbers of bytes/words being transferred simultaneously.

3.3 Fetch and write mechanisms

3.3.1 Fetch policy

We can identify three strategies for fetching bytes or blocks (lines) from the main memory to the cache, namely:

1. Demand fetch.
2. Prefetch.
3. Selective fetch.

The first two strategies have already been encountered in paging systems (Chapter 2). *Demand fetch* is the name given to fetching a block when it is needed and is not already in the cache, i.e. to fetch the required block on a miss. This strategy is the simplest and requires no additional hardware or tags in the cache recording the references, except to identify the block in the cache to be replaced.

Prefetch is the name given to the strategy of fetching blocks before they are requested. A simple prefetch strategy is to prefetch the $(i + 1)$ th block when the i th block is initially referenced (assuming that the $(i + 1)$ th block is not already in the cache) on the expectation that it is likely to be needed if the i th block is needed. Sequential prefetch can reduce the miss ratio by 50 per cent if the cache is large. Unfortunately, fetching the $(i + 1)$ th block means that some other block must be displaced, and this block might be more likely to be referenced than $(i+1)$ th block. On the simple prefetch strategy, not all first references will induce a miss, as some will be to prefetched blocks. Prefetching could be limited to when there has been a miss to the i th block (*prefetching on a miss*).

Selective fetch describes the policy of not always fetching blocks, dependent upon some defined criterion, and in these cases using the main memory rather than the cache to hold the information. For example, shared writable data might be easier to maintain if it is always kept in the main memory and not passed to a cache for access, especially in multiprocessor systems. Cache systems need to be designed so that the processor can access the main memory directly and bypass the cache. Individual locations could be tagged as non-cacheable. Because instructions should never be altered, caches could be split into two parts, an *instruction cache* and a *data cache*, and the write mechanism need only be applied to the data cache. It is necessary to enforce the policy that instructions are not modified during execution. This enforcement policy can be done only on a completely new system and when software need not be brought from a previous system without this policy.

Examples of separate instruction and data cache systems include the National Semiconductor NS32532 32-bit microprocessor (Maytal *et al.*, 1989) which has a 512-byte instruction cache and a 1024-byte data cache on chip, both with a 16-byte block size. The instruction cache is directly mapped. The data cache uses a two-way set-associative cache with a least recently used replacement algorithm and write-through policy (see page 77). Most other microprocessor caches are, or can be, divided into separate instruction and data caches.

3.3.2 Write operations

As reading the required word in the cache does not affect the cache contents, there can be no discrepancy between the cache word and the copy held in the main memory after a memory read instruction. However, in general, writing can occur to cache words and it is possible that the cache word and copy held in the main memory may be different. It is necessary to keep the cache and the main memory identical if input/output transfers operate on the main memory contents, or if

multiple processors operate on the main memory, as in a shared memory multiple processor system.

If we ignore the overhead of maintaining consistency and the time for writing data back to the main memory, then the average access time is given by the previous equation, i.e. $t_a = t_c + (1 - h)t_m$, assuming that all accesses are first made to the cache. The cache access time, t_c , is the time taken to interrogate the cache and discover whether the data is present, and to produce it if it is. If the data item is not present, then it takes t_m seconds to fetch the data item to the processor, including any additional time to load the cache. We can separate the cache access time into a cache interrogate time, t_{ci} , and a subsequent cache read time, t_{cr} . Then the average access time is given by:

$$t_a = t_{ci} + ht_{cr} + (1 - h)t_m$$

The average access time including write operations will depend upon the mechanism used to maintain data consistency. If it is not necessary to keep both main memory and cache memory contents consistent, for example in a system in which all processors and input/output devices access data through the one cache, then the average access time is given as before.

Though the average access time of the cache is a major factor in the performance of the system and will be computed for various read/write strategies in the next sections, the overall speed of computation is influenced by various factors. In particular, the instruction execution speed is determined by various internal operations in addition to instruction fetches and operand accesses. Also, differences in machine architecture can have a profound influence on the overall performance. An increase in the block size without any increase in the cache memory/main memory data paths might even result in a decrease in the overall performance as multiple transfers appear on the data path.

There are two principal alternative mechanisms to update the main memory, namely the *write-through* mechanism and the *write-back* mechanism.

3.3.3 Write-through mechanism

In the *write-through* mechanism, every write operation to the cache is repeated to the main memory, normally at the same time. The additional write operation to the main memory will, of course, take much longer than to the cache and will dominate the access time for write operations. Fortunately, there are usually several read operations between write operations (typically between three and ten). Smith (1982), for example, reports that in one of his studies, 16 per cent of references were write references, though for different programs the percentage varied from 5 per cent to 34 per cent. The average access time of write-through with transfers from main memory to the cache on all misses (read and write) is given by:

78 Computer design techniques

$$\begin{aligned}
 t_a &= t_c + (1 - h)t_b + w(t_m - t_c) \\
 &= (1 - w)t_c + (1 - h)t_b + wt_m \\
 &= (1 - w)t_c + (1 - h + w)t_m \quad \text{if } t_b = t_m
 \end{aligned}$$

where t_b = time to transfer block to cache and w = fraction of write references.

The term $(t_m - t_c)$ is the additional time to write the word to main memory whether a hit or a miss has occurred, given that both cache and main memory write operations occur simultaneously but the main memory write operation must complete before any subsequent cache read/write operations can proceed. For medium and large computer systems, the time to transfer a block is the same as the time to transfer a word/byte, as the memory data path is designed to match the cache, i.e. $t_b = t_m$. For smaller microprocessor systems, separate data transfers are needed for each byte/word transfer of a block. Then $t_b = bt_m$ when there are b bytes in the block.

Suppose $t_c = 25$ ns, $t_m = 200$ ns, $h = 99$ per cent, $w = 20$ per cent, and the memory data path fully matches the cache block size. The average access time would be 62 ns, with misses accounting for 2 ns and write policy accounting for 35 ns. When the data path does not match the block size and more than one transfer is required, the misses become more significant. For example, if the block size is sixteen ($b = 16$) the average access time is 92 ns, with misses accounting for 32 ns and the write policy accounting for 35 ns.

On a cache miss, a block could be transferred from the main memory to the cache whether the miss was caused by a write or by a read operation. The term *fetch on write* is used to describe a policy of bringing a word/block from the main memory into the cache for a write operation. In write-through, fetch on write transfers are often not done on a miss. The information will be written back to the main memory but not kept in the cache. Then the average access time is given by:

$$\begin{aligned}
 t_a &= t_c + (1 - w)(1 - h)t_b + w(t_m - t_c) \\
 &= (1 - w)(t_c + (1 - h)t_b) + wt_m
 \end{aligned}$$

The hit ratio will generally be slightly lower than for the fetch on write policy because altered blocks will not be brought into the cache and might be required during some read operations, depending upon the program. Suppose the hit ratio and other parameters were the same as before, then the average access time is 61.6 ns or, with $b = 16$, 85.6 ns.

The write-through scheme can be enhanced by incorporating buffers, as shown in Figure 3.6, to hold information to be written back to the main memory, freeing the cache for subsequent accesses. (Buffers are also found in write-back schemes, see page 80). For write-through, each item to be written back to the main memory is held in a buffer together with the corresponding main memory address if the transfer cannot be made immediately. The capacity to store more than one data/address pair

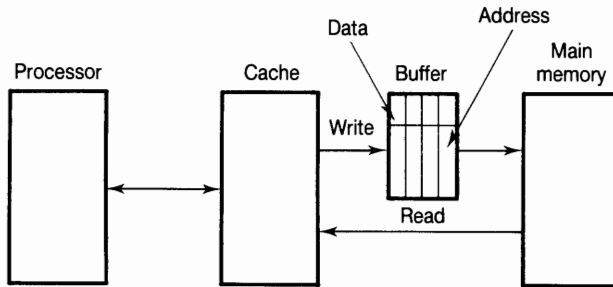


Figure 3.6 Cache with write buffer

is preferable. A capacity of four data/address items is typically sufficient and is used, for example on the IBM 3033, with write-through cache. If the write-through is totally transparent to the cache operation, the average cycle time reduces to that given in Section 3.3.2. Buffers require considerable additional logic to ensure that any request to main memory (by the processor/cache mechanism or another device) checks the buffers. All memory reference addresses have to be compared to the addresses stored in the buffers.

Immediate writing to main memory when new values are generated ensures that the most recent values are held in the main memory and hence that any device or processor accessing the main memory should obtain the most recent values immediately, thus avoiding the need for complicated consistency mechanisms. There will be a latency before the main memory has been updated, and the cache and main memory values are not consistent during this period. If the processor fails, the system can be restored relatively easily. The main memory often has error detection/correction circuitry based upon parity and Hamming codes, but the cache memory may not have this circuitry.

Write-through caches can be found in, for example, the IBM 3033 and VAX-11/780 and earlier computers. They can also be found in many microprocessor cache systems, being easy to implement and to maintain consistently on a single bus system. The Intel 82385 cache controller (Intel, 1987) for the Intel 80386 32-bit microprocessor uses a *posted write-through* policy. Data words are written to the main memory and if copies are maintained in the cache, the cache is updated, otherwise it is unaffected (i.e. no fetch on a write miss). Operands written to the main memory are buffered. The processor continues with the next operation while the cache controller updates the main memory, i.e. the location is identified for write-through, but the processor does not wait for the operation to be completed.

3.3.4 Write-back mechanism

In the *write-back* mechanism, the write operation to the main memory is only done at block replacement time. At this time, the block displaced by the incoming block might be written back to the main memory irrespective of whether the block has been altered. The policy is known as *simple write-back*, and leads to an average access time of:

$$t_a = t_c + (1 - h)t_b + (1 - h)t_b = t_c + 2(1 - h)t_b$$

where one $(1 - h)t_b$ term is due to fetching a block from memory and the other $(1 - h)t_b$ term is due to writing back a block. Write-back normally handles write misses as fetch on write, as opposed to write-through, which often handles write misses as no fetch on write.

The write-back mechanism usually only writes back blocks that have been altered. To implement this policy, a 1-bit tag is associated with each cache block and is set whenever the block is altered. At replacement time, the tags are examined to determine whether it is necessary to write the block back to the main memory. The average access time now becomes:

$$t_a = t_c + (1 - h)t_b + w_b(1 - h)t_b = t_c + (1 - h)(1 + w_b)t_b$$

where w_b is the probability that a block has been altered (fraction of blocks altered). The probability that a block has been altered could be as high as the probability of write references, w , but is likely to be much less, as more than one write reference to the same block is likely and some references to the same byte/word within the block are likely. The hit ratio will be the same as the simple write-back and the same as the write-through with fetch on write. For relatively little extra hardware, the average access time has been reduced by $(1 - h + w_b)t_b$, which is quite significant. However, under this policy the complete block is written back, even if only one word in the block has been altered, and thus the policy results in more traffic than is necessary, especially for memory data paths narrower than a block, but still there is usually less memory traffic than write-through, which causes every alteration to be recorded in the main memory.

The write-back scheme can also be enhanced by incorporating buffers to hold information to be written back to the main memory, just as is possible and normally done with write-through. Apart from the two main types of write policies, write-back and write-through, there is a variation called *write-once*, which is particularly applicable to multiple processor systems (see Section 3.8, page 9).

3.4 Replacement policy

3.4.1 Objectives and constraints

When the required word of a block is not held in the cache, we have seen that it is necessary to transfer the block from the main memory into the cache, displacing an existing block if the cache is full. Except for direct mapping, which does not allow a replacement algorithm, the existing block in the cache can be chosen by one of the algorithms described in Chapter 2 for virtual memory systems. For cache systems, the least recently used algorithm is most commonly used. We shall discuss these algorithms in the context of cache memory systems in the following sections. The replacement mechanism must be implemented totally in hardware, preferably such that the selection can be made completely during the main memory cycle for fetching the new block. Ideally, the block replaced will not be needed again in the future. However, such future events cannot be known and a decision has to be made based upon facts that are known at the time.

In Chapter 2, algorithms were classified as *fixed partition algorithms* or *variable partition algorithms*. In fixed partition algorithms, a fixed amount of memory is allocated. In variable partition algorithms the memory allocation may be altered by the algorithm as in the working set algorithm. Cache memory replacement algorithms always use fixed partition algorithms, given the relatively small size of caches. Replacement algorithms are also classified in Chapter 2 as *usage-based* or *non-usage-based*. For a cache, usage and non-usage algorithms are both candidates for the replacement algorithm; a critical factor is often the amount of hardware necessary to implement the algorithm, as the differences in performance might be less important than the differences in cost.

A usage-based replacement algorithm for the fully associative cache needs to take the usage (references) to all stored blocks into account. A usage-based replacement algorithm for a set-associative cache needs to take only the blocks in one set into account at replacement time, though a record needs to be kept of relative usage of the blocks in each set.

Whatever type of algorithm (usage-based or non-usage-based), there are generally fewer blocks to consider in a set-associative cache than in a similar sized fully associative cache, and the logic is simpler. For a two-way set-associative cache, only one bit per set is needed to indicate which item should be replaced. The associativity of caches is often small (two or four) but some large systems have set sizes up to sixteen (e.g. IBM 3033) and then the required logic and its speed to implement the replacement algorithm must be carefully considered.

3.4.2 Random replacement algorithm

Perhaps the easiest replacement algorithm to implement is a pseudorandom replacement algorithm. A true random replacement algorithm would select a block to replace in a totally random order, with no regard to memory references or previous selections; practical random replacement algorithms can approximate this algorithm in one or several ways. For example, one counter for the whole cache could be incremented at intervals (for example after each clock cycle, or after each reference, irrespective of whether it is a hit or a miss). The value held in the counter identifies the block in the cache (if fully associative) or the block in the set if it is a set-associative cache. The counter should have sufficient bits to identify any block. For a fully associative cache, an n -bit counter is necessary if there are 2^n words in the cache. For a four-way set-associative cache, one 2-bit counter would be sufficient, together with logic to increment the counter.

3.4.3 First-in first-out replacement algorithm

The first-in first-out replacement algorithm removes the block which has been in the cache for the longest time. The first-in first-out algorithm would naturally be implemented with a first-in first-out queue of block addresses, but can be more easily implemented with counters, only one counter for a fully associative cache or one counter for each set in a set-associative cache, each with a sufficient number of bits to identify the block.

3.4.4 Least recently used algorithm for a cache

The least recently used algorithm (LRU) is popular for cache systems and can be implemented fully when the number of blocks involved is small. There are several ways the algorithm can be implemented in hardware for a cache, these include:

1. Counters.
2. Register stack.
3. Reference matrix.
4. Approximate methods.

In the counter implementation, a counter is associated with each block. A simple implementation would be to increment each counter at regular intervals and to reset a counter when the associated block had been referenced. Hence the value in each counter would indicate the age of a block since last referenced. The block with the largest age would be replaced at replacement time.

The algorithm for these *aging registers* can be modified to take into account the fact that the counters have a fixed number of bits and that only a relative age is

required, as follows. When a hit occurs, the counter associated with the hit block is reset to 0, indicating that it is the most recently used, and all counters having a smaller value than the “hit block” counter originally are incremented by 1. All counters having a larger value are unaffected. On a miss when the cache is not full, the counter associated with the incoming block is reset to 0 and all other counters are incremented by 1. On a miss when the cache is full, the block with a counter set at the maximum value (three for a 2-bit counter and four sets) is chosen for replacement and then the counter is reset to 0, and all other counters incremented by 1. The counter with the largest value identifies the least recently used block. For example, suppose there are four blocks in the set of a set-associative cache. A 2-bit counter is sufficient for each block. Let the counters in one set be C_0 , C_1 , C_2 and C_3 . Initially, all the counters are set to 0. As an example, we obtain the sequence in Table 3.1 for the conditions specified.

In the register stack implementation, a set of n -bit registers is formed, one for each block in the set to be considered. The most recently used block is recorded at the “top” of the stack and the least recently used block at the bottom. Actually, the set of registers does not form a conventional stack, as both ends and internal values are accessible. The value held in one register is passed to the next register under certain conditions. When a block is referenced, starting at the top of the stack, the values held in the registers are shifted one place towards the bottom of the stack until a register is found to hold the same value as the incoming block identification. Subsequent registers are not shifted. The top register is loaded with the incoming block identification. This has the effect of moving the contents of the register holding the incoming block number to the top of the stack. It is left as a logic design exercise to devise the required logic. It will be found that the logic is fairly substantial and slow, and not really a practical solution, given the alternative reference matrix method.

The reference matrix method centers around a matrix of status bits. There is more than one version of the method. In one version (Smith, 1982), the upper triangular matrix of a $B \times B$ matrix is formed without the diagonal, if there are B blocks to

Table 3.1 Least recently used algorithm using counters – set size of four blocks

Block referenced	C_0	C_1	C_2	C_3	Subsequent actions
Initialization	0	0	0	0	
Miss	0	1	1	1	Block 0 filled
Miss	1	0	2	2	Block 1 filled
Block 0	0	1	2	2	Block 0 accessed
Miss	1	2	0	3	Block 2 filled
Miss	2	3	1	0	Block 3 filled
Block 1	3	0	2	1	Block 1 accessed
Miss	0	1	3	2	Block 0 replaced
Miss	1	2	0	3	Block 2 replaced

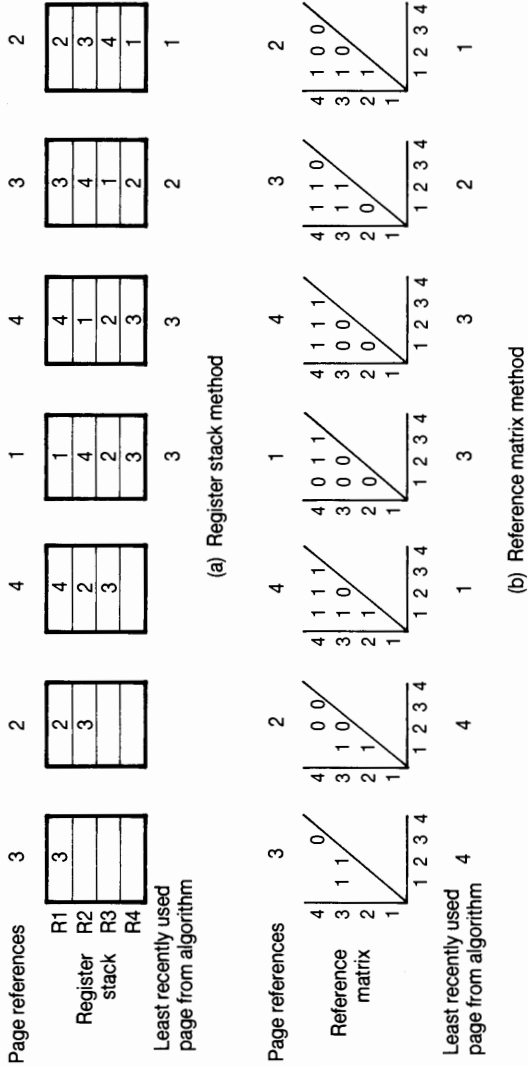


Figure 3.7 Least recently used replacement algorithm implementation
(a) Register stack method **(b)** Reference matrix method

consider. The triangular matrix has $(B \times (B - 1))/2$ bits. When the i th block is referenced, all the bits in the i th row of the matrix are set to one and then all the bits in the i th column are set to zero. The least recently used block is one which has all zeros in its row and all ones in its column, which can be detected easily by logic. The method is demonstrated in Figure 3.7 for $B = 4$ and the reference sequence 3, 2, 4, 1, 4, 3, 3, together with the values that would be obtained using a register stack. Maruyama (1975) extends the reference matrix method to select m least recently used pages where m can be greater than one.

When the number of blocks to consider increases above about four to eight, approximate methods are necessary for the LRU algorithm. Figure 3.8 shows a two-stage approximation method with eight blocks, which is applicable to any replacement algorithm and has been used with the least recently used algorithm (IBM 370/168-3). The eight blocks are divided into four pairs, and each pair has one status bit to indicate the most/least recently used block in the pair (simply set or reset by reference to each block). The least recently used replacement algorithm now only considers the four pairs. Six status bits are necessary (using the reference matrix) to identify the least recently used pair which, together with the status bit of the pair, identifies the least recently used block of a pair. The method can be extended to further levels. For example, sixteen blocks can be divided into four groups, each group having two pairs. One status bit can be associated with each pair, identifying the block in the pair, and another with each group, identifying the group in a pair of groups. A true least recently used algorithm is applied to the groups. In fact, the scheme could be taken to its logical conclusion of extending to a full binary tree. It is left as an exercise to determine whether this would make a reasonable design choice.

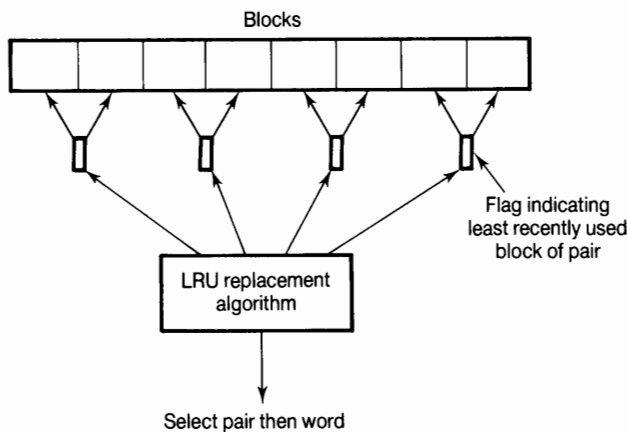


Figure 3.8 Two-stage replacement algorithm

3.5 Cache performance

For any given cache, if the size of the cache is increased, the miss ratio decreases and generally the performance of the system increases. The actual miss ratio depends very heavily on the programs being executed and the overall workload, such that an exact fixed value of miss ratio cannot be found for any particular computer system. The miss ratio also depends upon the cache organization chosen, the size of the internal divisions of the cache, the write policy and the replacement algorithm.

In the design phase of a computer system incorporating a cache, there are three basic methods of obtaining an estimate of the miss ratio:

1. Trace-driven simulation.
2. Direct measurement.
3. Mathematical modelling.

The trace-driven method is perhaps the most popular, giving miss ratios in actual situations which can be varied. In this method, programs are selected for execution on a computer system not necessarily having a cache. (It does not matter whether the system has a cache of any kind but the system should have a processor of the type in the system under investigation.) A record of the instruction and data references is kept. The processor trace facility (assuming there is one) is generally used. After each test program instruction has been executed, a trace interrupt causes a special routine to be executed; this records the instruction and data references. The routine usually has to recognize the effective addresses of the test program instructions, but this is relatively straightforward to accomplish. For example, the Intel 8086 microprocessor (and many other processors) has an instruction which returns the effective address rather than the addressed operand (LEA instruction – load effective address) which can be used in many cases.

Specific cache operations are then simulated, using the instruction/data references that have been gathered, to determine the miss ratio. Generally, a very large number of references needs to be gathered to obtain accurate figures, usually at least several hundred thousand references. When the cache simulation is begun, a relatively large number of initial misses will be due to the cache containing no information. This of course occurs in practice; such *cold starts* produce a disproportionate number of misses. Also occurring in practice are *context switch misses*, which occur frequently in a multiprogramming environment. In such an environment, the computer system will execute part of one program and will then return to the operating system to select another program. It will then execute part of this program and the procedure will be repeated with all user programs. Even in a single-user system, there will be operating system calls causing a change in context. However, many trace/simulation experiments will assume that warm start results are required. For *warm start* results (when the effects of the start-up and context changes are not considered) traces of

perhaps 300 000–1 000 000 references, or thereabouts, are required. Since trace programs will operate substantially more slowly than the test program alone, experiments need to be performed for many hours. There are techniques for substantially reducing the number of references but still obtaining accurate values and the reader is directed to Stone (1987) for further details. Other trace experiments have been directed towards transient behavior of cold start and context switch environments (Strecker, 1983).

Figure 3.9 shows representative warm start results obtained by the trace method for three programs, A, B and C, and is based on the results of Smith (1987a). There can be significant differences between individual programs; in fact there can be enormous differences. As expected, as the overall size of the cache is increased, the hit ratio initially decreases significantly but this change reduces as the cache size is increased further and after a certain cache size, the change in miss ratio becomes unnoticeable.

When computed as a function of block size, as shown in Figure 3.10, for a given cache size, the miss ratio decreases with increasing cache size but often a minimum is reached whereafter the miss ratio increases. Hence there is an optimum value for the block size for a particular cache size. For example, a good block size for a 256 byte cache, according to Figure 3.9, would be perhaps 64 bytes. As the cache size is increased, the optimum block size increases. The overall effect is caused partly by program locality, when programs reference more than one contiguous area and fewer of these areas can reside in the cache simultaneously as the block size increases. For example, when the number of blocks is half the cache size, only two different contiguous areas could be stored in the cache simultaneously. More significantly, as the block size increases, there is more contiguous information in the cache and more likelihood that some of this information will not be wanted. This

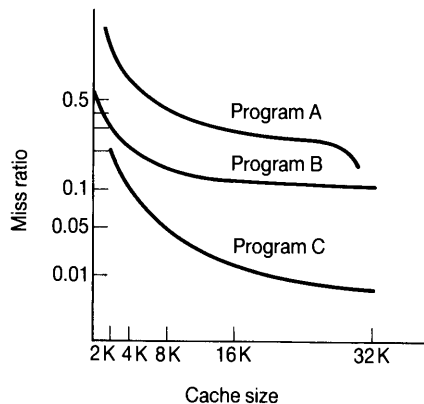


Figure 3.9 Miss ratio against cache size

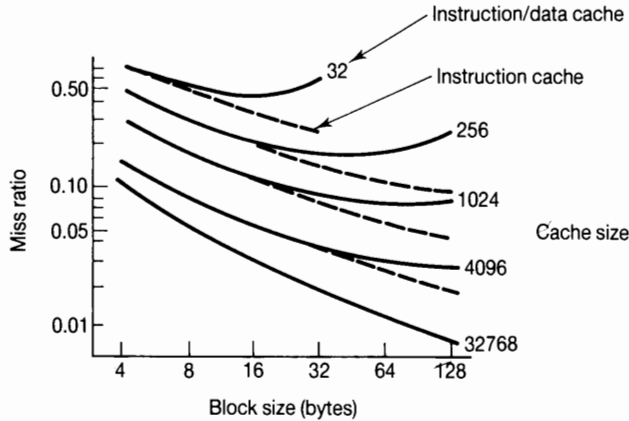


Figure 3.10 Miss ratio against block size

unwanted information displaces information that might be wanted in the future (an effect called *memory pollution*). We note also that as the block size increases, more information must be transferred into the cache (assuming that the whole block is transferred at one time), which decreases the operational speed if some of the information is not wanted.

The miss ratio for a particular cache organization differs for instruction references and data references. For data references, we might get a more pronounced “knee” in the miss ratio curve as plotted against block size, particularly for small cache sizes, whereas for instruction fetches (shown by the dotted lines in Figure 3.10) there is a general decrease in miss ratio for increasing block size. This effect is because instructions are more likely to be referenced sequentially and more of the information in larger blocks is likely to be needed. (However, data items are also often stored in contiguous locations.)

The memory reference sequence could be obtained by direct measurement by attaching special monitoring hardware to the computer system to record the memory references. (In fact, for the study reported by Smith (1987a) simulation trace references and hardware monitored references were both used.) Monitoring hardware is feasible for relatively slow and medium speed systems, including microprocessor systems, but might be difficult, if not impossible, to implement for the fastest computer system operating at full speed. It might be possible to decrease the speed of operation of these fast computer systems by decreasing the clock rate so that the memory references could be captured. Once the sequence has been obtained, it can be processed by a cache simulation program, as before.

One advantage of direct measurement is that all memory references can be obtained. Some instructions might not be easily traced in software, notably operating

system reserved instructions. Also, hardware monitors operate at a much greater speed than software monitors and may be left on a system during a normal operation without appreciably affecting the operation. In contrast, software trace monitors slow the system down drastically. To simulate context switches, several programs can be traced and each program executed during the cache simulation for, say, 20 000 references in sequence. For example, in the study by Smith (1987a), twenty-seven program traces were selected from five different types of computer system as a representative sample.

An alternative method of gathering cache results is to construct the cache physically in a system and make direct measurements. As a design tool before finally deciding on the cache size and organization, the method has the disadvantage that cache parameters cannot be easily altered. However, it can be performed after a cache design has been selected to confirm that the choice was appropriate.

Mathematical models of caches can be developed based upon differential equations, statistical and probabilistic techniques. After a mathematical model has been obtained, it is generally compared to experimental simulation results. Mathematical curve fitting expressions can also be derived based upon trace-driven results, and extrapolated for designs not covered in the trace-driven simulation.

The contents of a cache in a multiprogrammed system will exhibit constant changes from one program to another. During the transition between programs, a much higher miss ratio will occur as the new program displaces the old program in the cache. The miss ratio will reduce until a steady state is reached with the cache holding the new program. This important aspect is called the *transient behavior* of caches and has been studied mathematically by Strecker (1983), and by Thiebaut and Stone (1987). Strecker has developed a formula the rate of change of the number of locations filled in the cache as:

$$\frac{dn}{dt} = m(n)p(n)$$

where $m(n)$ is the miss ratio with n locations filled in the cache and $p(n)$ is the probability that a miss results in a new location being filled. ($p(n)$ is zero if the cache is filled, one if the cache is not filled and any free location can be used, i.e. in a fully associative cache, and less than one with direct and set-associative caches, which place restraints upon the availability of locations for incoming blocks.) Strecker assumes that the probability is numerically equal to the fraction of free cache locations, i.e.:

$$p(n) = \frac{s - n}{s}$$

where s is the size of the cache. The reasonably good approximation to the miss

ratio is given as:

$$m(n) = \frac{a + bn}{a + n}$$

where a and b are constants to be found from trace results. Hence we obtain:

$$\frac{dn}{dt} = \frac{(a + bn)(s - n)}{(a + n)s}$$

It is left as an exercise to solve this equation (see Strecker, 1983).

Thiebaut and Stone (1987) introduced the term *footprint* to describe the active portion of a process that is present in the cache. Footprints of two processes reside in the cache during a transition from one program to another. Probabilistic equations are derived (see Stone, 1987).

Mathematical modelling is useful in helping to see the effect of changing parameters, but mathematical models cannot capture the vast differences in programs.

3.6 Virtual memory systems with cache memory

In a computer system with virtual memory, we can insert the cache after the virtual–real address translation, so that the cache holds real address tags and the comparison of addresses is done with real addresses. Alternatively, we can insert the cache before the virtual–real translation so that the cache holds virtual address tags and the comparison of addresses is done using virtual addresses. Let us first consider the former case, which is much less complicated and has fewer repercussions on the rest of the system design.

3.6.1 Addressing cache with real addresses

Though it is not necessary for correct operation, it is common to perform the virtual–real translation at the same time as some independent part of the cache selection operation to gain an improvement in speed. The overlap is done in the following way. As we have seen in Chapter 2, the address from the processor in a paged virtual memory system is divided into two fields, the most significant field identifying the page and the least significant field identifying the word (line) within the page. The division into page and line is fixed for a particular system and made so that a suitable sized block of information is transferred between the main and the secondary memories. In a cache system, the address is also divided into fields – a most significant field (the tag field corresponding to the tags stored in the cache) and a less significant field (to select the set (in set-associative cache) and to select the block and word within the block). If the tag field corresponds directly to the page field in the real address, then the set selection can be done with the next significant bits of the address before the virtual address translation is done, and the virtual address translation can be

performed while the set selection is being done. When the address translation has been done, and a real page address produced, this address can be compared with the tags selected from the cache, as shown in Figure 3.11. On a cache miss, the real address is immediately available for selecting the block in main memory, assuming a page fault has not occurred and the block can be transferred into the cache directly.

Clearly, as described, the overlap mechanism relies on the page size being the same as the overall cache size (irrespective of the organization), although some variations in the lengths of the fields are possible while still keeping some concurrent operations. In particular, the page size can be larger, so that there are more bits for the line than needed for the set/block/word selection in the cache. The extra bits are then concatenated with the real page address before being compared with the tags.

3.6.2 Addressing cache with virtual addresses

If the cache is addressed with virtual addresses these are immediately available for selecting a word within the cache and there is a potential increase in speed over a real addressed cache. Only on a cache miss would it be necessary to translate a virtual address into a real address, and there is more time then. Clearly, if the tag field of the virtual address is larger than the real address, the tag fields in the cache would be larger and there would be more associated comparators. Similarly, if the

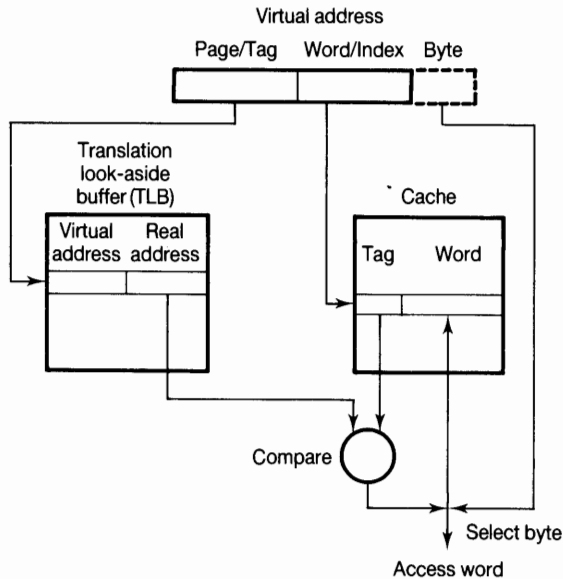


Figure 3.11 Cache with translation look-aside buffer

virtual address is smaller than the real address, the tag fields in the cache would be smaller and there would be fewer comparators. Often though, the virtual and real address tags have the same number of bits. A particular advantage of a virtual addressed cache is that there is no need for overlap between the virtual/real address translation and the cache operation, as there is no translation mechanism for cache hits. So the division of addresses into fields in the virtual/real addresses and the division of fields in the cache selection mechanism can be designed separately and need not have any interrelationship.

However, though the virtual addressed cache is an apparently attractive solution, it has a complication concerned with the relationship between virtual addresses in different processes which may be in the cache together. It is possible for different virtual addresses in different processes to map into the same real address. Such virtual addresses are known as *synonyms* – from the word denoting the same thing(s) as another but suitable for different contexts. Synonyms are especially likely if the addressed location is shared between processes, but can also occur if programs request the operating system to use different virtual addresses for the same real address. Synonyms can occur when an input/output device uses real addresses to access main memory accessible by the programs. They can also occur in multi-processor systems when processors share memory using different virtual addresses. It is also possible for the same virtual address generated in different processes to map into different real addresses.

Process or other tags could be attached to the addresses to differentiate between virtual addresses of processes, but this adds a complication to the cache design, and would still allow multiple copies of the same real block in the cache simultaneously. Of course, synonyms could be disallowed by placing restrictions on virtual addresses. For example, each location in shared code could be forced to have only one virtual address. This approach is only acceptable for shared operating system code and is done in the IBM MVS operating system.

Otherwise, synonyms are handled in virtual addressed caches by the use of a *reverse translation buffer* (RTB), also called an *inverse translation buffer* (ITB). On a cache miss, the virtual address is translated into a real address using the virtual–real translation look-aside buffer (TLB) to access the main memory. When the real address has been formed, a reverse translation occurs to identify all virtual addresses given under the same real address. This reverse translation can be performed at the same time as the main memory cycle. If the real address is given by another virtual address already existing in the cache, the virtual address is renamed to eliminate multiple copies of the same block. The information from the main memory is not needed and is discarded. If a synonym does not exist, the main memory information is accepted and loaded into the cache.

When there are direct accesses to the main memory by devices such as a direct memory access (DMA) input/output device, the associated block in the cache, if present, must be recognized and invalidated (see Section 3.2.2). To identify the block, a real–virtual address translation also needs to be performed using a reverse translation buffer.

3.6.3 Access time

The average access time of a system with both a cache and a paged virtual memory has several components, depending on one of several situations arising – whether the real address (assuming a real addressed cache) is in the translation look-aside buffer, the cache or the main memory and whether the data is in the cache or the main memory. The translation look-aside buffer is used to perform the address translation when the virtual page is in the translation look-aside buffer. If there is a miss in the translation look-aside buffer, the translation is performed by accessing a page table which may be in the cache or in the main memory. There are six combinations of accesses, namely:

1. Address in the translation look-aside buffer, data in the cache.
2. Address in the translation look-aside buffer, data in the main memory.
3. Address in the cache, data in the cache.
4. Address in the cache, data in the main memory.
5. Address in the main memory, data in the cache.
6. Address in the main memory, data in the main memory.

(Part of the page table could be in the secondary memory, but we will not consider this possibility.) Suppose there is no overlap between translation look-aside buffer translation and cache access and the following times apply:

Translation look-aside buffer address translation time (or to generate a TLB miss)	=	25 ns
Cache time to determine whether address in cache	=	25 ns
Cache data fetch if address in cache	=	25 ns
Main memory read access time	=	200 ns
Translation look-aside buffer hit ratio	=	0.9
Cache hit ratio	=	0.95

the access times and probabilities of the various access combinations are given in Table 3.2.

Table 3.2 Access times and probabilities of the various access combinations

Access times		Probabilities	
25 + 25 + 25	= 75 ns	0.9×0.95	= 0.855
25 + 25 + 200	= 250 ns	0.9×0.05	= 0.045
25 + 25 + 25 + 25 + 25	= 125 ns	$0.1 \times 0.95 \times 0.95$	= 0.09025
25 + 25 + 25 + 25 + 200	= 300 ns	$0.1 \times 0.95 \times 0.05$	= 0.00475
25 + 25 + 200 + 25 + 25 + 25	= 325 ns	$0.1 \times 0.05 \times 0.95$	= 0.00475
25 + 25 + 200 + 25 + 25 + 200	= 500 ns	$0.1 \times 0.05 \times 0.05$	= 0.00025

94 Computer design techniques

The average access time is given by:

$$(75 \times 0.855) + (250 \times 0.045) + (125 \times 0.09025) + (300 \times 0.00475) + (325 \times 0.00475) + (500 \times 0.00025) = 89.75 \text{ ns (64.125 ns on a cache hit)}$$

If the virtual memory system also incorporates two-level paging or segments, further combinations exist. The calculation can easily be modified to take into account partial overlap between the TLB access and cache access.

3.7 Disk caches

The concept of a cache can be applied to the main memory/secondary memory interface. A disk cache is a random access memory introduced between the disk and the normal main memory of the system. It can be placed within the disk unit, as shown in Figure 3.12, or within the computer system proper. The disk cache has considerable capacity, perhaps greater than 8 Mbytes, and holds blocks from the disk which are likely to be used in the near future. The blocks are selected from previous accesses in much the same way as blocks are placed in a main memory cache. A disk cache controller activates the disk transfers. The principle of locality, which makes main memory caches effective, also makes disk caches effective and reduces the effective input/output data page transfer time, perhaps from 20–30 ms to 2–5 ms, depending upon the size of page transfer to the main memory. The disk cache is implemented using semiconductor memory of the same type as normal main memory, and clearly such memory could have been added to the main memory as a design alternative. It is interesting to note that some operating systems, such as UNIX, employ a software cache technique of maintaining an input/output buffer in the main memory.

The unit of transfer between the disk and the disk cache could be a sector, multiple sectors or one or more tracks. A minimum unit of one track is one

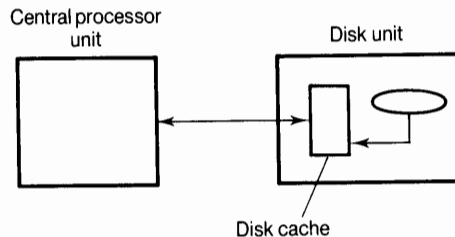


Figure 3.12 Disk cache in disk unit

candidate (Grossman, 1985), as is transferring the information from the selected sector to the end of the track. A write-through policy has the advantage of simplifying error recovery. Not all the information from/to the disk need pass through the disk cache and some data/code might be better not using the cache. One possibility is to have a dynamic cache on/off mechanism which causes the cache to be bypassed under selected circumstances.

Perhaps one of the main attractions of placing the additional cache memory in the disk unit is that existing software and hardware may not need to be changed and substantial improvements in speed can be obtained in an existing system. Most commercial disk caches are integrated into the disk units. Examples include the IBM 3880 Model 23 Cache Storage Controls with an 8–64 Mbyte cache. Disk caches have also been introduced into personal computer systems. It is preferable to be able to access the disk cache from the processor and to allow disk cache transfers between the disk cache and disk simultaneously, as disk transfers might be one or more tracks and such transfers can take a considerable time. Some early commercial disk caches did not have this feature (for example the IBM 3880 Model 13).

Disk caches normally incorporate error detection and correction. For example the disk cache incorporated into the IBM 3880 Model 23 has error detection/correction to detect all triple-bit errors, and correct all double-bit errors and most triple-bit errors. The earlier IBM 3880 Model 13, having a 4–8 Mbyte cache, could detect double errors and correct single-bit errors (Smith, 1985). Both these disk drives maintain copies of data in the cache using a least recently used replacement algorithm.

3.8 Caches in multiprocessor systems

In this section we will briefly review the methods suggested to cope with multiple processors each having caches, or having access to caches. Multiprocessor systems will be discussed in detail in subsequent chapters. In a situation of more than one cache, it is possible that copies of the same code/data are held in more than one cache, and are accessed by different processors. Reading different copies of the same code/data does not generally cause a problem. A complication only exists if individual processors alter their copies of data, because shared data copies should generally be kept identical for correct operation. We note that write-through is not sufficient, or even necessary, for maintaining cache coherence, as more than one processor writing-through the cache does not keep all the values the same and up to date. Several possibilities exist to maintain *cache coherence*, in particular:

1. Shared caches.
2. Non-cacheable items.
3. Sloop bus mechanism.
4. Broadcast write mechanisms.
5. Directory methods.

Clearly, a single cache shared by all processors with the appropriate controls would maintain cache coherence. Also, a shared cache might be feasible for DMA devices accessing the cache directly rather than relying on a write policy. However, with several processors the performance of the system would seriously degrade, due to contention. In a multiprocessor system with more than one memory module accessible by all the processors, an appropriate place for each cache is attached to the processor, as shown in Figure 3.13. It would also be possible to place the caches in front of each memory module, but this arrangement would not decrease the interconnection traffic and contention.

Cache coherence problems only occur on data that can be altered, and such writable data could be maintained only in the shared main memory and not placed in the cache at all. Additional software mechanisms are needed to keep strict control on the shared writable data, normally through the use of critical sections and semaphores (see Chapter 6).

The *snoop bus* mechanism or *bus watcher* is particularly suitable for single bus systems, as found in many microprocessor systems. In the snoop bus mechanism, a bus watcher unit for each processor/cache observes the transactions on the bus and in particular monitors all memory write operations. If a location in main memory is altered and a copy exists in the cache, the bus watcher unit invalidates the cache copy by resetting the corresponding valid bit in the cache. This action requires the unit to have access not only to the valid bits in the cache, but also to the tags in the cache, or copies of the cache tags, in order to compare the main memory address tag with the cache tag. Alternatively, the cache word/block with the same index as the main memory location can be invalidated, whether or not the tags correspond. The unit then does not need to access the tags, though access to the valid bits is still

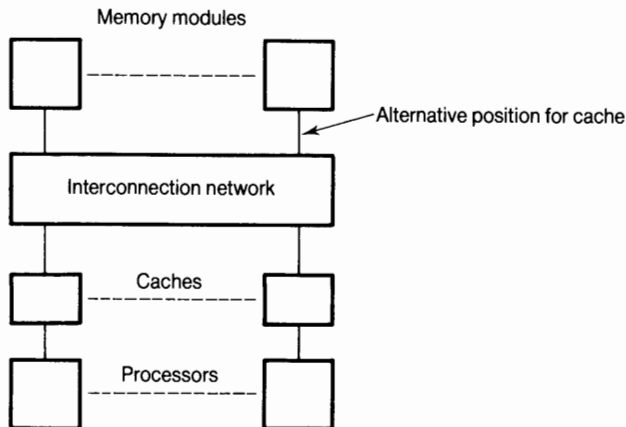


Figure 3.13 Multiprocessor with local caches

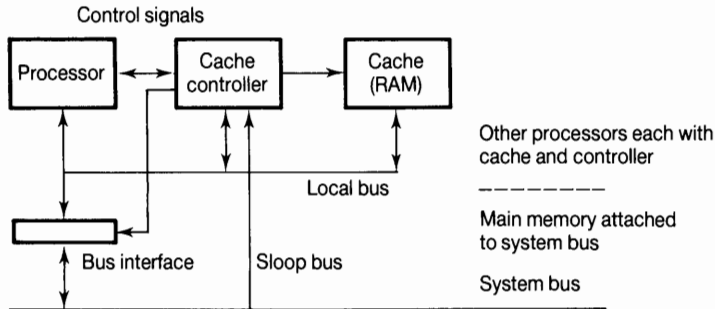


Figure 3.14 Cache with sloop bus controller

necessary. However, the unit might mark as invalid a cache block which does not correspond to an altered main memory word, because the cache location with the same index as the main memory location would be invalidated, irrespective of the values of the tags.

Figure 3.14 shows a representative microprocessor implementation based upon an 80386 processor and an 82385 sloop bus cache controller (Intel, 1987b). The processor accesses the cache through a local bus, all accesses being controlled by the cache controller. For a cache miss that requires access to the main memory on the system bus, the cache controller sends the request through the system bus to the main memory and loads the returning data into the cache. Write accesses with a sloop bus are conveniently handled by write-through.

In *write-once*, the first time a processor makes write reference to a location in the cache, the main memory is also updated in a write-through manner. The fact is recorded in such a way that other processors can recognize that the location has been updated and now cannot be accessed by them. If the stored information was also stored in any other cache, these copies are invalidated by resetting valid bits in the caches. Subsequently, if the first processor again performs a write operation to the location, only the cache is altered, and the main memory is updated only when the block is replaced as in write-back.

In *broadcast writes*, every cache write request is also sent to all other caches in the system. Such broadcast writes interrogate the caches to discover whether each cache holds the information being altered. The copy is then either updated (update write) or an invalidated bit associated with the cache line is set to show that the copy is now incorrect. The use of invalidating words is generally preferable to update writes as multiple update writes by different processors to the same location might cause inconsistencies between caches. In any event, significant additional memory transactions occur with the broadcast method, though it has been implemented on large computer systems (for example IBM 3033).

In one *directory method* (Smith, 1982), if a block has not been modified it may

exist in several caches simultaneously, but once the block is altered by one processor, all other copies are invalidated, and a valid block then exists only in one cache, initially the cache associated with the processor that made the alteration. A subsequent read operation to that block by another processor causes the block to be transferred to the requesting cache so that multiple copies exist again, until a write operation occurs, which invalidates the copies not immediately updated.

The mechanism is achieved through the use of a directory of bits created in the main memory. One set of bits is created for each block that can be transferred into the caches. One bit in each of these sets is for each cache, as shown in Figure 3.15. Each set of bits has one further bit – a *block modified* bit to indicate that the block has been altered. When this occurs only one cache may hold the block and only one of the other bits can be set. If the block has not been altered, the modified bit is reset. Then, it is possible for more than one cache to hold the block and correspondingly more than one bit set in the directory to indicate this fact.

Each block in each cache has a bit which is set if the block is the only valid copy. This bit is set upon a write operation and the block has been transferred into the cache from another cache. There are various situations that can arise in the multiple cache system (i.e. combinations of read/write, hit/miss, present in another cache/not present in another cache, altered/not altered) and the directory method must cope with these situations. On a cache read operation when the block is already in the cache, no directory and private bit operations are necessary. On a read operation when the block is not in the cache, the modified bit of the block in the main directory must be checked to see whether it has been altered in some other cache. If the block is altered, it must be transferred into the cache and other copies invalidated. The main directory is also updated, including resetting the modified bit. If the missing block has not been altered, the copy is sent to the cache and the directory is updated.

On a cache write operation when the block is in the cache, first the private bit is checked to see whether it owns the only copy of the block. If it does own the only

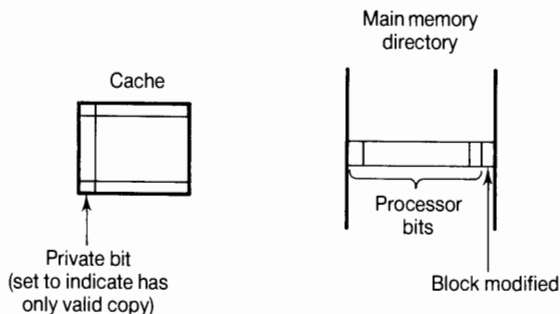


Figure 3.15 A directory method to maintain cache coherence

copy, the block is simply updated. If it does not own the only copy, the main directory is examined to find the other copies. These copies are invalidated if the directory allows a change in ownership. On a write operation when the block is not in the cache, the main directory is updated and the block is transferred to the cache.

There are several variations on basic cache coherence techniques. Mathematical performance analysis of seven different multiprocessor cache coherence techniques for single bus systems is given in Yang, Bhuyan and Liu (1989).

PROBLEMS

3.1 Choose suitable memory interleaving to obtain an average access time of less than 50 ns given that the main memory has an access time of 150 ns and a cache has an access time of 35 ns. If ten locations hold a loop of instructions and the loop is repeated sixty times, what is the average access time?

3.2 What is the average access time of a system having three levels of memory – a cache memory, a semiconductor main memory and magnetic disk secondary memory – if the access times of the memories are 20 ns, 200 ns and 2 ms, respectively? The cache hit ratio is 80 per cent and the main memory hit ratio is 99 per cent.

3.3 A computer employs a 1 Mbyte 32-bit word main memory and a cache of 512 words. Determine the number of bits in each field of the address in the following organizations:

1. Direct mapping with a block size of one word.
2. Direct mapping with a block size of eight words.
3. Set-associative mapping with a set size of four words.

3.4 Derive an expression for the hit ratio of a direct mapped cache assuming there is an equal likelihood of any location in the main memory being accessed (in practice this assumption is not true). Repeat for a two-way set-associative mapped cache. Determine the size of memory at which the direct mapped cache has a hit ratio within 10 per cent of the set-associative cache.

3.5 Design the logic to implement the least recently used replacement algorithm for four blocks using a register stack.

3.6 Design the logic to implement the least recently used replacement algorithm for four blocks using the reference matrix method.

3.7 Solve the equation given in Section 3.5:

$$\frac{dn}{dt} = \frac{(a + bn)(s - n)}{(a + n)s}$$

for n where n locations are filled in the cache, s is the size of the cache, and a and b are constants.

3.8 Determine the conditions in which a write-through policy creates more misses than simple write-back policy, given that the hit ratio is the same in both cases.

3.9 Determine the conditions in which a write-through policy with no fetch on write creates more misses than a write-through policy with fetch on write, given that fetch on write creates 10 per cent higher hit ratio.

3.10 Determine the average access time in a computer system employing a cache, given that the main memory access time is 125 ns, the cache access time is 30 ns and the hit ratio is 85 per cent. The write-through policy is used and 20 per cent of memory requests are write requests.

3.11 Repeat Problem 3.10 assuming a write-back policy is used, and the block size is sixteen words fully interleaved.

3.12 Using aging counters to implement the least recently used algorithm, as described in Section 3.4.4, derive the numbers held in the counters after each of the following pages has been referenced:

2, 6, 9, 7, 2, 3, 2, 9, 6, 2, 7, 4

given that the cache holds four pages.

3.13 Show how a reference matrix as described in Section 3.4.4, can be used to implement the least recently used algorithm with the sequence:

2, 6, 9, 7, 2, 3, 2, 9, 6, 2, 7, 4

given that the cache holds four pages.

3.14 A cache in a system with virtual memory is addressed with a real address. Both the real addresses and virtual addresses have thirty-two bits and the page size is 512 words. The set size is two. Determine the division of fields in the address to achieve full overlap between the page translation and set selection. Suppose the cache must have only two

pages, give a design showing the components and address formats.

3.15 A disk cache is introduced into a system and the access time reduces from 20 ms to 3 ms. What is the access time of the disk cache, given that the hit ratio is 70 per cent?

3.16 Work through all combinations of actions that can occur in the directory method described in Section 3.8, drawing a flow diagram and the values of the bits in the directories.

3.17 Choose a real computer system or processor with both a cache and virtual memory and identify those methods described in Chapters 2 and 3 which have been employed. Describe how the methods have been implemented (block diagram, etc.) and comment on the design choices made.

Overlap and the associated concept, *pipelining*, are methods which can be used to increase the speed of operation of the central processor. They are often applied to the internal design of high speed computers, including advanced microprocessors, as a type of multiprocessing. In this chapter, we will describe how pipelining is applied to instruction processing and include some of the methods of designing pipelines.

4.1 Overlap and pipelining

4.1.1 Technique

Overlap and pipelining really refer to the same technique, in which a task or operation is divided into a number of subtasks that need to be performed in sequence. Each subtask is performed by its own logical unit, rather than by a single unit which performs all the subtasks. The units are connected together in a serial fashion with the output of one connecting to the input of the next, and all the units operate simultaneously. While one unit is performing a subtask on the i th task, the preceding unit in the chain is performing a different subtask on the $(i+1)$ th task, as shown in Figure 4.1.

The mechanism can be compared to a conveyor belt assembly line in a factory, in which products are in various stages of completion. Each product is assembled in stages as it passes along the assembly line. Similarly, in overlap/pipelining, a task is presented to the first unit. Once the first subtask of this task is completed, the results are presented to the second unit and another task can be presented to the first unit. Results from one subtask are passed to the next unit as required and a task is completed when the subtasks have been processed by all the units.

Suppose each unit in the pipeline has the same operating time to complete a subtask and that the first task is completed and a series of tasks is presented. The time to perform one complete task is the same as the time for one unit to perform one subtask of the task, rather than the summation of all the unit times. Ideally, each

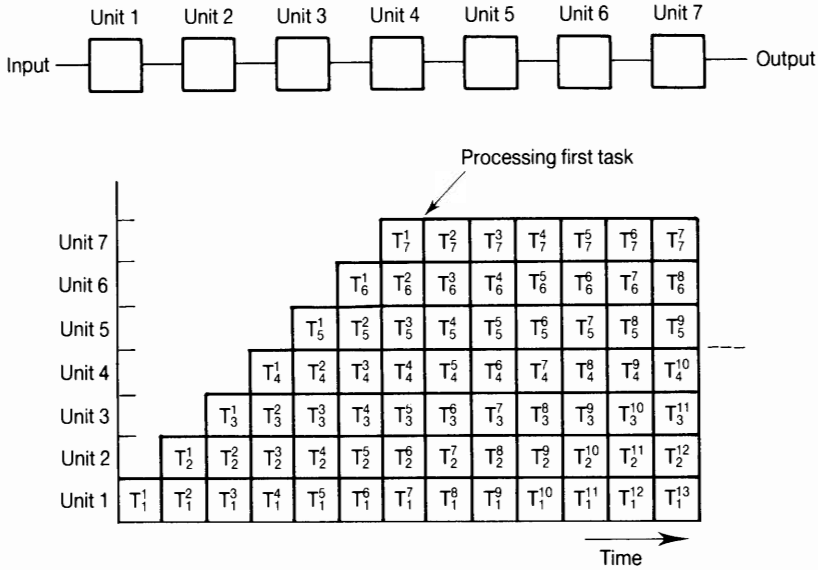


Figure 4.1 Pipeline processing ($T_i^j = j$ th subtask in the i th task)

subtask should take the same time, but if this is not the case, the overall processing time will be that of the slowest unit, with the faster units being delayed. It may be advantageous to equalize stage operating times with the insertion of extra delays. We will pursue this technique later.

The term pipelining is often used to describe a system design for achieving a specific computation by splitting the computation into a series of steps, whereas the term overlap is often used to describe a system design with two or more clearly distinct functions performed simultaneously. For example, a floating point arithmetic operation can be divided into a number of distinct pipelined suboperations, which must be performed in sequence to obtain the final floating point result. Conversely, a computer system might perform central processor functions and input/output functions with separate processors – a central processor and an input/output processor – operating at the same time. The central processor and input/output processor operations are overlapped.

4.1.2 Pipeline data transfer

Two methods of implementing the data transfer in a pipeline can be identified:

1. Asynchronous method.
2. Synchronous method.

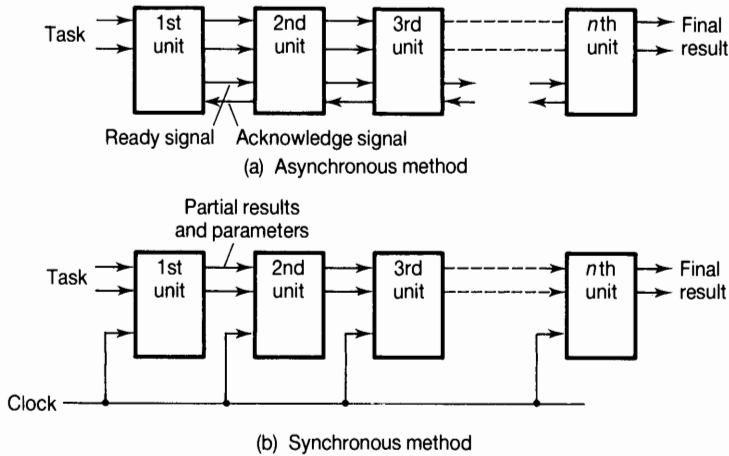


Figure 4.2 Transfer of information between units in a pipeline (a) Asynchronous method (b) Synchronous method

These are shown in Figure 4.2. In the asynchronous method, a pair of “handshaking” signals are used between each unit and the next unit – a ready signal and an acknowledge signal. The ready signal informs the next unit that it has finished the present operation and is ready to pass the task and any results onwards. The acknowledge signal is returned when the receiving unit has accepted the task and results. In the synchronous method, one timing signal causes all outputs of units to be transferred to the succeeding units. The timing signal occurs at fixed intervals, taking into account the slowest unit.

The asynchronous method provides the greatest flexibility in stage operating times and naturally should make the pipeline operate at its fastest, limited as always by the slowest unit. Though unlikely in most pipeline designs, the asynchronous method would allow stages to alter the operating times with different input operands. The asynchronous method also lends itself to the use of variable length first-in first-out buffers between stages, to smooth the flow of results from one stage to the next. However, most constructed instruction and arithmetic pipelines use the synchronous method. An example of a pipeline that might use asynchronous handshaking is in dataflow systems when nodal instructions are only generated when all their operands are received (see Chapter 10). Other examples include the pipeline structures formed with transputers, as described in Chapter 9.

Instruction and arithmetic pipelines almost always use the synchronous method to reduce logic timing and implementation problems. There is a staging latch between each unit and the clock signal activates all the staging latches simultaneously, as shown in Figure 4.3.

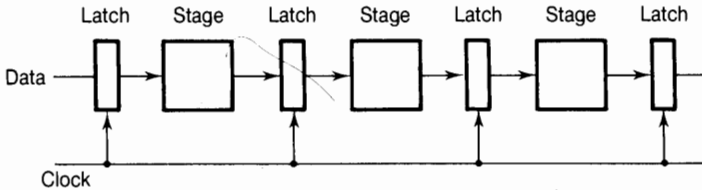


Figure 4.3 Pipeline with staging latches

Pipelines could have been designed without staging latches between pipeline stages and without a synchronizing clock signal – pipeline stages could produce their outputs after natural logic delays, results could percolate through the pipeline from one stage to the next and the final output could be sampled at the same regular frequency as that at which new pipeline inputs are applied. This type of pipeline is called a *maximum-rate pipeline*, as it should result in the maximum speed of operation. Such pipelines are difficult to design because logic delays are not known exactly – the delays vary between devices and depend upon the device interconnections. Testing such pipelines would be a distinct challenge. However, Cray computers do not use staging latches in their pipelines, instead, path delays are equalized.

4.1.3 Performance and cost

Pipelining is present in virtually all computer systems, including microprocessors. It is a form of parallel computation; at any instant more than one task is being performed in parallel (simultaneously). Pipelining is therefore done to increase the speed of operation of the system, although as well as potentially increased speed, it has the advantage of requiring little more logic than a non-pipelined solution in many applications, and sometimes less logic than a high speed non-pipelined solution. An alternative parallel implementation using n replicated units is shown in Figure 4.4. Each unit performs the complete task. The system achieves an equivalent increased speed of operation by applying n tasks simultaneously, one to each of the n units, and producing n results n cycles later. However, complete replication requires much more logic. As circuit densities increase and logic gate costs reduce, complete replication becomes attractive. Replicated parallel systems will be described in later chapters. We can make a general comment that pipelining is much more economical than replication of complete units.

We see from Figure 4.1 that there is a staircase characteristic at the beginning of pipelining; there is also a staircase characteristic at the end of a defined number of tasks. If s tasks are presented to an n -stage pipeline, it takes n clock periods before the first task has been completed, and then another $s - 1$ clock periods before all the tasks have been completed. Hence, the number of clock periods necessary is given by $n + (s - 1)$. Suppose a single, homogeneous non-pipelined unit with equivalent function can perform s tasks in sn clock periods. Then the speed-up

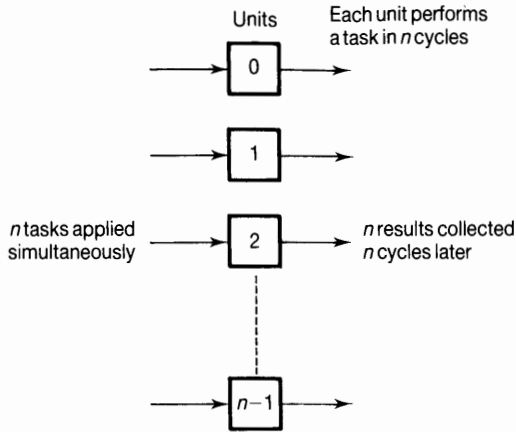


Figure 4.4 Replicated units

available in a pipeline can be given by:

$$\text{Speed-up} = \frac{T_1}{T_n} = \frac{sn}{n + (s - 1)}$$

The potential maximum speed-up is n , though this would only be achieved for an infinite stream of tasks and no hold-ups in the pipeline. The assumption that a single homogeneous unit would take as long as the pipelined system to process one task is also not true. Sometimes, a homogeneous system could be designed to operate faster than the pipelined version.

There is a certain amount of inefficiency in that only in the steady state of a continuous submission of tasks are all the units operating. Some units are not busy during start-up and ending periods. We can describe the efficiency as:

$$\begin{aligned} \text{Efficiency} &= \frac{\sum_{i=1}^n t_i}{n \times (\text{overall operating time})} \\ &= \frac{s}{n + (s - 1)} \\ &= \frac{\text{Speed-up}}{n} \end{aligned}$$

where t_i is time unit i operates. Speed-up and efficiency can be used to characterize pipelines.

Pipelining can be applied to various subunits in a traditional uniprocessor computer and to the overall operation. First, we will consider pipelining applied to overall instruction processing, and then we shall consider how the arithmetic operations within the execution phase of an instruction can be pipelined. Staging latches are assumed to be present in the following.

4.2 Instruction overlap and pipelines

4.2.1 Instruction fetch/execute overlap

The fetch and execute cycles of a processor are often overlapped. Instruction processing requires each instruction to be fetched from memory, decoded, and then executed, in this sequence. In the first instance, we shall assume one fetch cycle fetching a complete instruction and requiring one execute cycle, and no further decomposition.

This technique requires two separate units, a fetch unit, and an execute unit, which are connected together as shown in Figure 4.5(a). Both units have access to the main memory, the fetch unit to access instructions and the execute unit to fetch operands and to store the result if either or both of these actions are necessary. Processor registers, including the program counter, are accessible by the units if necessary. Some dedicated processor registers might be contained within either unit, depending upon the design.

The fetch unit proceeds to fetch the first instruction. Once this operation has been completed, the instruction is passed to the execute unit which decodes the instruction and proceeds to execute it. While this is taking place, the fetch unit fetches the next instruction. The process is continued with the fetch unit fetching the i th instruction while the execute unit is executing the $(i-1)$ th instruction, as shown in Figure 4.5(b).

The execute time is often variable and depends upon the instruction. With fixed length instructions, the fetch time would generally be a constant. With variable length multibyte/word instructions, the fetch time would be variable if the complete instruction needed to be formed before the instruction was executed. Figure 4.5(c) shows a variable instruction fetch and execution times. In this figure, the i th fetch and the $(i-1)$ th execute operations always begin operating together, irrespective of the longer operating time of the previous execute and fetch operations. The overall processing time is given by:

$$\text{Processing time} = \sum_{i=1}^{n+1} \text{Max}(T(F_i), T(E_{i-1}))$$

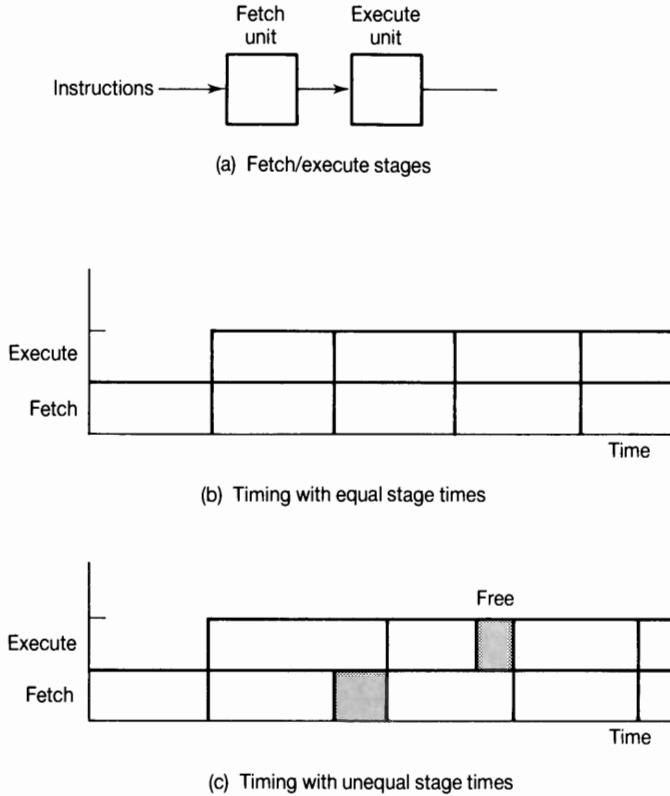
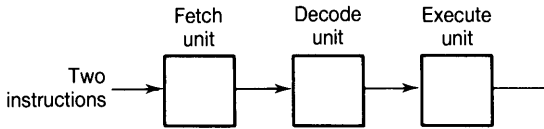


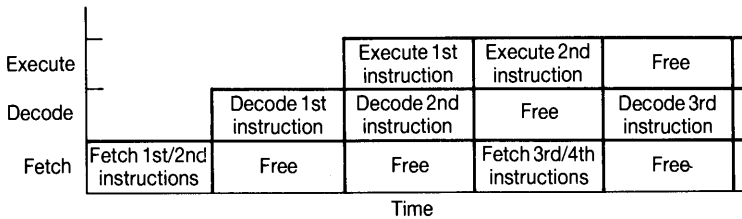
Figure 4.5 Fetch/execute overlap (a) Fetch/execute stages (b) Timing with equal stage times (c) Timing with unequal stage times

where $T(F_i)$ = time of i th fetch operation and $T(E_i)$ = time of i th execute operation.

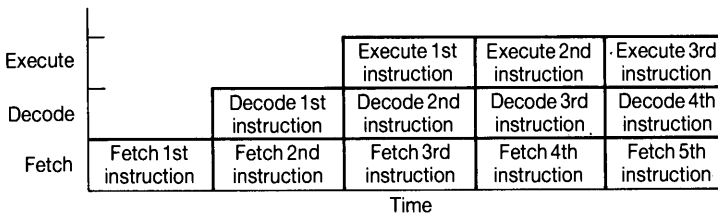
Clearly the execute unit may operate at a different time to the fetch unit. In particular, it is likely to require more time for complicated instructions, and will dominate the processing time. To reduce this effect, the execute unit could be split into further separate units. A separate instruction decode unit could be provided after the fetch unit, followed by an execute unit, as shown in Figure 4.6(a). This scheme is known as *three-level overlap*. The decode unit is responsible for identifying the instruction, including fetching any values from memory in order to compute the effective operand address. However, it is not usually possible for the fetch unit to fetch an instruction and a decode unit to fetch any operands required for the previously fetched instruction at the same time, if the program and data are held in the same memory, as only one unit can access the memory at any instant.



(a) Fetch/decode/execute stages



(b) Fetching two instructions simultaneously



(c) Ideal overlap with interleaved memory

Figure 4.6 Fetch/decode/execute overlap (a) Fetch/decode/execute stages (b) Fetching two instructions simultaneously (c) Ideal overlap with interleaved memory

One method of overcoming this problem is to fetch more than one instruction at a time using multiple memory modules or using true memory interleaving (Section 1.3.4, page 20). In Figure 4.6(b), the fetch unit fetches two instructions simultaneously and then becomes free while the decode unit can access the memory. However, none of the units is operating all of the time, and only two instructions are processed in every three cycles. Figure 4.6(c) shows the ideal utilization using two-way interleaved memory. The usage might be different for particular instructions. The fetch unit fetches an instruction and the decode unit fetches an operand from memory if necessary in the same cycle. Instructions are processed at the maximum rate of one per cycle. Clearly, memory contention will arise if both the fetch unit and decode unit request the same memory module. Contention can be reduced with a greater degree of interleaving. In one scheme, the fetch unit fetches

two instructions simultaneously and becomes free on every alternate cycle, but still allows the system to process one instruction per cycle.

Further instruction processing decomposition can be made. For example, we could have five stages:

1. Fetch instruction.
2. Decode instruction.
3. Fetch operand(s).
4. Execute operation (ADD, etc.).
5. Store result.

This is shown in Figure 4.7. As we divide the processing into more stages, we hope to reduce and equalize the stage processing times. Of the five stages given, stage 1 always requires access to the memory. Stages 3 and 5 require access to the memory if the operands and results (respectively) are held in memory. However, the instructions of many computer systems, particularly microprocessor systems (the 68000 being one exception), do not allow direct memory to memory operations, and provide only memory to processor register, register to register and register to memory operations, which forces the use of processor registers as temporary holding locations. In such situations, stages 3 and 5 do not occur in the same instruction, and only one, at most, needs access to memory.

Unfortunately, at any given time during the processing, stage 3 will be processing instruction n and stage 5 will be processing instruction $n-2$ and both might require

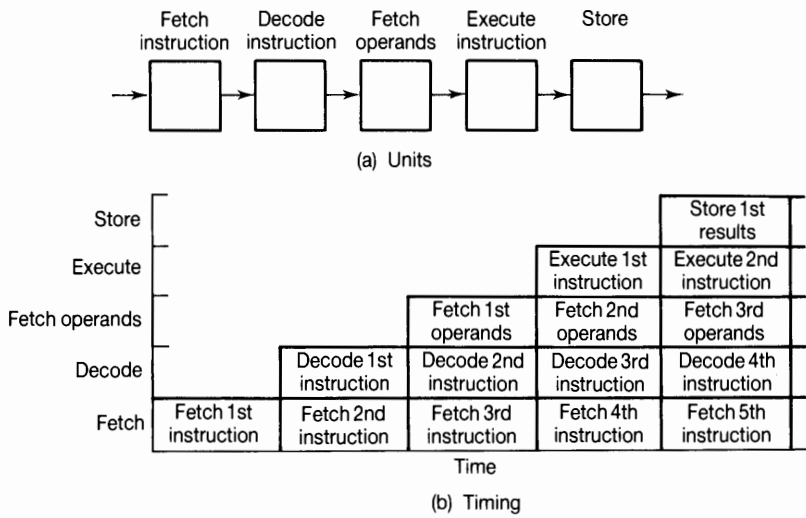


Figure 4.7 Five-stage instruction pipeline (a) Units (b) Timing

access to memory. When it is not possible to guarantee only one stage requesting use of individual memory modules, or any other shared resource, additional logic must be incorporated to arbitrate between requests.

In fact, there are several different hardware and software conditions that might lead to hesitation in the instruction pipeline. Overlap and pipelining assume that there is a sequence of tasks to be performed in one order, with no interaction between tasks other than passing the results of one unit on to the next unit. However, although programs are written as a linear sequence, the execution of one instruction will often depend upon the results of a previous instruction, and the order of execution may be changed by branch instructions.

We can identify three major causes for breakdown or hesitation of an instruction pipeline:

1. Branch instructions.
2. Data dependencies between instructions.
3. Conflict in hardware resources (memory, etc.).

We will consider these factors separately in the following sections. The term “branch” instructions will be used to include “jump” instructions.

4.2.2 Branch instructions

Given no other mechanism, each branch instruction (and the other instructions that follow) could be processed normally in an instruction pipeline. When the branch instruction is completely executed, or at least when the condition can be tested, it would be known which instruction to process next. If this instruction is not the next instruction in the pipeline, all instructions in the pipeline are abandoned and the pipeline cleared. The required instruction is fetched and must be processed through all units in the same way as when the pipeline is first started, and we obtain a space-time diagram such as that shown in Figure 4.8.

Typically, 10–20 per cent of instructions in a program are branch instructions and these instructions could reduce the speed of operation significantly. For example, if a five-stage pipeline operated at 100 ns steps, and an instruction which subsequently cleared the pipeline at the end of its execution occurred every ten instructions, the average instruction processing of the ten instructions would be:

$$\frac{9 \times 100 \text{ ns} + 1 \times 500 \text{ ns}}{10} = 140 \text{ ns}$$

The longer the pipeline, the greater the loss in speed due to conditional branch instructions. Very few instruction pipelines have more than twenty stages. We have ignored the step-up time of the pipeline, that is, the time to fill the pipeline initially when the system starts executing instructions.

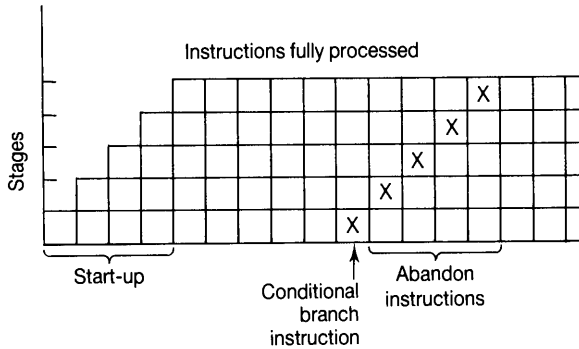


Figure 4.8 Effect of conditional branch instruction in a pipeline

Unconditional branch instructions always cause a change in the sequence and the change is predictable and fixed, but can also affect the pipeline. The fetch unit responsible for fetching instructions takes the value held in the program counter as the address of the next instruction and the program counter is then incremented. Therefore for normal sequential processing, the address of the next instruction is available for the fetch unit immediately the program counter has been incremented, and the fetch unit can keep fetching instructions irrespective of the execution of the instructions. The address of the next instruction for unconditional branch instructions is held in the instruction, or in a memory or register location, or is computed from the contents of addressed locations. If the address is held in the instruction, it would be available after the decode operation, otherwise it would be available after the operand fetch operation if the more complex effective address computations are done then. In any event, the fetch unit does not have the information immediately available and, given no other mechanism, would fetch the next instruction in sequence.

The fetch and decode units could be combined. Then, the fetch/decode unit might have obtained the next address during decoding. In a two-stage pipeline having an instruction fetch/decode unit and an instruction execution unit, the address of the next instruction after an unconditional instruction would be available after the fetch/decode unit has acted upon the unconditional branch instruction. It is often assumed that unconditional branch instructions do not cause a serious problem in pipelines. This is not justified with complex effective addresses computed in stages.

Conditional branch instructions do not always cause a change in the sequence, or even necessarily cause a change in the majority of instances, but this is dependent upon the use of the branch instruction. Conditional branch instruction might typically cause a change 40–60 per cent of the time, on average over a wide range of applications, though in some circumstances the percentage could be much greater or much less. Conditional branch instructions are often used in programs for:

1. Creating repetitive loops of instructions, terminating the loop when a specific condition occurs (loop counter = 0 or arithmetic computational result occurs).
2. To exit a loop if an error condition or other exceptional condition occurs.

The branch usually occurs in 1, but in 2 it does not usually occur. The same instruction might be used for both applications, say branch if positive. Alternatively, different instructions or different conditions might be used, say branch if not zero for a loop, and branch if zero for an error condition, dependent upon the program. The use is not generally known by the system. The programmer could be guided in the choice, given a particular pipeline design which makes a fixed selection after a conditional branch. As with an unconditional branch instruction, even a fixed selection is not possible in hardware if the effective address has not yet been computed.

Strategies exist to reduce the number of times the pipeline breaks down due to conditional branch instructions, using additional hardware, including:

1. Instruction buffers to fetch both possible instructions.
2. Prediction logic to fetch the most likely next instruction after a branch instruction.
3. Delayed branch instructions.

Instruction buffers

A first-in first-out instruction buffer is often used to hold instructions fetched from the memory before passing them to the next stage of instruction pipeline. The buffer becomes part of the pipeline as additional delay stages, and extends the length of the pipeline. The advantage of a first-in first-out buffer is that it smoothes the flow of instructions into the instruction pipeline, especially when the memory is also accessed by the operand fetch unit. It also enables multiple word instructions to be formed. However, increasing the pipeline with the addition of buffers increases the amount of information in the pipeline that must be discarded if the incorrect instructions are fetched after a branch instruction. Most 16-/32-bit microprocessors have a pipelined structure with buffers between stages. For example, the Intel 80286 and 80386 have a prefetch queue in the instruction fetch unit for instruction words fetched from memory, and a decoded instruction queue in the subsequent decode unit leading to the execute unit.

Figure 4.9 shows two separate first-in first-out instruction buffers to fetch both possible instruction sequences after a conditional branch instruction. It is assumed that both addresses are available immediately after fetching the branch instruction. Conditional branch instructions cause both buffers to fill with instructions, assumed from an interleaved memory. When the next address has been discovered, instructions are taken from the appropriate buffer and the contents of the other buffer are discarded. The scheme is sometimes called *multiple prefetching* or *branch bypassing*.

A major disadvantage of multiple prefetching is the problem encountered when

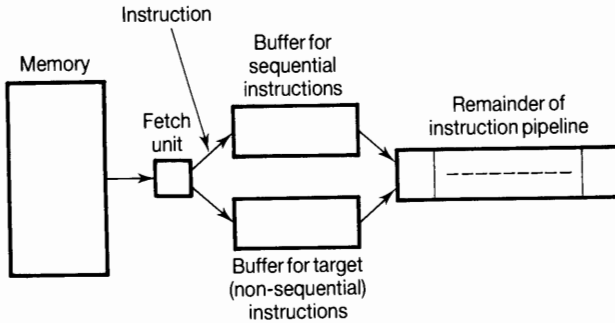


Figure 4.9 Instruction buffers

more than one conditional branch instruction appears in the instruction stream. With two sequential conditional branch instructions, there are four alternative paths, with three instructions, eight alternative paths and, in general, there are 2^n alternative paths when there are n conditional branch instructions. The number of possible conditional branch instructions to be considered will be given by the number of stages in the pipeline. Of course it is unreasonable to provide a buffer for all alternative paths except for small n .

A technique known as *branch folding* (Lilja, 1988) can be used with a two-stage instruction pipeline having an instruction fetch/decode unit (an I unit) and an instruction execute unit (an E unit). An instruction cache-type buffer is inserted between the I and the E units. Instructions are fetched by the I unit, recognized, and the decoded instructions placed in the instruction buffer, together with the address of the next instruction in an associated field for non-branch instructions. If an unconditional branch instruction is decoded, the next address field of the previous (non-branch) instruction fetched is altered to correspond to the new target location, i.e. the unconditional branch instruction folds into the previous instruction. Conditional branch instructions have two next address fields in the buffer, one for each of the next addresses. The execution unit selects one of the next address paths and the other address is carried through the pipeline with the instruction until the instruction has been fully executed and the branch can be resolved. At that time, either the fetched path is used and the next address carried with the instruction is discarded, or the path of the next address carried with the instruction is used and the pipeline is cleared.

Prediction logic and branch history

There are various methods of predicting the next address, mainly based upon expected repetitive usage of the branch instruction, though some methods are based upon expected non-repetitive usage.

To make a prediction based upon repetitive historical usage, an initial prediction

is made when the branch instruction is first encountered. When the true branch instruction target address has been discovered, it is stored in a high speed memory look-up table, and used if the same instruction at the same address is encountered again. Subsequently, the stored target address will always be the address used on the last occasion. A stored bit might be included with each table entry to indicate that a previous prediction has been made.

There are variations in the prediction strategy; for example, rather than update the predicted address when it was found to be wrong, allow it to remain until the next occasion and change it then if it is still found to be wrong. This algorithm requires an additional bit stored with each entry to indicate that the previous prediction was correct, but might produce better results.

One form of prediction table is a *branch history table*, which is implemented in a similar fashion to a cache. A direct mapping scheme could be used, in which target addresses are stored in locations whose addresses are the same as the least significant bits of the addresses of the instructions, together with most significant bit address bits. We note that, as in the directly mapped data/instruction cache, all branch instructions stored in the cache must have addresses with different least significant bits. Alternatively, a fully associative or a set-associative cache-type table could be employed, as shown in Figure 4.10, when a replacement algorithm, as used in caches, is required. In any event, only a limited number of instruction addresses can be stored.

The branch history table can be designed to be accessed after the decode operation, rather than immediately the instruction is fetched. Then the target address will often be known and hence it is only necessary to store a bit to indicate whether the target address should be taken, rather than the full target address, and the table

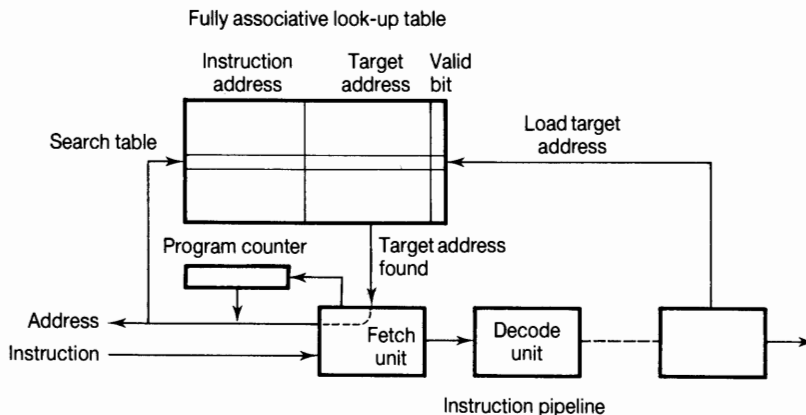


Figure 4.10 Instruction pipeline with branch history table (prediction logic not shown – sequential instructions taken until correct target address loaded)

requires fewer bits. This type of table is called a *decode history table* (Stone, 1987), but it has the disadvantage that the next instruction will have been fetched before the table has been interrogated and so this instruction may have to be discarded.

Delayed branch instructions

In the delayed branch instruction scheme, branch instructions operate such that the sequence of execution is not altered immediately after the branch instruction is executed (if at all) but after one or more subsequent non-branch instructions, depending upon the design. The subsequent instructions are executed irrespective of the branch outcome. For example, in a two-stage pipeline, a branch instruction might be designed to have an effect after the next instruction, so that this instruction need not be discarded in the pipeline. For an n -stage pipeline, the branch instruction could be designed to have an effect after $n - 1$ subsequent instructions, as shown in Figure 4.11. Clearly, the instructions after the branch do not affect the branch outcome, and must be such that the computation is still correct by placing the instructions after the branch instruction. It becomes more difficult for the programmer or compiler to find an increasing number of suitable independent instructions to place after a branch instruction. Typically, one instruction can be rearranged to be

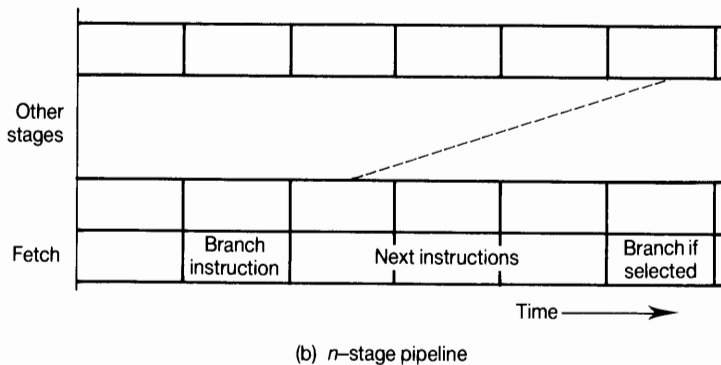
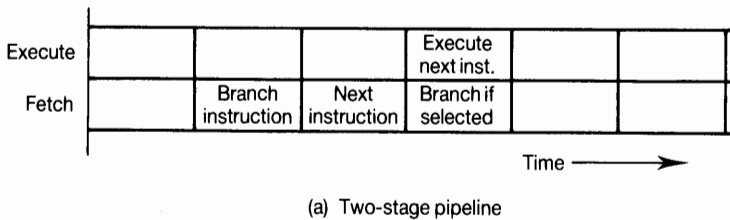


Figure 4.11 Delayed branch technique (a) Two-stage pipeline (b) n -stage pipeline

after the branch instruction in about 70 per cent of occasions, but additional instructions are harder to find.

A one-instruction delayed branch technique has been used extensively in micro-programmed systems at the microinstruction level because microinstructions can often be executed in one cycle and hence can use a two-stage microinstruction fetch/execute pipeline. The one-stage delayed branch instruction has also found application in RISCs (reduced instruction set computers) which have simple instructions executable in one cycle (see Chapter 5).

A number of refinements have been suggested and implemented to improve the performance of delayed branch instructions. For example, in the *nullification* method for loops, the instruction following a conditional branch instruction at the end of the loop is made to be the instruction at the top of the loop. When the loop is terminated, this instruction is converted into a *no-op*, an instruction with no operation and achieving nothing except an instruction delay.

4.2.3 Data dependencies

Suppose we wish to compute the value of $C = 2 \times (A + \text{contents of memory location } 100)$ with the program sequence given in 8086 code as:

```
ADD  AX,[100] ;Add memory location 100 contents
                ;to AX register
SAL  AX,1     ;Shift AX one place left
MOV  CX,AX    ;Copy AX into CX register
```

and these instructions are in a five-stage pipeline, as shown in Figure 4.12. (The 8086 does not have the pipeline shown.) It would be incorrect to begin shifting AX before the add instruction and, similarly, it would be incorrect to load CX before the shift operation. Hence, in this program each instruction must produce its result before the next instruction can begin.

Should the programmer know that a pipeline organization exists in the computer used, and also the operation of this pipeline, it may be possible to rewrite some programs to separate data dependencies. Otherwise, when a data dependency does occur, there are two possible strategies:

1. Detect the data dependencies at the input of the pipeline and then hold up the pipeline completely until the dependencies have been resolved (by instructions already in the pipeline being fully executed).
2. Allow all instructions to be fetched into the pipeline but only allow independent instructions to proceed to their completion, and delay instructions which are dependent upon other, not yet executed, instructions until these instructions are executed.

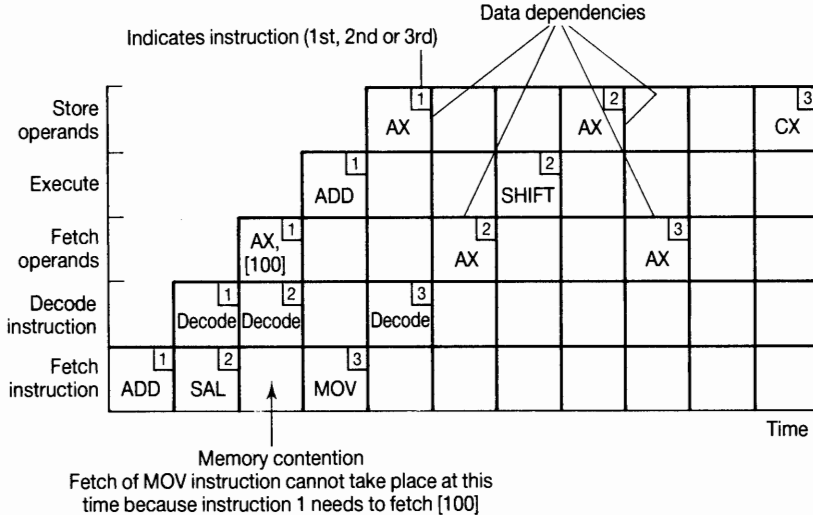


Figure 4.12 Five-stage pipeline with data dependencies and memory contention

Data dependencies can be detected by considering read and write operations on specific locations accessible by the instructions (including all operations such as arithmetic operations which alter the contents of the locations).

In terms of two operations – read and write – operating upon a single location, a *write-after-write* hazard exists if there are two write operations upon a location such that the second write operation in the pipeline completes before the first. Hence the written value will be altered by the first write operation when it completes. A *read-after-write* hazard exists if a read operation occurs before a previous write operation has been completed, and hence the read operation would obtain an incorrect value (a value not yet updated). A *write-after-read* hazard exists when a write operation occurs before a previous read operation has had time to complete, and again the read operation would obtain an incorrect value (a prematurely updated value). *Read-after-read* hazards, in which read operations occur out of order, do not normally cause incorrect results. Figure 4.13 illustrates some of these hazards in terms of an instruction pipeline in which read and write operations are done at various stages.

An instruction can usually only alter one location, but might read two locations. For a two-address instruction format, one of the locations read will be the same as the location altered. Condition code flags must also be included in the hazard detection mechanism. The number of hazard conditions to be checked becomes quite large for a long pipeline having many partially complete instructions.

We can identify a potential hazard between instruction *i* and instruction *j* when one of the following conditions occurs:

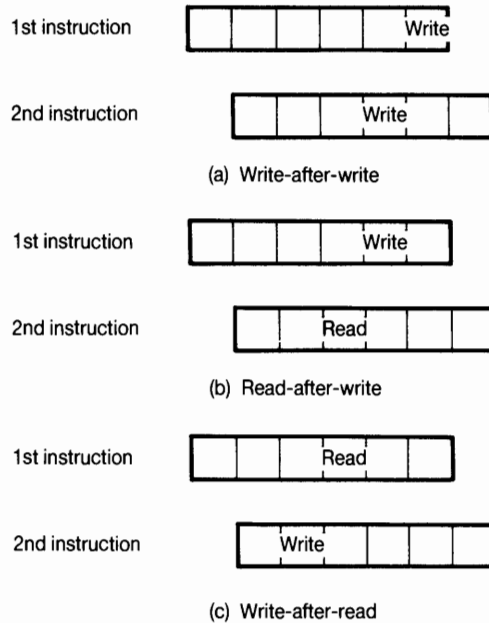


Figure 4.13 Read/write hazards (a) Write-after-write (b) Read-after-write
(c) Write-after-read

For write-after-write	$W(i) \cap W(j) \neq 0$
For read-after-write	$W(i) \cap R(j) \neq 0$
For write-after-read	$R(i) \cap W(j) \neq 0$

$W(i)$ indicates the set of locations altered by instruction i ; $R(i)$ indicates the set of locations read by instruction i , and 0 indicates an empty set. For no hazard, neither of the sets on the left hand side of each condition includes any of the same elements. Clearly these conditions can cover all possible read/write arrangements in the pipeline. It would be better to limit the detection only to the situations that are possible.

Detecting the hazard at the beginning of the pipeline and stopping the pipeline completely until the hazard has passed is obviously much simpler than only stopping the specific instruction creating the hazard from entering the pipeline, because a satisfactory sequence of operations must be maintained to obtain the desired result (though not necessarily the same order as in the program). Hazard detection must also include any instructions held up at the entrance to the pipeline.

A relatively simple method of maintaining a proper sequence of read/write operations is to associate a 1-bit tag with each operand register. This tag indicates

whether a valid result exists in the register, say 0 for not valid and 1 for valid. A fetched instruction which will write to the register examines the tag and if the tag is 1, it sets the tag to 0 to show that the value will be changed. When the instruction has produced the value, it loads the register and sets the tag bit to 1, letting other instructions have access to the register. Any instruction fetched before the operand tags have been set has to wait. A form of this *scoreboard* technique is used on the Motorola MC88100 RISC microprocessor (Motorola, 1988a). The MC88100 also has delayed branch instructions.

Figure 4.14 shows the mechanism in a five-stage pipeline having registers read only in stage 3 and altered only in stage 5. In this case, it is sufficient to reset the valid bit of the register to be altered during stage 3 of a write instruction in preparation for setting it in stage 5. Figure 4.14 shows a write instruction followed by two read instructions. Both read instructions must examine the valid bit of their source registers prior to reading the contents of the registers, and will hesitate if they cannot proceed.

Notice that the five-stage pipeline described only has read-after-write hazards; write-after-read and write-after-write hazards do not occur if instruction sequencing is maintained, i.e. if instructions are executed in the order in the program, and if the pipeline is “stalled” by hazards, as in Figure 4.12. A somewhat more complicated scoreboard technique was used in the CDC 6600. The CDC 6600 scoreboard kept a

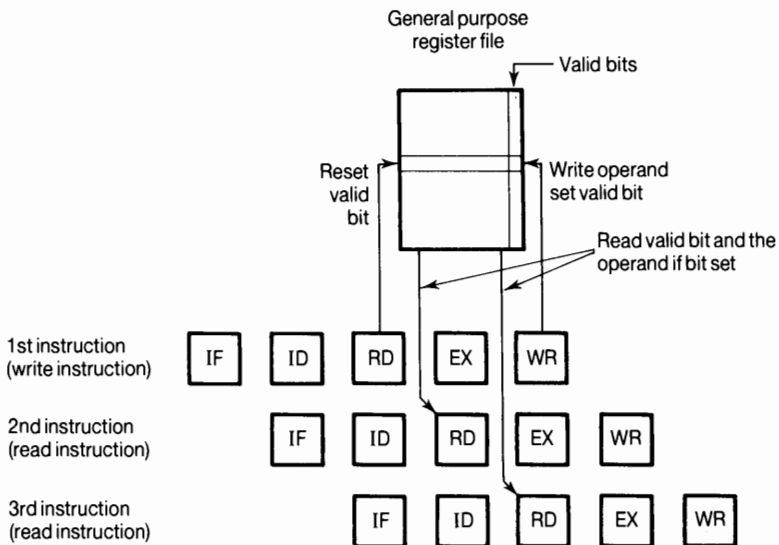


Figure 4.14 Register read/write hazard detection using valid bits (IF, instruction fetch; ID, instruction decode; RD, read operand; EX, execute phase; WR write operand)

record of the availability of operands and functional units for instructions as they were being processed to allow instructions to proceed as soon as possible and out of sequence if necessary. The interested reader should consult Thornton (1970).

4.2.4 Internal forwarding

The term *forwarding* refers to the technique of passing the result of one instruction to another instruction via a processor register without storing the result in a memory location. Forwarding would generally increase the speed of operation, as access to processor operand registers is normally faster than access to memory locations. Three types of forwarding can be identified:

1. Store-fetch forwarding.
2. Fetch-fetch forwarding.
3. Store-store overwriting.

Store and fetch refer to writing operands into memory and reading operands from memory respectively. In each case, unnecessary memory references can be eliminated.

In store-fetch forwarding, fetching an operand which has been stored and hence is also held in a processor operand register is eliminated by taking the operand directly from the processor operand register. For example, the code:

```
MOV  [200],AX  ;Copy contents of AX register
                ;into memory location 200
ADD  BX,[200]  ;Add memory contents 200 to register BX
```

could be reduced to:

```
MOV  [200],AX
ADD  BX,AX
```

which eliminates one memory reference (in the final ADD instruction).

In fetch-fetch forwarding, multiple accesses to the same memory location are eliminated by making all accesses to the operand in a processor operand register once it has been read into the register. For example:

```
MOV  AX,[200]
MOV  BX,[200]
```

could be reduced to:

```
MOV  AX,[200]
MOV  BX,AX
```

122 Computer design techniques

In store-store overwriting, one or more write operations without intermediate operations on the stored information can be eliminated. For example:

```
MOV  [200],AX
MOV  [200],BX
```

could be reduced to:

```
MOV  [200],BX
```

though the last simplification is unlikely in most programs. Rearrangements could be done directly by the programmer when necessary, or done automatically by the system hardware after it detects the forwarding option, using internal forwarding.

Internal forwarding is hardware forwarding implemented by processor registers not visible to the programmer. The most commonly quoted example of this type of internal forwarding is in the IBM 360 Model 91, as reported by Tomasulo (1967). The IBM 360 Model 91 is now only of historical interest and was rapidly superseded by other models with caches (the Model 85 and the Model 195). In internal forwarding, the results generated by an arithmetic unit are passed directly to the input of an arithmetic unit, by matching the destination address carried with the result with the addresses of the units available. Operand pairs are held in buffers at the input of the units. Operations are only executed when a unit receives a full complement of operands, and then new results, which may become new source operands, are generated. It may be that instructions are not executed in the sequence in which they are held in memory, though the final result will be the same. The IBM 360 Model 91 internal forwarding mechanism is similar to dataflow computing described in Chapter 10 and predates the implementation of the latter. A cache could be regarded as a forwarding scheme which short-circuits the main memory. The complicated forwarding scheme of the Model 91 may not be justified if a cache is present. RISCs often use relatively simple internal forwarding (see Chapter 5).

4.2.5 Multistreaming

We have assumed that the instructions being processed are from one program and that they depend upon the immediately preceding instructions. However, many large computer systems operate in a multiuser environment, switching from one user to another at intervals. Such activities often have a deleterious effect on cache-based systems, as instructions/data for a new program need to be brought into the cache to replace the instructions/data of a previous program. Eventually, the instructions/data of a replaced program will need to be reinstated in the cache.

In contrast, this process could be used to advantage in a pipeline, by interleaving instructions of different programs in the pipeline and by executing one instruction from each program in sequence. For example, if there are ten programs to be

executed, every tenth instruction would be from the same program. In a ten-stage pipeline, each instruction would be completely independent of the other instructions in the pipeline and no hazard detection for conditional jump instructions or data dependencies would be necessary. The instructions of the ten programs would execute at the maximum pipeline rate of one instruction per cycle. This technique necessitates a complete set of processor registers for each program, i.e. for ten programs, ten sets of operand registers, ten program counters, ten memory buffers, if used, and tags are also needed in the instruction to identify the program. In the past, such duplication of registers might have been difficult to justify, but now it may be a reasonable choice, given that the maximum rate is obtained under the special conditions of several time-shared programs and no complicated hazard detection logic is necessary. The scheme may be difficult to expand to more time-shared programs than the number of stages in the pipeline.

4.3 Arithmetic processing pipelines

4.3.1 General

In the previous sections we considered the arithmetic units as single entities. In fact, arithmetic operations of the execute phase could be decomposed further into several separate operations. Floating point arithmetic, in particular, can be naturally decomposed into several sequential operations. It is also possible to pipeline fixed point operations to advantage, especially if several operations are expected in sequence. We will briefly consider how arithmetic operations might be pipelined in the following sections.

Note that an arithmetic pipeline designed to perform a particular arithmetic operation, say floating point addition, could only be supplied with continuous tasks in an instruction pipeline if a series of floating point instructions were to be executed. Such situations arise in the processing of the elements of vectors, and hence pipelined arithmetic units find particular application in computers which can operate upon vectors and which have machine instructions specifying vector operations. Such computers are called *vector computers*, and the processors within them are *vector processors*. For general purpose (scalar) processors only capable of operating upon individual data elements, pipelined arithmetic units may not be kept fully occupied. Pipelined arithmetic units in scalar processors should be used for the following reasons:

1. Increased performance should a series of similar computations be encountered.
2. Reduced logic compared to non-pipelined designs in some cases.
3. Multifunction units might be possible.

Multifunction arithmetic pipelines can be designed with internal paths that can be

reconfigured statically to produce different overall arithmetic functions, or can be reconfigured dynamically to produce different arithmetic functions on successive input operands. In a dynamic pipeline, different functions are associated with sets of operands as they are applied to the entrance of the pipeline. The pipeline does not need to be cleared of existing partial results when a different function is selected and the execution of previous functions can continue unaffected. Multifunction pipelines have not been used much in practice because of the complicated logic required, but they should increase the performance of a single pipeline in scalar processors. Multifunction pipelines do not seem to have an advantage in vector computers, as these computers often perform the same operation on a series of elements fed into the pipeline.

4.3.2 Fixed-point arithmetic pipelines

The conventional method of adding two integers (fixed point numbers) is to use a parallel adder consisting of cascaded full adder circuits. Suppose the two numbers to be added together have digits $A_{n-1} \dots A_0$ and $B_{n-1} \dots B_0$. There are n full adders in the parallel adder. Each full adder adds two binary digits, A_i and B_i , together with a “carry-in” from the previous addition, C_{i-1} , to produce a sum digit, S_i , and a “carry-out” digit, C_i , as shown in Figure 4.15(a). A pipelined version of the parallel adder is shown in Figure 4.15(b). Here, the n full adders have been separated into different pipeline stages.

A multifunction version of the parallel adder pipeline giving both addition and subtraction can be achieved easily. Subtraction, $A - B$, can be performed in a parallel adder by complementing the B digits and setting the carry-in digit to the first stage to 1 (rather than to 0 for addition). Hence, one of each pair of digits passed on to the adjacent stage needs to be complemented and this operation can be incorporated into the pipeline stage. The adder/subtractor pipeline could be static. In this case, the complementing operation occurs on the appropriate bits of each pair of operands applied to the pipeline as they pass through the pipeline. Alternatively, the adder/subtractor pipeline could be dynamic, and the complementing operation performed upon specific operands. These operands could be identified by attaching a tag to them; the tag is passed from one stage to the next with the operands. Additional functions could be incorporated, for example, single operand increment and decrement. Multiplication and division might be better performed in a separate unit, though it is possible to design a multifunction pipeline incorporating all of the basic arithmetic operations.

The previous addition pipeline is based upon a parallel adder in which the carry signal “ripples” from one pipeline stage to another. In a non-pipelined version, the speed of operation is limited by the time it takes for the carry to ripple through all the full adders. (This is also true in the pipelined version, but other additions can be started while the process takes place.) A well-known method of reducing ripple time is to predict the carry signals at each full adder by using *carry-look-ahead* logic

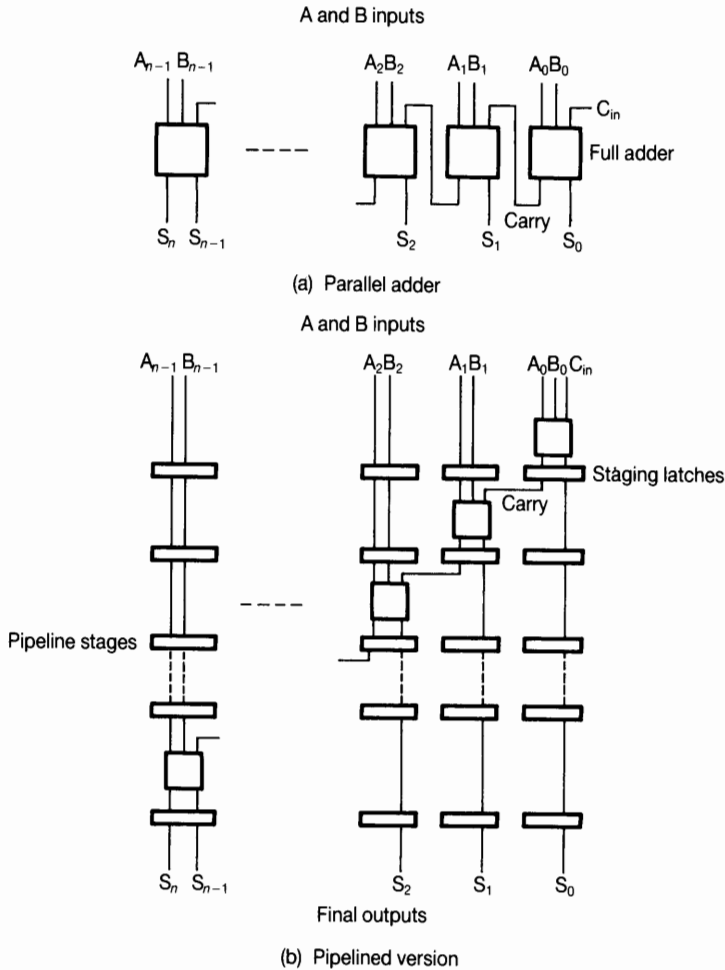


Figure 4.15 Pipelined parallel adder (a) Parallel adder (b) Pipelined version

rather than waiting for each to be generated by adjacent full adders. Such prediction logic can also be pipelined. The full details of carry-look-ahead adders can be found in Baer (1980).

There are also various ways to perform multiplication. Most of these are suitable for arrangement as a pipeline as they involve a series of additions, each of which can be done in a pipeline stage. The conventional method to implement multiplication is a shift-and-add process using a parallel adder to successively add A to an accumulating sum when the appropriate bit of B is 1. Hence, one pipeline solution would be to unfold the iterative process and have n stages, each consisting of a parallel adder.

126 Computer design techniques

One technique applicable to multiplication is the *carry-save* technique. As an example, consider the multiplication of two 6-bit numbers:

$$\begin{array}{r}
 A \quad 110101 \\
 B \quad 101011 \\
 \hline
 110101 \\
 110101 \\
 000000 \\
 110101 \\
 000000 \\
 110101 \\
 \hline
 100011100111 \\
 \hline
 \end{array}$$

The partial products are divided into groups, with three partial products in each group. Therefore we have two groups in this example. The numbers in each group are added simultaneously, using one full adder for each triplet of bits in each group, without carry being passed from one stage to the next. All three inputs of the full adders are used. This process results in two numbers being generated for each group, namely a sum word, and a carry word:

	110101		110101
Group 1	110101	Group 2	000000
	000000		110101
	<hr/>		<hr/>
Sum 1	01011111	Sum 2	11100001
Carry 1	01000000	Carry 2	00101000
	<hr/>		<hr/>

Each carry word is moved one place left to give it the correct significance. The true sum of the three numbers in each case could be obtained by adding together the sum and carry words. The final product is the summation of Sum 1, Carry 1, Sum 2 and Carry 2. Taking three of these numbers, the carry-save process is repeated to produce Sum 3 and Carry 3, i.e.:

$$\begin{array}{r}
 \text{Sum 1} \quad 01011111 \\
 \text{Carry 1} \quad 01000000 \\
 \text{Sum 2} \quad 11100001 \\
 \hline
 \text{Sum 3} \quad 11100010111 \\
 \text{Carry 3} \quad 00010010000 \\
 \hline
 \end{array}$$

The process is repeated taking Sum 3, Carry 3 and Carry 2 to produce Sum 4 and Carry 4:

Sum 3	11100010111
Carry 3	00010010000
Carry 2	00101000
Sum 4	11011000111
Carry 4	01000100000

Finally, Sum 4 and Carry 4 are added together using a parallel adder:

Sum 4	11011000111
Carry 4	01000100000
Final sum	100011100111

Each step can be implemented in one stage of a pipeline, as shown in Figure 4.16. The partial product bits can be generated directly from the logical AND of the corresponding A and B bits. The first partial product has the bits $A_{n-1}B_0 \dots A_1B_0, A_0B_0$. The second partial product has the bits $A_{n-1}B_1 \dots A_1B_1, A_0B_1$, etc.

The multiplier using carry-save adders lends itself to become a *feedback pipeline* to save on components, as shown in Figure 4.17. Here, the carry-save adders are reused one or more times, depending upon the number of bits in the multiplier, and on the organization. Note that the advantage of being able to submit new operands for multiplication on every cycle is now lost.

Another multiplication technique involves having a two-dimensional array of cells. Each cell performs a 3-bit full adder addition. There are several versions of the array multiplier, each of which passes on signals in different ways. The shift-and-add multiplier is in fact a form of an array multiplier when reconfigured for a pipeline. The reader is referred to Jump *et al.* (1978) for a study of array multipliers arranged for pipelining. Array multipliers are suitable for VLSI implementation.

4.3.3 Floating point arithmetic pipelines

Floating point arithmetic is particularly suitable for pipeline implementation as a sequence of steps can be readily identified. It is perhaps the commonly quoted example for pipeline implementation. Even in a non-pipelined computer system, floating point arithmetic would normally be computed as a series of steps (whereas fixed point arithmetic might be computed in one step.)

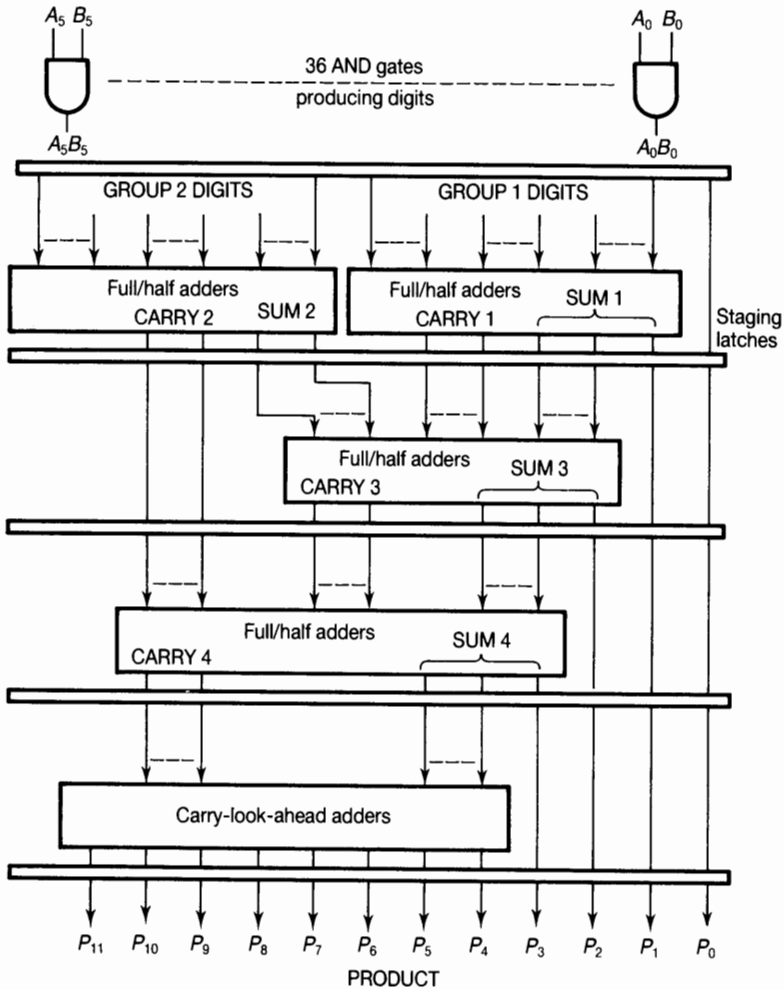


Figure 4.16 6-bit \times 6-bit carry-save multiplier

Each floating point number is represented by a mantissa and exponent, given by:

$$\text{number} = \text{mantissa} \times 2^{\text{exponent}}$$

where the base of the number system is 2. (The base could also be power of 2, for example, it is occasionally 16). The mantissa and exponent are stored as two numbers. The sign of the number is shown by a separate sign bit and the remaining mantissa is a positive number (i.e. the full mantissa is represented in the sign plus magnitude representation). A biased exponent representation is often used for the

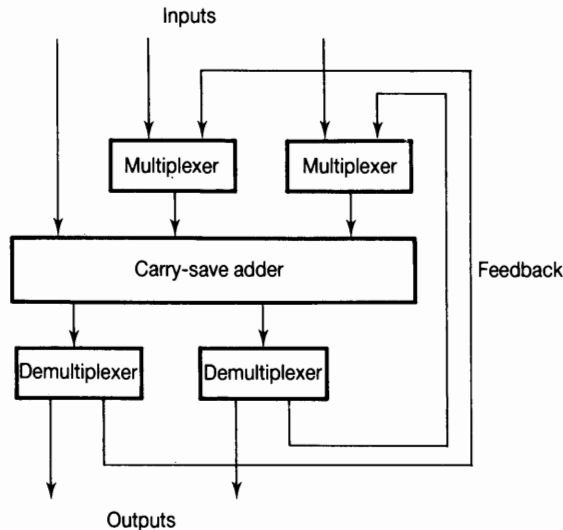


Figure 4.17 Carry-save adder with feedback

exponent such that the stored exponent is always positive, even when representing a negative exponent. In the biased exponent system, the stored exponent = actual exponent + bias. The bias is usually either 2^{n-1} or $2^{n-1} - 1$, where there are n bits in the number. The biased exponent representation does not affect the basic floating point arithmetic algorithms but makes it easier to implement the comparison of exponents which is necessary in floating point addition (not in floating point multiplication).

Numbers are also usually represented in a normalized form in which the most significant digit of the (positive) mantissa is made to be non-zero (i.e. 1, with a base of 2) and the exponent adjusted accordingly, to obtain the greatest possible precision of the number (the greatest number of significant digits in the mantissa). In fact, the most significant bit need not be stored in base two system if it is always 1. The stored mantissa is normally a fraction, i.e. the binary point is to the immediate left of the stored mantissa, and the exponents are integers. The position of the binary point is immaterial to the algorithm.

The addition of two normalized floating point numbers, represented by the mantissa/exponent pairs, m_1e_1 and m_2e_2 , requires a number of sequential steps, for example:

1. Subtract exponents e_1, e_2 , and generate the difference $e_1 - e_2$.
2. Interchange mantissa m_1 and m_2 , if $e_1 - e_2$ is negative and make the difference positive. Otherwise no action is performed in this step.

3. Shift mantissa m_2 by $e_1 - e_2$ places right.
4. Add mantissas to produce result mantissa replacing m_2 .
5. Normalize result as follows. If mantissa greater than 1, shift one place right and add 1 to exponent. If mantissa less than 0.5, shift mantissa left until leftmost digit = 1 and subtract number of shifts from exponent. If mantissa = 0, load special zero pattern into exponent. Otherwise do nothing. Check for underflow (number too small to be represented) or overflow (number too large to be represented) and in such cases generate the appropriate actions.

Some steps might be divided further, and any group of sequential steps in a pipeline can be formed into one step.

Floating point multiplication is conceptually easier, having the steps:

1. Add exponents e_1 and e_2 .
2. Multiply mantissa m_1 and m_2 .
3. Normalize if necessary.
4. Round mantissa to a single length result.
5. Renormalization if necessary (rounding may increase mantissa one digit which necessitates renormalization).

However, the mantissa multiplication operation would typically be divided into two or more stages (perhaps iterative stages with feedback) which would make floating point multiplication a longer process than floating point addition. It is possible to combine floating point multiplication with addition, as the exponent addition of the floating point multiplication and the exponent subtraction of floating point addition could both be performed with a parallel adder. Also, both operations require normalization.

A floating point multiply/divide unit can be designed as a feedback pipeline by internally feeding back partial product results until the final result is obtained. The general motive for designing feedback pipelines is reduction in hardware, compared to a non-feedback pipeline. New inputs cannot be applied to a feedback pipeline (at least not when the feedback is to the input) until previous results have been generated and consumed, and hence this type of pipeline does not necessarily increase throughput, and externally the unit may not be regarded as a pipeline. We will use the term *linear pipeline* to describe a pipeline without feedback paths.

4.4 Logical design of pipelines

4.4.1 Reservation tables

The reservation table is central to pipeline designs. A *reservation table* is a two-dimensional diagram showing pipeline stages and their usage over time, i.e. a

space–time diagram for the pipeline. Time is divided into equal time periods, normally equivalent to the clock periods in a synchronous pipeline. If a pipeline stage is used during a particular time period, an X is placed in the reservation table time slot. The reservation table is used to illustrate the operation of a pipeline and also used in the design of pipeline control algorithms.

A reservation table of a five-stage linear pipeline is shown in Figure 4.18. In this particular case, each of the five stages operate for one time period, and in sequence. It is possible to have stages operate for more than one time period, which would be shown with Xs in adjacent columns of one row. More than one X in one row, not necessarily adjacent columns, could also indicate that a stage is used more than once in a feedback configuration. A reservation table with more than one X in a column would indicate that more than one stage is operating simultaneously on the same or different tasks. Operating on the same task would indicate parallel processing, while operating on different tasks would generally indicate some form of feedback in the pipeline.

A reservation table describes the actions performed by the pipeline during each time period. A single function pipeline has only one set of actions and hence would have one reservation table; a multifunction pipeline would have one reservation table for each function of the pipeline. In a static multifunction pipeline, only one function can be selected for all entering tasks until the whole pipeline is reconfigured for a new function, and only one of the reservation tables is of interest at any instant corresponding to overall function selected. In a dynamic multifunction pipeline, different overall functions can be performed on entering data, and all of the reservation tables of functions selected need to be considered as a set.

Pipelines generally operate in synchronism with a common clock signal and each time slot would be related to this clock period, the boundary between two adjacent slots normally corresponding to clocking the data from one pipeline stage to the next stage. Note though, that the reservation table does not show the specific paths taken by information from one stage to another, and it is possible for two different pipelines to have the same reservation table.

The reservation table does help determine whether a new task can be applied after

		Time				
Clock periods →		0	1	2	3	4
Stages	1	X				
	2		X			
	3			X		
	4				X	
	5					X

Figure 4.18 Reservation table of a five-stage linear pipeline

the last task has been processed by the first stage. Each time the pipeline is called upon to process a new task is an *initiation*. Pipelines may not be able to accept initiations at the start of every period. A *collision* occurs when two or more initiations attempt to use the same stage in the pipeline at the same time.

Consider, for example, the reservation table of a static pipeline shown in Figure 4.19. This table has adjacent Xs in rows. Two consecutive initiations would cause a *collision* at slots 1–2. Here, the stage is still busy with the first initiation when the second reaches the input of the stage. Such collisions need to be avoided by delaying the progress of the second initiation through this particular pipeline until one cycle later. A potential collision can be identified by noting the distance in time slots between Xs in each row of the reservation table. Two adjacent Xs have a “distance” of 1 and indicate that two initiations cannot be applied in successive cycles. A distance of 2 would indicate that two initiations could be separated by an extra cycle.

		Time							
		0	1	2	3	4	5	6	7
Stages	1	X				X	X		
	2		X	X					
	3				X				
	4							X	X

Figure 4.19 Reservation table with collision

A *collision vector* is used to describe the potential collisions and is defined for a given reservation table in the following way:

$$\text{Collision vector } C = C_{n-1}C_{n-2} \dots C_2C_1C_0$$

where there are n stages in the pipeline. $C_i = 1$ if a collision would occur with an initiation i cycles after an initiation (taking into account all existing tasks in the pipeline), otherwise $C_i = 0$. We note that C_0 will always be 1, as two simultaneous initiations would always collide. Hence, sometimes C_0 is omitted from the collision vector. C_n and subsequent bits are always 0, as initiations so separated would never collide. All previous initiations would have passed through the pipeline completely.

The *initial collision vector* is the collision vector after the first initiation is presented to the pipeline. To compute this it is only necessary to consider the distance between all pairs of Xs in each row of the reservation table. The distances between all pairs in the reservation table shown in Figure 4.19 are (5,4,1,0) and the initial collision vector is 110011 (including C_0).

4.4.2 Pipeline scheduling and control

Now let us consider the situations when a pipeline should not accept new initiations on every cycle because a collision would occur sometime during the processing of the task. The pipeline needs a control or scheduling mechanism to determine when new initiations can be accepted without a collision occurring.

Latency is the term used to describe the number of clock periods between two initiations. The *average latency* is the average number of clock periods between initiations generally over a specific repeating cycle of initiations. The *forbidden latency set* contains those latencies which cause collisions, e.g. (5, 4, 1, 0) previously. This set is also represented in the collision vector. The smallest average latency considering all the possible sequences of tasks (initiation cycles) is called the *minimum average latency* (MAL). Depending upon the design criteria, the optimum scheduling strategy might be one which produces the minimum average latency.

The following scheduling strategy is due to Davidson (1971). A pipeline can be considered in a particular state; it changes from one state to another as a result of accepted initiations. A diagram of linked states becomes a *state diagram*. Each state in the state diagram is identified by the collision vector (sometimes called a *status vector* in the state diagram) which indicates whether a new initiation may be made to the pipeline. The initial state vector of an empty pipeline before any initiations have been made is $00 \dots 00$, since no collision can occur with the first initiation. After the first initiation has been taken, the collision vector becomes the initial collision vector and C_1 in the collision vector will define whether another initiation is allowed in the next cycle.

First the collision vector is shifted one place right and 0 is entered into the left side. If $C_0 = 1$, indicating that an initiation is not allowed, the pipeline is now in another state defined by the shifted collision vector. If $C_0 = 0$, indicating that an initiation is allowed, there are two possible new states – one when the initiation is not taken, which is the same as when $C_0 = 1$, and one when the initiation is taken. If the initiation is taken, the initial collision vector is bit-wise logically ORed with the shifted collision vector to produce a new collision vector. This logical ORing of the shifted collision vector with the initial collision vector incorporates into the collision vector the effect of the new initiation and its effect on potential collisions.

Figure 4.20 illustrates the algorithm for computing the collision vector for a pipeline when initiations may or may not be taken. It immediately leads to a possible scheduling algorithm, i.e. after shifting the collision vector, if $C_0 = 0$, an initiation is taken and a new collision vector is computed by logically ORing operations. The strategy of always taking the opportunity of submitting an initiation to the pipeline when it is known that a collision will not occur, i.e. choosing the minimum latency on every suitable occasion, is called a *greedy strategy*. Unfortunately, a greedy strategy will not necessarily result in the minimum average latency (an optimum strategy), though it normally comes fairly close to the optimum strategy, and is particularly easy to implement.

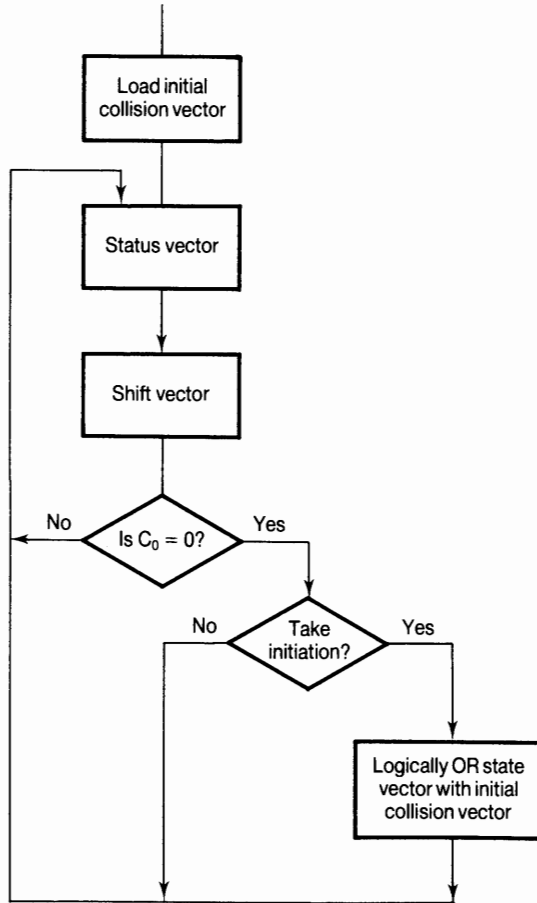


Figure 4.20 Davidson's pipeline control algorithm

The state diagram for the collision vector 110011 (the reservation table in Figure 4.19) is shown in Figure 4.21. All possible states are included, whether or not an initiation is taken. Clearly such state diagrams could become very large.

The state diagram can be reduced to only showing changes in state when an initiation is taken. The various possible cycles of initiations can be easily located from this *modified* (or *reduced*) *state diagram*. The modified state diagram is shown in Figure 4.22. The number beside each arc indicates the number of cycles necessary to reach a state. We can identify possible closed *simple* cycles (cycles in which a state is only visited once during the cycle), as given by 3,3,3,3,..., 2,6,2,6,..., 3,6,3,6,..., and 6,6,6,6,... These simple cycles would be written as (3), (2,6), (3,6), and (6).

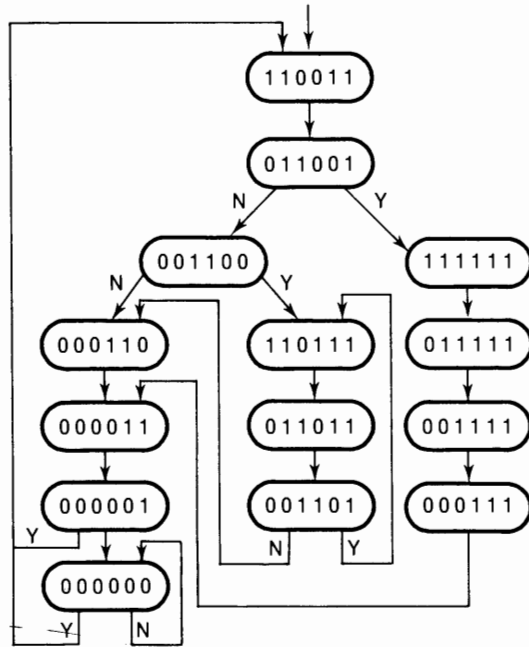


Figure 4.21 State diagram for collision vector 110011

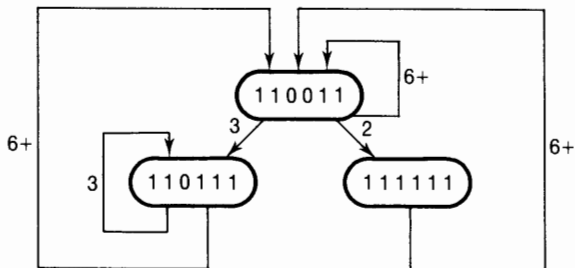


Figure 4.22 Modified state diagram
(6+ = 6 or more cycles to reach state)

There is usually more than one greedy cycle if the starting point for a cycle can be other than the initial state. In Figure 4.22, the greedy cycles are (2,6) starting at the initial state 110011 and (3) starting at 110111. The average latency of any greedy (simple) cycle is less than or equal to the number of 1s in the initial collision vector (see Kogge, 1981). More complex cycles exist, in which states are visited more than once. However it has been shown (see Kogge (1981) for proof) that for any complex cycle with a given average latency, there is at least one simple cycle with an average latency no greater than this latency. In searching for an optimum strategy there is no need to consider complex cycles, as a simple cycle exists with the same or better latency, assuming the criterion is minimum latency.

The minimum average latency is always equal to or greater than the maximum number of Xs in the rows of the reservation table. This condition gives us the lower bound on latency and can be deduced as follows: let the maximum number of Xs in a reservation table row be n_{\max} , which equals the number of times the most used stage is used by one initiation. Given t time slots in the reservation station, the maximum possible number of initiations is limited by the most used stage which, of course, can be used by one initiation at a time. Hence the maximum number of initiations = t/n_{\max} . The minimum latency = $t/(\text{maximum number of initiation}) = n_{\max}$.

We now have the conditions: maximum number of Xs in row \leq minimum average latency (MAL) \leq greedy cycle average latency \leq number of initial collision vector 1s, giving upper and lower bounds on the MAL.

A given pipeline design may not provide the required latency. A method of reducing the latency is to insert delays into the pipeline to expand the reservation table and reduce the chances of a collision. In general, any fixed latency equal to or greater than the lower bound can be achieved with the addition of delays, though it may never be possible to achieve a particular cycle of unequal latencies. Mathematical methods exist to determine whether a particular cycle could be achieved (see Kogge (1981)).

Given a simple cycle of equal latencies, and that all stages (Xs) in the reservation table depend upon previously marked stages, we have the following algorithm to identify where to place delays for a latency of n cycles:

1. Starting with the first X in the original reservation table, enter the X in a revised table and mark every n cycles from this position to indicate that these positions have been reserved for the initiations every n cycles. Mark with, say, an F (for forbidden).
2. Repeat for subsequent Xs in the original reservation table until an X falls on an entered forbidden F mark. Then delay the X one or more positions until a free position is found for it. Re-mark delayed positions with a D. Delay all subsequent Xs by the same amount.

All Ds in the reservation table indicate where delays must be generated in the pipeline.

Figure 4.23(a) shows a reservation table with a collision vector 11011. There is one simple cycle (2,5) giving an MAL of 3.5. However, the lower bound (number of Xs in any row) is 2. The previous algorithm is performed for a cycle of (2) in Figure 4.23(b).

Only one delay is necessary in Figure 4.23. This delay consists of a stage in the pipeline which simply holds the information for one cycle as it passes from one stage to the next. It can be implemented using only one extra stage latch. Multiple delays between processing stages, had they been required, might be best implemented using a dual port memory in which different locations can be read and written simultaneously, as shown in Figure 4.24. Locations read are those which were written n cycles previously, when an n -cycle delay was required.

		Time					
		0	1	2	3	4	5
Stages	1	X				X	
	2		X			X	
	3			X	X		
	4						X

(a) Original reservation table

		Time						
		0	1	2	3	4	5	6
Stages	1	X				D	X	
	2		X			X		
	3			X	X			
	4							X

(b) Reservation table with delay added

Figure 4.23 Adding delays to reduce latency (a) Original reservation table (b) Reservation table with delay added

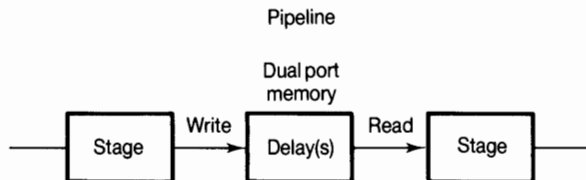


Figure 4.24 Using dual port memory for delay

The algorithm described assumes that Xs must be maintained in the same order as in the original reservation table. It may be that certain stages could be executed before others, though the relationship between the stages is not shown in the reservation table. In that case, it would not be necessary to delay all subsequent Xs, only those which depended upon the delayed stage.

Apart from having a strategy for accepting initiations, pipeline control logic is necessary to control the flow of data between stages. A flexible method of control is by microprogramming, in which the specific actions are encoded in a control memory. This method can be extended so that the specific actions are encoded in words which pass from one stage to the next with the data.

4.5 Pipelining in vector computers

We conclude this chapter with some remarks on the design of large, very high speed *vector computers*, these being a very successful application of pipelining. Apart from normal “scalar” instructions operating upon one or two single element operands, vector computers have instructions which can operate on strings of numbers formed as one-dimensional arrays (vectors). Vectors can contain either all integers or all floating point numbers. A vector computer might handle sixty-four element vectors. One operation can be specified on all the elements of vectors in a single instruction. Various vector instructions are possible, notably arithmetical/logical operation requiring one or two vectors, or one scalar and one vector producing a vector result, and an arithmetical/logical operation on all the elements of one vector to produce a scalar result. *Vector processors* can also be designed to attach to scalar computers to increase their performance on vector computations. *Supercomputers* normally have vector capability.

Vector computers can be register-to-register type, which use a large number of processor registers to hold the vectors (e.g. Cray 1, 2, X-MP, Y-MP computers) or memory-to-memory type, which use main memory locations to hold the vectors (e.g. Cyber 205). Most systems use vector registers. In either case, the general architecture is broadly as shown in Figure 4.25, where the data elements are held in main memory or processor registers. As in all stored program computers described, instructions are read from a program memory by a processor. The vector processor accepts elements from one or two vectors and produces a stream of result elements.

Most large, high speed computer systems have more than one functional unit to perform arithmetical and logical operations. For example, in a vector computer, separate scalar and vector arithmetical functional units can be provided, as can different functional units for addition/subtraction and multiplication/division. Functional units can be pipelined and fed with operands before previous results have been generated if there are no hazard conditions. Figure 4.26 shows multiple functional units using vector registers to hold vector operands, as in Cray computers; scalar register would also exist. The units take operands from vector registers and

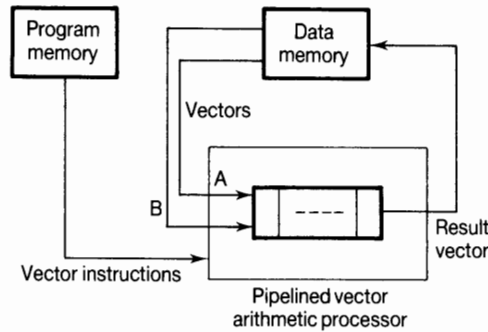


Figure 4.25 Pipelined vector processing

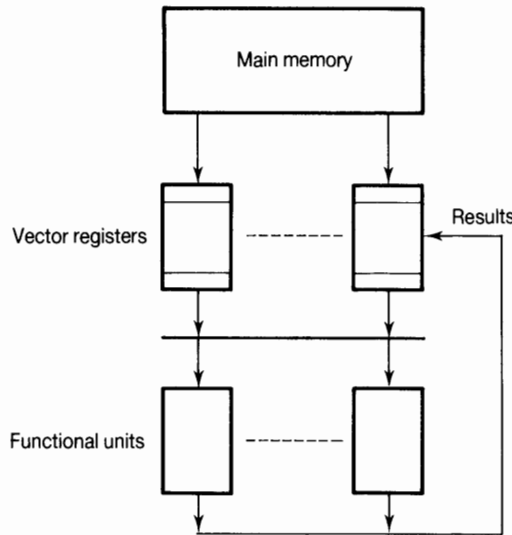


Figure 4.26 Multiple functional units

return results to the vector registers. Each vector register holds the elements of one vector, and individual elements are fed to the appropriate functional unit in succession.

Typically, a series of vector instructions will be received by the processor. To increase the speed of operation, the results of one functional unit pipeline can be fed into the input of another pipeline, as shown in Figure 4.27. This technique is known as *chaining* and overlaps pipeline operations to eliminate the “drain” time of the

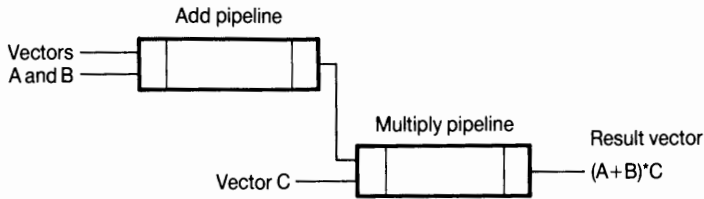


Figure 4.27 Chaining

first pipeline. More than two pipelines can be chained when available. Details of vector pipelining and chaining in large vector processor systems can be found in Cheng (1989).

PROBLEMS

4.1 Derive an expression for the minimum clock period in a ten-stage synchronous pipeline in terms of the stage operating time, t_{stage} , stage latch set-up time, $t_{\text{set-up}}$, and the clock propagation time from one stage to the next, t_{prop} , assuming that the clock passes from one stage to the next stage.

4.2 A microprocessor has two internal units, an instruction fetch unit and an instruction execute unit, with fetch/execute overlap. Compute the overall processing time of eight sequential instructions, in each of the following cases.

1. $T(F_i) = T(E_i) = 100$ ns for $i = 1$ to 8
2. $T(F_i) = 50$ ns, $T(E_i) = 100$ ns for $i = 1$ to 8
3. $T(F_i) = 100$ ns, $T(E_i) = 50, 75, 125, 100, 75$ and 50 ns for $i = 1, 2, 3, 4, 5, 6, 7$ and 8 respectively.

where $T(F_i)$ is the time to fetch the i th instruction and $T(E_i)$ is the time to execute the i th instruction.

4.3 A computer system has a three-stage pipeline consisting of an instruction fetch unit, an instruction decode unit and an instruction execute unit, as shown in Figure 4.6. Determine the time to execute twenty sequential instructions using two-way interleaved memory if the fetch unit fetches two instructions simultaneously. Draw the timing diagram for maximum concurrency given four-way interleaved memory.

4.4 A microprocessor has five pipelined internal units, an instruction fetch unit (IF), an instruction decode unit (ID), an operand fetch unit (OF), an operation execute unit (OE) and a result operand store unit (OS). Different instructions require particular units to operate for the periods shown in Table 4.1 (in cycles, one cycle = 100 ns).

Table 4.1 Pipeline unit operating times for instructions in Problem 4.4

Instruction	T(IF)	T(ID)	T(OF)	T(OE)	T(OS)
Load memory to register	1	1	1	0	0
Load register to register	1	1	0	1	0
Store register to memory	1	1	0	0	1
Add memory to register	1	1	1	1	0

Compute the overall processing time of sequential instructions, in each of the following cases.

1. MOV AX, [100] ;Copy contents of location 100
;into AX register
 MOV BX, [200]
 MOV CX, [300]
 MOV DX, [400]
2. MOV AX, [100] ;Copy contents of location 100
;into AX register
 MOV BX, [200] ;Copy contents of location 200
;into BX register
 ADD AX, BX ;Add contents of BX to AX
 MOV [200], AX ;Copy contents of AX
;into location 200

4.5 Given that an instruction pipeline has five units, as described in Problem 4.4, deduce the times required for each unit to process the following instructions:

```

ADD AX, [102]
SUB BX, AX
INC BX
MOV AX, [DX] ;Copy the contents of the
              ;location whose address is in
              ;register DX into register AX.
```

Identify three types of instructions in which $T(\text{OF}) = T(\text{OE}) = T(\text{OS}) = 0$ ns.

4.6 What is the average instruction processing time of a five-stage instruction pipeline if conditional branch instructions occur as follows: third instruction, ninth instruction, tenth instruction, twenty-fourth instruction, twenty-seventh instruction, given that there are thirty-six instructions to process? Assume that the pipeline must be cleared after a branch instruction has been decoded.

4.7 Identify potential data dependency hazards in the following code:

```

MOV  AX, [100]
ADD  AX, BX
MOV  CX, 1      ;load the literal 1 into CX register
MOV  [100], AX
MOV  [200], BX
ADD  CX, [200]

```

given a five-stage instruction pipeline. Suppose that hazards are recognized at the input to the pipeline, but that subsequent instructions are allowed to pass through the pipeline. Determine the sequence in which the instructions are processed.

4.8 Design a dynamic arithmetic pipeline which performs fixed point (integer) addition or subtraction.

4.9 Design an arithmetic pipeline which performs shift-and-add unsigned integer multiplication.

4.10 Design a static multifunction pipeline which will perform floating point addition or floating point multiplication.

4.11 Draw the reservation table for the pipeline shown in Figure 4.28, and draw an alternative pipeline which has the same reservation table.

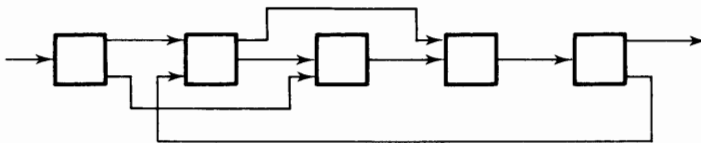


Figure 4.28 Pipeline for Problem 4.11

4.12 Determine the initial collision vector for the reservation table shown in Figure 4.29. Derive the state diagram and simplify the diagram into a reduced state diagram. List the simple cycles, and give the minimum average latency (MAL).

		Time						
		0	1	2	3	4	5	6
Stages	1	X						
	2		X	X	X		X	
	3				X	X		
	4						X	X

Figure 4.29 Reservation table for Problem 4.12

4.13 For the reservation table shown in Figure 4.30, introduce delays to obtain the cycle (3), i.e. an initiation every third cycle.

		Time						
		0	1	2	3	4	5	6
Stages	1	X	X		X			
	2			X			X	
	3				X		X	
	4					X		X

Figure 4.30 Reservation table for Problem 4.13

Reduced instruction set computers

In this chapter the concept of providing a limited number of instructions within the processor (reduced instruction set computers, RISCs) as an alternative to the more usual large number of instructions (complex instruction set computers, CISCs) will be discussed. This is a major departure from the previous trend of increasingly complex instructions, and is concerned with improving the performance of the processor.

5.1 Complex instruction set computers (CISCs)

5.1.1 Characteristics

The choice of instructions in the instruction set of the processor is a major design factor. Chapter 1 stated that operations in instructions are reduced to a simple form. However, throughout the development of computers until the 1980s, the instructions provided in the instruction set became more complex as more features were added to aid the software development and close the so-called *semantic gap* between the hardware and software. Mostly, a simple instruction format was retained with one operation, one or two operands and one result, but specialized operations and addressing modes were added. The general argument for providing additional operations and addressing modes is that they can be performed at greater speed in hardware than as a sequence of primitive machine instructions.

Let us look first at the possibilities for more complex instructions provided in *complex instruction set computers* (CISCs). Complex instructions can be identified in the following areas:

1. To replace sequences of primitive arithmetic operations.
2. For alternative indirect methods of accessing memory locations.
3. For repetitive arithmetic operations.
4. In support of procedure calls and parameter passing.

5. In support of the operation system.
6. In support of multiprocessor systems.

Less common composite operations include checking for error conditions. For example, the Motorola MC68000 has a “check register against bounds” (CHK) instruction to compare the value held in a register with an upper bound. If the upper bound is exceeded, or the register value is below zero, an exception (internal interrupt) occurs. The upper bound is held in another register or in memory.

More than one arithmetic/logic operation could be specified in one instruction, for example, to add two operands and shift the result one or more places left or right, as in the Nova minicomputer of the early 1970s. Clearly the number of instances in a program that such operations are required in sequence is limited. Arithmetic operations followed by shift operations can be found in microprogrammed devices, for example in the 4-bit Am2901 microprogram device introduced in 1975. One application at the microprogram level is to implement multiplication and division algorithms.

Apart from adding more complex operations to increase the speed of the system, complex addressing modes have also been introduced into systems. Addressing modes can be combined, for example index register addressing and base register addressing (i.e. base plus index register addressing). Indirect addressing could be multilevel. In multilevel indirect memory addressing, the address specifies a memory location which holds either the address of the operand location or, if the most significant bit is set to 1, the remaining bits are interpreted as an address of another memory location. The contents of this location are examined in the same manner. The indirection mechanism will continue until the most significant bit is 0 and the required operand address is obtained. Such multilevel indirection was provided in the NOV computer of the 1970s. Multilevel indirection is an example of a mechanism which is relatively simple to implement but which is of limited application and is now rarely found.

Support for common repetitive operations is appealing because one instruction could initiate a long sequence of similar operations without further instruction fetches. Examples include instructions to access strings and queues, and many CISCs have support for strings and queues. The Intel 8086 microprocessor family has several instructions which access a consecutive sequence of memory locations. The Motorola MC68000 microprocessor family has postincrement and predecrement addressing modes, in which the memory address is automatically incremented after a memory access and decremented prior to a memory access respectively. (Similar addressing can also be found in the VAX family.)

Multiple operations are needed during procedure calls and returns. In addition to saving and restoring the return address, more complex call and return instructions can save all the main processor registers (or a subset) automatically. Mechanisms for passing procedure parameters are helpful, as procedure calls and returns occur frequently and can represent a significant overhead.

It is helpful for the operating system if some instructions (e.g. input/output

instructions) simply cannot be executed by the user and are only available to the operation system. In addition, access to areas of memory are restricted. We have seen in Chapter 2 that memory protection can be incorporated into the memory management system. Finally, multiprocessor systems (as we shall discuss in subsequent chapters) require hardware support in the form of special instructions to maintain proper access to shared locations.

CISCs often have between 100 and 300 instructions and 8–20 addressing modes. An often quoted extreme example of a CISC is the VAX-11/780, introduced in 1978, having 303 instructions and 16 addressing modes with complex instruction encoding. Microprocessor examples include the Intel 80386, with 111 instructions and 8 addressing modes, and the Motorola MC68020, with 109 instructions and 18 addressing modes. In many cases, the development came about by extending previous system designs and because of the view that the greatest speed can be achieved by providing operations in hardware rather than using software routines.

Large numbers of operations and addressing modes require long instructions for their specification. They also require more than one instruction format because different operations require different information to be specified. In a CISC, a general technique to reduce the instruction lengths and the program storage requirements, though increasing the complexity even further, is to encode those instructions which are most likely to be used into a short length.

5.1.2 Instruction usage and encoding

To discover which instructions are more likely to be used, extensive analyses for application programs are needed. It has been found that though high level languages allow very complex constructs, many programs use simple constructs. Tanenbaum (1990) identifies, on average, 47 per cent of program statements to be assignment statements in various languages and programs, and of these assignment statements, 80 per cent are simply assigning a value to a constant. Other studies have shown that the complex addressing modes are rarely used. For example, DEC found during the development of the VAX architecture that 20 per cent of the instructions required 60 per cent of the microcode but were only used 0.2 per cent of the time (Patterson and Hennessy, 1985). This observation led to the micro VAX-32 having a slightly reduced set of the full VAX instruction set (96 per cent) but a very significant reduction in control memory (five-fold).

Hennessy and Patterson (1990) present instruction frequency results for the VAX, IBM 360, Intel 8086 and their paper design, DLX processor. Table 5.1 is based upon the 8086 results. Three programs are listed, all running under MS-DOS 3.3 on an 8086-processor IBM PC. The first is the Microsoft assembler, MASM, assembling a 500-line assembly language program. The second is the Turbo C compiler compiling the Dhrystone benchmark and the third is a Lotus 1-2-3 program calculating a 128 cell worksheet four times. The Dhrystone benchmark has been proposed as a benchmark program embodying operations of a “typical” program. This benchmark,

and the other widely quoted benchmark program – the Whetstone benchmark – have been criticized as not being able to predict performance (see for example Hennessy and Patterson (1990), pp. 73 and 183). The test done here refers to the compiler, not to the execution of the Dhrystone benchmark.

Of course, each instruction frequency study will give different results depending upon benchmark programs, the processor and other conditions. However, register accesses generally account for a large percentage of accesses, and a significant percentage are move operations (for example 51 per cent register addressing, 29 per cent MOV and 12 per cent PUSH/POP in Table 5.1). Conditional jump instructions also account for a significant percentage of instructions (10 per cent in Table 5.1) and, though not shown in Table 5.1, instructions using small literals are very commonly used for counters and indexing lists.

Table 5.1 8086 Instruction usage

	MASM assembler (%)	Turbo C compiler (%)	Lotus 1-2-3 (%)	Average (%)
Operand access				
Memory	37	43	43	41
Immediate	7	11	5	8
Register	55	46	52	51
Memory access addressing				
Indirect	12	9	15	12
Absolute	36	18	34	30
Displacement	52	73	51	59
Instruction type				
Data transfer				
MOV	30	30	21	29
PUSH/POP	12	18	8	12
LEA	3	6	0	3
Arithmetic/logical				
CMP	9	3	3	7
SAL/SHR/RCR	0	3	12	5
INC/DEC	3	3	3	5
ADD	3	3	3	3
OR/XOR	1.5	4.5	3	3
Other each				3
Control/call				
JMP	3	1.5	1.5	2
LOOP	0	0	12	4
CALL/RET	3	6	3	4
Cond. jump	12	12	6	10

CISC processors take account of this characteristic by using variable length instructions in units of bytes or 16-bit words. Totally variable length instructions, using Huffman coding, can be used and, in one study, led to a 43 per cent saving in code size (Katevenis, 1985). The Intel 432 microprocessor uses bit-encoded instructions, having from 6 to 321 bits. Instructions can be limited to be multiples of bytes or words, which leads to 35 and 30 per cent savings, respectively. Limiting instructions in this way is often done because it matches the memory byte/word fetch mechanism. For example, an MC68000 instruction can be between one and five 16-bit words. An 8086 instruction can be between 1 and 6 bytes. The VAX-11/780 takes this technique to the extreme with between 2 and 57 bytes in an instruction.

The following frequently used operations are candidates for compact encoding:

1. Loading a constant to a register.
2. Loading a small constant (say 0 to 15) to a register.
3. Loading a register or memory with 0.
4. Arithmetic operations with small literals.

The MC68000 has “quick” instructions (move/add/subtract quick) in compact encoding with small constants. Similarly, the 8086 family has compact encoding for some register operations.

A significant consequence of complex instructions with irregular encoding is the need for complex decode logic and complex logic to implement the operations specified. Most CISCs use microcode (Chapter 1) to sequence through the execution steps, an ideal method of complex instructions. This can lead to a very large control store holding the microcode. Again, an extreme example is the 456 Kbyte microcode control store of the VAX-11/780. A consequence of bit-, byte- and word-encoded instructions is that the decoding becomes a sequential operation. Decoding continues as further parts of the instruction are received.

5.2 Reduced instruction set computers (RISCs)

5.2.1 Design philosophy

The policy of complex machine instructions with complex operations and long microprograms has been questioned. An alternative design surfaced in the early 1980s, that of having very simple instructions with few operations and few addressing modes, leading to short, fast instructions, not necessarily microprogrammed. Such computers are known as *reduced instruction set computers* (RISCs) and have been established as an alternative to complex instruction set computers. The general philosophy is to transfer the complexity into software when this results in improved overall performance. The most frequent primitive operations are provided in hardware. Less frequent operations are provided only if their inclusion does not adversely

affect the speed of operation of the existing operations. The prime objective is to obtain the greatest speed of operation through the use of relatively simple hardware.

The following issues lead to the concept of RISCs:

1. The effect of the inclusion of complex instructions.
2. The best use of transistors in VLSI implementation.
3. The overhead of microcode.
4. The use of compilers.

Inclusion of complex instructions

The inclusion of complex instructions is a key issue. As we have mentioned, it was already recognized prior to the introduction of RISCs that some instructions are more frequently used than others. The CISC solution was to have shorter instruction lengths for commonly used instructions; the RISC solution is not to have the infrequently used instructions at all. To paraphrase Radin (1983), even if adding complex instructions only added one extra level of gates to a ten-level basic machine cycle, the whole CPU has been slowed down by 10 per cent. The frequency and performance improvement of the complex functions must first overcome this 10 per cent degradation and then justify the additional cost.

VLSI implementation

One of the arguments put forward for the RISC concept concerns VLSI implementation. In the opening paragraph of his award-winning thesis, Katevenis (1985) makes the point that “it was found that hardware support for complex instructions is not the most effective way of utilizing the transistors in a VLSI processor”. There is a trade-off between size/complexity and speed. Greater VLSI complexity leads directly to decreased component speeds due to circuit capacitances and signal delays. With increasing circuit densities, a decision has to be made on the best way to utilize the circuit area. Is it to add complex instructions at the risk of decreasing the speed of other operations, or should the extra space on the chip be used for other purposes, such as a larger number of processor registers, caches or additional execution units, which can be performed simultaneously with the main processor functions? The RISC proponents argue for the latter. Many RISCs employ silicon MOS technology; however, the RISC concept is also applicable to the emerging, lower density gallium arsenide (GaAs) technology and several examples of GaAs RISC processors have been constructed.

Microcode

A factor leading to the original RISC concept was changing memory technology. CISCs often rely heavily on microprogramming, which was first used at a time when the main memory was based upon magnetic core stores and faster read-only control memory could be provided. With the move to semiconductor memory, the gap between the achievable speed of operation of main memory and control memory narrows; the cache memory concept has also been developed. Now, a considerable

overhead can appear in a microprogrammed control unit, especially when a simple operation might correspond to one microinstruction. Microprogramming, in which the programmer uses the microinstructions directly, was tried in the 1970s, by providing writable control stores, but is now not popular.

Compilers

There is an increased prospect for designing optimizing compilers with fewer instructions. Some of the more exotic instructions are rarely used, particularly in compilers which have to select an appropriate instruction automatically, as it is difficult for the compiler to identify the situations where the instructions can be used effectively. A key part of the RISC development is the provision for an optimizing compiler which can take over some of the complexities from the hardware and make best use of the registers. Many of the techniques that can be used in an optimizing RISC compiler are known and can be used in CISC compilers.

Further advantages of the RISC concept include simplified interrupt service logic. In a RISC, the processor can easily be interrupted at the end of simple instructions. Long, complex instructions would cause a delay in interrupt servicing or necessitate complex logic to enable an interrupt to be servicing before the instruction had completed. A classic example of a complex instruction which could delay an interrupt service is a string instruction.

The growth of RISC systems can be evidenced by the list of twenty-one RISC processors given by Gimarc and Milutinović (1987), all developed in the 1980s; a list which does not include the MC88100 introduced by Motorola just afterwards and early prototype systems. The MC88100 is considered in Section 5.3.3 as representative of commercial RISCs.

There are claims against the RISC concept. Disadvantages include the fact that if the machine instructions are simple, it is reasonable to expect the programs to be longer. There is some dispute over this point, as it is argued that compilers can produce better optimized code from RISC instruction sets, and in any event, more complex instructions are longer than RISC instructions. Certain features identified with a RISC might also improve a CISC. For example, RISCs usually call for a large number of general purpose registers. A large register file, with a suitable addressing mechanism, could improve the performance of a CISC. Similarly, optimizing compilers using information on the internal structure of the processor can improve the performance of a CISC.

5.2.2 RISC characteristics

Though the RISC philosophy can be achieved after various architectural choices, there are common characteristics. The number of different instructions is limited to 128, or fewer, carefully selected instructions which are likely to be most used. These instructions are preferably encoded in one fixed-size word and execute in one cycle without microcoding. Perhaps only four addressing modes are provided.

Indexed and PC-relative addressing modes are probably a minimum requirement; others can be obtained from using these two addressing modes. All instructions conform to one of a few instruction formats. Memory operations are limited to load and store and all arithmetic/logical operations operate upon operands in processor registers. Hence it is necessary to have a fairly large number of general purpose processor registers, perhaps between thirty-two and sixty-four.

A memory stack is not often used for passing procedure parameters – internal processor registers are used instead, because procedure calls and returns have been identified as very common operations which could incur a heavy time penalty if they require memory accesses.

A three-register address instruction format is commonly chosen for arithmetic instructions, i.e. the operation takes operands from two registers and places the result in a third register. This reduces the number of instructions in many applications and differs from many CISC microprocessors, which often have two register, or one register/one memory, address instructions.

In keeping all instructions to a fixed size, some do not use all the bits in the instruction for their specification, and unused bits would normally be set to zero. Such wastage is accepted for simplicity of decoding. At least with fixed instruction length we do not have the problem of instructions crossing over page boundaries during a page fault. An implication of fixed instruction word length, say 32 bits, is that it is not possible to specify a 32-bit literal in one instruction – at least two instructions are needed if a 32-bit literal is necessary. It may be necessary to shift one literal before adding to another literal. Similarly, it is not possible to specify a full 32-bit address in one instruction.

Those instructions which are likely to be used need to be identified; this usually involves tracing program references of typical applications and identifying instruction usage. In CISCs, a wide range of applications is supported. One possible approach for RISCs is to limit the application area and provide instructions suitable for that area, such as embedded computers for signal processing, artificial intelligence or multiprocessing systems.

Like all processors, RISCs rely on pipeline processing. A two-stage pipeline would seem appropriate for a RISC, one stage to fetch the instruction and one to execute it. Branch instructions provided usually include the option of single cycle delayed branch instructions (described in Chapter 4) which match a two-stage pipeline well. Some RISCs do not conform to a two-stage pipeline, though all have short pipelines. For register-to-register processing, an instruction could be divided into four steps:

1. Instruction fetch/decode.
2. Register read.
3. Operate.
4. Register write.

Two-, three- and four-stage pipelines assuming register-to-register operations are

shown in Figure 5.1. In all pipelines, each register reads calls for two accesses to the internal register file to obtain both operands. Both accesses should preferably be performed simultaneously, and then a two-port register file is necessary. The actual implementation may put further requirements and constraints upon register/memory accesses, for example, because of the need to precharge buses in a VLSI implementation.

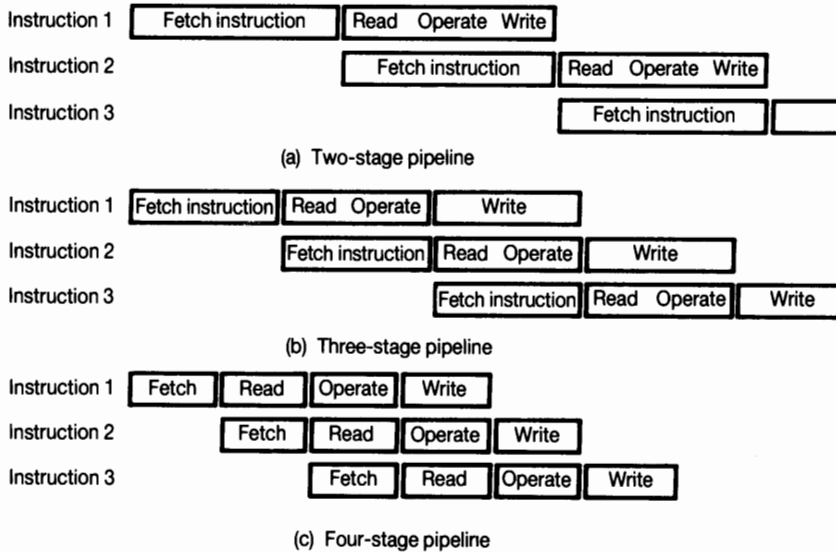


Figure 5.1 Pipelines for register-to-register operations (a) Two-stage pipeline (b) Three-stage pipeline (c) Four-stage pipeline

The two-stage pipeline assumes that an instruction fetch operation requires the same time as the read-operate-write execution phase, a reasonable assumption for register read-write operations and a main memory without a cache. A cache would bring the instruction fetch time closer to register access times. With three stages, an instruction fetch time equates with a read-operate and write times; with four stages the four steps (fetch, read, operate and write) should all take the same time, including any circuit precharging.

With three or more stages in the pipeline, there may be register read-write hazards (Chapter 4). For example, an instruction may attempt to read the contents of a register whose value has not yet been updated by a previous instruction in the pipeline. Logic can be introduced to detect the hazards (e.g. scoreboard bits) or, keeping with the RISC philosophy, such hazards could be recognized by the compiler and the instruction sequence modified accordingly.

There may also be scope for internal forwarding; when a value is written into a register it could also be transferred directly as one of the sources of a subsequent instruction, saving a register read operation. A three-stage pipeline calls for the

execution of the read/operate part of one instruction at the same time as the register write of another instruction. This would suggest a three-port register file with three buses. This can be reduced to a two-port register file by arranging for the write operation to occur during the operate part of the next instruction. The four-stage pipeline would need a three-port register file with three buses, two read and one write.

RISCs have to access the main memory for data, though with a large number of registers such access can be reduced. Accessing memory typically requires more time than register read/write. During a memory access in some designs the pipeline is stalled for one cycle, rather than having complex pipeline logic incorporated to keep it busy with other operations. There is also a potential memory conflict between an instruction fetch and a data access. Separate instruction and data memory modules with separate buses can eliminate the memory bottleneck. Some RISCs employ separate memory for data and instructions (Harvard architecture).

RISCs can employ pipelining much more extensively than the simple 2- to 4-stage pipelining described, especially if they have multiple pipelined arithmetic units which can be arranged to operate simultaneously. Memory accesses for both data and instructions may be pipelined internally.

5.3 RISC examples

5.3.1 IBM 801

The first computer system designed on RISC principles was the IBM 801 machine, designed over the period 1975–79 and publicly reported in 1982 (see Radin, 1983). The work marks the time when increasing computer instruction set complexity was first questioned. The 801 establishes many of the features for subsequent RISC designs. It has a three-register instruction format, with register-to-register arithmetic/logical operations. The only memory operations are to load a register from memory and to store the contents of a register in memory. All instructions have 32 bits with regular instruction formats. Immediate operands can appear as 16-bit arithmetic/logical immediate operands, 11-bit mask constants, 16-bit constant displacement for PC relative branch instructions and 26-bit offset for PC relative addressing or absolute addressing. The system was constructed using SSI/MSI ECL components with a cycle time of 66 ns.

Programming features include:

- 32 general purpose registers.
- 120 32-bit instructions.
- Two addressing modes:
 - base plus index;
 - base plus immediate.
- Optimizing compiler.

Architectural features include:

- Separate instruction cache and data cache.
- Four-stage pipeline:
 - Instruction fetch;
 - Register read or address calculation;
 - ALU operation;
 - Register write.
- Internal forwarding paths in pipeline.
- Interrupt facility implemented in a separate controller.

Register fields in the instruction are 5-bits (to specify one of thirty-two registers). The three-register format is carried out “pervasively” throughout. For example, it allows shift operations to specify a source register, a destination register and the number of shifts in a third register. Instruction memory contents cannot be altered except to load the instructions. Instructions are provided for cache management to reduce unnecessary cache load and store operations. Procedure parameters are passed through registers when possible. A memory stack is not used. Data is stored aligned to boundaries; words on word boundaries, half word (bytes) on half word boundaries, instructions on word boundaries.

Branch instructions come in two versions, “delayed branch with execute” and “delayed branch”. The delayed branch with execute delays execution of the branch until after the next instruction, but executes the next instruction regardless of the outcome of the branch instruction. The compiler will attempt to use the delayed branch with execute version if possible, placing a suitable instruction immediately after the branch, otherwise the non-delayed version is used.

Memory load instructions are also delayed instructions. When an instruction which will load a register is fetched, the register is locked so that subsequent instructions in the pipeline do not access it until it has been loaded properly. The compiler attempts to place instructions which do not require access to the register immediately after the “delayed load” instruction. Notice how the compiler must know the operation of the pipeline intimately to gain the greatest possible speed in the RISC. It is reported that 30 per cent of 801 instructions are load/store (Radin, 1983).

The 801 design team wanted all user programming to be done in a high level language, which means that the only assembly language programming necessary will be that for system programs. In conventional systems, hardware is provided to protect the system against the “user”. For example, in memory management, protection mechanisms exist to stop users accessing operating system memory and operating system instructions. The 801 team argument is that complex protection would slow down the system. All users should use compilers supplied with the system, and the complex protection is undesirable and unnecessary. Without hardware complexity it becomes easier to accommodate changes in technology. The 801 programming source language is called PL.8, which is based upon PL/1, but is without certain

features, such as those which would call for absolute pointers to Automatic or Static storage (Radin, 1983).

A key aspect of the project was the design of an optimizing compiler. The project depended upon being able to transfer complexity from the architecture into an optimizing compiler. From a source code program, intermediate language code is first produced and then optimized by the compiler. Conventional optimizing techniques applicable to any system are used. For example, constants are evaluated at compile time, loops are shortened by moving constant expressions to outside the loop, intermediate values are reused when possible and some procedures are expanded in-line to reduce register saving.

Allocation of variables to registers is done by considering all of the variables, rather than local variables only. Register allocation is illustrated in Figure 5.2. First, an arbitrary large number of registers is assumed to be present and the compiler uses one register for each variable in the program. The "lifetime" of each variable is identified, i.e. the time between the first and last use of the variables. Then the variables are mapped onto the available set of registers in such a manner as to minimize memory accesses. In the example of Figure 5.2, four registers are available

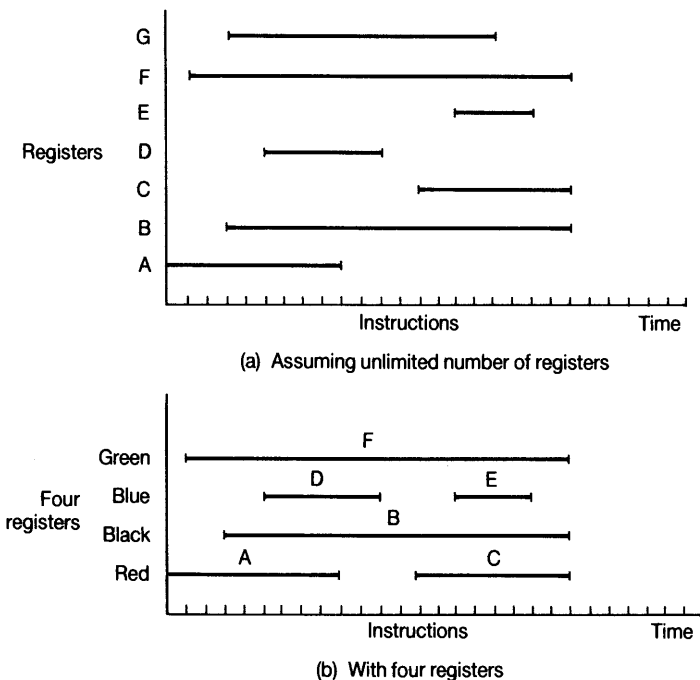


Figure 5.2 Register allocation with limited number of registers (a) Assuming unlimited number of registers (b) With four registers

(called red, black, blue and green) and seven variables, initially calling for seven registers (A, B, C, D, E, F and G). Those variables which cannot be allocated registers are held in memory, for example G in Figure 5.2. The algorithm used in the IBM project is fully described by Chaitin *et al.* (1981) and is based upon the notion that the register allocation problem can be seen as a graph coloring problem. There are other register allocation algorithms. Notice that the “lifetime” of a variable may not always represent its usage. A register with a short lifetime might be referenced many times, and hence should be held in register, while another variable might have long lifetime but is not referenced very often and would have a lower overhead if held in memory. Figure 5.2 does not show this aspect.

5.3.2 Early university research prototypes – RISC I/II and MIPS

The first university-based RISC project was probably at the University of California at Berkeley (Patterson, 1985 and Katevenis, 1985), very closely followed by the MIPS (Microprocessor without Interlocked Pipeline Stages) project at Stanford University. Both projects resulted in the first VLSI implementations of RISCs, the Berkeley RISC I in 1982, and the Stanford MIPS and the Berkeley RISC II, both in 1983. These early VLSI RISCs did not have floating point arithmetic, though it was anticipated that floating point units could be added to operate independently of other units in the processor. Floating point operations are regarded as candidates for inclusion in the instruction set, especially for numeric applications.

Features of these early VLSI RISCs are shown in Table 5.2. All processors are 32-bit, register-to-register processors and do not use microcode. Regular instruction formats are used.

Figure 5.3 shows the two instruction formats of the RISC II, where R_{s1} and R_{s2} refer to the two source registers and R_d refers to the destination register. These registers are specified by a 5-bit number, i.e. one from a group of 32 registers which can be identified from the 138 registers at any instant. (A register window pointer register is preloaded to specify which group of 32 registers is being referenced, see page 159 for more details.) The flag SCC (Set Condition Codes) specifies whether

Table 5.2 Features of early VLSI RISCs

Features	VLSI RISC		
	RISC I	RISC II	MIPS
Registers	78	138	16
Instructions	31	39	55
Addressing modes	2	2	2
Instruction formats	2	2	4
Pipeline stages	2	3	5

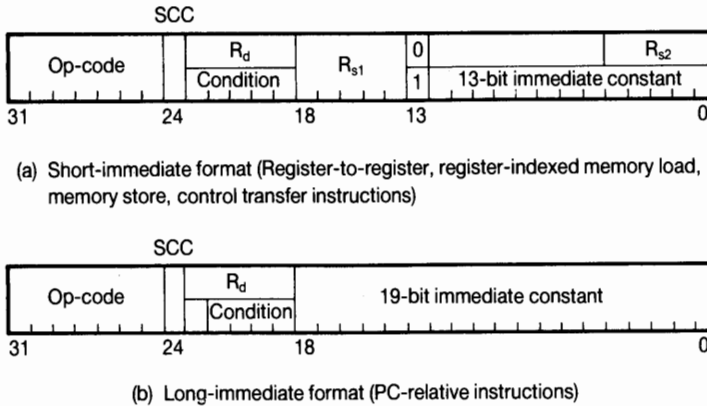


Figure 5.3 RISC I/II instruction formats (register-to-register, register-indexed memory load, memory store, control transfer instructions) **(a)** Short-immediate format **(b)** Long-immediate format (PC-relative instructions)

the condition code flags are to be set according to the result of the operation. The short-immediate format shown in Figure 5.3(a) is used for register-to-register, register-indexed memory load, memory store and control transfer instructions. Two fields in this format each have alternative interpretations, as shown. For non-conditional instructions, a destination register, R_d , is specified. For conditional instructions, a 4-bit condition is specified instead. One source operand can be held in a register, R_{s1} , or given as a 13-bit immediate constant in the instruction. The long-immediate format, shown in Figure 5.3(b), is used for PC-relative instructions. As indicated earlier, two instructions are necessary to load a 32-bit constant into a register.

Figure 5.4 shows the internal arrangement of the RISC II processor (slightly simplified). The 138 word register file is addressed from busEXT and has two buses, busA and busB. SHFTR is a 32-bit shifter using the left and right shift buses, busL and busR. BusR is also used to load the BI input of the 32-bit arithmetic/logic unit (ALU) and busL can be used to load data/constants into the data path. A full description of the operation of the RISC II can be found in Katevenis (1985). Notice the use of multiple program counters to specify the instructions in the pipeline. This characteristic can be found in subsequent RISCs.

The three-stage pipeline of the RISC II is shown in Figure 5.5. In Figure 5.5(a) all instructions are register-to-register. In Figure 5.5(b), the effect of a memory instruction is shown. Subsequent instructions are suspended while the memory access is in progress. Internal forwarding is implemented. Dataflow of operands internally forwarded to two subsequent instructions is shown by arrows. For example, while a

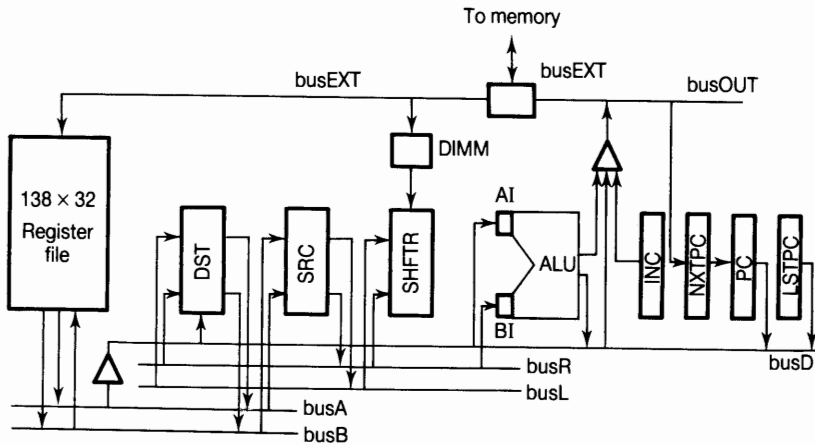


Figure 5.4 RISC II processor (DST, destination latch (a temporary pipeline latch); SRC, source latch for the shifter; DIMM, combined data in/immediate latch (holding data from memory or an immediate constant from the instruction); PC, program counter (holding the address of the instruction being executed during the current cycle); NXP, next-program counter (holding the address of the instruction being fetched during the current cycle); LSTPC, last-PC-register (holding the address of the instruction last executed or attempted to be executed); INC, incrementer which generates NXP + 4.)

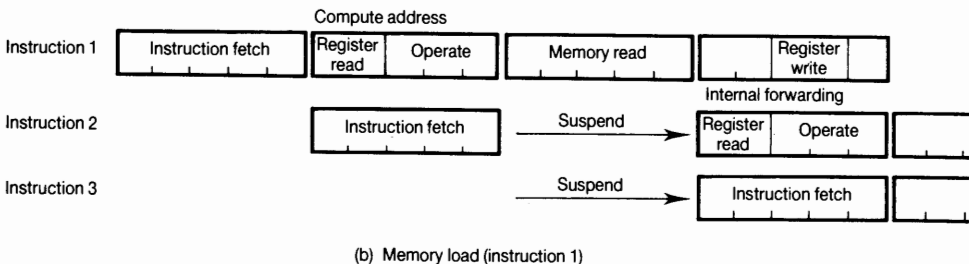
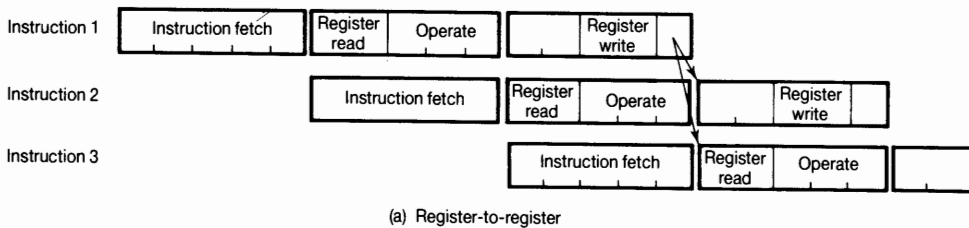


Figure 5.5 RISC II pipeline (a) Register-to-register (b) Memory load (instruction 1)

register has been loaded with a value, this value becomes immediately available without the subsequent instructions having to read the contents of the register.

The Berkeley RISC project introduced the concept of a *register window* to simplify and increase the speed of passing parameters between nested procedures. The internal register file holds parameters passed between procedures, as shown in Figure 5.6. Each procedure has registers in the file allocated for its use. The central registers are used only within the procedure. The upper portion is used by the procedure and by the procedure that called it. The lower portion is used by the procedure and the procedure it calls, i.e. both the upper and lower portions of the registers allocated to one procedure overlap with the allocation of registers of other procedures. In this way, it is not necessary to save parameters in memory during procedure calls, assuming a sufficient number of registers is provided for the procedures, otherwise main memory must be used to store some of the register contents. Another potential disadvantage occurs when multiple tasks are performed which would necessitate allocating some of the registers for particular tasks or saving registers when tasks are swapped.

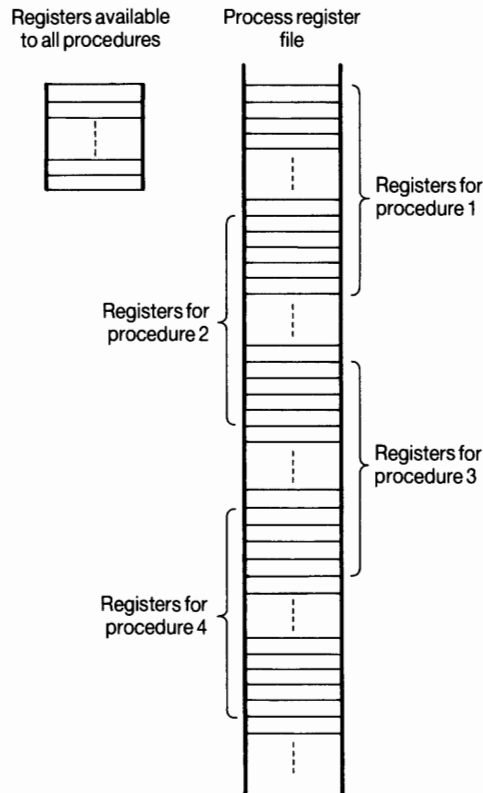


Figure 5.6 RISC register window

The seventy-eight registers of the RISC I are configured as six windows, each of fourteen registers with two groups of four overlapping registers and eighteen global registers accessible by all procedures. Each window had six local registers available to the procedure alone. The next version, the RISC II, has 138 registers configured as eight windows, each of twenty-two registers with two groups of 6 overlapping registers and 10 global registers. It was found that procedures are not usually nested to a depth of greater than eight and very rarely greater than ten or eleven, especially over any reasonably short period of the computation.

The register windows can be viewed arranged in a circular fashion, as shown in Figure 5.7 (for the RISC II). The current window pointer, CWP, points to the window that can be accessed. The specific register within the window is specified as a register number in the instruction. The register address is made up of a 3-bit window address concatenated to a 5-bit register number. Note how a register in an overlapping group has two addresses. For example, register 1:26 in window 1 is also register 2:10 in window 2. Register numbers between 0 and 9 always refer to the global registers irrespective of the current window. We would expect that during a period in the computation, the procedures would nest to a limited extent, and the circular nature of the windows accommodates this characteristic well.

5.3.3 A commercial RISC – MC88100

The Motorola MC88100 RISC 32-bit microprocessor, introduced in 1988 (Motorola, 1988a), is one of the first RISCs to be produced by a major CISC microprocessor manufacturer. The main characteristics of the MC88100 are:

1. Register-to-register (3-address) instructions, except load/store.
2. Thirty-two general purpose registers.
3. Fifty-one instructions.
4. All instructions fixed 32-bit length.
5. No microcode.
6. Four pipelined execution units that can operate simultaneously.
7. Separate data and address paths (Harvard architecture).

The instruction format is regular in that the destination and source specifications are in the same places in the instruction, though there are several instruction formats. The fifty-one instructions are given in Table 5.3, and include the common integer and floating point arithmetic and logical operations. Unusual instructions include a number of instructions for manipulating bit fields within registers. “Extract Unsigned Bit Field”, *extu*, copies a group of bits in a source register into the least significant end of the destination register. “Extract Signed Bit Field”, *ext*, is similar but sign extends the result. The position of the field in the source register is specified in terms of an offset from the least significant end and a width giving the number of bits in the field. Offset and width are held either in the instruction or in a second

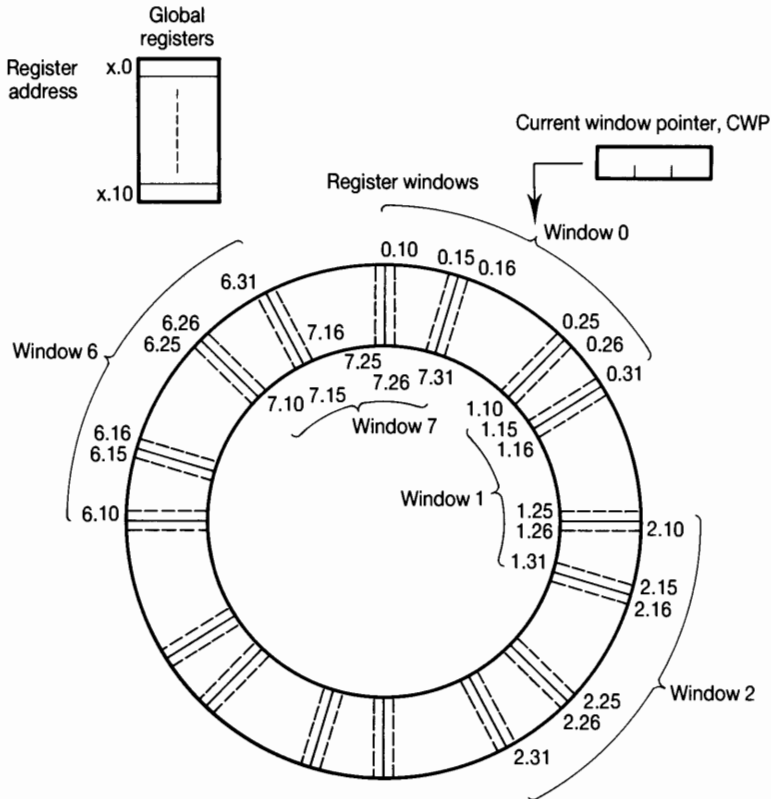


Figure 5.7 Register window addresses

source register. The reverse operation of copying a number of the least significant bits of a source register into a destination register in a specified field position is also available (“Make Bit Field”, `mak`). Fields can be set to 1s with “Set Bit Field”, `set`, or cleared to 0s with “Clear Bit Field”, `clr`. The instruction `ext` can be used for shift operations, the only specific shift instruction provided being `rot`, which rotates the contents of a source register right by a specified number of places. Another unusual instruction is “Find First Bit Clear”, `ff0`, which scans the source register from the most significant bits towards the least significant bit and loads the destination register with the bit number of the first bit found to be clear (0). “Find First Bit Set”, `ff1`, loads the bit number of the most significant bit set.

Table 5.3 MC88100 Instruction Set
courtesy of Motorola Inc.

<i>Integer arithmetic:</i>		<i>Load/store/exchange:</i>	
add	Add	ld	Load register from memory
addu	Add unsigned	lda	Load address
cmp	Compare	ldcr	Load from control register
div	Divide	st	Store register to memory
divu	Divide unsigned	stcr	Store to control register
mul	Multiply	xcr	Exchange control register
sub	Subtract	xmem	Exchange register with memory
subu	Subtract unsigned		
<i>Floating point arithmetic:</i>		<i>Flow-control:</i>	
fadd	Floating point add	bb0	Branch on bit clear
fcmp	Floating point compare	bb1	Branch on bit set
fdiv	Floating point divide	bcnd	Conditional branch
fldcr	Load from floating point control register	br	Unconditional branch
flt	Convert integer to floating point	bsr	Branch to subroutine
fmul	Floating point multiply	jmp	Unconditional jump
fstcr	Store to floating point control register	jsr	Jump to subroutine
fsub	Floating point subtract	rte	Return from exception
fxcr	Exchange floating point control register	tb0	Trap on bit clear
int	Round floating point to integer	tb1	Trap on bit set
nint	Round floating point to nearest integer	tbnd	Trap on bounds check
trnc	Truncate floating point to integer	tend	Conditional trap
<i>Logical:</i>		<i>Bit-field:</i>	
and	AND	clr	Clear bit field
mask	Logical mask immediate	ext	Extract signed bit field
or	OR	extu	Extract unsigned bit field
xor	Exclusive OR	ff0	Find first bit clear
		ff1	Find first bit set
		mak	Make bit field
		rot	Rotate register
		set	Set bit field

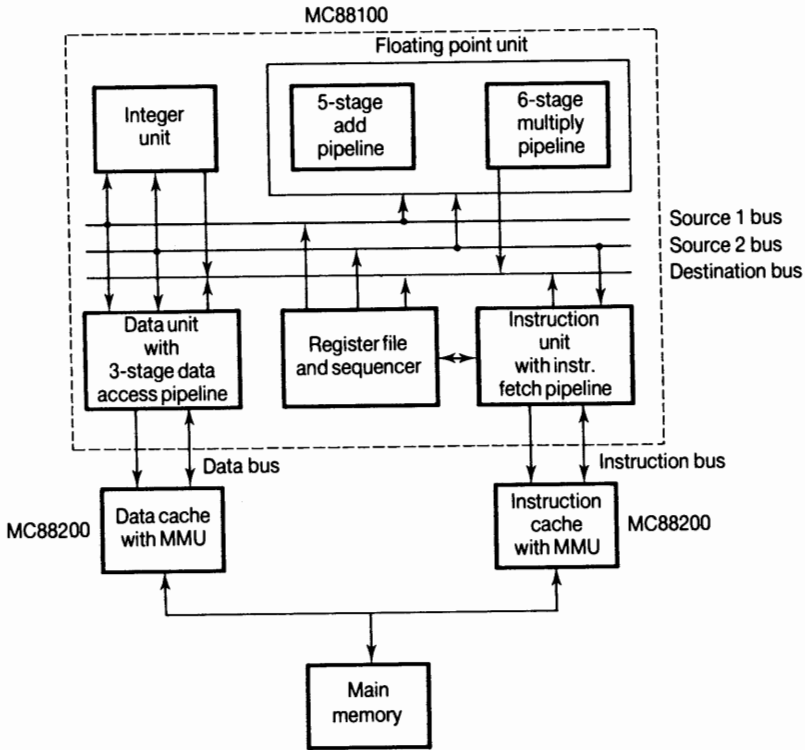


Figure 5.8 MC88100 system

There are seven addressing modes, three for accessing data and four for generating instruction addresses, namely:

Data addressing

1. Register indirect with unsigned immediate index.
2. Register indirect with register index.
3. Register indirect with scaled register index.

Instruction addressing

1. Register with 9-bit vector number.
2. Register with 16-bit signed displacement.
3. Instruction pointer relative (26-bit signed displacement).
4. Register direct.

The internal architecture of the MC88100 is shown in Figure 5.8. We would expect a RISC system to execute instructions in a single cycle and to produce a result after each cycle, and the MC88100 can achieve this. Integer arithmetic/logical instruc-

tions execute in a single cycle. However, because of the multiple pipelined units, it is possible for units to complete their operations in a different order to the one in which they were started. An internal scoreboard technique is used to keep a record of registers being updated.

Figure 5.9 shows the thirty-two general purpose registers and their usage. Except `r0` and `r1`, the uses are software conventions suggested by Motorola to aid software compatibility. Register `r0` holds the constant zero which can be read but cannot be altered. (This idea was present in the Berkeley RISC processors.) Register `r1` is loaded with a return pointer by `bsr` and `jsr`. Other registers exist in the system for floating point numbers, the supervisor state, three program counters, execute instruction pointer (XIP), next instruction pointer (NIP) and fetch instruction pointer (FIP).

r0	Zero
r1	Subroutine return pointer
r2	Called procedure parameter registers
r3	
r4	
r5	
r6	
r7	
r8	
r9	Called procedure temporary registers
r10	
r11	
r12	
r13	Calling procedure reserved registers
r14	
r15	
r16	
r17	
r18	
r19	
r20	
r21	
r22	
r23	
r24	
r25	
r26	Linker
r27	Linker
r28	Linker
r29	Linker
r30	Frame pointer
r31	Stack pointer

Figure 5.9 MC88100 general purpose registers

5.3.4 The Inmos transputer

The Inmos transputer was certainly one of the first processors to embody the principles of the RISC; in fact the early work on the transputer took place at the same time as the IBM 801, but independently and without knowledge of the latter, though the actual implementation of the transputer was not made available for some time afterwards. The transputer is a VLSI processor with external communication links to other transputers in a multiprocessor system. The multiprocessor aspect of the device, and its high level programming language *occam*, are considered in detail in Chapter 9. Occam is normally used in preference to assembly language. Here we are interested in the RISC aspect of the device and hence will mention some details of the machine language.

Basic machine instructions have one byte with the format shown in Figure 5.10. The first 4 bits specify a data operand (from 0 to 15) and the second 4 bits specify a function. The sixteen functions are allocated as follows:

- Thirteen frequently occurring functions.
- Two prefix/negative prefix functions.
- One operate function.

The thirteen frequently occurring functions include the load/store functions:

- Load constant.
- Load/store local.
- Load local pointer.
- Load/store non-local.

and also:

- Add constant.
- Jump.
- Conditional jump.
- Call.

“Local” locations are relative to a *workspace pointer*, an internal processor registers and sixteen local locations can be specified in single byte instructions. “Non-local” locations are relative to the processor A register. Inmos claims that the instructions

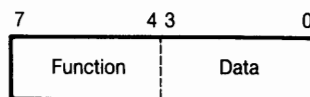


Figure 5.10 Transputer instruction format

chosen for single byte encoding lead to 80 per cent of executed instructions encoded in one byte.

The two prefix functions allow the operand to be extended in length in further bytes. Operands specified in all instructions are stored in an internal *operand register* and, apart from the prefix instructions, the operand registers are cleared of their contents after the instruction has been executed. The prefix instruction loads the 4-bit operand of the instruction in the operand register and then shifts the contents four places to the left. Thus, by including one prefix instruction before another instruction, the operand can be increased up to eight bits. The operand register in 32-bit transputers has thirty-two bits and can be completely filled using three prefix instructions and a non-prefix instruction. The “negative prefix” instruction loads the operand register but complements the contents before it shifts the contents four places left.

The “operate” function interprets the operand stored in the operand register as an operation on operands held in an internal stack. Hence, without prefix, the operate function extends the number of instructions to twenty-nine (i.e. thirteen frequently occurring functions plus sixteen operate functions). Arithmetic instructions are encoded as operate functions. With prefix, the number of instructions can be extended further, and less frequently used instructions are encoded with a single prefix.

Transputer instructions have either one address or zero address formats, the operate instructions being zero address. Three processor registers, A, B and C, are provided as an evaluation stack for zero address stack instructions (among other purposes). For example, “load local/non-local” (load onto the evaluation stack) moves the contents of B into C and the contents of A into B before loading A. “Store local/non-local” moves B into A, copies the contents of C into B and stores A. The *add* instruction adds the contents of A and B, putting the result in the A register. One address instruction uses the A register (top of the stack) inherently and the specified location is usually relative to the workspace pointer. A literal can be used.

5.4 Concluding comments on RISCs

The RISC concept has been established as a design philosophy leading away from complex instructions. This is not to say that CISCs will not be designed, especially those processors which must be compatible with existing CISC processors. For example, the Motorola 68000 family, a true CISC processor family, has been enhanced with various products since the introduction of the 68000 in 1979, including the 16-bit 68010 and 68020, and the 32-bit 68030 and 68040. The more recent trend, as in the 68040, is to have multiple pipelined units so that instructions can be executed in close to one cycle, on average (as in RISCs). Without the constraint of hardware compatibility with CISCs, RISC designs such as the

Motorola 88100 are concerned fully with performance. It seems likely that to obtain the greatest performance, processors will need to take on board RISC concepts.

PROBLEMS

5.1 A certain processor has 100 instructions in its instruction set and six addressing modes. It has four instruction formats, one 16-bit and three 32-bits. What additional information, if any, would be needed to be able to categorize the processor as a RISC or CISC?

5.2 Show how the addressing modes indexed plus literal and PC-relative can be used to implement all other common addressing modes (as given in Chapter 1, Section 1.1).

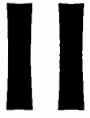
5.3 A processor has four general purpose registers (an artificially low number for this problem). By trial and error, allocate four registers to hold variables so as to minimize the number of variables held in memory, given the following lifetimes:

Variable	Lifetime
a	1 to 10
b	1 to 8
c	4 to 12
d	6 to 8
e	10 to 13
f	6 to 8
g	1 to 5
h	1 to 13

The lifetime is given in execution periods of the program. Identify the variables held in memory. When would the assignment result in non-optimum processor speed (i.e. what additional information might be needed for an optimum assignment)?

5.4 Design the logic required to decode the register addresses in the register window given in Figure 5.7.

PART



Shared memory multiprocessor systems

Multiprocessor systems and programming

This chapter identifies various types of multiprocessor systems and outlines their operation. Software techniques applicable to general purpose multiprocessors are presented, in preparation for further study of general purpose multiprocessor architectures in subsequent chapters.

6.1 General

In previous chapters, we considered methods of improving the performance of single processor computer systems. Now we will consider the extension of the stored program computer concept to systems having more than one processor. Such systems are called *multiprocessor systems*. Each processor executes the same or different instructions simultaneously, depending upon the type of system. The principal motive behind developing multiprocessors is to increase the speed of operation. (There are sometimes other motives, such as fault tolerance and matching the application.) It seems apparent that increased speed should result when more than one processor operates simultaneously. The best possible increase in speed would be proportional to the number of processors, and would occur when all the processors are operating simultaneously all the time on the application and there is no additional computation or data transfer involved in the multiprocessor implementation. Such an increase in speed is rarely achieved in practice because it is rarely possible to get all the processors doing useful work on a single problem at the same time and there is normally a substantial overhead in the communication between processors. However, considerable speed-up can be achieved.

The terms *parallelism* and *concurrency* are used to describe the simultaneous operation of multiple processors (and hence *parallel programming*, *parallel computers*, *concurrent programming*); the term *parallelism* is occasionally restricted to processors all executing the same code, though we will not make that distinction here.

There is a continual demand for greater computational speed than is possible with

available computer systems. General areas which require great computational speed include modeling, simulation and prediction, which often need repetitive calculations on large amounts of data to give valid results. Commonly quoted application examples include weather forecasting, economic forecasting and aerodynamic simulation for aircraft and space vehicles. Computer design also requires modeling and extensive calculations to simulate the systems and VLSI integrated circuits prior to manufacture. As the VLSI circuits become more complex, it takes increasingly more time to simulate them; a simulation which takes two weeks to reach a solution is usually unacceptable in a manufacturing environment. The time needs to be short enough for the designer to work effectively. Certain applications have specific target times. For example, two days for forecasting the weather the next day would make the prediction useless.

There has been a debate on whether one fast processor would be faster and more cost-effective than a system with more than one slower but less expensive processor. Most manufacturers of large computer systems in the 1970s chose to develop pipelined single processor systems with the fastest technology available if the highest speeds were required. Sometimes these processors were designed or extended to operate upon vectors as well as scalars, to improve the performance with vectors. These systems often used emitter-coupled logic (ECL) gates and interconnection delays significantly affected the overall speed operation. Prototype research machines designed and constructed during the 1970s were based upon much slower microprocessor devices although these systems generally did not demonstrate that multiple slower devices could be harnessed together to be faster and more cost-effective than ECL-based high speed single processor systems. However, in the 1980s, even manufacturers of the fastest computer systems had to develop multiprocessor versions of their single processor systems, e.g. the four-processor Cray 2 system. It is now apparent that multiprocessors must be used to obtain improvements in speed even though the difficulties of using more than one processor on a single problem have not been resolved.

Research into multiprocessor architectures continues, with each plan calling for more processors – sometimes for thousands of processors – and in the research community there have been conferences which will only accept papers describing results concerned with no less than a thousand processors. The main software aspect is how to use the processors together on a single problem. Competitions have even been organized to find the greatest possible speed-up over a single processor solution. The main architectural problem for large scale multiprocessors is how to interconnect the processors in a feasible manner. This topic is examined in Chapter 8.

Multiprocessor systems are also designed to gain fault tolerance, i.e. to be able to continue operating in the presence of hardware (and possibly software) faults. Hardware fault tolerance is achieved by the addition of circuits that are not necessary for normal operation but that enable the system to continue if faults occur. The number of faults that can be present is limited and dependent upon the system design. Specific applications exist in which computer systems must continue operating for as long as possible or over an initial time period. For example, the

computer system in an aircraft must continue working while operating the aircraft, and a faulty system could lead to loss of life. In the commercial field, computer breakdown could lead to financial loss. Manufacturing plants controlled by computers need to exhibit fault tolerance if possible, sometimes for safety, sometimes to avoid financial loss. Fault tolerance is also extremely important in the military areas (missile control, etc.).

6.2 Multiprocessor classification

6.2.1 Flynn's classification

A normal single processor stored program computer (von Neumann computer) generates a single stream of instructions which acts upon single data items. Flynn (1966) called this type of computer a *single instruction stream–single data stream* (SISD) computer. In a general purpose multiprocessor system, one instruction stream is generated for each processor. Each instruction acts upon different data. Flynn called this type of computer a *multiple instruction stream–multiple data stream* (MIMD) computer. Apart from these two extremes, it is possible (and there are some advantages in doing so) to design a computer in which a single instruction stream is generated by a single control unit, and the instructions are broadcast to more than one processor. Each processor executes the same instruction, but using different data. The data items form a vector and the instructions act upon the complete vector in one instruction cycle. Flynn called this type of computer a *single instruction stream–multiple data stream* (SIMD) computer. The fourth combination, *multiple instruction stream–single data stream* (MISD) computer does not exist, unless one specifically classifies pipelined architectures in this group, or possibly some fault tolerant systems. Flynn's classifications are shown in Figure 6.1. The SIMD computer has an array of processing elements, one for each element in the vectors being processed, and hence is also called an *array computer*.

The original stored program computer has the SISD form, with instructions operating sequentially upon integers, and later upon floating point numbers. The use of pipelining in the stored program computer does not really alter the general classification as a single instruction stream still exists and operates upon a single data stream. The SISD classification does not define the type of data items to be processed. Some large scale vector computers operate upon vectors as well as scalars (integers or floating point numbers) where a vector is a one-dimensional array of scalar elements, using the pipelining principle to process the series of elements (see Chapter 4, Section 4.5). We shall classify such pipelined vector computers as SISD. The SIMD computer also processes vectors. Hence, there are two types of vector computer, one using the pipeline technique and one using an SIMD array of processing elements

It is possible to have combined systems; for example a MSIMD system (multiple

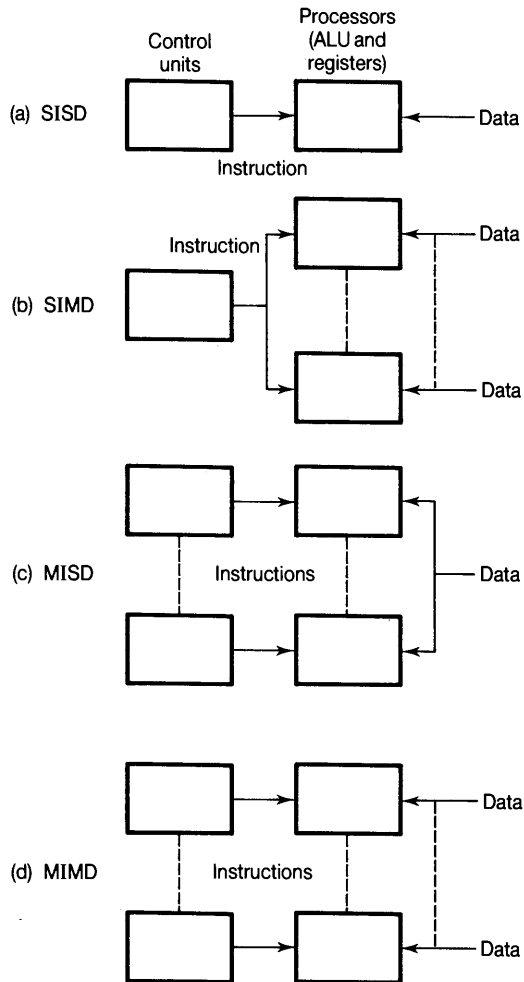


Figure 6.1 Flynn's classification (1966) (a) SISD (b) SIMD (c) MISD (d) MIMD

single instruction stream–multiple data stream computer) consists of more than one SIMD system each controlled separately.

6.2.2 Other classifications

Flynn's classifications are now very old and several other attempts have been made to classify computer systems. Feng (1972) classified systems in terms of number of bits and number of words that can be processed simultaneously, given as a tuple (bits, words). Classifications have been also proposed by Reddi and Feurstel (1976), Händler (1977), Skillicorn (1988) and Dasgupta (1990). However, as none of these classifications have been as widely used as Flynn's classifications, we shall use the latter. Classifications play a part in the study of computer architectures by cataloging existing systems and exposing architectures perhaps not yet developed.

6.3 Array computers

6.3.1 General architecture

In an array (SIMD) computer, a program memory holds the sequence of instructions to be executed and a centralized control unit extracts each instruction from the program memory in the same way as a normal von Neumann stored program (SISD) computer – by using a program counter. As in a simple SISD computer, one instruction is generally executed in its entirety, followed by the next instruction, etc. The instructions in the instruction set comprise a set of normal single processor instructions, such as integer arithmetical/logical instruction control instructions and jump/branch instructions, which are executed directly by the control unit, i.e. the control unit has the ability to perform arithmetical and control processor functions. There is also a group of instructions, especially vector instructions, which are not executed by the control unit but by the array of processing elements. These instructions are recognized by the control unit which broadcast the instructions to the processing elements for execution. Each processing element performs the operation defined upon data stored in data memory connected directly or indirectly to the processing elements. The data memory may be local to each processing element or it may be global memory connected to the processing elements through an interconnection network. The two possible schemes are shown in Figure 6.2. Suitable interconnection networks are presented in Chapter 7.

A key feature of the systems is that all processing elements operate in lock step fashion, all starting and finishing together. Each processing element has local addressable registers and the ability to perform arithmetical and logical operations upon a different data element which would be part of a vector or an array. A typical vector instruction might be $A := A + D$, where A is an array of accumulators and

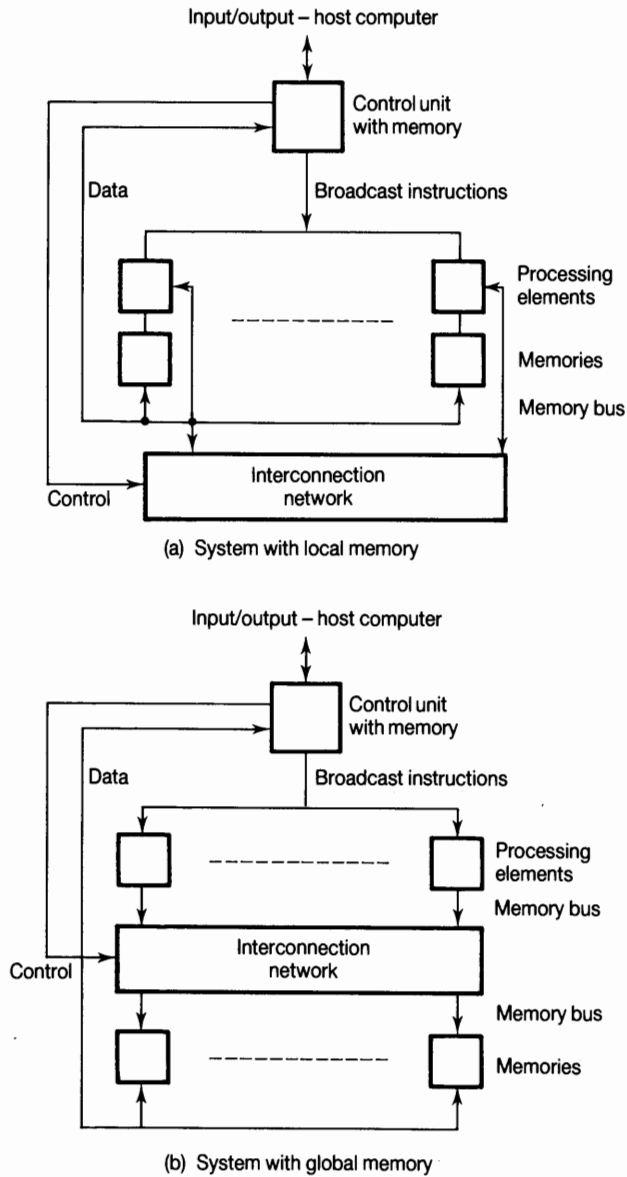


Figure 6.2 SIMD architectures (a) System with local memory (b) System with global memory

D is a data array. The i th processing element performs $A_i := A_i + D_i$, where A_i is the i th accumulator of A, which is in the i th processing element, and D_i identifies the i th data item in the array D. In the architecture shown in Figure 6.2(a), it is necessary for the i th data item to be in the local memory of the i th processing element before the instruction can be executed correctly.

Branch instructions (conditional and unconditional) are executed by the control unit. In a single processor system, the conditions for the jump instructions are related to the result after an arithmetic/logical operation. The condition might be zero result, positive or negative result, etc. Individual flags in a condition code register indicate specific conditions and others may be inferred by combining conditions (i.e. positive or zero). In an array computer with n processing elements, there can be n identical arithmetical/logical operations taking place simultaneously with n different accumulator registers (one in each processing element).

Typically, in an array computer we might like to perform a computation for those results that exhibit a particular condition, and perhaps a different computation for those results that exhibit the converse condition. This situation and other situations are handled by providing a vector of condition flags for each condition, with one bit for each processing element. This vector of condition code flags might be transferable to a control unit register under program control and then recognized by jump instructions. Masking operations on this register will identify the processing elements in which the condition is present and subsequent actions can be selected as appropriate. Specific control unit instructions can be provided for recognizing, for example, the most significant 1 in the register. A specific broadcast instruction is provided to load the register with the conditions prevailing in the processing elements. Conditional jump instructions might jump on the condition that all, any, or none of the bits are 1.

A masking mechanism is also introduced to inhibit selected processing elements from responding to the broadcast instructions. A mask is generated by specific instructions. Processing elements are given individual address index registers to modify the address of the data element to be accessed. Hence, it is possible to access rows, columns or every alternative element by each processing element modifying its index register. Each processing element can have more than one index register, additional data registers, a status/condition code register and other registers.

6.3.2 Features of some array computers

In this section we will highlight particular features of some array computers.

Illiac IV

The Illiac IV has historical importance, being the first major attempt at constructing an array computer. The Illiac IV computer system (Bouknight *et al.*, 1972) was developed in the late 1960s by the University of Illinois (hence ILLInois Array Computer) and constructed by the Burroughs Corporation in 1972. The general

architecture corresponds to the type shown in Figure 6.2(a), though the interconnection network paths are limited. The original design called for four quadrants of sixty-four processing elements (256 processing elements in all) but only one quadrant of sixty-four processing elements was finally constructed because of economic reasons and schedule delays. Each element can communicate directly with four neighboring processing elements using direct links between processing elements in an 8×8 matrix pattern as shown in Figure 6.3. The processing elements are numbered 0 to 63, so that the i th processing element can communicate directly with the $i-1$ th, $i+1$ th, $i-8$ th and the $i+8$ th processing elements. This restricted interconnection scheme can achieve any processing element connection, if not to neighboring processing elements then via intermediate processing elements. A maximum of six intermediate processing elements is necessary (seven communication steps), but most applications require less than the worst case communication overhead.

A particular feature of the system is that the control unit does not have a separate program memory described in the general scheme. Instead, the memory of the processing elements is used in an interleaved fashion, i.e. the first addressed location is in the local memory of the first processing element, the second addressed location is in the local memory of the second processing element, and so on, with every sixty-fourth location in one processing element memory. Eight 64-bit words

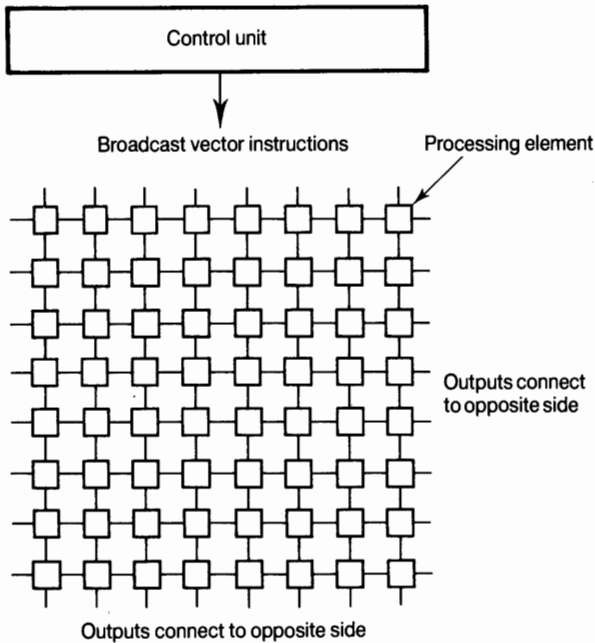


Figure 6.3 Array SIMD computer using nearest neighbor connections

can be transferred simultaneously from the processing element memories to the control unit using a 512-bit bus between the processing elements and the control unit. Each processing element has local memory consisting of 2048×64 -bit words. The control unit connects to all of the processing elements via a common data bus to pass instructions to the processing elements, and to all of the local memories through a common control bus to obtain program instructions.

The sixty-four processing elements operate in lock step fashion and receive broadcast instructions for a control unit as described previously with scalar instructions executed by the control unit. Control unit operations can overlap processing element operations. Each instruction has a fixed length of thirty-two bits and sixteen instructions are taken into an instruction buffer within the control unit simultaneously. The instruction buffer can hold sixty-four 64-bit words (128 instructions) and when execution has reached halfway through the block of instructions fetched, the next block is prefetched, replacing the oldest block. The control unit also has a 64-word data buffer (data cache) which can be loaded from the processing element memories, 512 bits at a time, and a 64-bit ALU, a program counter and four accumulators. Instructions for the processing elements (the vector and associated instructions) have their effective address calculated, using a 24-bit address adder, before passing on to the processing elements. Each processing element can perform 64-bit floating point, 32-bit floating point, 64-bit unsigned integer or 8-bit unsigned integer arithmetic.

Burroughs' Scientific Processor

The Burroughs' Scientific Processor (BSP) was developed in the mid-1970s as an attempt to construct a commercial SIMD array computer after Burroughs' involvement in the Illiac IV, though this project was abandoned and no systems were marketed. The BSP system employs architecture of the type shown in Figure 6.2(b).

A particular feature of the BSP is the use of seventeen memory modules with sixteen arithmetic processing elements. Only sixteen of the seventeen memory modules can connect to the processing elements at any instant. Two full cross-bar switch interconnection networks (Chapter 8, page 252) are used to make the connection between the arithmetic processing elements and memory modules. An "output" interconnection network is used for transfers from the sixteen processing elements to sixteen memories and an "input" interconnection network is used for transfers from the memories to the processing elements. The use of seventeen modules rather than a number which is a power of two is a unique feature of the system. The technique allows contention-free communication between the arithmetic processing elements and memory modules for most types of accesses to arrays stored in the memory modules. Only when the accesses are to elements separated by seventeen consecutively addressed locations, or a multiple of seventeen, is there memory conflict, as then the elements are inserted in the same memory module.

Spreading array elements across memory modules to avoid contention is also done in MIMD systems; a standard parallel programming technique is to spread an $n \times n$ matrix across $n + 1$ memory modules (see Rettberg and Thomas, 1986).

GF-11

The GF-11 is an array computer system developed by IBM in the early 1980s as a research vehicle primarily for numerical calculations of some of the predictions in quantum chromodynamics, a proposed theory of particles which participate in nuclear interactions (Beetem, Denneau and Weingarten, 1987). The architecture is of the type shown in Figure 6.2(a) with full interconnectivity provided between processing elements (as opposed to the limited nearest neighbor connections of the Iliac IV). There are 576 arithmetic processors, each capable of achieving 20 Mflops (million floating point operations per second). Each processor has separate fixed point and floating point units. The floating point unit has two multiply and two arithmetic units capable of addition, subtraction, computing absolute values and fixed/floating point conversion. There are three levels of local memory provided with each processor, a 12.5 ns cycle-time 256 word register file, a 50 ns cycle-time 16 Kword static memory and a 512 Kword dynamic memory arranged in two banks and providing one access every 200 ns (one word = 32 bits). The control unit broadcasts 180-bit microcode words to the processors at a 50 ns rate. The controller microcode memory consists of 512×200 -bit words. A host computer connects to the control unit.

A three-stage interconnection network, which allows all permutations of connections between the 576 processors, is used. The basic switching cell is a 9-bit wide (including parity) 24×24 cross-bar switch, and each stage has twenty-four cells. Three stages call for seventy-two cells. (See Chapter 8, page 263 for more details of multistage interconnection networks.)

The GF-11 system has a definite goal; to reduce the time it takes to perform some particularly time-consuming calculations which typically require 1×10^{17} arithmetic operations. At 20 Mflops the calculation would take 150 years; at 10 Gflops the calculation takes four months.

Concluding comments

SIMD array vector computers have a long history, with the Iliac IV a landmark in the development of the idea which can be traced back to the late 1950s. After the Iliac IV, a few subsequent systems were constructed and refined the basic concept. None of the systems were developed into commercial products and commercial computer manufacturers have been less than enthusiastic about the idea, preferring to keep to traditional SISD systems, enhanced by vector instructions using pipelining. (Some people do refer to pipelined systems as a form of SIMD.)

6.3.3 Bit-organized array computers

The array computers so far described use processing elements which operate upon binary words (*word-organized* or *word-slice array computers*); it is possible to design an array computer in which the processing elements operate upon one bit, or possibly a few bits, of a word each. Such *bit-organized* or *bit-slice array computers* have particular applications in image processing. Picture elements (*pixels*) of the

images are represented by an array of bits, and commonly the same Boolean operation needs to be performed upon each picture element simultaneously at great speed. Alternatively, bit-organized array computers might be designed such that processing elements can be linked together to process a selected word size in much the same way as bit-slice microprocessors are linked together to create processors operating upon words.

Examples of bit-organized array computers include the CLIP computer, developed in the mid-1970s at University College, London, for visual applications, the DAP (Distributed Array Processor) also developed in the mid-1970s by ICL, for general array computations, and the MPP (Massively Parallel Processor) (Batcher, 1980) developed by Goodyear Aerospace in the early 1980s. The MPP is really a massively parallel processor system having 16 384 bit-slice microprocessors arranged as a 128×128 array. The connection machine (Hillis, 1985) also comes under the bit-organized array computer classification.

BLITZEN project

The BLITZEN project (Blevins *et al.*, 1988) developed a system similar to the MPP using VLSI technology. Each array chip in the BLITZEN system has 128 processors, each with 1K-bit local static memory and with processors locally interconnected using an X-grid interconnection network, as shown in Figure 6.4. (The MPP used the

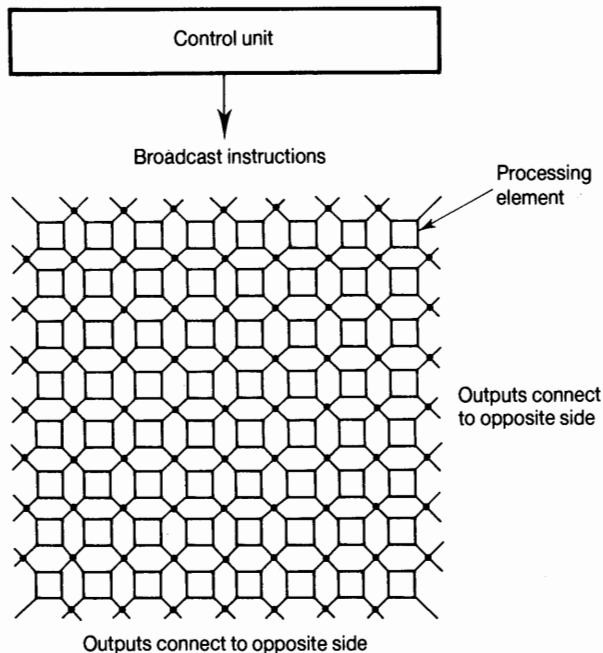


Figure 6.4 Array SIMD computer using X-network

north–south–east–west nearest neighbor network). In the X-network, the cross-over point of the diagonal paths are joined together. The network provides north, south, east, west and diagonal transfers but only with all transfers in the same direction at any instant. For example, suppose a north direction is required. Each processing element can send information on the north–east diagonal path and accept information on the south–east diagonal path. For SIMD operations, all transfers in one step are in the same direction.

We will not investigate these systems further, and will leave array and vector computers to concentrate on MIMD computer systems. The reader is directed to the references quoted for further information on array computers.

6.4 General purpose (MIMD) multiprocessor systems

6.4.1 Architectures

In a general purpose MIMD computer system, a number of independent processors operate upon separate data concurrently. Hence each processor has its own program memory or has access to program memory. Similarly, each processor has its own data memory or access to data memory. Clearly there needs to be a mechanism to load the program and data memories and a mechanism for passing information between processors as they work on some problem. There are several possible architectures for general purpose MIMD multiprocessor systems. First, we can divide the multiprocessor systems into two types:

1. Shared memory multiprocessor systems.
2. Message-passing multiprocessor systems (without shared memory).

Shared memory multiprocessors use the centralized memory for communication purposes, while *message-passing multiprocessors*, without shared memory, use direct links to pass data between processors/processing elements.

Shared memory multiprocessor systems

In a shared memory multiprocessor, given that each processor must connect to program memory and data memory (which are usually the same memory), most architectures consist of a set of processors, perhaps with local memory connecting to one or more shared memory modules, as shown in Figure 6.5.

Different systems employ different interconnection methods. We can identify the following interconnection methods:

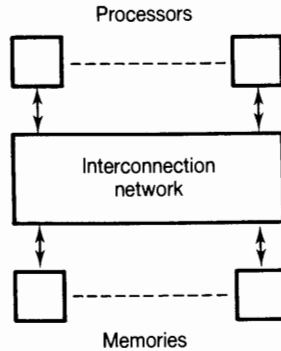


Figure 6.5 Shared memory multiprocessor system

1. Single bus.
2. System and local buses.
3. Multiple buses.
4. Cross-bar switch.
5. Multiport memory.
6. Multistage networks.

These interconnection methods are shown in Figure 6.6. The time-shared bus system (Figure 6.6(a)) is particularly suitable as a multiprocessor extension to a normal single processor microprocessor system, or other computer systems. In a single bus system with more than one processor attached to the bus, individual processors can access any of the memory modules attached to the bus, though only one data or instruction transfer can occur on the bus at any instant. The time-shared bus system has been taken up by microprocessor manufacturers for multiple processor operation, and there are several standard buses which can support more than one microprocessor.

To reduce bus contention and increase performance, each processor can be given local memory, which can act as a cache. The local memory can attach to the associated processor using a local bus. The local buses then connect to a system bus. Global memory is provided on the system bus as shown in Figure 6.6(b).

The multiple bus multiprocessor architecture shown in Figure 6.6(c) is a direct extension of the single bus architecture but with more than one bus connecting to all of the processors, memory modules and input/output interfaces. Processors can use any free bus to make a connection to a memory, but only B such connections can be made simultaneously, given B buses. Arbitration logic is necessary to resolve

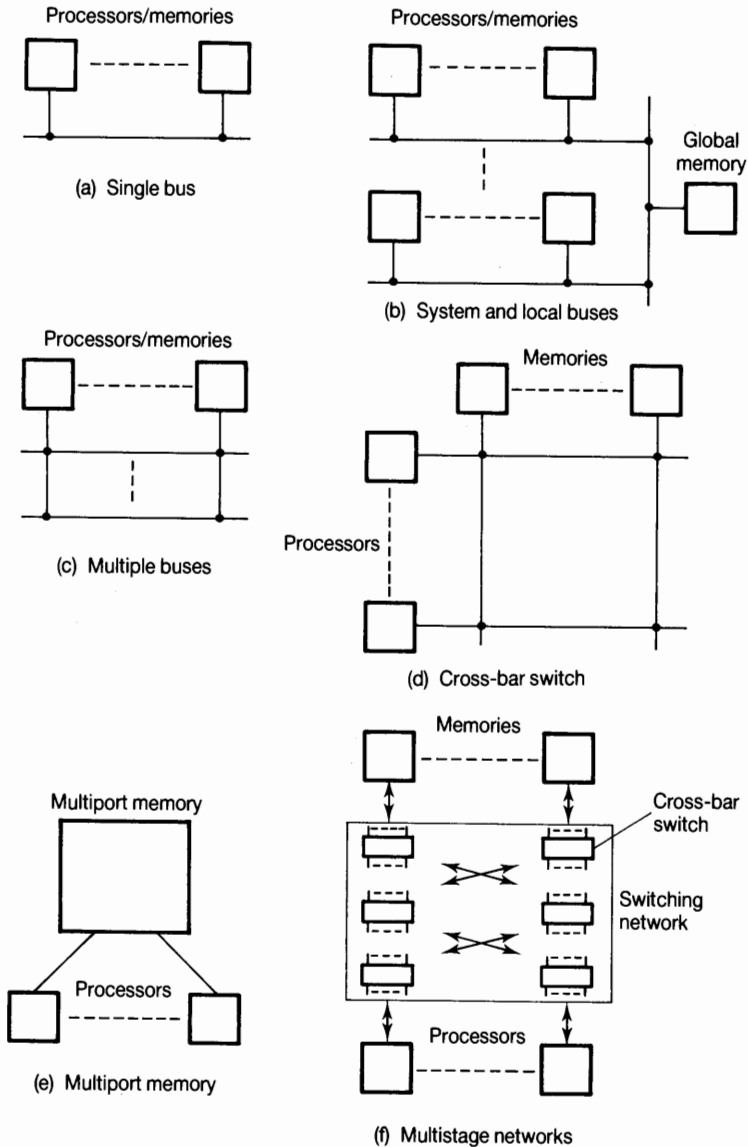


Figure 6.6 Shared memory architectures (a) Single bus (b) System and local buses (c) Multiple buses (d) Cross-bar switch (e) Multiport memory (f) Multistage networks

simultaneous requests, first to select up to one request for each memory module and then to select up to B of those requests to use the buses.

In the cross-bar switch system (Figure 6.6(d)), a direct path is made between each processor and each memory module using one electronic bus switch to interconnect the processor and memory module. Each bus switch connects the set of processor bus signals, perhaps between forty and eighty signals, to a memory module. The cross-bar architecture eliminates bus contention completely, though not memory contention, and can allow processors and memory to operate at their maximum speed.

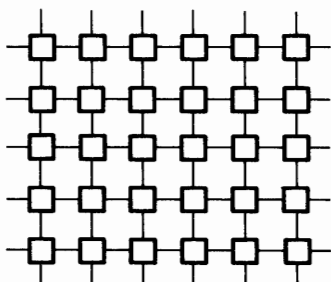
The multiport memory architecture, as shown in Figure 6.6(e), uses one multiport memory connecting to all the processors. Multiport memory is designed to enable more than one memory location to be accessed simultaneously, in this case by different processors. If there are, say, sixteen processors, sixteen ports would be provided into the memory, one port for each processor. Though large multiport memory could be designed, the design is too complex and expensive and consequently “pseudomultiport” memory is used, which appears to access more than one location simultaneously but in fact accesses the locations sequentially at high speed. Pseudomultiport memory can be implemented using normal single-port high speed random access memory with the addition of arbitration logic at the memory–processor interface to allow processors to use the memory on a first-come first-served basis. Using normal memory components, it is necessary for the memory to operate substantially faster than the processors. To service N simultaneous requests, the memory would need to operate at least N times faster than when servicing a single request. The multiport architecture with pseudomultiport memory can be considered as a variation of the cross-bar switch architecture, with each column of cross-bar switches moved to be close to the associated memory module.

The cost and complexity of the cross-bar switch grows as $O(N^2)$ where there are N processors and memory modules. Hence, the cross-bar interconnection network would be unsuitable for large numbers of processors and memory modules. In such cases, a multistage network (Figure 6.6(f)) can be used to reduce the number of switches. In such networks, a path is established through more than one switching element in an array of switching elements. Most multistage networks have three or more stages and each path requires one switch element at each stage.

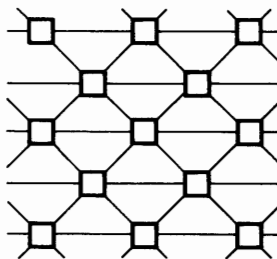
Message-passing multiprocessor systems

There are various possible direct link (*static*) interconnection networks for message-passing multiprocessor systems; some examples are shown in Figure 6.7. A very restricted static interconnection network, but a particularly suitable scheme for VLSI fabrication, is to connect processors directly to their nearest neighbors, perhaps to other processors, in a two-dimensional array of processors. Four links are needed to make contact with four other processors, as shown in Figure 6.7(a) and $3n$ links in all for n processors. In general, $n(m - 1)$ bidirectional links are needed in the array to connect n processors to m other processors (each processor having m shared

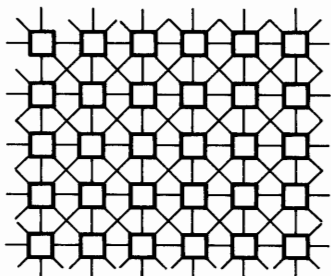
links). In a system of many concurrent processes in individual processors, processes are likely to communicate with the neighbors. Many multiprocessor algorithms are structured to create this characteristic, to map on to static array connected multiprocessors. We will consider static networks in Chapter 8 and message-passing systems in Chapter 9.



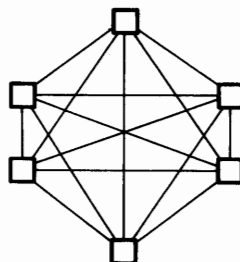
(a) Nearest neighbor mesh



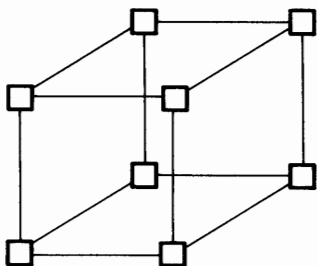
(b) Nodes with six links



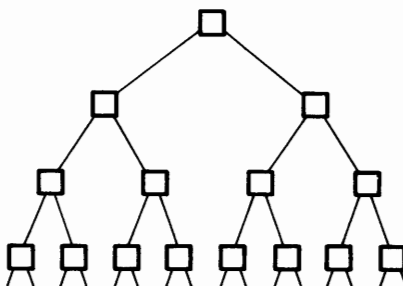
(c) Nodes with eight links



(d) Exhaustive



(e) Cubic



(f) Tree

Figure 6.7 Some static interconnection networks (a) Nearest neighbor mesh (b) Nodes with six links (c) Nodes with eight links (d) Exhaustive (e) Cubic (f) Tree

Fault tolerant systems

We mentioned in Section 6.1 that multiprocessor systems are sometimes designed to obtain increased reliability. The reliability of a system can be increased by adding redundant components. If the probability that a single component is working (the *reliability*) is given by P , the probability that at least one component is working with n duplicated components is given by $1 - (1 - P)^n$, i.e. one minus the probability that all of the components have failed. As n increases, the probability of failure decreases. In this example, the fault tolerant system with duplicated components must be designed so that only one of the components need work.

We can duplicate parts at the system level (extra systems), gate level (extra gates) or component level (extra transistors, etc.). To be able to detect failures and continue operating in the face of faults, the duplicate parts need to repeat actions performed by other parts, and some type of combining operation is performed which disregards the faulty actions. Alternatively, error detecting codes could be used; this requires extra gates.

One arrangement for system redundancy is to use three systems together with a voter circuit which examines the outputs of the systems, as shown in Figure 6.8. Each system performs the same computations. If all three systems are working, the corresponding outputs will be the same. If only two of the three systems are working, the voter chooses the two identical outputs. If more than one system is not working, the system fails. The probability that the system will operate is given by $P_s = P^3 + 3P^2(1-P)$, i.e. the probability of all three systems operating or three combinations of two systems working and one not working. The triplicated system reliability is greater than for a single system during an initial operating period, but becomes less reliable later if the reliability decreases with time (see Problem 6.4). It is assumed that there is negligible probability of two faulty systems producing the same output, and that the voter will not fail. The concept can be extended to handle two faulty systems using five systems.

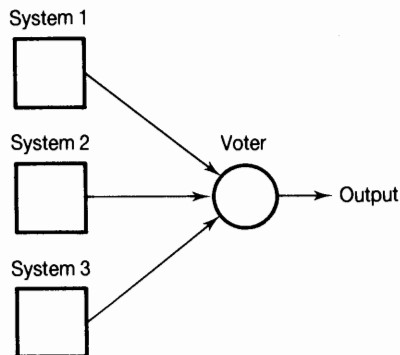


Figure 6.8 Triplicated system with a voter

6.4.2 Potential for increased speed

To achieve an improvement in speed of operation through the use of parallelism, it is necessary to be able to divide the computation into tasks or processes which can be executed simultaneously. We might use a different computational algorithm with a multiprocessor rather than with a uniprocessor system, as it may not always be the best strategy simply to take an existing sequential computation and find the parts which can be executed simultaneously. Hence, a direct comparison is somewhat complicated by the algorithms chosen for each system. However, let us ignore this point for now. Suppose that a computation can be divided, at least partially, into concurrent tasks for execution on a multiprocessor system. A measure of relative performance between a multiprocessor system and a single processor system is the *speed-up factor*, $S(n)$, defined as:

$$S(n) = \frac{\text{Execution time using one processor (uniprocessor system)}}{\text{Execution time using a multiprocessor with } n \text{ processors}}$$

which gives the increase in speed in using a multiprocessor. The efficiency, E , is defined as:

$$E = \frac{S(n)}{n} \times 100\%$$

We note that the maximum efficiency of 100 per cent occurs when the speed-up factor, $S(n)$, = n .

There are various possible divisions of processes onto processors depending upon the computation, and different divisions lead to different speed-up factors. Also, any communication overhead between processors should be taken into account. Again, there are various possible communication overheads, from exhaustive communication between all processors to very limited communication between processors. The communication overhead is normally an increasing function of the number of processors. Here we will investigate some idealized situations. We shall use the term *process* to describe a contained computation performed by a processor; a processor may be scheduled to execute more than one process.

Equal duration process

The computation might be such that it can be divided into equal duration processes, with one process mapped onto one processor. This ideal situation would lead to the maximum speed-up of n , given n processors, and can be compared to a full pipeline system (Chapter 4). The speed-up factor becomes:

$$S(n) = \frac{t}{t/n} = n$$

where t is the time on a single processor. Suppose there is a communication overhead such that each process communicates once with one other process, but concurrently, as in a linear pipeline. The communications all occur simultaneously and thus appear as only one communication, as shown in Figure 6.9. Then the speed-up would be:

$$S(n) = \frac{t}{t/n + ct/n} = \frac{n}{1 + c}$$

where c is the fractional increase in the process time which is taken up by communication between a pair of processes. If $c = 1$ then the time taken to communicate between processes is the same as the process time, $S(n) = n/2$, a reduction to half the speed-up.

In more general situations, the communication time will be a function of the number of processes and the communications cannot be fully overlapped.

Parallel computation with a serial section

It is reasonable to expect that some part of a computation cannot be divided at all into concurrent processes and must be performed serially. For example the computation might be divided as shown in Figure 6.10. During some period, perhaps an initialization period or period before concurrent processes are being set up, only one processor is doing useful work and, for the rest of the computation, all of the available processors (n processors) are operating on the problem, i.e. the remaining part of the computation has been divided into n equal processes.

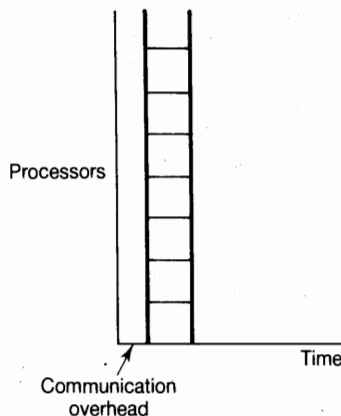


Figure 6.9 Equal duration tasks

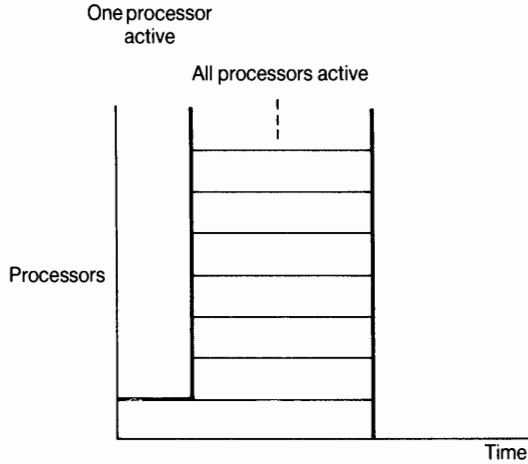


Figure 6.10 Parallel computation with serial section

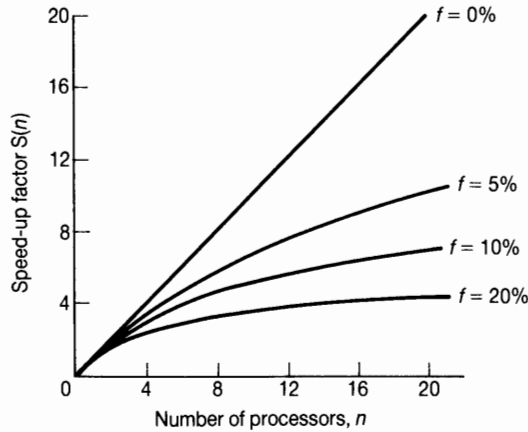
If the fraction of the computation that cannot be divided into concurrent tasks is f , and no overhead incurs when the computation is divided into concurrent parts, the time to perform the computation with n processors is given by $ft + (1-f)t/n$ and the speed-up factor is given by:

$$S(n) = \frac{t}{ft + (1-f)t/n} = \frac{n}{1 + (n-1)f}$$

This equation is known as Amdahl's law. Figure 6.11 shows $S(n)$ plotted against number of processors and plotted against f . We see that indeed a speed improvement is indicated, but the fraction of the computation that is executed by concurrent processes needs to be a substantial fraction of the overall computation if a significant increase in speed is to be achieved. The point made in Amdahl's law is that even with an infinite number of processors, the maximum speed-up is limited to $1/f$. For example, with only 5 per cent of the computation being serial, the maximum speed-up is 20, irrespective of the number of processors.

In fact, the situation could be worse. There will certainly be an additional computation to start the parallel section and general communication overhead between processes. In the general case, when the communication overhead is some function of n , say $tf_c(n)$, we have the speed-up given by:

$$S(n) = \frac{n}{1 + (1-f)n + nf_c(n)}$$



(a) Speed-up against number of processors

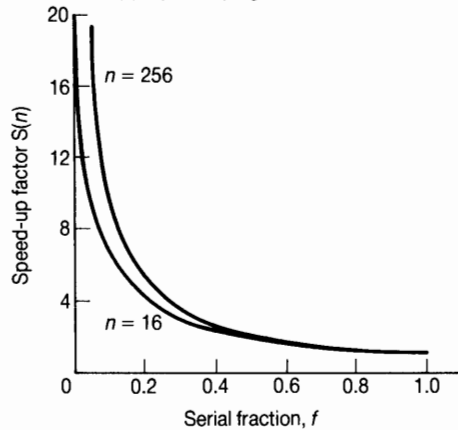

 (b) Speed-up against serial fraction, f

Figure 6.11 Speed-up factor (a) Speed-up factor against number of processors -
 (b) Speed-up factor against serial fraction, f

In practice we would expect computations to use a variable number of processors, as illustrated in Figure 6.12.

Optimum division of processes

We need to know whether utilizing all the available processors and dividing the work equally among processors is the best strategy, or whether an alternative strategy is better. Stone (1987) investigated this point and developed equations for different communication overheads, finding that the overhead eventually dominates, after which it is better not even to spread the processes among all the processors,

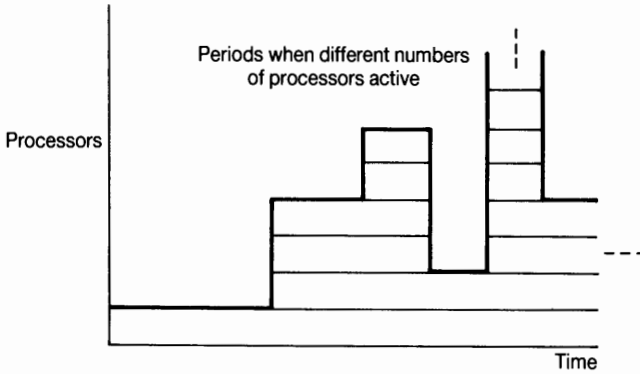


Figure 6.12 Parallel computation with variable processor usage

but to let only one processor do the work, i.e. a single processor system becomes faster than a multiprocessor system. In our equations, this point is reached when the denominator of the speed-up equations equals or exceeds n , making $S(n)$ equal or less than one. Stone confirms that if dividing the process is best, spreading the processes equally among processors is best (assuming that the number of processes will divide exactly into the number of processors).

Speed-up estimates

It was once speculated that the speed-up is given by $\log_2 n$ (Minsky's conjecture). Lea (1986) used the term *applied parallelism* for the parallelism achieved on a particular system given the restricted parallelism processing capability of the system, and suggested that the applied parallelism is typically $\log_2 n$. He used the term *natural parallelism* for the potential in a program for simultaneous execution of independent processes and suggested that the natural parallelism is $n/\log_2 n$.

Hwang and Briggs (1984) presented the following derivation for speed-up: $\leq n/\log_e n$. Suppose at some instant i processors are active and sharing the work equally with a load $1/i$ (seconds). Let the probability that i processors are active simultaneously be $P_i = 1/n$ where there are n processors. There is an equal chance of each number of processors ($i = 1, 2, 3 \dots n$) being active. The (normalized) overall processing time on the multiprocessor is given by:

$$T_n = \frac{1}{n} \sum_{i=1}^n \frac{1}{i}$$

The speed-up factor is given by:

$$S(n) = \frac{1}{n} \leq \frac{n}{\log_e n} < \frac{n}{\log_2 n}$$

$$\frac{1}{n} \sum_{i=1}^n \frac{1}{i}$$

Figure 6.13 shows the speed-up estimates. If these values could not be improved upon in practice they would lead to the conclusion that multiprocessors give poor speed-up on large numbers of processors! The challenge is to disprove this statement in practical situations and with specific multiprocessor designs. In fact, some studies have achieved nearly perfect speed-up. For example, a 256-processor Butterfly multiprocessor system has achieved a speed-up of 230 on a range of numerical calculations (Rettberg and Thomas, 1986), whereas $n/\log_2 n$ gives a value of 32.

6.5 Programming multiprocessor systems

6.5.1 Concurrent processes

A *process* or *task* is a computation performed by a processor with defined sets of inputs and outputs (results). A process could be one machine instruction, but is much more likely to be a group of machine instructions executed in sequence. The

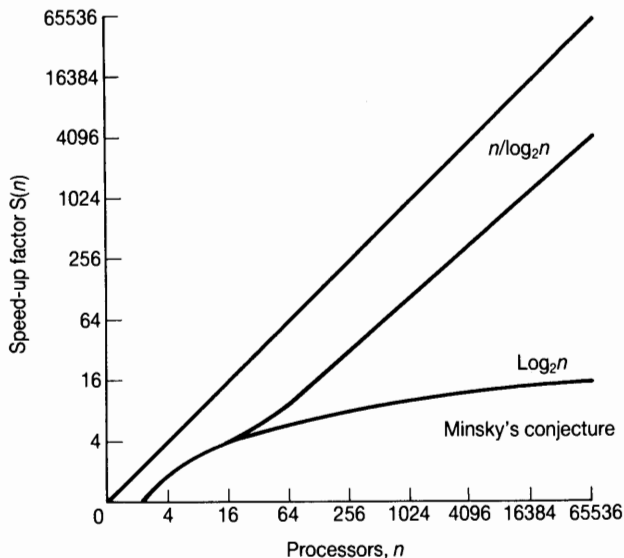


Figure 6.13 Speed-up factor estimates

size of a process and its input/output requirements have a profound effect on the performance of the system. A graphical representation of concurrent processes may be made, as shown in Figure 6.14.

Some computations might immediately suggest a solution involving concurrent processes. Such computations also tend to suggest specialized system architectures, for example, arrays of processors for visual applications. Here we will start with a computation which has been specified in a sequential manner; a transformation is then necessary to obtain a parallel computation. Transformations can be achieved in one of two principal ways:

1. By the programmer recognizing and specifying the parts which are to be executed in parallel, i.e. *explicit parallelism*.
2. By a compiler recognizing potential parallel parts and performing a restructuring algorithm, i.e. *implicit parallelism*.

A third method is to have a hardware structure which finds the parallelism when it exists. The dataflow technique, considered in Chapter 10, has this property.

6.5.2 Explicit parallelism

Most of the techniques under explicit parallelism or “programmer-defined parallelism” are centered upon providing the programmer with structures within a programming language which can be used to define the parallel processes.

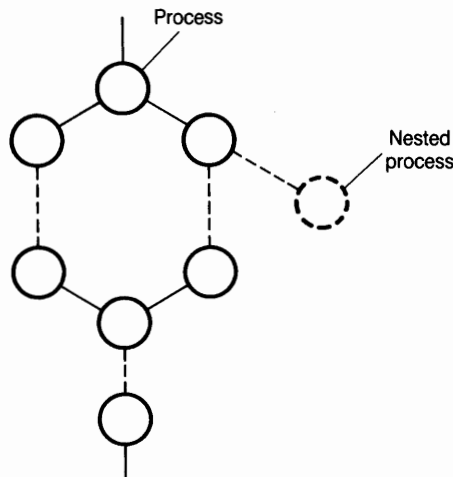


Figure 6.14 Parallel and serial processes

Constructs for FORTRAN-like languages

Perhaps the first example of programming language structures to specify parallelism is the FORK-JOIN group of statements, introduced by Conway (1963). (Conway refers to earlier work and it appears that the idea was known before 1960.) FORK-JOIN constructs have been applied as extensions to FORTRAN and, much more recently, to the UNIX operating system. In the original FORK-JOIN construct, a FORK statement generates one new path for a concurrent process and the concurrent processes use JOIN statements at their ends. When both JOIN statements have been reached, processing continues in a sequential fashion. For more concurrent processes, additional FORK statements are necessary either in sequence or at the head of the spawned processes to create further concurrent processes. The FORK-JOIN constructs are shown nested in Figure 6.15. Each spawned process requires a JOIN statement at its end which brings together the concurrent processes to a single terminating point. Only when all concurrent processes have completed can the subsequent statements be executed, and typically a counter is used to keep a record of processes not completed.

The FORK statement has the general form:

```
FORK A, J, N
```

where A is a label indicating the beginning of the spawned process and J identifies a counter which is set to the value N. If N is omitted, the counter is incremented. If J

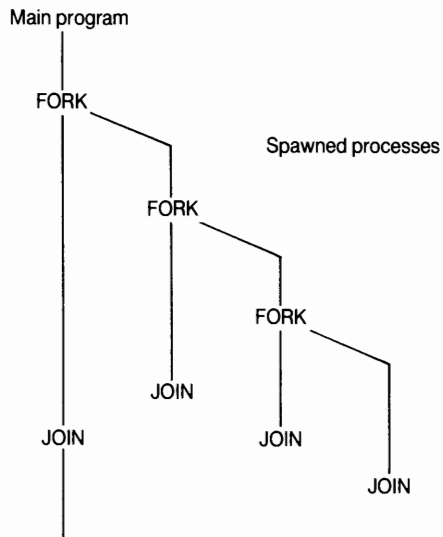


Figure 6.15 FORK-JOIN construct

is also omitted, a counter is not involved in the operation. The counter is used to indicate the number of concurrent processes which have not finished.

The JOIN statement has the general form:

```
JOIN J, B
```

where J identifies a counter which is decremented. If the counter contents then become zero, the statement specified by the label B is executed. B can be omitted, in which case the statement at the next location is executed. If the counter contents are not zero, the processor executing the JOIN statement is released.

The JOIN statement has three sequential operations. First, a counter has to be decremented. Next, the value held in the counter must be examined and finally, associated processing is either terminated or continued, dependent upon the value of the counter (not zero or zero). It is important that JOIN statements operate on the counter separately; if more than one JOIN statement were to operate on a counter simultaneously, there would be instances when incorrect program sequences would subsequently be followed. Hence, the decrement and test operations of JOIN statements must be made "indivisible" by, for example, the use of indivisible test and set machine instructions or "locked" instruction sequences (see page 204). There are also instances when FORK and JOIN statements must not operate upon the same counter simultaneously.

There are variations to the FORK-JOIN syntax described. FORK statements can be defined as spawning a number of processes, rather than only one additional process. These processes are identified by a list of parameters, for example:

```
FORK L1, L2, L3 ... Ln
```

generates n separate processes. The first process has the label $L1$ at its start, the second $L2$ and so on. The CREATE construct creates a new instance of a subroutine which is executed on a separate processor. A simple example of using CREATE is to add together the elements of a 1000 element array by dividing the summation into 10 routines, each adding up 100 elements, as in the following:

```

      .
      .
DO 100 J = 1, 9
CREATE ('SUM', ACC(J), A(I), 100)
I = I + 100
CONTINUE
CALL SUM (ACC(10), A(10), 100)
JOIN
CALL SUM (SUM, ACC, 10)
      .
      .
END
```

```

SUBROUTINE SUM (ACC,A,N)
DIMENSION A(100)
ACC = 0.0
DO 10 I = 1,N
ACC = ACC + A(I)
10 CONTINUE
RETURN
END

```

Nine instances of the subroutine SUM are created, in addition to one instance in the main program before the JOIN construct. Each instance uses a separate global accumulator, ACC(1), ACC(2) ... ACC(10). After the JOIN, the main program adds together the results held in the accumulators to produce the final result in SUM. Further examples of the CREATE construct are given in Karp (1987).

Other constructs have been proposed to specify n identical processes or similar processes, for example DOPAR-DOEND:

```

DOPAR L1,i = 1,10,1
statements
L1: PAREND

```

generates ten identical sequences. The parameters 1,10,1 are initial value, final value and step value of a variable i . Successive values are used in a different process. The construct can be compared to the FORTRAN DO-CONTINUE construct.

Constructs for block structured languages

The FORK-JOIN constructs have their origins at the time of the FORTRAN sequential programming language, which employs labels to identify the start of new program sequences selected by GOTO statements. Such methods have generally lost favor and block structured languages are now preferred, at least in the academic community. Parallel programming structures can be introduced into block structured languages such as Pascal. For example, a PARBEGIN-PAREND construct (or COBEGIN-COEND construct) can identify a group of statements which are to be executed simultaneously in Pascal-derived parallel languages, as shown below:

```

PARBEGIN
S1;
S2;
.
.
Sn;
PAREND

```

198 Shared memory multiprocessor systems

Here the statements $S_1, S_2 \dots S_n$ are specified as executed simultaneously. Each statement could be a block of statements, i.e.:

```
PARBEGIN
  BEGIN
    .
    .
  END;
  BEGIN
    .
    .
  END;
  BEGIN
    .
    .
  END;
  .
  .
PAREND
```

and hence a set of multistatement processes can be specified as being executed simultaneously. Single statement processes might incur an unacceptable communication overhead, though the construct allows this possibility.

An earlier example of this approach can be found in ALGOL-68. The order of execution of statements (or compound statements) separated by a comma instead of a semicolon was not defined, i.e. the statements would be executed in any order in a single processor system, and could be executed simultaneously in a multiprocessor system.

The PARFOR construct, found in concurrent versions of Pascal, generates a number of separate processes as specified in the construct of the form:

```
PARFOR i := 1 TO n DO
  BEGIN
    S1;
    S2;
    .
    .
    Sm
  END
```

which generates n processes each consisting of the statements $S_1, S_2 \dots S_m$. Each process uses a different value of i . For example:

```

PARFOR i := 1 TO 5 DO
  BEGIN
    A[i] := 0
  END

```

clears $A[1]$, $A[2]$, $A[3]$, $A[4]$ and $A[5]$ to zero concurrently. Examples of the similar FORALL construct for the C language are given in Terrano, Dunn and Peters (1989).

6.5.3 Implicit parallelism

In this section we will consider the detection of parallelism in programs.

Bernstein's conditions

Bernstein (1966) established a set of conditions which are sufficient to determine whether two processes can be executed in parallel. These conditions, which we will reduce to a simple form here, relate to memory locations used by the processes (usually statements). Generally, these memory locations would be used to hold variables which are to be altered or read during the executing of the processes or statements. Let us define two sets of memory locations, I (input) and O (output), such that:

I_i is the set of memory locations read by a process P_i .

O_i is the set of memory locations altered by a process P_i .

For two processes P_1 and P_2 to be executed simultaneously, the input to process P_1 must not be the output of P_2 , and the input of P_2 must not be the output of P_1 , i.e.:

$$I_1 \cap O_2 = \phi$$

$$I_2 \cap O_1 = \phi$$

where ϕ is an empty set. The output of each process must also be different, i.e.:

$$O_1 \cap O_2 = \phi$$

We will refer to the three conditions as Bernstein's conditions. We should mention in passing that Bernstein differentiated between read-only operations, write-only operations, read then write operations and write then read-only operations. He also included the set read by the process after the two processes considered for parallelization, which led to the third condition being $I_3 \cap O_2 \cap O_1 = \phi$. As noted by Baer (1980), the third condition reduces to $O_1 \cap O_2 = \phi$ for high level language statements.

200 Shared memory multiprocessor systems

If the three conditions are all satisfied, the two statements can be executed concurrently. The conditions can be applied to processes of any complexity. A process can be a single statement, when it can be determined whether the two statements can be executed simultaneously. I_i corresponds to the variables on the right hand side of the statements and O_i corresponds to the variables on the left hand side of the statements.

Example:

Suppose the two statements were (in Pascal):

A := X + Y;

B := X + Z;

we have:

$I_1 = (X, Y)$

$I_2 = (X, Z)$

$O_1 = (A)$

$O_2 = (B)$

and the conditions:

$I_1 \cap O_2 = \phi$

$I_2 \cap O_1 = \phi$

$O_1 \cap O_2 = \phi$

are satisfied. Hence the statements A := X + Y, B := X + Z can be executed simultaneously. Suppose the statements were:

A := X + Y

B := A + B

the condition $I_2 \cap O_1 \neq \phi$. Hence the two statements cannot be executed simultaneously.

The technique can be extended to processes involving more than two statements. Then, the set of inputs and outputs to each process, rather than each statement, is considered. The technique can also be used to determine whether several statements can be executed in parallel. In this case, the conditions are:

$I_i \cap O_j = \phi$

$I_j \cap O_i = \phi$

$O_i \cap O_j = \phi$

for all i, j (excluding $i = j$).

Example:

```
A := X + Y
B := X * Z
C := Y - X
```

Here $I_1 = (X, Y)$, $I_2 = (X, Z)$, $I_3 = (Y, X)$, $O_1 = (A)$, $O_2 = (B)$ and $O_3 = (C)$.
All the conditions:

$$\begin{array}{lll} I_1 \cap O_2 = \phi & I_1 \cap O_3 = \phi & I_2 \cap O_3 = \phi \\ I_2 \cap O_1 = \phi & I_3 \cap O_1 = \phi & I_3 \cap O_2 = \phi \\ O_1 \cap O_2 = \phi & O_1 \cap O_3 = \phi & O_2 \cap O_3 = \phi \end{array}$$

are satisfied and hence the three statements can be executed simultaneously (or in any order).

Parallelism in loops

Parallelism can be found in high level language loops. For example the Pascal loop:

```
FOR i := 1 TO 20 DO
  A[i] := B[i]
```

could be expanded to:

```
A[1] := B[1];
A[2] := B[2];
A[3] := B[3];
.
.
A[19] := B[19];
A[20] := B[20]
```

and, given twenty processors, these could all be executed in parallel (Bernstein's conditions being satisfied). If the result of the statement(s) within the body of the loop does depend upon previous loop iterations, it may still be possible to split the sequential statements into partitions which are independent. For example the Pascal loop:

```
FOR i := 3 TO 20 DO
  A[i] := A[i-2] + 4
```

202 Shared memory multiprocessor systems

computes:

```
A[3] := A[1] + 4;
A[4] := A[2] + 4;
A[5] := A[3] + 4;
.
.
A[19] := A[17] + 4;
A[20] := A[18] + 4
```

Hence A[5] can only be computed after A[3], A[4] after A[2] and so on. The computation can be split into two independent sequences (partitions):

```
A[3] := A[1] + 4;      A[4] := A[2] + 4;
A[5] := A[3] + 4;      A[6] := A[4] + 4;
.
.
A[17] := A[15] + 4;    A[18] := A[16] + 4;
A[19] := A[17] + 4
```

or written as two DO loops:

```
i := 3;                i := 4;
FOR j := 1 TO 9 DO      FOR j := 1 TO 8 DO
  BEGIN                 BEGIN
    i := i + 2;          i := i + 2;
    A[i] := A[i-2] + 4    A[i] := A[i-2] + 4
  END                   END
```

Each loop can be executed by a separate processor in a multiprocessor system. The approach can be applied to generate a number of partitions, dependent upon the references within the body of the loop.

A *parallelizing compiler* accepts a high level language source program and makes translations and code restructuring to create independent code which can be executed concurrently. There are various recognition algorithms and strategies that can be applied and incorporated into a parallelizing compiler apart from the methods outlined previously. Further information can be found in Padua, Kuck and Lawrie (1980) and Padua and Wolfe (1986). Some parallelizing compilers are designed to translate code into parallel form for vector computers. Padua and Wolfe use the term *concurrentizing* for code translation to create multiprocessor computations.

6.6 Mechanisms for handling concurrent processes

6.6.1 Critical sections

Suppose we have obtained, by either explicit or implicit parallelism, a set of processes that are to be executed simultaneously. A number of questions arises. First, we need a mechanism for processes to communicate and pass data, even if this only occurs when a process terminates. Coupled with this, we need a mechanism to ensure that communication takes place at the correct time, i.e. we need a synchronization mechanism. A synchronization mechanism is also required to terminate processes, as we have seen in the `JOIN` construct. If processes are to access common variables (memory locations) or interact in some other way, we need to ensure that incorrect data is not formed while two or more processes attempt to alter variables.

A process typically accesses a shared resource from time to time. The shared resource might be physical, such as an input/output device or a database contained within shared memory, and may accept data from, or provide data to, the process. More than one process might wish to access the same resource from time to time.

A mechanism for ensuring that only one process accesses a particular resource at a time is to establish sections of code involving the resource as so-called *critical sections* and arrange that only one such critical section is executed at a time, i.e. *mutual exclusion* exists. The first process to reach a critical section for a particular resource executes the critical section (“enters the critical section”) and prevents all other processes from executing a critical section for the same resource by some as yet undefined mechanism. Once the process finishes the critical section, another process is allowed to enter it for the same resource.

6.6.2 Locks

The simplest mechanism for ensuring mutual exclusion of critical sections is by the use of a *lock*. A lock is a 1-bit variable which is set to 1 to indicate that a process has entered the critical section and reset to 0 to indicate that no process is in the critical section, the last process having left the critical section. The lock operates like a door lock. A process coming to the “door” of a critical section and finding it open may enter the critical section, locking the door to prevent other processes entering. Once the process has finished the critical section, it unlocks the door and leaves.

Suppose that a process reaches a lock which is set, indicating that the process is excluded from the critical section. It now has to wait until it is allowed to enter the critical section. The process might need to examine the lock bit continually in a tight loop, for example, equivalent to:

204 Shared memory multiprocessor systems

```
WHILE Lock = 1 DO SKIP;           Skip means no operation
Lock := 1;                        enter critical section
Critical Section
Lock := 0;                        leave critical section
```

Such locks are called *spin locks* and the mechanism is called *busy waiting*. Busy waiting is inefficient of processors as no useful work is being done while waiting for the lock, though this is a common approach with locks.

Other computations could be done in place of SKIP. In some cases it may be possible to deschedule the process from the processor and schedule another process while waiting for a lock to open, though this in itself incurs an overhead in saving and reading process information. If more than one process is busy waiting for a lock to be reset, and the lock opens, a mechanism might be necessary to choose the best or highest priority process to enter the critical section, rather than let this be resolved by indeterminate busy waiting. Such a mechanism is incorporated into the semaphore operation (see Section 6.6.3).

It is important that more than one process does not set the lock (open the door) and enter the critical section simultaneously, or that one process finds the lock reset (door open) but before it can set it (close the door) another process also finds the door open and enters the critical section. Hence the actions of examining whether a lock is set and of setting it must be done as one uninterruptable operation, and one during which no other process can operate upon the lock. This exclusion mechanism is generally implemented in hardware by having special *indivisible* machine instructions which perform the complete operation sequence. Most recent microprocessors have such indivisible machine instructions.

Intel 8086 lock prefix/signal

The Intel 8086 microprocessor implements a lock operation by providing a special 1-byte LOCK instruction which prevents the next instruction from being interrupted by other bus transactions. The LOCK instruction causes a LOCK signal to be generated for the duration of the LOCK instruction and the next instruction, whatever type of instruction this may be. The LOCK signal is used with external logic to inhibit bus transactions of other processors. If a bus request is received by the processor, the request is recorded internally but not honored until after the LOCK instruction and the next instruction. The exact timing is described by Intel (1979).

The lock operation preceding a critical section could be implemented in 8086 assembly language as follows:

```
L2:    MOV CX,FFFFH    ;Set up value to load into lock
        LOCK          ;Make next instruction indivisible
        XCHG Lock,CX  ;Set lock
        JCXZ L1       ;Start critical section if
                        ; lock not originally set
        JP L2         ;Wait for lock to open
L1:    ;Critical section
```

In this sequence, XCHG Lock, CX exchanges the contents of memory location Lock and register CX. The exchange instruction takes two bus cycles to complete. Without the LOCK prefix, the exchange operation could be interrupted between bus cycles in a multiprocessor system, and lead to an incorrect result.

Motorola MC68000 Test and Set instruction

The MC68000 microprocessor has one indivisible instruction, the TAS instruction (test and set an operand), having the format:

```
TAS effective address
```

where effective address identifies a byte location using any of the 68000 “data alterable addressing” modes (Motorola, 1984). There are two sequential operations, “test” and “set”. First, the value read from the addressed location is “tested” for positive/negative and zero, i.e. the N (negative) and Z (zero) flags in the condition code register are set according to the value in the location. The Z flag is set when the bit is zero and the N flag is set when the whole number held is negative. Next, the most significant bit of the addressed location is set, irrespective of the previous test, i.e. whether or not the bit was 1, it is set to 1 during the TAS instruction. The addressed location is read, modified as necessary and the result written in one indivisible read-modify-write bus cycle. A lock operation before a critical section could be encoded using a TAS instruction in 68000 assembly language as:

```
L1: TAS Flag
    BPL L1 ;Repeat if lock already set (positive)
```

The 68000 also has a *test a bit and set* instruction (BSET) which is not indivisible and could not be used alone as a lock operation. Most processors have some form of indivisible instruction. The 32-bit MC68020 microprocessor has an indivisible *compare and swap* (CAS) instruction which can be used to maintain linked lists in a multiprocessor environment. This instruction can also be found on mainframe computers such as the IBM 370/168 (see Hwang and Briggs (1984) for more details).

Though indivisible instructions simplify the locks, locks with mutual exclusion can be implemented without indivisible TAS instructions. For example, one apparent solution is given below using two variables A and B:

Process 1	Process 2
A := 0;	B := 0;
Non-critical section	Non-critical section
A := 1;	B := 1;
WHILE B = 1 DO SKIP;	WHILE A = 1 DO SKIP;
Critical section	Critical section
A := 0;	B := 0;
Non-critical section	Non-critical section

However, this scheme can easily be deadlocked. In *deadlock*, the processes cannot proceed as each process is waiting for others to proceed. The code will deadlock when both A and B are set to 1 and tested simultaneously. SKIP could be replaced with code to avoid this type of deadlock.

The solution is still susceptible to both Process 1 and Process 2 entering the critical section together if the sequence of instructions is not executed as specified in the program, which is possible in some systems. We have seen, in Chapter 4, for example, that some pipelined systems might change the order of execution (Section 4.2.3). Memory contention and delays might also change the order of execution, if queued requests for memory are not executed in the order presented to the memory. The effect of such changes of execution was first highlighted by Lamport (1979) who used code similar to that given for Process 1 and Process 2 to elucidate a solution, namely that the following conditions must prevail:

1. Each processor issues memory requests in the order specified by its program.
2. Memory requests from all processors issued to an individual memory location are serviced from a single first-in first-out queue (in the order in which they are presented to the memory).

In fact, it is only necessary for memory requests to be serviced in the order that they are made in the program, but in practice that always means that the two separate Lamport conditions are satisfied.

To eliminate the busy waiting deadlock condition and maintain at most one process in the critical section at a time, a third variable, P, can be introduced into the code as below:

Process 1	Process 2
A := 0;	B := 0;
Non-critical section	Non-critical section
A := 1;	B := 1;
P := 2;	P := 1;
WHILE B = 1 AND P = 2 DO SKIP;	WHILE A = 1 AND P = 1 DO SKIP;
Critical section	Critical section
A := 0;	B := 0;
Non-critical section	Non-critical section

Irrespective of whether any of the instructions of one process are separated by instructions of the other process, P can only be set to Process 1 or Process 2 and hence the conditional loop will resolve the conflict and one process will be chosen to enter its critical section. It does not matter whether both conditional loops are performed simultaneously or are interleaved, though it is assumed that only one process can access a variable at a time (read or write), which is true for normal computer memory. Also, assuming that each critical section executes in a finite

time, both processes will eventually have the opportunity to enter their critical sections (i.e. the algorithm is fair to both processes). It is left as an exercise to determine whether Lamport's conditions must still be satisfied.

6.6.3 Semaphores

Dijkstra (1968) devised the concept of a semaphore which is a positive integer (including zero) operated upon by two operations named **P** and **V**. The **P** operation on a semaphore, s , written as $\mathbf{P}(s)$, waits until s is greater than zero and then decrements s by one and allows the process to continue. The **V** operation increments s by one. The **P** and **V** operations are performed indivisibly. (The letter **P** is from the Dutch word "passeren" meaning to pass, and the letter **V** is from the Dutch word "vrijgeven" meaning to release.)

A mechanism for activating waiting processes is also implicit in the **P** and **V** operations, though the exact algorithm is not specified; the algorithm is expected to be fair. Delayed processes should be activated eventually, commonly in the order in which they are delayed. Processes delayed by $\mathbf{P}(s)$ are kept in abeyance until released by a $\mathbf{V}(s)$ on the same semaphore. Processes might be delayed using a spin lock (busy waiting) or more likely by descheduling processes from processors and allocating in its place a process which is ready.

Mutual exclusion of critical sections of more than one process accessing the same resource can be achieved with one semaphore having the value 0 or 1 (a *binary semaphore*) which acts as a lock variable, but the **P** and **V** operations include a process scheduling mechanism. The semaphore is initialized to 1, indicating that no process is in its critical section associated with the semaphore. Each mutually exclusive critical section is preceded by a $\mathbf{P}(s)$ and terminated with a $\mathbf{V}(s)$ on the same semaphore, i.e.:

Process 1	Process 2	Process 3	...
Non-critical section	Non-critical section	Non-critical section	
$\mathbf{P}(s)$	$\mathbf{P}(s)$	$\mathbf{P}(s)$	
Critical section	Critical section	Critical section	...
$\mathbf{V}(s)$	$\mathbf{V}(s)$	$\mathbf{V}(s)$	
Non-critical section	Non-critical section	Non-critical section	

Any process might reach its $\mathbf{P}(s)$ operation first (or more than one process may reach it simultaneously). The first process to reach its $\mathbf{P}(s)$ operation, or to be accepted, will set the semaphore to 0, inhibiting the other processes from proceeding past their $\mathbf{P}(s)$ s, but any process reaching its $\mathbf{P}(s)$ operation will be recorded in a first-in first-out queue. The accepted process executes its critical section. When the process reaches its $\mathbf{V}(s)$ operation, it sets the semaphore s to 1 and allows one of the processes waiting to proceed into its critical section.

A general semaphore (or counting semaphore) can take on positive values other than zero and one. Such semaphores provide, for example, a means of recording the number of “resource units” available or used. Consider the action of a “producer” of data linked to a “consumer” of data through a first-in first-out buffer. The buffer would normally be implemented as a circular buffer in memory, using a pointer to indicate the front of the queue and a different pointer to indicate the back of the queue. The locations currently not holding valid data are those locations between the front and back pointer, in the clockwise direction, not including the locations pointed at by each pointer. The locations holding valid items to be taken by the consumer are those locations between the front and back pointer in the counter-clockwise direction, including the locations pointed at by each pointer.

Loading the queue and taking items from the queue must be indivisible and separate operations. Two counting semaphores can be used, one called empty, to indicate the number of empty locations in the complete circular queue, and one called full, to indicate the number of data items in the queue ready for the consumer. When the queue is full, $full = n$, the total number of locations in the queue, and $empty = 0$. When the queue is empty, the initial condition, $full = 0$ and $empty = n$. The two semaphores can be used as shown below:

Producer	Consumer
Produce data message	
$\mathbf{P}(empty)$	$\mathbf{P}(full)$
Load buffer	Take next message from queue
$\mathbf{V}(full)$	$\mathbf{V}(empty)$

Notice that the \mathbf{P} and \mathbf{V} operations surrounding each critical section do not operate on the same semaphore as in the previous example of a mutually exclusive critical section.

When the producer has a message for the queue, it performs a $\mathbf{P}(empty)$ operation. If $empty = 0$, indicating that there are no empty locations, the process is delayed until $empty \neq 0$, indicating that there is at least one free location. Then the empty semaphore is decremented, indicating that one of the free locations is to be used and the producer enters its critical section to load the buffer using the back pointer of the queue, updating the back pointer accordingly. On leaving the critical section, a $\mathbf{V}(full)$ is performed, which increments the full semaphore to show that one location has been filled.

When the consumer wants to take the next message from the queue, it performs a $\mathbf{P}(full)$ operation which delays the process if $full = 0$, i.e. if there are no messages in the queue. When $full \neq 0$, i.e. when there is at least one message in the queue, full is decremented to indicate that one message is to be taken from the queue. The consumer then enters its critical section to take the next message from the queue, using the front pointer and updating this pointer accordingly. On leaving the critical section, a $\mathbf{V}(empty)$ is performed which increments the empty semaphore to show that one more location is free.

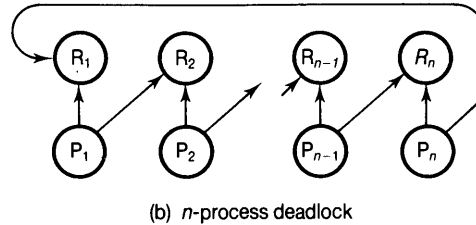
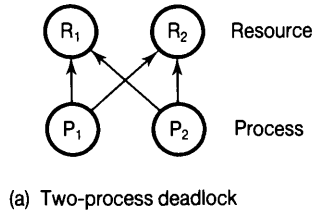


Figure 6.16 Deadlock (deadly embrace) (a) Two-process deadlock
(b) n -process deadlock

The previous example can be extended to more than one buffer between a producer and a consumer, and with more than two processes. An important factor is to avoid deadlock (sometimes called a *deadly embrace*) which prevents processes from ever proceeding. Deadlock can occur with two processes when one requires a resource held by the other, and this process requires a resource held by the first process, as shown in Figure 6.16(a). In this figure, each process has acquired one of the resources. Both processes are delayed and unless one process releases a resource wanted by the other process, neither process will ever proceed.

Deadlock can also occur in a circular fashion, as shown in Figure 6.16(b), with several processes having a resource wanted by another. Process P_1 requires resource R_2 , which is held by P_2 , process P_2 requires resource R_3 , which is held by process P_3 , and so on, with process P_n requiring resource R_1 held by P_1 , thus forming a deadlock situation. Given a set of processes having various resource requests, a circular path between any group indicates a potential deadlock situation. Deadlock cannot occur if all processes hold at most only one resource and release this resource in a finite time. Deadlock can be eliminated between two processes accessing more than one resource if both processes make requests first for one resource and then for the other.

It is widely recognised that semaphores, though capable of implementing most critical section applications, are open to human errors in use. For example, for every \mathbf{P} operation on a particular semaphore, there must be a corresponding \mathbf{V} operation on the same semaphore. Omission of a \mathbf{P} or \mathbf{V} operation, or misnaming the semaphore, would create havoc. The semaphore mechanism is a very low level mechanism

programmed into processes.

Semaphores combine two distinct purposes; first, they achieve mutual exclusion of critical sections and second, they achieve synchronization of processes. Mutual exclusion is concerned with making sure that only one process accesses a particular resource. The separate action of making sure that processes are delayed until another process has finished with the resource has been called *condition synchronization*, which leads to a *conditional critical section*, proposed independently by Hoare and by Brinch Hanson (see Andrews and Schneider (1983) for details). Another technique is to use a *monitor* (Hoare, 1974), a suite of procedures which provides the only method to access a shared resource. Reading and writing can only be done by using a monitor procedure and only one process can use a monitor procedure at any instant. If a process requests a monitor procedure while another process is using one, the requesting process is suspended and placed on a queue. When the active process has finished using the monitor, the first process in the queue (if any) is allowed to use a monitor procedure (see Grimsdale (1984)). A study of these techniques is beyond the scope of this book.

PROBLEMS

6.1 Suggest two advantages of MIMD multiprocessors and two advantages of SIMD multiprocessors.

6.2 Suggest two advantages of shared memory MIMD multiprocessor systems and two advantages of message-passing MIMD multiprocessors.

6.3 How many systems are necessary to survive any four systems failing in a fault tolerant system with a voter?

6.4 Determine when a triplicated system becomes less reliable than a single system, given that the reliability of a single system is given by $e^{-\lambda t}$. λ is the failure rate.

6.5 Identify unique features of each of the following array computers:

1. Illiac IV.
2. BSP.
3. GF-11.
4. Blitzen.

6.6 Determine the execution time to add together all elements of a 33×33 element array in each of the following multiprocessor systems:

1. An MIMD computer system with sixty-four independent processors accessing a shared memory through an interconnection network.
2. An SIMD computer system with sixty-four processors connected through a north-south-east-west nearest neighbor connection network. The processors only have local memory.
3. As 2. but with sixteen processors.
4. An SIMD system having sixty-four processors connected to shared memory through an interconnection network.

One addition takes t_a sec. Make and state any necessary assumptions.

6.7 Show a suitable path taken between two nodes which are the maximum distance apart in the Illiac IV system (with an 8×8 mesh nearest neighbor network). Develop a routing algorithm to establish a path between any two nodes. Repeat assuming that paths can only be left to right or top to bottom (in Figure 6.3).

6.8 Develop the upper and lower bound for the speed-up factor of a multiprocessor system given that each processor communicates with four other processors but simultaneous communications are not allowed.

6.9 In a multiprocessor system, the time each processor executes a critical section is given by t_c . Prove that the total execution time is given by:

$$T_p = fT_1 + (1 - f)T_1/p + t_c$$

and hence prove that the best case time becomes:

$$T_p = fT_1 + t_c + \max((1 - f)T_1/p, (p-1)t_c)$$

where T_1 is the total execution time with one processor, p is the number of processors in the system and f is the fraction of the operations which must be performed sequentially. Differentiate the first expression to obtain the number of processors for the minimum execution time. Assume that a sufficient number of processors is always available for any program.

6.10 Using Bernstein's conditions, identify the statements that can be executed simultaneously in the following:

```

A := D*E;
D := A*E;
E := A*D;
B := A*B;
E := E+1;

```

Are there any statements that can be executed simultaneously and are not identified by Bernstein's conditions? Is it possible for such statements to be present?

6.11 Separate the following Pascal nested loop into independent loops which can be executed on different processors simultaneously:

```

FOR i := 2 TO 12 DO
  FOR j := 1 TO 10 DO
    X[i] := X[i+j]*X[i]

```

6.12 Deduce what the following parallel code achieves (given in two versions, one "C-like" and one "Pascal-like"):

C-like:

```

PARFOR (i = 1, j = 1; i <= 10; i++, j++) {
  pixel[i][j] = (pixel[i][j+1]+pixel[i+1][j]
                +pixel[i][j-1]+pixel[i-1][j])/4;
}

```

Pascal-like:

```

j := 1;
PARFOR i = 1 TO 10 DO
  BEGIN
    j := i;
    pixel[i, j] = (pixel[i, j+1]+pixel[i+1, j]
                  +pixel[i, j-1]+pixel[i-1, j])/4
  END

```

In what aspect is the Pascal version inefficient?

6.13 Identify the conditions (if any) which lead to deadlock or incorrect operation in the code for a lock using the three shared variables A, B and P (Section 6.6.2).

Single bus multiprocessor systems

This chapter will consider the use of a bus to interconnect processors, notably microprocessors. Substantial treatment of the arbitration function is given and the extension of the single bus system to incorporate system and local buses is considered. The operation of coprocessors on local buses is presented with microprocessor examples.

7.1 Sharing a bus

7.1.1 General

Microprocessor systems with one processor normally use a bus to interconnect the processor, memory modules and input/output units. This method serves well for transferring instructions from the memory to the processor and for transferring data operands to or from the memory. A single bus can be used in a multiprocessor system for interconnecting all the processors with the memory modules and input/output units, as shown in Figure 7.1. Clearly, only one transfer can take place on the

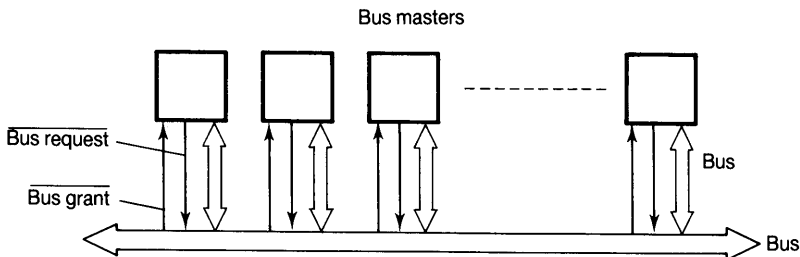


Figure 7.1 Time-shared bus system

bus at any instant; however, the scheme is practical and has been adopted by microprocessor manufacturers.

In a single bus multiprocessor system, all memory modules connecting to the bus become a single memory, available to all processors through the bus. Processors make requests to bus arbitration circuitry for the bus, and one processor is allowed to use the bus at a time. This processor can access any memory module and the performance is unaffected by the selection of the memory module. Processors compete for the use of the bus and a mechanism must be incorporated into the system to select one processor at a time to use the bus. When more than one processor wishes to use the bus, *bus contention* occurs.

A single bus can only improve processing speed if each processor attached to it has times when it does not use the bus. If each processor requires the bus continuously, no increase in speed will result, because only one processor will be active and all the other processors will be waiting for the bus. Most processors have times when they do not require the bus, though processors without local memory require the bus perhaps 50–80 per cent of the time. If a processor requires the bus 50 per cent of the time, two processors could use it alternately, giving a potential increase of speed of 100 per cent over a single processor system.

A synchronous system could achieve this speed. For example, the Motorola 6800 8-bit microprocessor operates on a two phase clock system with equal times in each phase. Memory references are only made during one phase. Hence, two processors could be arranged to operate on memory in opposite phases, and no bus arbitration circuitry would be required. If the processors each required the bus $1/n$ of the time, then n processors could use the bus in an interleaved manner, resulting in an n -fold increase in speed. If further similar processors were added, no further increase in speed would result. Below maximum utilization of the bus there is a linear increase in speed, while at the point the bus is fully utilized, no increase in speed results as further processors are added.

Synchronizing memory references is rather unusual and not applicable to more recent microprocessors; microprocessors have times when they use the bus, which change depending upon the instructions. For an asynchronous multiprocessor system where processors have to compete for the bus, processors will sometimes need to wait for the bus to be given to them, and the speed-up becomes less than in a synchronous system. A mathematical analysis is given in Section 7.3. It is rare for it to be worthwhile to attach more than 4–5 processors to a single bus.

Processors can be provided with local cache-holding instructions and data which will reduce the number of requests for memory attached to the bus and reduce bus contention. First, though, let us discuss the various mechanisms for transferring control of the bus from one processor to another. Processors, or any other device that can control the bus, will be called *bus masters*. The processor controlling the bus at any instant will be called the *current bus master*. Bus masters wishing to use the bus and making a request for it will be called *requesting bus masters*.

7.1.2 Bus request and grant signals

There are two principal signals used in the transfer of control of the bus from one bus master to another, namely the *bus request* signal and the *bus grant* signal, though other signals are usually also present and the signals are variously entitled depending upon the system designer or microprocessor manufacturer. Transfer of the control of the bus from one bus master to another uses a handshaking scheme. The bus master wishing to use the bus makes a request to the current bus master by activating the bus request signal. The current bus master releases the bus some time later, and passes back a bus grant signal to the requesting bus master, as shown in Figure 7.2(a). The exact timing is system dependent. Figure 7.2(b) shows one possible timing using the two signals described. Bus request causes, in due course, bus grant to be returned. When bus grant is received, bus request is deactivated, which causes bus grant to be deactivated. Bus control signals are often *active-low*, meaning that the quiescent state is 1 and that 0 indicates action. Such signals are shown with a bar over their name. We shall use the word “activated” to indicate action.

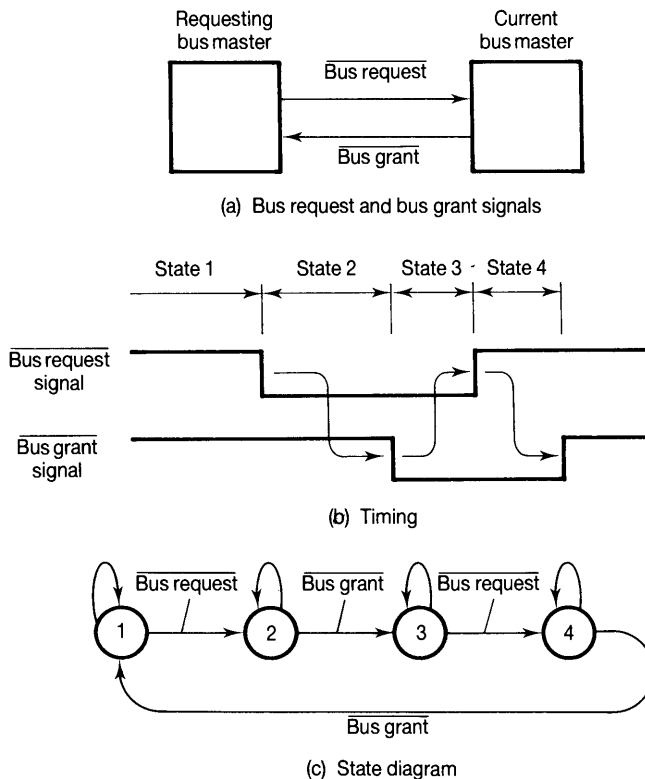


Figure 7.2 Bus request/grant mechanism (a) Bus request and bus grant signals (b) Timing (c) State diagram

Buses can be classified as either *synchronous* or *asynchronous*. For all bus transactions in the synchronous bus, the time for each transaction is known in advance, and is taken into account by the source device in accepting information and generating further signals. In the asynchronous bus, the source device does not know how long it will take for the destination to respond. The destination replies with an acknowledgement signal when ready. When applied to transferring control of the bus, the asynchronous method involves not only a request signal from the requesting bus master and a grant signal from the current bus master, but also a further *grant acknowledge* signal from the current bus master acknowledging the grant signal.

In a synchronous bus, the two signal handshake system is often augmented with a *bus busy* signal, which indicates whether the bus is being used. It may be that an *acknowledge* signal, rather than a grant signal, is returned from the current bus master to the requesting bus master after the request has been received. The current bus master then releases the bus busy line when it eventually releases the bus, and this action indicates that the requesting master can take over the bus, as shown in Figure 7.3.

Microprocessors designed for multiprocessor operation have request/acknowledge/grant signals at the pin-outs although, when there are more than two processors in the system, additional logic may be necessary to resolve multiple requests for particular schemes.

7.1.3 Multiple bus requests

It is necessary for the current bus master to decide whether to accept a particular request and to decide between multiple simultaneous requests, should these occur. In both cases, the decision is normally made on the basis of the perceived priority of the incoming requests, in much the same way as deciding whether to accept an interrupt signal in a normal single processor microprocessor system. Individual bus masters are assigned a priority level, with higher priority level masters being able to take over the bus from lower priority bus masters. The priority level may be fixed by making specific connections in a priority scheme (i.e. static priority/fixed priority) or, less commonly, altered by hardware which alters the priority according to some algorithm (dynamic priority).

Arbitration schemes can generally be:

1. Parallel arbitration schemes.
2. Serial arbitration schemes.

In parallel arbitration schemes, the bus request signals enter the arbitration logic separately and separate bus grant signals are generated. In serial arbitration schemes, a signal is passed from one bus master to another, to establish which requesting bus master, if any, is of higher priority than the current bus master. The serial configuration is often called a daisy chain scheme.

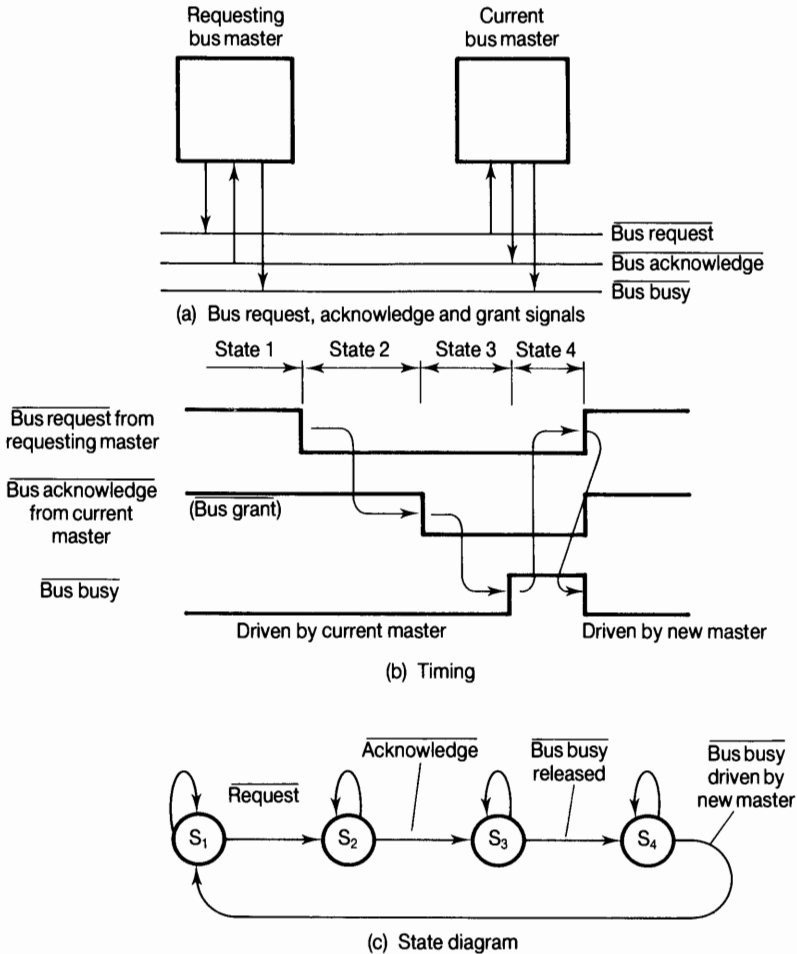


Figure 7.3 Bus request/acknowledge/busy mechanism (a) Bus request, acknowledge and busy signals (b) Timing (c) State diagram

Arbitration schemes can also be:

1. Centralized arbitration schemes.
2. Decentralized arbitration schemes.

In centralized schemes, the request signals, either directly or indirectly, reach one central location for resolution and the appropriate grant signal is generated from this

point back to the bus masters. In decentralized schemes, the signals are not resolved at one point – the decision to generate a grant/acknowledge signal may be made at various places, normally at the processor sites. The decentralized schemes often (but not always) have the potential for fault tolerance, whereas the centralized schemes are always susceptible to point failures. Parallel and serial arbitration schemes can either be centralized or decentralized, though the centralized forms are most common.

7.2 Priority schemes

7.2.1 Parallel priority schemes

The general centralized parallel priority scheme is shown in Figure 7.4. Each bus master can generate a bus request signal which enters the centralized arbitration logic (arbiter). One of the requests is accepted and a corresponding grant signal is returned to the bus master. A bus busy signal is provided; this is activated by the bus master using the bus. A bus master may use the bus when it receives a grant signal and the bus is free, as indicated by the bus busy line being inactive. While a bus master is using the bus, it must maintain its request and bus busy signals active. Should a higher priority bus master make a request, the arbitration logic recognizes the higher priority master and removes the grant from the current bus master. It also provides a grant signal to the higher priority requesting bus master, but this bus master cannot take over the bus until the current bus master has released it. The current bus master recognizes that it has lost its grant signal from the arbitration logic, but it will usually not be able to release the bus immediately if it is in the process of making a bus transfer. When a suitable occasion has been reached, the

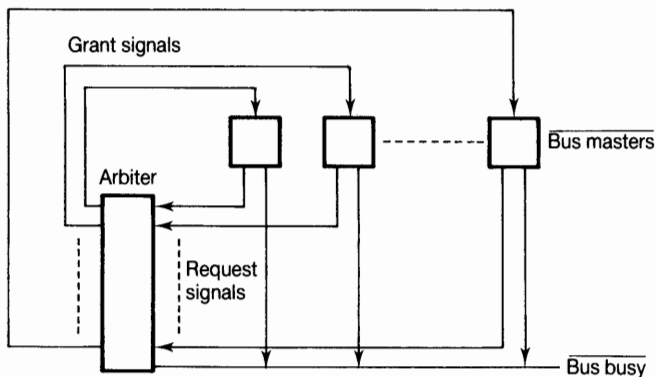


Figure 7.4 Centralized parallel arbitration

current bus master releases the bus and the bus busy line, which signals to the requesting master that it can take over the bus.

Notice that it is necessary to provide a bus busy signal because bus masters are incapable of releasing the bus immediately when they lose their grant signal. Hence we have a three signal system. There are various priority algorithms which can be implemented by the arbitration logic to select a requesting bus master, all implemented in hardware as opposed to software because of the required high speed of operation. We have already identified static and dynamic priority. In the first instance, let us consider static priority. Dynamic priority in parallel priority schemes is considered on page 225.

In static (fixed) priority, requests always have the same priority. For example, suppose that there were eight bus masters 0, 1, 2, 3, 4, 5, 6 and 7 with eight request signals REQ0, REQ1, REQ2, REQ3, REQ4, REQ5, REQ6 and REQ7, and eight associated grant signals GRANT0, GRANT1, GRANT2, GRANT3, GRANT4, GRANT5, GRANT6 and GRANT7. Bus master 7 could be assigned the highest priority, with the other bus masters assigned decreasing priority such that bus master 0 has the lowest priority. If the current master is bus master 3, any of the bus masters 7, 6, 5 and 4 could take over the bus from the current master, but bus masters 2, 1 and 0 could not. In fact, bus master 0 could only use the bus when it was not being used and would be expected to release it to any other bus master wishing to use it.

Static priority is relatively simple to implement. For eight request inputs and eight "prioritized" grant signals, the Boolean equations to satisfy are:

$$\begin{aligned} \text{GRANT7} &= \text{REQ7} \\ \text{GRANT6} &= \overline{\text{REQ7}} \cdot \text{REQ6} \\ \text{GRANT5} &= \overline{\text{REQ7}} \cdot \overline{\text{REQ6}} \cdot \text{REQ5} \\ \text{GRANT4} &= \overline{\text{REQ7}} \cdot \overline{\text{REQ6}} \cdot \overline{\text{REQ5}} \cdot \text{REQ4} \\ \text{GRANT3} &= \overline{\text{REQ7}} \cdot \overline{\text{REQ6}} \cdot \overline{\text{REQ5}} \cdot \overline{\text{REQ4}} \cdot \text{REQ3} \\ \text{GRANT2} &= \overline{\text{REQ7}} \cdot \overline{\text{REQ6}} \cdot \overline{\text{REQ5}} \cdot \overline{\text{REQ4}} \cdot \overline{\text{REQ3}} \cdot \text{REQ2} \\ \text{GRANT1} &= \overline{\text{REQ7}} \cdot \overline{\text{REQ6}} \cdot \overline{\text{REQ5}} \cdot \overline{\text{REQ4}} \cdot \overline{\text{REQ3}} \cdot \overline{\text{REQ2}} \cdot \text{REQ1} \\ \text{GRANT0} &= \overline{\text{REQ7}} \cdot \overline{\text{REQ6}} \cdot \overline{\text{REQ5}} \cdot \overline{\text{REQ4}} \cdot \overline{\text{REQ3}} \cdot \overline{\text{REQ2}} \cdot \overline{\text{REQ1}} \cdot \text{REQ0} \end{aligned}$$

which could be implemented as shown in Figure 7.5. This arrangement could be extended for any number of bus masters and standard logic components are available to provide static priority (for example the SN74278 4-bit cascadable priority component (Texas Instruments, 1984) which also has flip-flops to store requests).

Static priority circuit devices can generally be cascaded to provide further inputs and outputs, as shown in Figure 7.6. In this figure, each priority circuit has an enable input, $\overline{\text{EI}}$, which must be activated to generate any output, and an enable output, $\overline{\text{EO}}$, which is activated when any one of the priority request outputs is active. The $\overline{\text{EI}}$ of the highest priority circuit is permanently activated. When a request is received by a priority circuit, the outputs of the lower priority circuits are disabled.

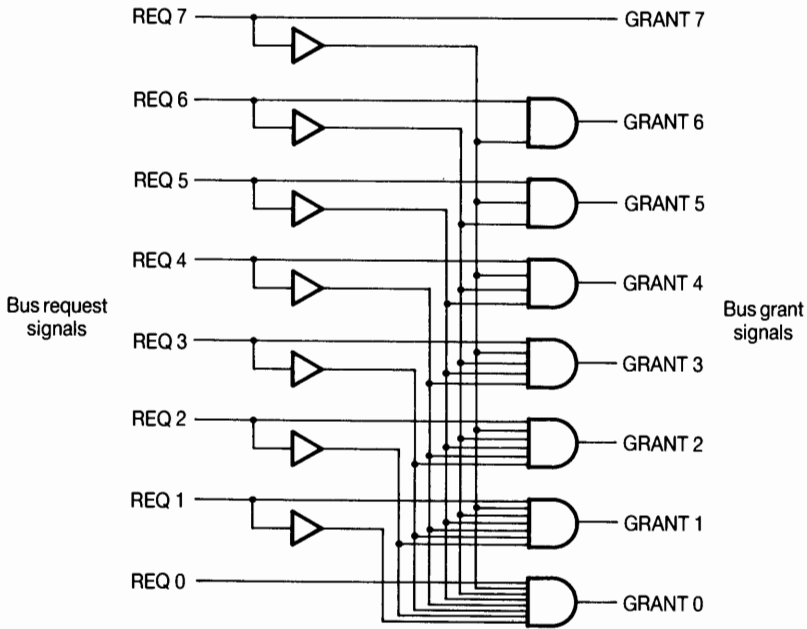


Figure 7.5 Parallel arbitration logic

Hence, after all requests have been applied and sufficient time has elapsed for all logic levels to be established, only the highest priority grant signal will be generated. The previous Boolean equations can easily be modified to incorporate enable signals.

To prevent transient output changes due to simultaneous asynchronous input changes, the request input information can be stored in flip-flops, as in the SN74278. This necessitates a clock input and, as in any synchronizing circuit, there is a finite probability that an asynchronous input change occurs at about the same time as the clock activation and this might cause maloperation.

The speed of operation of cascaded priority arbiters is proportional to the number of circuits cascaded. Clearly, the method is unsuitable for a large number of requests. To improve the speed of operation of systems with more than one arbiter, two-level, parallel bus arbitration can be used, as shown in Figure 7.7. Groups of requests are resolved by a first level of arbiters and a second level arbiter selects the highest priority first level arbiter. For larger systems, the arrangement can be extended to further levels.

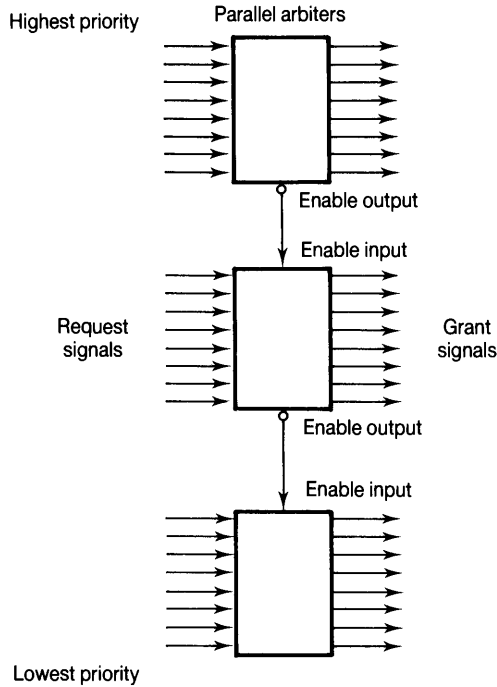


Figure 7.6 Cascaded arbiters

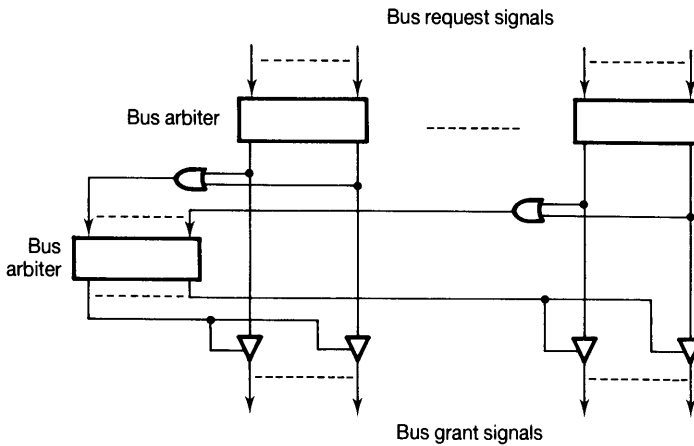


Figure 7.7 Two-level parallel bus arbitration

Microprocessor example

The Motorola MC68452 Bus Arbitration Module (BAM) (Motorola, 1985a) is an example of a microprocessor arbitration device designed to interface to the MC68000 microprocessor. The MC68452 BAM can accept up to eight device bus requests $\overline{DBR0}$, $\overline{DBR1}$, $\overline{DBR2}$, $\overline{DBR3}$, $\overline{DBR4}$, $\overline{DBR5}$, $\overline{DBR6}$ and $\overline{DBR7}$ and has eight corresponding device bus grant outputs $\overline{DBG0}$, $\overline{DBG1}$, $\overline{DBG2}$, $\overline{DBG3}$, $\overline{DBG4}$, $\overline{DBG5}$, $\overline{DBG6}$ and $\overline{DBG7}$ generated according to a static priority ($\overline{DBR7}$ is the highest priority, through to $\overline{DBR0}$, the lowest priority). The BAM generates a bus request signal, \overline{BR} , indicating that it has received one or more requests according to the Boolean AND function $\overline{BR} = \overline{DBG0} \cdot \overline{DBG1} \cdot \overline{DBG2} \cdot \overline{DBG3} \cdot \overline{DBG4} \cdot \overline{DBG5} \cdot \overline{DBG6} \cdot \overline{DBG7}$. The \overline{BG} input enables the \overline{DBG} outputs.

An asynchronous three signal handshake system is used for the transfer of bus control. This consists of a bus request signal, \overline{DBRn} , a bus grant signal, \overline{DBGn} , and a bus grant acknowledge signal, \overline{BGACK} . This three signal handshaking system matches the general bus operation of the MC68000. The timing of the signals is shown in Figure 7.8. When one or more bus requests is received and the grant outputs are enabled, the BAM generates a bus grant signal corresponding to the highest priority bus request input. The bus request signal is returned to the requesting bus master, which must then acknowledge receipt of the signal by activating the common bus grant acknowledge signal. The requesting bus master can then take over the bus immediately. While the bus master is using the bus, it must maintain the acknowledgement, \overline{BGACK} low, and return \overline{BGACK} high when it has finished with the bus. The request, \overline{DBRn} , must be returned high before \overline{BGACK} . The requesting bus masters must maintain their requests low until an acknowledgement is received.

The MC68000 does not generate a bus request signal directly at its pin-out; external processor bus request circuitry is necessary to produce this signal, which is dependent upon the system configuration. A bus request signal needs to be

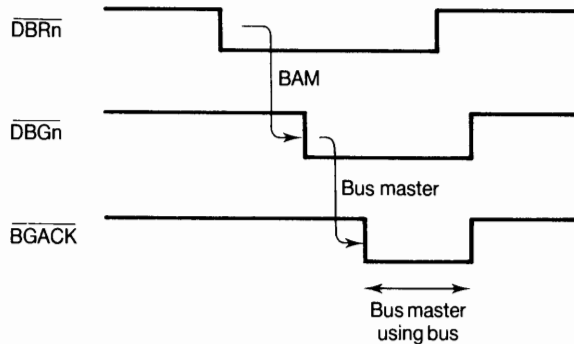


Figure 7.8 MC68000 request/grant/acknowledge sequence

generated for every bus transaction. If there is a local bus, the logic needs to incorporate a bus address decoder. Specific interface parts are available to interface to the VME bus (MC68172/3) and for arbitration (MC68174).

A bus master can use the bus as long as it wishes, which may be for one transaction, or for several, upon condition that it maintains \overline{BGACK} low throughout the transaction(s). There is no mechanism built into the BAM for forcing masters off the bus though a bus clear signal ($BCLR$) is generated whenever a higher priority bus master makes a request for the bus, and this signal could be used with additional circuitry. Also $BGACK$ must be generated by circuitry in addition to the BAM.

The BAM can operate as an arbiter for a system with a central processor and devices which can control the bus temporarily, such as DMA devices and display controllers, or in a multiprocessor system where the control of the bus is not returned to one particular processor. Figure 7.9 shows how the BAM can be used in a single processor system containing other devices which can temporarily control the bus. In this application, \overline{BR} is not connected to BG . Whenever any device connected to the BAM makes a request for the bus, the processor is informed via the \overline{BR} signal. Normally the MC68000 processor will relinquish the bus between 1.5 and 3.5 cycles after the request has reached it, and then return a bus grant signal to the BAM. The BAM then passes a grant signal to the highest priority requesting device.

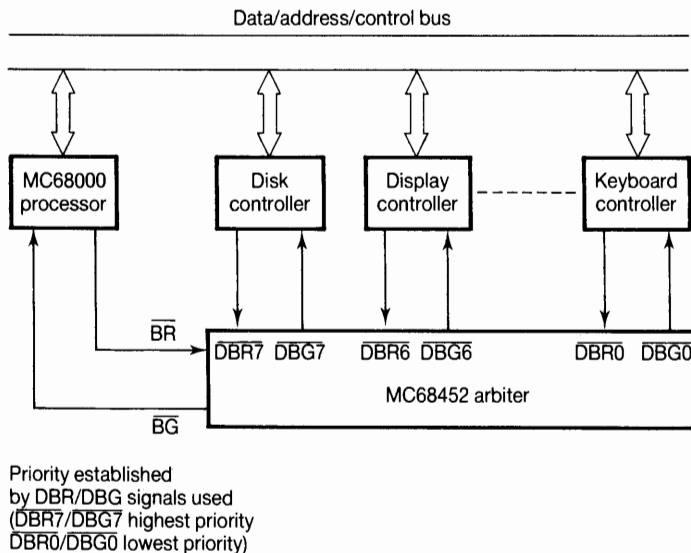


Figure 7.9 Using an MC68452 arbiter in a single processor system

Decentralized parallel priority scheme

In the decentralized parallel priority scheme, one arbiter is used at each processor site, as shown in Figure 7.10, to produce the grant signal for that processor, rather than a single arbiter producing all grant signals. All the request signals need to pass along the bus to all the arbiters, but individual processor grant signals do not need to pass to other processors. Each processor will use a different arbiter request input and corresponding arbiter grant output. An implementation might use wire links for the output of a standard arbiter part, as shown in Figure 7.10. Alternatively, the arbiter function could be implemented from the basic Boolean equations given earlier for parallel priority logic (see page 219), as shown in Figure 7.11. In this case, the total arbitration logic of the system would be the same as the centralized parallel priority scheme.

The decentralized parallel priority scheme is potentially more reliable than the centralized parallel priority scheme, as a failure of one arbiter should only affect the associated processor. An additional mechanism would be necessary to identify faulty arbiters (or processors), perhaps using a time-out signal. However, certain arbiter and processor faults could affect the complete system. For example, if an arbiter erroneously produced a grant signal which was not associated with the highest priority request, the processor would attempt to control the bus, perhaps at the same time as another processor. This particular fault could also occur on a centralized system.

An advantage of the scheme is that it requires fewer signals on the bus. It does not require grant signals on the bus. Also, in a multiboard system with individual processors on separate boards, a special arbiter board is not necessary. All processor boards can use the same design.

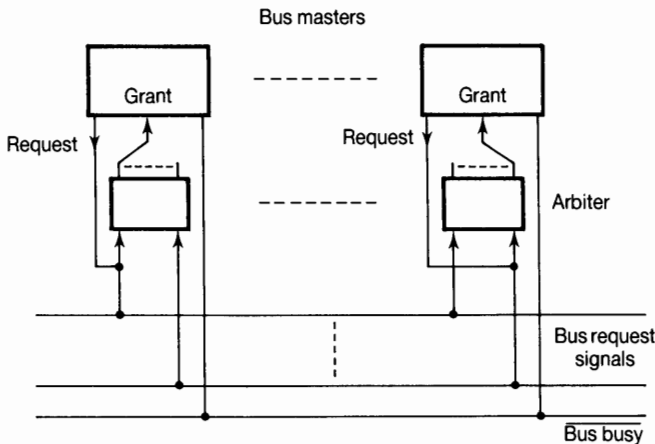


Figure 7.10 Decentralized parallel arbitration

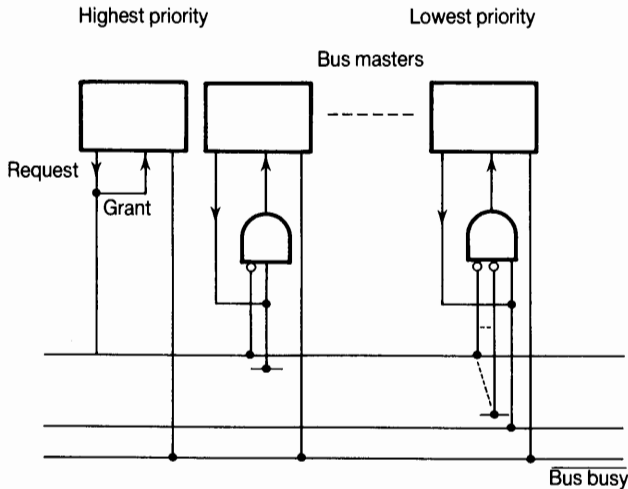


Figure 7.11 Decentralized parallel arbitration using gates

Dynamic priority in parallel priority schemes

The implementation of the parallel priority schemes so far described assigns a fixed priority to individual bus masters. More complex logic, which assigns different priorities depending upon conditions present in the system, can be provided at the arbitration sites. The general aim is to obtain more equitable use of the bus, especially for systems in which no single processor should dominate the use of the bus. Various algorithms can be identified, notably:

1. Simple rotating priority.
2. Acceptance-dependent rotating priority.
3. Random priority.
4. Equal priority.
5. Least recently used (LRU) algorithm.

After each arbitration cycle in *simple rotating priority*, all priority levels are reduced one place, with the lowest priority processor taking the highest priority. In *acceptance-dependent rotating priority* (usually called rotating priority), the processor whose request has just been accepted takes on the lowest priority and the others take on linearly increasing priority. Both forms of rotating policies give all processors a chance of having their request accepted, though the request-dependent rotating policy is most common. In *random priority*, after each arbitration cycle, the priority levels are distributed in a random order, say by a pseudorandom number generator. In *equal priority*, when two or more requests are made to the arbiter,

there is an equal chance of any one request being accepted. Equal priority is applicable to asynchronous systems in which requests are processed by the arbiter as soon as they are generated by processors operating independently. If two or more requests occur simultaneously, the arbiter circuit resolves the conflict. In the *least recently used algorithm*, the highest priority is given to the bus master which has not used the bus for the longest time. This algorithm could also be implemented in logic.

In the (acceptance-dependent) rotating priority algorithm, all possible requests can be thought of as sequential entries in a circular list, as shown in Figure 7.12, for a sixteen bus master system. A pointer indicates the last request accepted. The bus master associated with this request becomes of the lowest priority after being serviced. The next entry has the highest priority and subsequent requests in the list are of decreasing priority. Hence, once a request has been accepted, all other requests become of greater priority. When further requests are received, the highest priority request is accepted, the pointer adjusted to this request and a further request

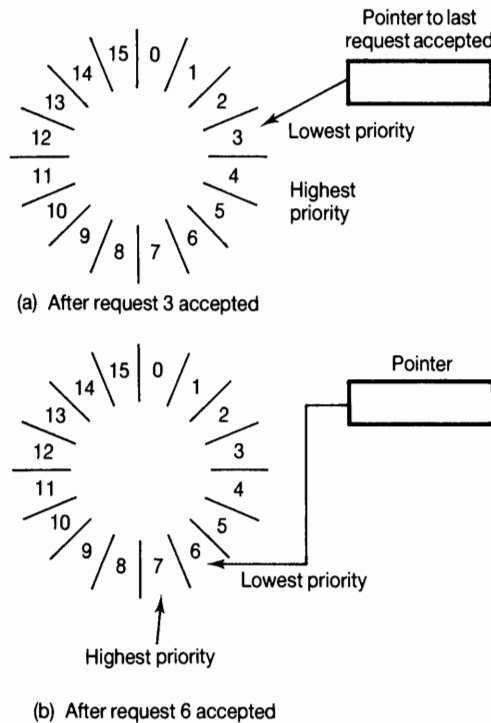


Figure 7.12 Rotating priority algorithm (a) After request 3 accepted
(b) After request 6 accepted

from this master becomes the lowest priority request. For example, the list shown in Figure 7.12(a) shows the allocation of sixteen devices after request 3 has been received and is serviced. In Figure 7.12(b) request number 6 has been received and the pointer is moved accordingly.

Rotating priority has been used in interrupt controllers, for example the Advanced Micro Devices Am9519, and in many ways the interrupt mechanism is similar to the bus control mechanism but uses interrupt request and acknowledge/grant signals rather than bus request and acknowledge/grant signals. Various features in the Am9519 device can be preprogrammed, including a fixed priority or rotating priority and particular responses to interrupts. Features such as mask registers to lock out specific requests are not normally found in bus arbitration systems. Rotating priority can also be performed in the serial priority scheme (see Section 7.2.2).

There are some schemes which assign priority according to some fixed strategy; these schemes are not strictly dynamic, in so far as the assignment does not necessarily change after each request is serviced. We can identify two such algorithms:

1. Queueing (first-come first-served) algorithm.
2. Fixed time slice algorithm.

The *queueing (first-come first-served) algorithm* is sometimes used in analytical studies of bus contention and assumes a queue of requests at the beginning of an arbitration cycle. The request accepted is the first request in the queue, i.e. the first request received. This algorithm poses problems in implementation and is not normally found in microprocessor systems. In the *fixed time slice algorithm*, each bus master is allocated one period in a bus arbitration sequence. Each bus master can only use the bus during its allocated period, and need not use the bus on every occasion. This scheme is suitable for systems in which the bus transfers are synchronized with a clock signal.

7.2.2 Serial priority schemes

The characteristic feature of serial priority schemes is the use of a signal which passes from one bus master to another, in the form of a *daisy chain*, to establish whether a request has the highest priority and hence can be accepted. There are three general types, depending upon the signal which is daisy chained:

1. Daisy chained grant signal.
2. Daisy chained request signal.
3. Daisy chained enable signal.

The *daisy chained grant scheme* is the most common. In this scheme the bus requests from bus masters pass along a common (wired-OR) request line, as shown

in Figure 7.13. A bus busy signal is also common and, when active, indicates that the bus is being used by a bus master. When one or more bus masters make a request, the requests are routed to the beginning of the daisy chain, sometimes through a special bus controller and sometimes by direct connection to the highest priority master. The signal is then passed from one bus master to the next until the highest priority requesting bus master is found. This bus master prevents the signal passing any further along the daisy chain and prepares to take over the bus.

In the *daisy chained request scheme*, as shown in Figure 7.14, the daisy chain connection is again from the highest priority bus master through to the lowest priority bus master, but with the request signal being routed along the daisy chain. Each requesting bus master generates a bus request signal which is passed along the daisy chain, eventually reaching the current bus master. This bus master is of lower priority than any of the requesting bus masters to the left of it, and hence will honor the request by generating a common (wired-OR) bus acknowledge/grant signal. All requesting bus masters notice this signal but only the one which has a request pending and does not have a request present at its daisy chain input responds, as it

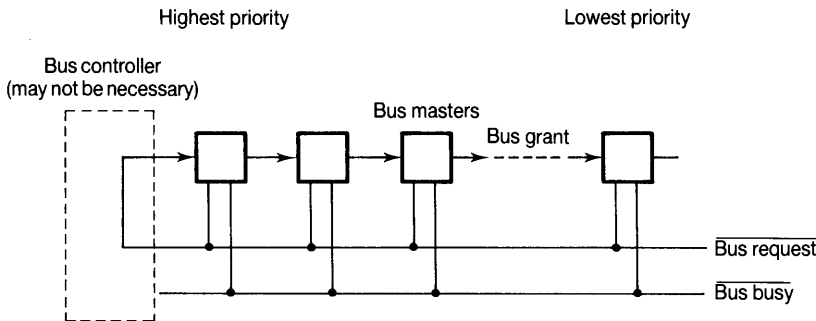


Figure 7.13 Centralized serial priority arbitration with daisy-chained grant signal

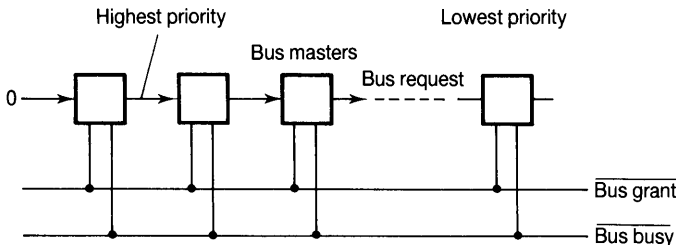


Figure 7.14 Centralized serial priority arbitration with daisy-chained request signal

must be the highest priority requesting bus master. Other requesting bus masters have an opportunity to compete for the bus in future bus arbitration cycles. The 8086 microprocessor supports a form of daisy-chained request arbitration.

In the *daisy chained enable scheme*, both the bus request and bus acknowledge/grant signals are common (wired-OR) signals and an additional enable signal is daisy chained. When a bus master makes a request it disables the daisy chained enable output, indicating to lower priority bus masters that a higher priority bus master has presented a request. The common request signal is routed to a bus controller, which generates a common (wired-OR) bus acknowledge signal to all bus masters. The highest priority requesting bus master will have its enable input activated and this condition will allow it to take over the bus. The daisy chained enable system was used in single processor Z-80 systems for organizing interrupts from input/output interfaces.

In all types of daisy chain schemes, a key point is that the mechanism must be such that a requesting bus master cannot take over the bus until it has been freed. A bus controller can be designed to issue an acknowledge/grant signal only when the bus is free. If there is no bus controller, there are two possible mechanisms, namely:

1. Bus masters are not allowed to make a request until the bus is free.
2. Bus masters are allowed to make a request at any time but are not allowed to take over the bus until the bus is free (and after receipt of a grant signal).

In 1, after the grant signal comes via the daisy chain, the bus master can take over the bus immediately. In 2, the bus master must wait until the bus is free. When the bus is taken over, the bus busy line is activated.

A strategy must be devised for terminating the control of the bus. One strategy would be to allow a bus master only one bus cycle and to make it compete with other bus masters for subsequent bus cycles. Alternatively, bus masters could be forced off the bus by higher priority requests (and perhaps requested to terminate by lower priority bus masters).

MC68000 microprocessor

The MC68000 microprocessor is particularly designed to use the daisy-chained acknowledge scheme with its three processor signals bus request input (\overline{BR}), bus grant output (\overline{BG}) and bus grant acknowledge input (\overline{BGACK}). The bus grant acknowledge signal is, in effect, a bus busy signal and is activated when a bus master has control of the bus. External circuitry is necessary to generate this signal for each bus master. Bus request indicates that at least one bus master is making a request for the bus. Again, external circuitry is necessary to generate this signal for each bus master. The bus grant signal, \overline{BG} , is generated by the processor and indicates that it will release the bus at the end of the current bus cycle in response to receiving the \overline{BR} signal. The requesting processor waits for all of the following conditions to be satisfied (Motorola, 1985b):

230 Shared memory multiprocessor systems

1. The bus grant, \overline{BG} , has been received.
2. The address strobe, \overline{AS} , is inactive indicating that the processor is not using the bus.
3. The data transfer acknowledge signal, \overline{DTACK} , is inactive indicating that neither memory or peripherals are using the bus.
4. The bus grant acknowledge signal, \overline{BGACK} , is inactive indicating that no other device still has control over the bus.

The scheme described allows masters to make requests even if the bus is being used, but the transfer of control is inhibited until the bus becomes free. Hence the arbitration cycle can be overlapped with the current bus cycle. In contrast, bus requests in a Z8000 multiprocessor are inhibited until the bus is free, when arbitration takes place to find the highest priority requesting bus master.

Decentralized serial priority scheme

Though the daisy chain distributes the arbitration among the bus master sites, the daisy chain signal originates at one point and subsequently passes along the daisy chain. Hence the daisy chain methods so far described are categorized as centralized priority schemes. The daisy chain grant method can be modified to be a decentralized scheme by making the current bus master generate the daisy chain grant signal and arranging a circular connection, as shown in Figure 7.15. The daisy chain signal now originates at different points each time control is transferred from one bus master to another, which leads to a rotating priority. The current bus master has the lowest priority for next bus arbitration. The bus master immediately to the right of the current bus master has the highest priority and bus masters further along the daisy chain have decreasing priority.

When a bus master has control of the bus, it generates a grant signal which is passed to the adjacent bus master. The signal is passed through bus masters that do not have a request pending. Whenever a bus master makes a request, and has a grant

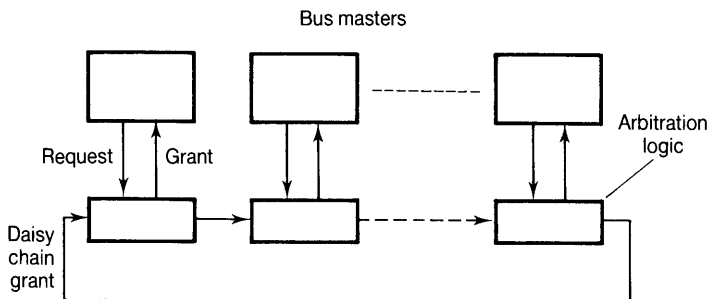


Figure 7.15 Rotating daisy chain

input signal active, it inhibits the grant signal from continuing along the daisy chain. However, it cannot take over the bus until the current bus master releases the bus (assuming a bus master is using the bus). When the current bus master finds that it has lost its daisy chained grant, it must release the bus at the earliest opportunity and release a common bus busy line. Then the requesting master can take over the bus. When more than one bus master makes a request for the bus, the requesting bus master nearest the current bus master in the clockwise direction is first to inhibit the daisy chain grant signal and claim the bus.

An implementation of the *rotating daisy chain* scheme typically requires one flip-flop at each bus master to indicate that it was the last to use the bus or that it is currently using the bus. One design is given by Nelson and Refai (1984). Flip-flops are usually activated by a centralized clock signal, and request signals should not change at about the time of the activating clock transition or the circuit might enter a metastable state for a short period (with an output voltage not at a binary level).

Finally, note that though the scheme is decentralized, it still suffers from single point failures. If one of the arbitration circuits fails to pass on the grant signal, the complete system will eventually fail as the daisy chain signal propagates to the failed circuit.

Combined serial–parallel scheme

The serial priority scheme is physically easy to expand though the speed of operation is reduced as the daisy chain length increases. The parallel priority scheme is faster but requires extra bus lines. The parallel scheme cannot be expanded easily in a parallel fashion beyond the original design since it is dependent upon the number of lines available on the bus for request and acknowledge/grant signals, and the arbitration logic. Typically eight or sixteen bus masters can be handled with a parallel priority scheme.

The parallel priority scheme can be expanded by daisy chaining each request or grant signal, thus combining the serial and parallel techniques. A scheme is shown in Figure 7.16. Here the bus request signals pass to the parallel arbitration circuit as before. However, these signals are wired-OR types and several bus masters may use each line. The grant signals are daisy chained for each master using the same request line, so that the requesting master can be selected. The operation is as follows: the requesting master produces a bus request signal. If accepted by the priority logic, the corresponding grant signal is generated. This signal passes down the daisy chain until it reaches the requesting master. At the same time, an additional common *bus clear* signal is generated by the priority logic and sent to all the bus masters. On receiving this signal the current master will release the bus at the earliest possible moment, indicating this by releasing the bus busy signal. The new master will then take over the bus.

The parallel and serial schemes are in fact two extremes of implementing the same Boolean equations for arbitration given in Section 7.2.1. From these equations, we can obtain the equations implemented at each bus master site in a daisy chain grant system. Defining IN_n as the n th daisy chain input and OUT_n as the n th daisy chain output, which are true if no higher priority request is present, then:

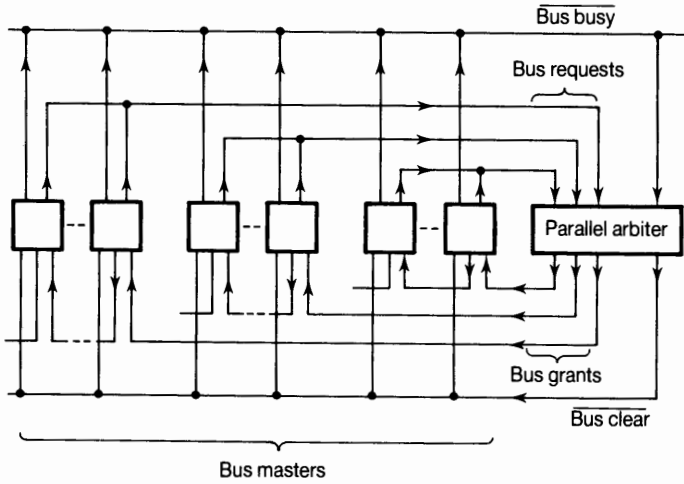


Figure 7.16 Parallel arbiter with daisy chained grant signals

$$\begin{aligned} IN_n &= \overline{OUT_n} \\ OUT_n &= \overline{REQ_n \cdot IN_n} \end{aligned}$$

$$\begin{aligned} GRANT_7 &= REQ_7 \\ OUT_7 &= IN_6 = \overline{REQ_7} \end{aligned}$$

$$\begin{aligned} GRANT_6 &= \overline{REQ_7} \cdot REQ_6 = \overline{IN_6} \cdot REQ_6 \\ OUT_6 &= IN_5 = \overline{REQ_7} \cdot REQ_6 = \overline{IN_6} \cdot REQ_6 \end{aligned}$$

$$\begin{aligned} GRANT_5 &= \overline{REQ_7} \cdot \overline{REQ_6} \cdot REQ_5 = \overline{IN_5} \cdot REQ_5 \\ OUT_5 &= IN_4 = \overline{REQ_7} \cdot \overline{REQ_6} \cdot REQ_5 = \overline{IN_5} \cdot REQ_5 \end{aligned}$$

$$\begin{aligned} GRANT_4 &= \overline{REQ_7} \cdot \overline{REQ_6} \cdot \overline{REQ_5} \cdot REQ_4 = \overline{IN_4} \cdot REQ_4 \\ OUT_4 &= IN_3 = \overline{REQ_7} \cdot \overline{REQ_6} \cdot \overline{REQ_5} \cdot REQ_4 = \overline{IN_4} \cdot REQ_4 \end{aligned}$$

$$\begin{aligned} GRANT_3 &= \overline{REQ_7} \cdot \overline{REQ_6} \cdot \overline{REQ_5} \cdot \overline{REQ_4} \cdot REQ_3 = \overline{IN_3} \cdot REQ_3 \\ OUT_3 &= IN_2 = \overline{REQ_7} \cdot \overline{REQ_6} \cdot \overline{REQ_5} \cdot \overline{REQ_4} \cdot REQ_3 = \overline{IN_3} \cdot REQ_3 \end{aligned}$$

$$\begin{aligned} GRANT_2 &= \overline{REQ_7} \cdot \overline{REQ_6} \cdot \overline{REQ_5} \cdot \overline{REQ_4} \cdot \overline{REQ_3} \cdot REQ_2 = \overline{IN_2} \cdot REQ_2 \\ OUT_2 &= IN_1 = \overline{REQ_7} \cdot \overline{REQ_6} \cdot \overline{REQ_5} \cdot \overline{REQ_4} \cdot \overline{REQ_3} \cdot REQ_2 = \overline{IN_2} \cdot REQ_2 \end{aligned}$$

$$GRANT_1 = \overline{REQ_7} \cdot \overline{REQ_6} \cdot \overline{REQ_5} \cdot \overline{REQ_4} \cdot \overline{REQ_3} \cdot \overline{REQ_2} \cdot REQ_1 = \overline{IN_1} \cdot REQ_1$$

$$\text{OUT1} = \text{IN0} = \overline{\text{REQ7}}.\overline{\text{REQ6}}.\overline{\text{REQ5}}.\overline{\text{REQ4}}.\overline{\text{REQ3}}.\overline{\text{REQ2}}.\overline{\text{REQ1}} = \text{IN1}.\overline{\text{REQ1}}$$

$$\text{GRANT0} = \overline{\text{REQ7}}.\overline{\text{REQ6}}.\overline{\text{REQ5}}.\overline{\text{REQ4}}.\overline{\text{REQ3}}.\overline{\text{REQ2}}.\overline{\text{REQ1}}.\text{REQ0} = \text{IN0}.\text{REQ0}$$

Alternatively, we could have grouped two grant circuits together to get:

$$\text{GRANT7} = \text{REQ7}$$

$$\text{GRANT6} = \overline{\text{REQ7}}.\text{REQ6}$$

$$\text{OUT7/6} = \text{IN5/4} = \overline{\text{REQ7}}.\overline{\text{REQ6}}$$

$$\text{GRANT5} = \overline{\text{REQ7}}.\overline{\text{REQ6}}.\text{REQ5} = \text{IN5/6}.\text{REQ5}$$

$$\text{GRANT4} = \overline{\text{REQ7}}.\overline{\text{REQ6}}.\overline{\text{REQ5}}.\text{REQ4} = \text{IN5/6}.\overline{\text{REQ5}}.\text{REQ4}$$

$$\text{OUT5/4} = \text{IN3/2} = \overline{\text{REQ7}}.\overline{\text{REQ6}}.\overline{\text{REQ5}}.\overline{\text{REQ4}} = \text{IN5/6}.\overline{\text{REQ5}}.\overline{\text{REQ4}}$$

$$\text{GRANT3} = \overline{\text{REQ7}}.\overline{\text{REQ6}}.\overline{\text{REQ5}}.\overline{\text{REQ4}}.\text{REQ3} = \text{IN3/2}.\text{REQ3}$$

$$\text{GRANT2} = \overline{\text{REQ7}}.\overline{\text{REQ6}}.\overline{\text{REQ5}}.\overline{\text{REQ4}}.\overline{\text{REQ3}}.\text{REQ2} = \text{IN3/2}.\overline{\text{REQ3}}.\text{REQ2}$$

$$\text{OUT3/2} = \text{IN1/0} = \overline{\text{REQ7}}.\overline{\text{REQ6}}.\overline{\text{REQ5}}.\overline{\text{REQ4}}.\overline{\text{REQ3}}.\overline{\text{REQ2}}$$

$$= \text{IN3/2}.\overline{\text{REQ3}}.\overline{\text{REQ2}}$$

$$\text{GRANT1} = \overline{\text{REQ7}}.\overline{\text{REQ6}}.\overline{\text{REQ5}}.\overline{\text{REQ4}}.\overline{\text{REQ3}}.\overline{\text{REQ2}}.\text{REQ1} = \text{IN1/0}.\text{REQ1}$$

$$\text{GRANT0} = \overline{\text{REQ7}}.\overline{\text{REQ6}}.\overline{\text{REQ5}}.\overline{\text{REQ4}}.\overline{\text{REQ3}}.\overline{\text{REQ2}}.\overline{\text{REQ1}}.\text{REQ0}$$

$$= \text{IN1/0}.\overline{\text{REQ1}}.\text{REQ0}$$

Similarly, groups of four arbitration circuits can be created with a daisy chain signal between them, i.e.:

$$\text{GRANT7} = \text{REQ7}$$

$$\text{GRANT6} = \overline{\text{REQ7}}.\text{REQ6}$$

$$\text{GRANT5} = \overline{\text{REQ7}}.\overline{\text{REQ6}}.\text{REQ5}$$

$$\text{GRANT4} = \overline{\text{REQ7}}.\overline{\text{REQ6}}.\overline{\text{REQ5}}.\text{REQ4}$$

$$\text{OUT7-4} = \text{IN3 0} = \overline{\text{REQ7}}.\overline{\text{REQ6}}.\overline{\text{REQ5}}.\overline{\text{REQ4}}$$

$$\text{GRANT3} = \overline{\text{REQ7}}.\overline{\text{REQ6}}.\overline{\text{REQ5}}.\overline{\text{REQ4}}.\text{REQ3} = \text{IN3 0}.\text{REQ3}$$

$$\text{GRANT2} = \overline{\text{REQ7}}.\overline{\text{REQ6}}.\overline{\text{REQ5}}.\overline{\text{REQ4}}.\overline{\text{REQ3}}.\text{REQ2} = \text{IN3 0}.\overline{\text{REQ3}}.\text{REQ2}$$

$$\text{GRANT1} = \overline{\text{REQ7}}.\overline{\text{REQ6}}.\overline{\text{REQ5}}.\overline{\text{REQ4}}.\overline{\text{REQ3}}.\overline{\text{REQ2}}.\text{REQ1}$$

$$= \text{IN3-0}.\overline{\text{REQ3}}.\overline{\text{REQ2}}.\text{REQ1}$$

$$\text{GRANT0} = \overline{\text{REQ7}}.\overline{\text{REQ6}}.\overline{\text{REQ5}}.\overline{\text{REQ4}}.\overline{\text{REQ3}}.\overline{\text{REQ2}}.\overline{\text{REQ1}}.\text{REQ0}$$

$$= \text{IN3 0}.\overline{\text{REQ3}}.\overline{\text{REQ2}}.\overline{\text{REQ1}}.\text{REQ0}$$

Figure 7.17 shows implementations for a purely serial approach, arbiters with groups of two request inputs and arbiters with groups of four request inputs.

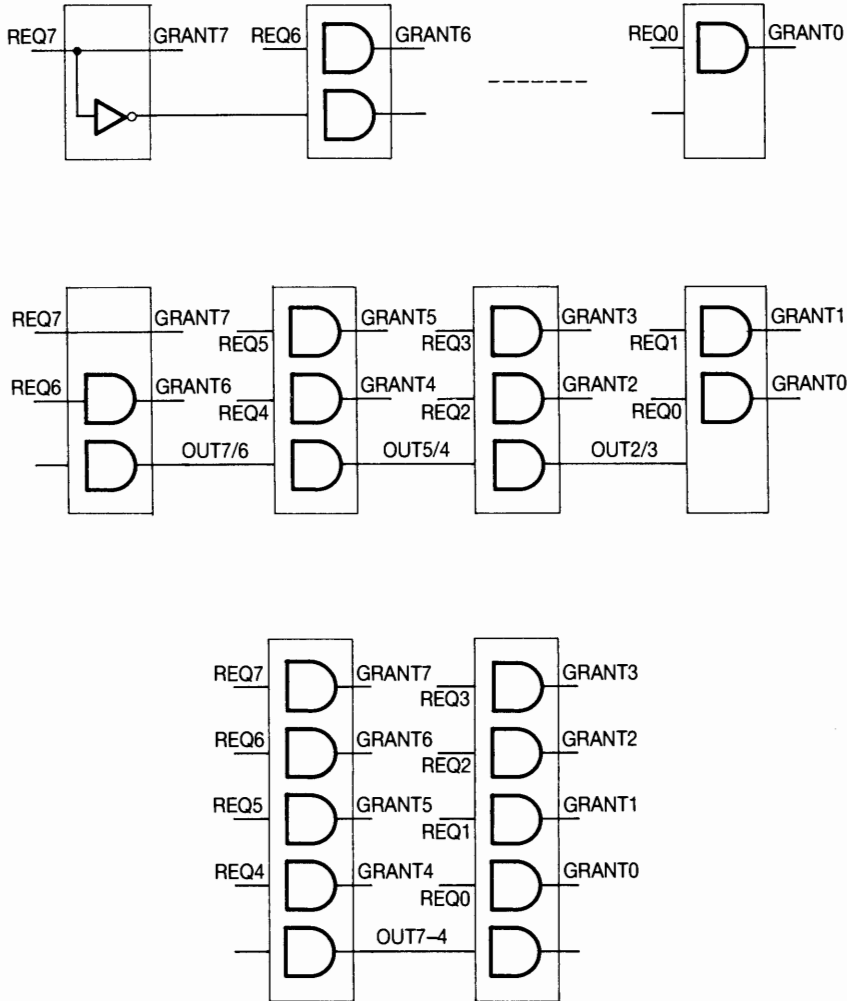


Figure 7.17 Serial-parallel arbitration logic (a) Serial (b) Groups of two requests (c) Groups of four requests

7.2.3 Additional mechanisms in serial and parallel priority schemes

Apart from the basic request, grant/acknowledge and bus busy signals, additional bus signals may be present in arbitration schemes. For example, there may be two

types of “bus request/clear” signals, one signal as a top priority clear, causing the current bus master to release the bus at the earliest possible moment, and one signal indicating a request which does not need to be honored so quickly. With the second request, the current bus master may complete a sequence of tasks, for example complete a DMA (direct memory access) block transfer. The top priority clear might be used for power down routines.

As described, the serial and parallel priority schemes with static priority will generally prevent lower priority bus masters obtaining control of the bus while higher priority bus masters have control. Consequently, it may be that these lower priority masters may never obtain control of the bus. This can be avoided by using a *common bus request* signal which is always activated whenever a bus request is made. If the requesting master has a higher priority than the current master, the normal arbitration logic will resolve the conflict and generate a bus request signal to the current bus master, causing the master to relinquish control of the bus at the end of the current cycle. If, however, the requesting master is of lower priority than the current bus master, a signal is not generated by the priority logic, but the bus master recognizes and takes note of the fact that the common bus request signal is activated. The current bus master continues but when it does not require the bus, perhaps while executing an internal operation, it releases the bus busy signal, thus allowing the requesting master access to the bus until the current bus master requires the bus again.

This scheme is particularly suitable if the master has an internal instruction queue with instruction prefetch, so that after the queue is full there may be long periods during which the bus is not required. Note that while the common bus request signal is not activated by a requesting bus master, the current bus master might not release the bus signal between bus transfers. The Intel 16-bit Multibus I system bus (the IEEE 796 bus) uses the common bus request mechanism with the signal $\overline{\text{CBRQ}}$ (Intel, 1979), though the actual microprocessors (e.g. 8086, 80286) do not generate the common bus signal.

7.2.4 Polling schemes

In polling schemes, rather than the bus masters issuing requests whenever they wish to take over control of the bus and a bus controller deciding which request to accept, a bus controller asks bus masters whether they have a request pending. Such polling schemes can be centralized, with one bus controller issuing request inquiries, or decentralized, with several bus controllers.

The mechanism generally uses special polling lines between the bus controller(s) and the bus masters to effect the inquiry. Given 2^n bus masters, 2^n lines could be provided, one for each bus master, and one activated at a time to inquire whether the bus master has a request pending. Alternatively, to reduce the number of polling lines, a binary encoded polling address could be issued on the polling lines and then only n lines would be necessary. In addition, there are bus request and busy lines.

Centralized polling schemes

A centralized polling scheme is shown in Figure 7.18. The bus controller asks each bus master in turn whether it has a bus request pending, by issuing each bus master polling address on the n polling lines. If a bus master has a request pending when its address is issued, it responds by activating the common request line. The controller then allows the bus master to use the bus. The bus master addresses can be sequenced in numerical order or according to a dynamic priority algorithm. The former is easy to implement using a binary counter which is temporarily halted from sequencing by the bus busy line.

Decentralized polling schemes

A decentralized polling scheme is shown in Figure 7.19. Each bus master has a bus controller consisting of an address decoder and an address generator. First, at the beginning of the polling sequence, an address is generated which is recognized by a controller. If the associated processor has a request outstanding, it may now use the bus. On completion, the bus controller generates the address of the next processor in the sequence and the process is repeated. It is usually necessary to have a handshaking system, as shown in Figure 7.19, consisting of a request signal generated by the address generator and an acknowledge signal generated by the address decoder.

The decentralized polling scheme, as described, is not resilient to single point failures, i.e. if a bus controller fails to provide the next polling address, the whole system fails. However, a time-out mechanism could be incorporated such that if a bus controller fails to respond in a given time period, the next bus controller takes over.

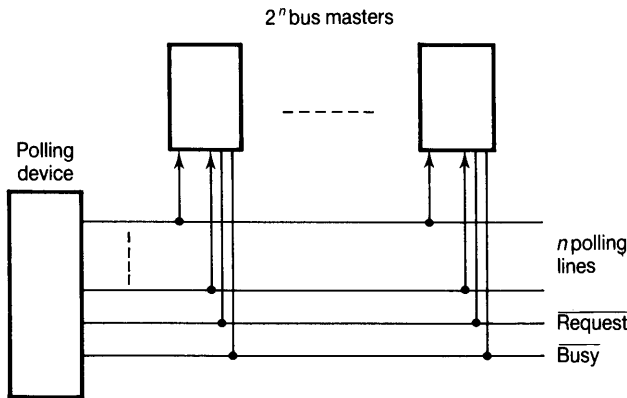


Figure 7.18 Centralized polling scheme

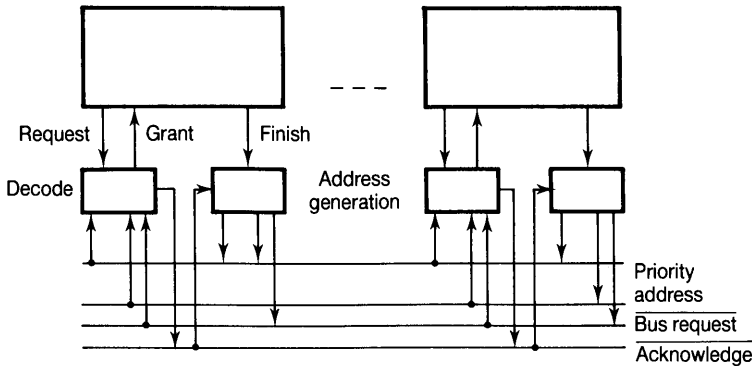


Figure 7.19 Decentralized polling scheme

Software polling schemes

Although all the priority schemes presented are implemented in hardware to obtain high speed of operation, the polling schemes lend themselves to a software approach. The arbitration algorithms could be implemented in software, using processor-based bus controllers if the speed of operation is sufficient. For example, the bus controller(s) in the polling scheme could store the next polling addresses, and these could be modified if a bus master is taken out of service. A degree of fault tolerance or the ability to reconfigure the system could be built into a polling scheme. For example, each bus master could be designed to respond on a common “I’m here” line when polled. No response could be taken as a fault at the bus master, or a sign that the bus master had been removed from service. However, such schemes are more appropriate for systems in which the shared bus is used to pass relatively long messages between computers, or message-passing systems.

7.3 Performance analysis

In this section we will present an analysis of the single bus system and the arbitration function. The methods will be continued in Chapter 8 for multiple bus and other interconnection networks.

7.3.1 Bandwidth and execution time

Suppose requests for memory are generated randomly and the probability that a processor makes a request for memory is r . The probability that the processor does

not make a request is given by $1 - r$. The probability that no processors make a request for memory is given by $(1 - r)^p$ where there are p processors. The probability that one or more processors make a request is given by $1 - (1 - r)^p$. Since only one request can be accepted at a time in a single bus system, the average number of requests accepted in each arbitration cycle (the bandwidth, BW) is given by:

$$BW = 1 - (1 - r)^p$$

which is plotted in Figure 7.20. We see that at a high request rate, the bus soon saturates.

If a request is not accepted, it would normally be resubmitted until satisfied, and the request rate, r , would increase to an *adjusted request rate*, say a . Figure 7.21 shows the execution time, T , of one processor with all requests accepted, and the execution time, T' , with some requests blocked and resubmitted on subsequent cycles (Yen *et al.*, 1982). Since the number of cycles without requests is the same in both cases, we have:

$$T'(1 - a) = T(1 - r)$$

Let P_a be the probability that a request will be accepted with the adjusted request rate, a , and BW_a be the bandwidth. With a fair arbitration policy, each request will have the same probability of being accepted and hence P_a is given by:

$$P_a = \frac{BW_a}{pa} = \frac{1 - (1 - a)^p}{pa}$$

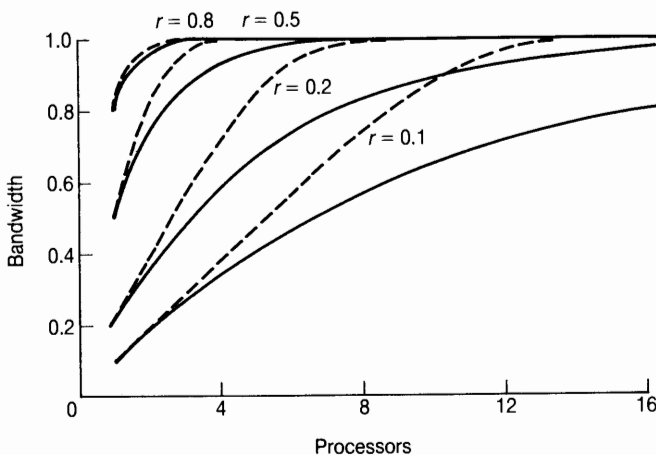


Figure 7.20 Bandwidth of a single bus system (--- using rate adjusted equations)

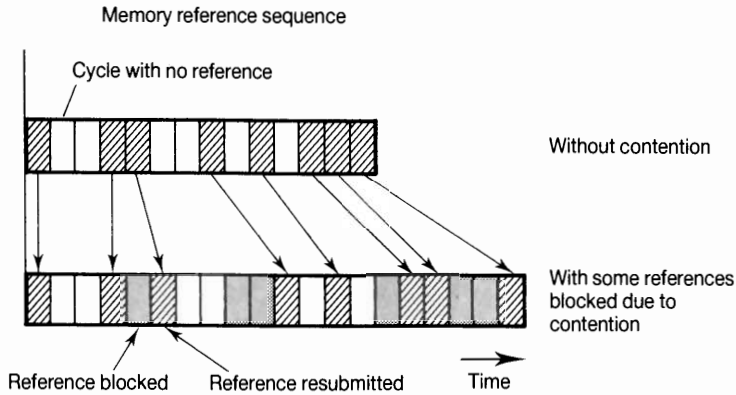


Figure 7.21 Memory references without contention and with contention

The number of requests before a request is accepted (including the accepted request) = $1/P_a$. Hence, we have:

$$T' = ((1 - r) + r/P_a)T$$

and then:

$$a = \frac{1}{1 + P_a(1/r - 1)}$$

Here the request rate with the presence of resubmissions is given in terms of the original request rate and the acceptance probability at the rate a . The equations for P_a and a can be solved by iteration.

On a single processor system, the execution time would be Tp . If all requests were accepted, the time on the multiprocessor would be T and the speed-up factor would be p . In the presence of bus contention and resubmitted requests, the execution time is T' and the speed-up factor is given by:

$$\text{Speed-up factor} = \frac{Tp}{T'} = \frac{p}{r/P_a + (1 - r)}$$

Figure 7.22 shows the speed-up factor using iteration to compute P_a . We see that the speed-up is almost linear until saturation sets in. The maximum speed-up is given by $1/r$. For example, if $r = 0.1$ (10 per cent) the maximum speed-up is 10, irrespective of the number of processors. With $r = 0.1$ and with ten processors, the speed-up is 9. Note that the derivation uses random requests; in practice the sequence may not be random.

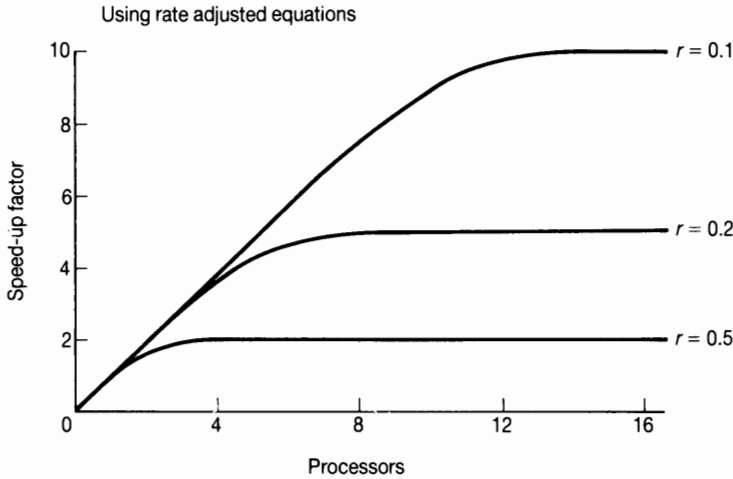


Figure 7.22 Speed-up factor of a single bus system

7.3.2 Access time

From the number of requests before a request is accepted being given by $1/P_i$, we obtain the time before a rejected request from the i th processor is finally accepted (the *access time*) as:

$$T_i = \frac{(1 - P_i)}{P_i}$$

where P_i is the probability that processor i successfully accesses the bus, that is, the probability that a submitted request is accepted. (An alternative derivation is given in Hwang and Briggs, 1984.) If a request is immediately accepted in the first arbitration cycle (i.e. $P_i = 1$), the access time is zero. The access time is measured in arbitration cycles. The probability that a processor successfully accesses the bus will depend upon the arbitration policy, and the initial request rate, r .

Fair priority

The probability, P_i , for a fair priority giving equal chance to all processors was given by P_a previously, i.e. $P_i = (1 - (1 - r)^p) / pr$ or, if the adjusted rate is used, $P_i = (1 - (1 - a)^p) / pa$. Figure 7.23 shows the access time against number of processors using the adjusted rate equations with iteration.

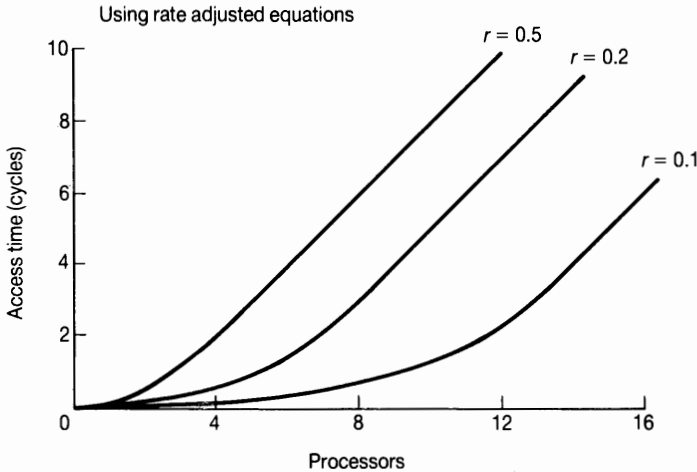


Figure 7.23 Access time of a single bus system

Fixed priority

Ignoring resubmitted requests, the probability, P_i , for fixed priority (e.g. daisy chain arbitration) would seem to be given by:

$$P_i = (1 - r)^{i-1}$$

which is the probability that none of the processors with higher priority than processor i makes a request. The lower processor number corresponds to the higher priority. (Processor i has priority i and processor $i-1$ is the next higher priority processor.) Resubmitted requests have a very significant effect on the computed access time with fixed priority. Unfortunately it is very difficult to incorporate an adjusted request rate into the equations as the probability of an individual processor not making a request is dependent upon other processors. The previous equation is invalid for $a = r$. Computer simulation can be performed to obtain the most accurate results.

7.4 System and local buses

We noted in Section 7.1 that a single bus cannot be used for all processor-memory transactions with more than a few processors and we can see the bus saturation in the previous analysis. The addition of a cache or local memory to each processor would reduce the bus traffic. This idea results in each processor having a local bus

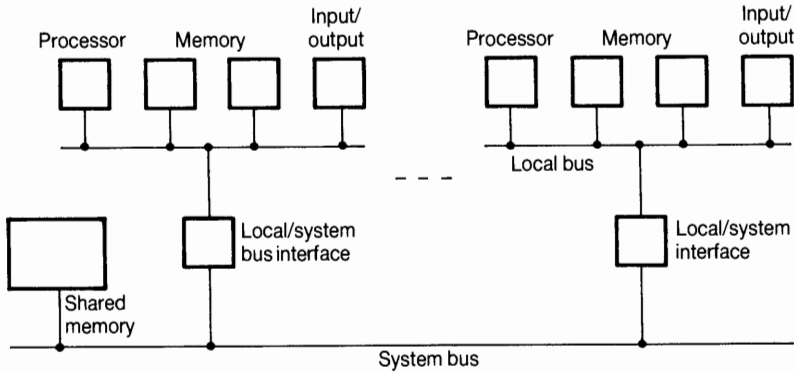


Figure 7.24 Multiple microprocessor system with local buses and system bus

for local memory and input/output, and a system bus for communication between local buses and to a shared memory, as shown in Figure 7.24. Now let us look at this type of architecture in detail. Bus arbitration is still necessary at the system bus level and possibly also at the local bus level.

A local/system bus interface circuitry connects the local and system buses together. Memory addresses are divided into system addresses referring to memory on the system bus, and local memory addresses referring to memory on the local bus. No local bus arbitration is required if there is only one processor on the local bus, but generally system bus arbitration logic is necessary to resolve multiple system bus requests to select one processor to use the system bus. When a processor issues a memory address, decode logic identifies the bus. Input/output addresses could be in local or system space, depending upon the design.

Since blocks of memory locations generally need to be transferred from the system memory to the local memory before being used, it is advantageous to provide a direct path between the system bus and the local memory using two-port memory. Two-port memory can be accessed by one of two buses, sometimes simultaneously. Small two-port memory with simultaneous access characteristics are available, but in any event two-port memory can be created (though without simultaneous access characteristics) using normal random access memory components and memory arbitration logic to select one of potentially two requests for the memory. In effect, the bus arbitration logic is replaced by similar memory arbitration logic. Care needs to be taken to ensure data consistency in the two-port memory using critical section locks (see Chapter 6). Most recent microprocessors have facilities for local and system buses, either built into the processors or contained in the bus controller interfaces.

Example of microprocessor with local and system bus signals

The 8-bit Zilog Z-280 microprocessor (Zilog, 1986) (a substantial enhancement to the Z-80 microprocessor) has the ability to distinguish between local bus addresses and system bus addresses using internal logic. The processor can operate in a multiprocessor configuration or not, by setting a bit in the 8-bit internal processor register called the *Bus Timing and Initialization* register. In the non-multiprocessor mode, only a single bus, the local bus, is supported and the processor is the controlling bus master for this bus. Other processor-like devices, such as DMA devices, must request the use of the bus from the Z-280 using the bus request signal ($\overline{\text{BUSREQ}}$) into the Z-280. The Z-280 acknowledges the request with the bus acknowledge signal ($\overline{\text{BUSACK}}$) and releases the bus by the time the acknowledgement is issued.

In the multiprocessor configuration mode, both local and global buses can be present and memory addresses are separated into those which refer to the local bus and those which refer to the global bus using the four most significant bits of the address. These four most significant bits can be selected as set to 1 or 0 for the local bus using a processor register called the *local address* register. Four base bits in this register are compared to the four address bits and if all four match, a local address reference is made, otherwise a global memory reference is made. The other four bits are mask enable bits to override global selection for each address bit when the corresponding mask bit is set to 0. If all mask bits are set to 0, all memory references are to the local memory. The Z-280 has four on-chip DMA channels, which may use the global bus in the same way as the Z-280 – using the local address register to ascertain whether addresses are local or global.

The local bus is controlled in the same way as in the non-multiprocessor mode, using $\overline{\text{BUSREQ}}$ and $\overline{\text{BUSACK}}$, but the processor must request the global bus. This request is made by issuing a Global Request output ($\overline{\text{GREQ}}$) from the processor, which is acknowledged by the Global Acknowledge input ($\overline{\text{GACK}}$) to the processor. $\overline{\text{GREQ}}$ would normally enter a global bus arbiter, which resolves multiple requests and priorities for the global bus, issuing $\overline{\text{GACK}}$ to the processor allowed to use the global bus.

7.5 Coprocessors

7.5.1 Arithmetic coprocessors

The local bus could, of course, carry more than one processor if suitably designed. More commonly, it carries *coprocessors* and DMA devices which are allowed to use the local bus, though overall control is always returned to the processor. Coprocessors are devices designed to enhance the capabilities of the central processor and operate in cooperation with the central processor. For example, an arithmetic coprocessor enhances the arithmetical ability of the central processor by providing additional arithmetical operations, such as floating point and high precision fixed point addition,

subtraction, compare, multiplication and division operations. Arithmetic coprocessors also include floating point trigonometric functions such as sine, cosine and tangent, inverse functions, logarithms and square root. The coprocessor can perform designed operations at the same time as the central processor is performing other duties.

Not all the binary patterns available for encoding machine instructions are used internally by a microprocessor, and it is convenient to arrange an arithmetic coprocessor to respond to some of the unused bit patterns as they are fetched from the memory. The main processor would expect the arithmetic coprocessor to supply the results of any such operations, and in this way the arithmetic coprocessor is seen simply as an extension to the processor. The coprocessor would be designed for particular processors.

8086 family coprocessors

The Intel 8087 (Intel, 1979) numeric coprocessor is designed to match the 16-bit Intel 8086 processors. The 80287 numeric coprocessor matches the 80286 processor. Figure 7.25 shows an 8087 coprocessor attached to an 8086 processor and a common bus. The 8086 processor fetches instructions in the normal way and all instructions are monitored by the 8087 coprocessor. Instructions which begin with the binary pattern 11011 are assigned in the 8086 instruction set for external coprocessor operation and are grouped as ESC (escape) instructions. If an ESC instruction is fetched, the 8087 prepares to act upon it. The ESC instruction also indicates whether an operand is to be fetched from memory. If an operand fetch is indicated, the address of the operand is provided in the third and fourth bytes of a multibyte instruction, and the 8086 fetches the address of the operand; otherwise, the 8086 will continue with the next instruction. The 8087 recognizes the ESC instruction and performs the encoded operation. If an operand address is fetched by the 8086, the address is accepted by the 8087. The 8087 will subsequently fetch the operand. It is possible for both processors to be operating simultaneously, with the 8086 executing the next instruction. The operations provided in the 8087 coprocessor include long word length, fixed point and floating point operations. The 8087 has an internal 8-word, 80 bit-word stack to hold operands. Some coprocessor instructions operate upon two operands held in the top two locations of the stack. Results can be stored in the stack or in memory locations.

The operations are performed about 100 times faster than if the 8086 had performed them using software algorithms. However, once the 8087 has begun executing an instruction, the two processors act asynchronously. When the 8087 is executing an instruction, its BUSY output is brought high. BUSY is usually connected to the $\overline{\text{TEST}}$ input of the 8086. The $\overline{\text{TEST}}$ input can be examined via the 8086 WAIT instruction. If $\overline{\text{TEST}} = 1$, the WAIT instruction causes the 8086 to enter wait states, until $\overline{\text{TEST}} = 0$. Then the next instruction is executed. Typically, the WAIT instruction would be executed before an ESC instruction, to ensure that the coprocessor is ready to respond to the ESC instruction. Hence the two processors can be brought back into synchronism. Other signals connect between the two processors, including bus request and grant signals to enable the two processors to share the bus. The 8086

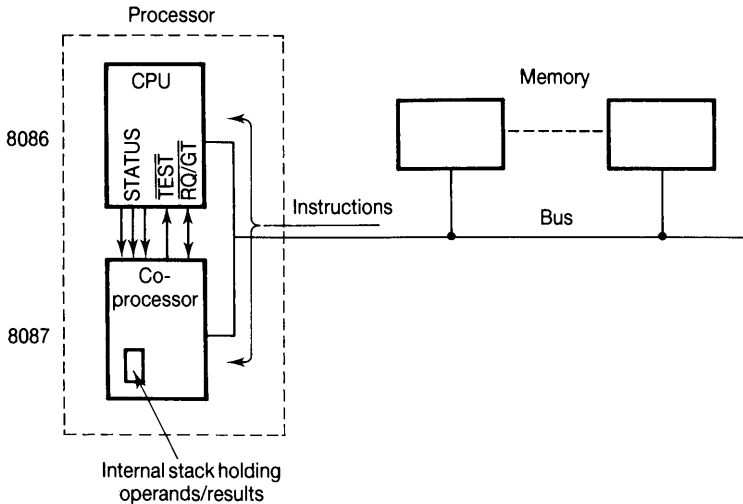


Figure 7.25 CPU with coprocessor

has an internal 6-byte queue used to hold instructions prior to their execution. The state of the queue is indicated by queue status outputs which the 8087 uses to ensure proper operation of `ESC` instructions. (The 8087 can also be connected to the 16-bit 8088 processor which has a 4-byte queue.)

MC68020 coprocessors

As with the 8086 family, the Motorola MC68000 family instruction set has some instruction encoding patterns not used by the processor, and some patterns are reserved for coprocessors (Beims, 1984). All instructions with the most significant four bits 1010 (A hexadecimal) or 1111 (F hexadecimal) in the first word are reserved for future expansion and external devices. Pattern 1111 (F) (called "line-F" op-codes) are reserved for coprocessor instructions. The MC68020 32-bit processor supports coprocessors, and coprocessors are attached to the local bus. Communication between the 68020 and the coprocessor is through data transfers to and from internal registers within the coprocessor.

The address space of the system is divided into eight spaces using a 3-bit function code (processor status outputs FC0–FC2) generated by the processor. In the MC68020, five are defined – user data (001), user program (010), supervisor data (101), supervisor program (110) and special processor-related functions (111), for example, breakpoint acknowledge, access level control, coprocessor communication or interrupt acknowledge. In function code 111, address lines 31 through to 20 are not used, and address bits 19, 18, 17 and 16 differentiate between the functions. Coprocessors use A19–A16 = 0010, and A15–A13 to identify the coprocessor, leaving twelve address bits plus upper/lower byte select lines to identify internal

registers within a particular coprocessor, i.e. up to 8192 bytes can be addressed within each coprocessor. Thirty-two bytes are defined as coprocessor registers used for communication with the main processor.

Coprocessor instructions include a 3-bit code in the first word to identify the coprocessor and the instructions may have extension words. In some cases, the first word includes the same 6-bit encoding of the effective address as internal MC68000 instructions, and the same addressing modes are available. The instructions are categorized in one of three groups – general, conditional and system control. In the general group, the first extension word contains a coprocessor command (defined by a particular coprocessor). In the conditional group, specific coprocessor tests are given in a condition selector field. In the system group, operations for virtual memory systems can be specified.

When the MC68020 fetches a coprocessor op-code (line-F op-code) the processor communicates with the coprocessor by writing a value into the coprocessor register. Coprocessors have eleven addressable registers used to hold commands and data passed to or from the MC68020 processor. For the general coprocessor instruction, the command in the fetched coprocessor instruction is transferred to the coprocessor command register. For conditional instructions, the condition selector is transferred to the coprocessor condition register.

The coprocessor should respond by issuing a 16-bit “primitive” command to the main processor. The encoding of the primitive commands allows up to sixty-four functions, though some are reserved. The functions are categorized into five groups – processor synchronization, instruction manipulation, exception handling, general operand transfer and register transfer. For example, in processor synchronization, the MC68020 can be instructed to proceed with the next instruction. In the general operand transfer group, the MC68020 can be instructed to evaluate the effective address of the coprocessor instruction and pass the stored data or the address to the coprocessor. If an addressed coprocessor does not exist in the system, hardware should issue a bus error signal, and typically the processor will enter a software routine to emulate the coprocessor operations. Bus error signals are normally generated if the processor does not receive an acknowledgement after a specific duration.

An example of a Motorola coprocessor is the MC68881 floating point coprocessor. The overhead of issuing and receiving commands is generally insignificant in typical coprocessor operations, which might take perhaps 50 microseconds to complete a complex floating point arithmetic operation.

Attached arithmetic processors

Some early attached arithmetic processors, for example the Intel 8231A Arithmetic Processing Unit (Intel, 1982), were simply memory mapped or input/output mapped devices which responded to particular commands from the central processor. Results were held in an internal stack, which could be examined by the processor under program control or under an interrupt scheme. These arithmetic processors did not require special coprocessor instructions in the central processor instruction set and could be attached to most microprocessor buses.

7.5.2 Input/output and other coprocessors

Apart from arithmetic coprocessors, coprocessors exist to perform other operations independently, notably:

1. I/O (DMA) controllers/coprocessors.
2. Local area network coprocessors.
3. Graphics coprocessors.

In all cases, the main processor can continue with other operations while the coprocessor is completing its task. Normally, these coprocessors are attached to the local bus, though it is possible to provide a separate local bus, as shown in Figure 7.26. This eliminates memory conflicts if the transactions can be completed totally on the coprocessor local bus. Coprocessors can be provided with their own instruction set and execute their programs from local memory on a separate bus.

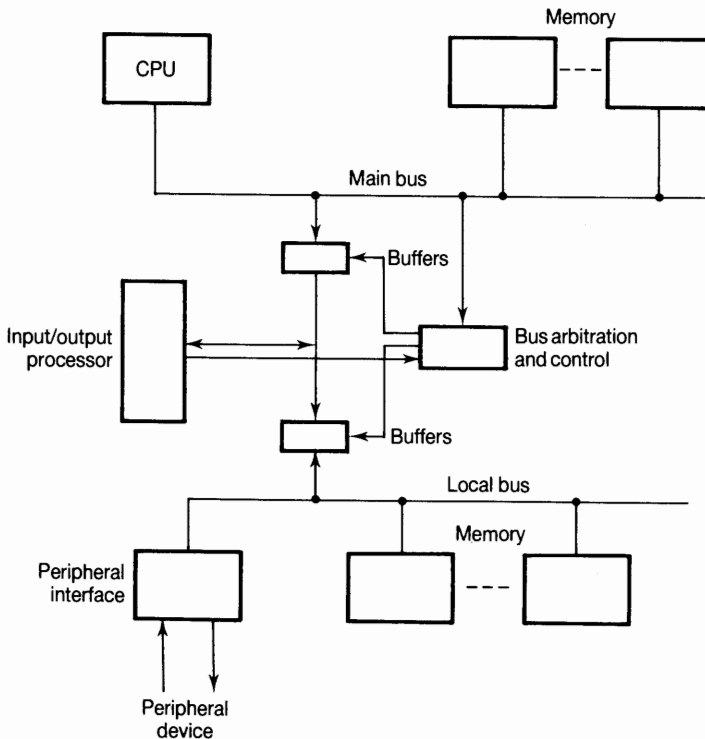


Figure 7.26 Input/output processor with local bus

PROBLEMS

7.1 Prove that the maximum speed-up of a multiprocessor system having n processors, in which each processor uses the bus for the fraction m of every cycle, is given by m .

7.2 Identify the relative advantages of the synchronous bus and the asynchronous bus.

7.3 Identify the relative advantages of parallel arbitration and serial arbitration.

7.4 Identify the relative advantages of centralized arbitration and decentralized arbitration.

7.5 Identify the relative advantages of the daisy chain grant arbitration scheme and the daisy chain request arbitration scheme. Which would you choose for a new microprocessor? Why?

7.6 A 3-to-8 line priority encoder is a logic component which accepts eight request inputs and produces a 3-bit number identifying the highest priority input request using fixed priority. A 3-to-8 line decoder accepts a 3-bit number and activates one of its eight outputs, as identified by the input number. Show how these components could be used to implement parallel arbitration. Derive the Boolean expressions for each component and show that these equations correspond to the parallel arbitration expressions given in the text.

7.7 Design a parallel arbitration system using three levels of parallel arbiter parts and determine the arbitration time of the system.

7.8 Suppose a rotating daisy chain priority circuit has the following signals:

BR	Bus request from bus master
BG	Bus grant to bus master
BRIN	Bus grant daisy chain input
BROUT	Bus grant daisy chain output

and contains one J-K flip-flop whose output, BMAST (bus master), indicates that the master is the current bus master. Draw a state table showing the eight different states of the circuit. Derive the Boolean expressions for the flip-flop inputs, and for BROUT. (See Nelson and Refai (1984) for solution.)

7.9 For any 16-/32-bit microprocessor that you know, develop the Boolean expressions and logic required to generate bus request and grant signals for both local and global (system) buses. The local bus addresses are 0 to 65535 and the global bus addresses are from 65536 onwards.

7.10 Derive Boolean expressions to implement a daisy chain scheme having three processors at each arbitration site.

7.11 Derive an expression for the arbitration time of a combined serial parallel arbitration scheme having m processors, using one n -input parallel arbiter. (m is greater than n .)

7.12 What is the access time for the highest and next highest priority processor in a system using daisy chain priority, given that the request rate is 0.25?

7.13 Suppose a new arithmetic coprocessor can have eight arithmetic operations. List those operations you would choose in the coprocessor. Justify.

7.14 Compare and contrast the features and mechanisms of 8086 coprocessors and 68020 coprocessors.

Interconnection networks are of fundamental importance to the design and operation of multiprocessor systems. They are required for processors to communicate either between themselves or with memory modules. This chapter will consider the interconnection network as applicable to a wide range of multiprocessor architectures, though with particular reference to general purpose MIMD computers. Multiple bus systems will be considered as an interconnection network, extending the single bus interconnection scheme of Chapter 7.

8.1 Multiple bus multiprocessor systems

In the last chapter, we considered single bus multiprocessor systems. We can extend the bus system to one with b buses, p processors and m memory modules, as shown in Figure 8.1(a), in which no more than one processor can use one bus simultaneously. Each bus is dedicated to a particular processor for the duration of a bus transaction. Each processor and memory module connects to each of the buses using electronic switches (normally three-state gates). A connection between two components is made, with two connections to the same bus. We will refer to processors and memory modules only. (Memory and I/O interfaces can be considered as similar components for basic bus transactions.) Processor–memory module transfers can use any free bus, and up to b requests for different memory modules can be serviced simultaneously using b buses. A two-stage arbitration process is necessary, as shown in Figure 8.1(b). In the first phase, processors make requests for individual memory modules using one arbiter for each memory module, and up to one request for each memory module is accepted (as only one request can be serviced for each module). There might be up to m requests accepted during this phase, with m memory modules. In the second phase, up to b of the requests accepted in the first phase are finally accepted and allocated to the b buses using a bus arbiter. If m is less than b , not all the buses are used. Blocked requests need to be honored later.

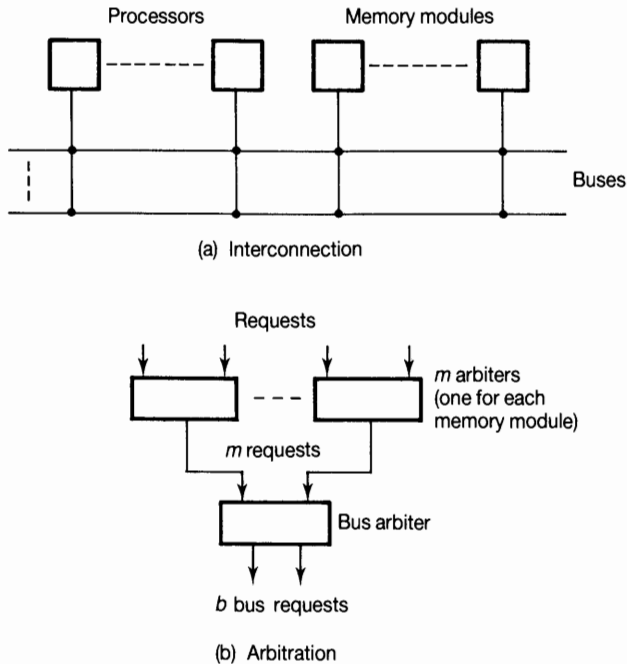


Figure 8.1 Multiple bus system (a) Interconnection (b) Arbitration

Clearly, bus contention will be less than in a single bus system, and will reduce as the number of buses increases; the complexity of the system then increases. Though extensive analytical studies have been done to determine the performance characteristics of multiple bus systems, few systems have been constructed for increased speed. Apart from such applications, multiple bus systems (especially those with two or three buses) have been used in fault tolerant systems. A single bus system collapses completely if the bus is made inoperative (perhaps through a bus driver short-circuited to a supply line).

Variations of the multiple bus system have been suggested. For example, not all the memory modules need to be connected to all the buses. Memory modules can be grouped together, making connections to some of the buses, as shown in Figure 8.2. Multilevel multiple bus systems can be created in which multiple bus systems connect to other multiple bus systems, either in a tree configuration or other hierarchical, symmetrical or non-symmetrical configurations.

Lang *et al.* (1983) showed that some switches in a multiple bus system can be removed (up to 25 per cent) while still maintaining the same connectivity and throughput (bandwidth). In particular, Lang showed that a single “rhombic” multiple

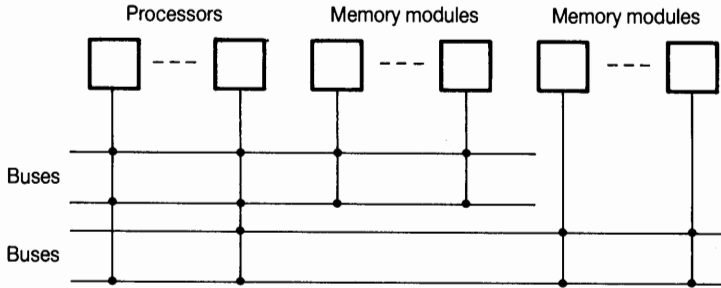


Figure 8.2 Partial multiple bus system

bus system can be designed with the same connectivity of a full multiple bus scheme and no reduction in performance whatever when:

1. $p - b + 1 \leq m_i \leq m$
2. $p + m + 1 - b - m_i \leq p_i \leq p$

where m_i memory modules and p_i processors are connected to bus i . Lang also showed that the minimum switch configuration can be achieved by keeping the processor connections complete and minimizing the memory module connections. We shall use Lang's observations in overlapping multiple bus networks (Section 8.5.2).

8.2 Cross-bar switch multiprocessor systems

8.2.1 Architecture

In the cross-bar switch system, processors and memories are interconnected through an array of switches with one electronic cross-bar switch for each processor-memory connection. All permutations of processor-memory connections are possible simultaneously, though of course only one processor may use each memory at any instant. The number of switches required is $p \times m$ where there are p processors and m memory modules.

Each path between the processors and memories normally consists of a full bus carrying data, address and control signals, and each cross-bar switch provides one simultaneous switchable connection. Hence the switch may handle perhaps 60–100 lines if it is to be connected between each processor and each memory. The address lines need only be sufficient to identify the location within the selected memory module. For example, twenty address lines are sufficient with 1 Mbyte memory

modules. Additional addressing is necessary to select the memory module. The memory module address is used to select the cross-bar switch. The cross-bar switch connections may be made by:

1. Three-state gates.
2. Wired-OR gates.
3. Analog transmission gates.
4. Multiplexer components .

The cross-bar switch connections could be fabricated in VLSI, though the number of input/output connections is significant. Analog transmission gates have the advantage of being intrinsically bidirectional.

Each processor bus entering the cross-bar network contains all the necessary signals to access the memory modules, and would include all the data lines, sufficient address lines and memory transfer control signals. The switch network can also be implemented using multiport memory. In effect, then, all of the switches in one column of the cross-bar are moved to be within one memory module.

The number of switches in a cross-bar network becomes excessive and impractical for large systems. However the cross-bar is suitable for small systems, perhaps with up to twenty processors and memories.

8.2.2 Modes of operation and examples

There are two basic modes of operation for cross-bar switch architectures, namely:

1. Master–slave architecture.
2. Architecture without a central control processor.

Each has distinct hardware requirements.

In the master–slave approach, one processor is assigned as the master processor and all the other processors are slave processors. All cross-bar switches are controlled by the master processor, as shown in Figure 8.3. The operating system for this architecture could also operate on a master–slave principle, possibly with the whole operating system on the master processor. Alternatively, the central part of the operating system could be on the master processor, with some dedicated routines passed over to slave processors which must report back to the master processor. The slave processors are available for independent user programs. In any event, only the master processor can reconfigure the network connections, and slave processors executing user programs must request a reconfiguration through the master processor. The master–slave approach is certainly the simplest, both for hardware and software design.

In the cross-bar switch system without central control, each processor controls the switches on its processor bus and arbitration logic resolves conflicts. Processors

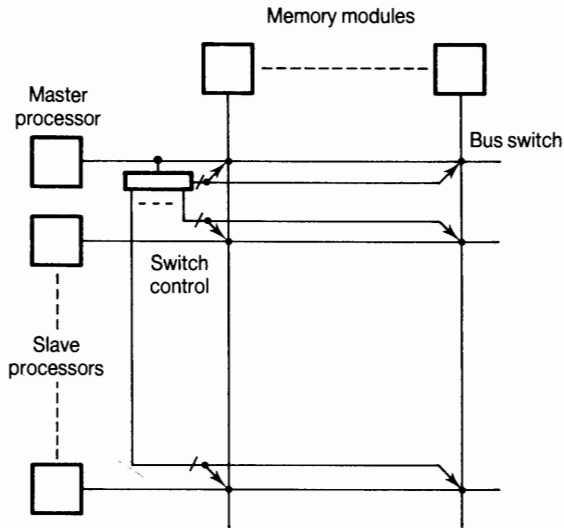


Figure 8.3 Cross-bar switch system with central control (master–slave)

make independent requests for memory modules. Each memory module/bus has its own arbitration logic and requests for that memory module are directed to the corresponding memory arbitration logic. Up to one request will be accepted for each memory module, and other requests are held back for subsequent arbitration cycles. Arbitration is effected by one arbiter for each memory module receiving requests for that module, as shown in Figure 8.4.

Perhaps the first example of a cross-bar switch multiple processor system (certainly the first commercial example) was the Burroughs D-825 four processor/sixteen memory module cross-bar switch system introduced in 1962 for military applications. Subsequently, commercial cross-bar switch systems have occasionally appeared, usually with small numbers of processors. There is at least one commercial example of a master–slave architecture, the IP-1 (International Parallel Machines Inc.). The basic configuration of the IP-1 has nine processors, one a master processor, with eight cross-bar switch memory modules. The system can be expanded to thirty-three processors. The cross-bar switch memory operates like multiport memory. There has been at least one small master–slave architecture research project (Wilkinson and Abachi, 1983).

A significant, influential and extensively quoted but now obsolete cross-bar switch system without central control called the C.mmp (Computer multi-miniprocessor) was designed and constructed in the early 1970s at Carnegie-Mellon University (Wulf and Harbison, 1978). C.mmp employed sixteen PDP-11 computer systems, each with local memory, connected to a sixteen memory module. In 1978,

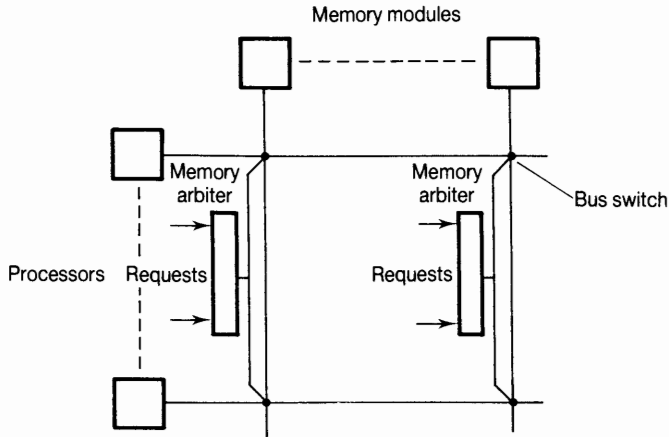


Figure 8.4 Cross-bar switch system without central control

at the end of the main investigation, the five original PDP-11s were PDP-11/20s and the eleven introduced in 1974 to complete the system were the faster PDP-11/40s. There were 3 Mbytes of memory in total (32 Mbytes possible). The total hardware cost of \$600 000 was divided into \$300 000 for the processors, \$200 000 for the shared memory and \$100 000 for the cross-bar switch. Apart from the cross-bar switch communication paths between the processors and memory, a communication path was established between the processors using an interprocessor (IP) bus. Input/output devices and backing memory were connected to specific processors.

PDP-11 processor instructions can specify a 16-bit address. This address is divided into eight 8 Kbyte pages (three most significant bits for the page and thirteen bits for the location within the page). This address is extended to eighteen bits on the local bus (Unibus) by concatenating two bits contained in the processor status word with the 16-bit address. The two processor status word bits cannot be changed by user programs and constrain user programs to operate within the 16-bit address space, i.e. within 64 Kbytes (eight 8 Kbyte pages).

In the C.mmp, shared memory is accessed via the cross-bar switch with 25-bit address, with the most significant four bits selecting the memory module, i.e. with high order interleaving. The 18-bit local bus address is translated into a 25-bit shared memory address by an address translation unit called Dmap, using a direct mapping technique (Section 2.2.2, page 32). Dmap contains four sets of relocation registers, with eight registers in each set. One set is selected by the two processor status bits and the register within the set is selected by the three next significant bits of the address, i.e. by the page bits. Each register contains a 12-bit frame address and three bits for memory management. The frame address selected is concatenated with the thirteen remaining address bits to obtain a 25-bit address. The frame bits are divided into a 4-bit port number selecting the memory module and an 8-bit page within port.

As C.mmp employed the approach without central control, any processor could execute any part of the operating system at any time. Shared data structures were accessed by only one process at a time, using one of two general mechanisms – either fast simple binary locks for small data structures, or semaphores with a descheduling and queueing mechanism for larger data structures. A widely reported disadvantage of the C.mmp, as constructed with PDP-11s, is the small user address space allowed by the 16-bit addresses.

The cross-bar switch architecture without central control has been used more recently, for example the S1 multiprocessor system developed for the United States Navy. The S1 also has sixteen processors connected to sixteen memory modules through a 16×16 cross-bar switch. However, the processors are specially designed very high speed ECL (emitter-coupled logic) processors.

In a cross-bar system, input/output devices and backing memory can be associated with particular processors, as in the C.mmp and S1. Alternatively, they can be made accessible to all processors by interconnecting them to the processors via the same cross-bar switch network as the memory modules; the cross-bar switch would then need to be made larger. Input/output devices and backing memory could also be connected to the processors via a separate cross-bar switch.

There are a number of possible variations in the arrangement of a cross-bar switch network. For example, Hwang *et al.* (1989) proposed the *orthogonal multiprocessor* using a network in which processors have switches to one of two orthogonal buses in the cross-bar network. At any instant, the processors can all connect to the vertical or horizontal buses. Each memory module needs to access only two buses. Hwang develops several algorithms for this system. Various memory access patterns are allowed. Overlapping connectivity networks including cross-bar versions, are considered in Section 8.5.

8.3 Bandwidth analysis

8.3.1 Methods and assumptions

One of the key factors in any interconnection network is the bandwidth, BW, which is the average number of requests accepted in a bus cycle. Bandwidth gives the performance of the system under bus contention conditions. Bandwidth and other performance figures can be found via one of four basic techniques:

1. Using analytical probability techniques.
2. Using analytical Markov queueing techniques.
3. By simulation.
4. By measuring an actual system performance.

Simplifying assumptions are often made for techniques 1 and 2 to develop a closed

form solution, which is then usually compared to simulations. Measurements on an actual system can confirm or contradict analytical and simulation studies for one particular configuration. We shall only consider probabilistic techniques. The principal assumptions made for the probabilistic analysis are as follows:

1. The system is synchronous and processor requests are only generated at the beginning of a bus cycle.
2. All processor requests are random and independent of each other.
3. Requests which are not accepted are rejected, and requests generated in the next cycle are independent of rejected requests generated in previous cycles.

If bus requests are generated during a cycle, they are only considered at the beginning of the next cycle. Arbitration actions are only taken at the beginning of each bus cycle. Asynchronous operation, in which requests can occur and be acted upon at any time, can be modelled by reducing the cycle time to that required to arbitrate asynchronous requests. In practice, most bus-based multiprocessor systems respond to bus requests only at the beginning of bus cycles, or sometimes only at the beginning of instruction cycles. Instruction cycles would generally be of variable time, but virtually all published probabilistic analyses assume a fixed bus cycle.

Assumption 2 ignores the characteristic that programs normally exhibit referential locality for both data and instruction references. However, requests from different processors are normally independent. A cross-bar switch system can be used to implement an interleaved memory system and some bandwidth analysis is done in the context of interleaved memory. Low order interleaving would generally ensure that references are spread across all memory modules, and though not truly in a random order, it would be closer to the random request assumption.

According to assumption 3, rejected requests are ignored and not queued for the next cycle. This assumption is not generally true. Normally when a processor request is rejected in one cycle, the same request will be resubmitted in the next cycle. However, the assumption substantially simplifies the analysis and makes very little difference to the results.

Though it is possible to incorporate memory read, write and arbitration times into the probabilistic analysis, we will refrain from adding this complexity. Markov queueing techniques take into account the fact that rejected requests are usually resubmitted in subsequent cycles.

8.3.2 Bandwidth of cross-bar switch

In a cross-bar switch, contention appears for memory buses but not for processor buses, because only one processor uses each processor bus but more than one processor might compete for a memory module and its memory bus. In the multiple bus system, to be considered later, both system bus contention and memory contention can limit the performance. In the cross-bar switch, we are concerned with the

probability that more than one request is made for a memory module as, in such cases, only one of the multiple requests can be serviced, and the other requests must be rejected.

First, let us assume that all processors make a request for some memory module during each bus cycle. Taking a small numerical example, with two processors and three memories, Table 8.1 lists the possible requests. Notice that there are nine combinations of two requests taken from three possible requests. The average bandwidth is given by the average number of requests that can be accepted. Fifteen requests can be accepted and the average bandwidth is given as $15/9 = 1.67$. Memory contention occurs when both processors request the same memory module. For our two processor/three memory system, we see that processor 1 makes a request for memory 1 three times, memory 2 three times and memory 3 three times, and similarly for processor 2. Hence there is a $1/3$ chance of requesting a particular memory.

Table 8.1 Processor requests with two processors and three memories

Memory requests		Number of requests accepted	Memory contention
Processors			
P_1	P_2		
1	1	1	YES
1	2	2	NO
1	3	2	NO
2	1	2	NO
2	2	1	YES
2	3	2	NO
3	1	2	NO
3	2	2	NO
3	3	1	YES

Now let us develop a general expression for bandwidth, given p processors and m memory modules. We have the following probabilities: The probability that a processor P_i makes a request for a particular memory module M_j is $1/m$ for any i and j (as there is equal probability that any memory module is requested by a processor). The probability that a processor, P_i , does not make a request for that memory module, M_j , is $1 - 1/m$. The probability that no processor makes a request for the memory module is given by $(1 - 1/m)^p$. The probability that one or more processors make a request for memory module M_j (i.e. the memory module has at least one request) is $(1 - (1 - 1/m)^p)$. Hence the cross-bar switch bandwidth, i.e. the number of memory modules with at least one request, is given by:

$$BW = m(1 - (1 - 1/m)^p)$$

The bandwidth function increases with p and m and is asymptotically linear for either p or m , given a constant p/m ratio (Baer, 1980). Alternative explanations and derivations of bandwidth exist, perhaps the first being in the context of interleaved memory (Hellerman, 1966). An early derivation for the bandwidth can be found in Ravi (1972), also in the context of interleaved memory.

The cross-bar switch bandwidth can be derived for the situation in which processors do not always generate a request during each bus cycle, for example, in a system having local memory attached to the processors. Let r be the probability that a processor makes a request. Then the probability that a processor makes a request for a memory module, $M_j = r/m$. For a simple derivation, this term can be substituted into the previous derivation to get the bandwidth as:

$$BW = m(1 - (1 - r/m)^p)$$

Patel (1981) offers an alternative derivation for the bandwidth with requests not necessarily always generated.

Figure 8.5 shows the bandwidth function. Simulation results (Lang *et al.*, 1982; Wilkinson, 1989) are also shown using a random number generator to specify the requests and with blocked requests resubmitted on subsequent cycles. For a request rate of 1, the bandwidth equation derived will give a value higher than that found in simulation and in practice, because rejected requests which are resubmitted in the next cycle will generally lead to more contention. At request rates of less than 1, the simulation results can give a higher bandwidth than analysis because there is then an opportunity for blocked requests to be satisfied later, when some other processors do not make requests.

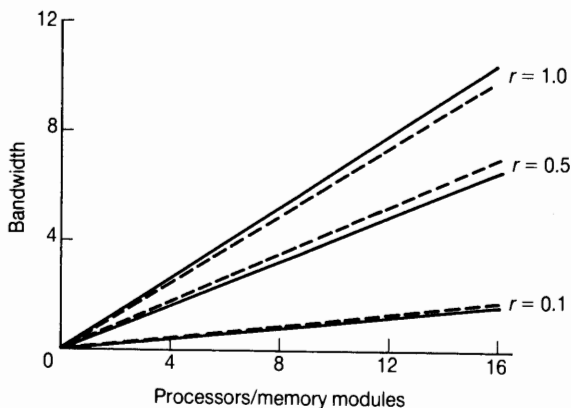


Figure 8.5 Bandwidth of cross-bar switch network (— analysis, ---- simulation)

The probability that an arbitrary request is accepted is given by:

$$P_a = \frac{BW}{rp} = (m/rp)(1 - (1 - r/m)^p)$$

and the expected wait time for a request to be accepted is $(1/P_a - 1)t_c$ where t_c is the bus cycle time.

8.3.3 Bandwidth of multiple bus systems

In the multiple bus system, processors and memory modules connect to a set of b buses, and the bandwidth will depend upon both memory contention and bus contention. Only a maximum of b requests can be serviced in one bus cycle, and then only if the b requests are for different memory modules. We noted that servicing a memory request can be regarded as a two stage process. First, up to m memory requests must be selected from all the requests. This mechanism has already been analyzed in the cross-bar switch system as it is the only selection process. We found that the probability that a memory has at least one request is $1 - (1 - r/m)^p = q$ (say). Second, of all the different memory requests received, only b requests can be serviced, due to the limitation of b buses. The probability that exactly i different memory modules are requested during one bus cycle is given in Mudge *et al.* (1984) (see also Goyal and Agerwala, 1984):

$$f(i) = \binom{p}{i} q^i (1 - q)^{p - i}$$

where $\binom{p}{i}$ is the binomial coefficient. The overall bandwidth is given by:

$$BW = \sum_{i=b}^p b f(i) + \sum_{i=1}^{b-1} i f(i)$$

The first term relates to b or more different requests being made and all b buses being in use, and the second term relates to fewer than b different requests being made and fewer than b buses being used. Figure 8.6 shows the bandwidth function and also simulation results (Lang *et al.*, 1982). As with the cross-bar switch for a request rate of 1, the simulation bandwidth is slightly less than the analytical value,

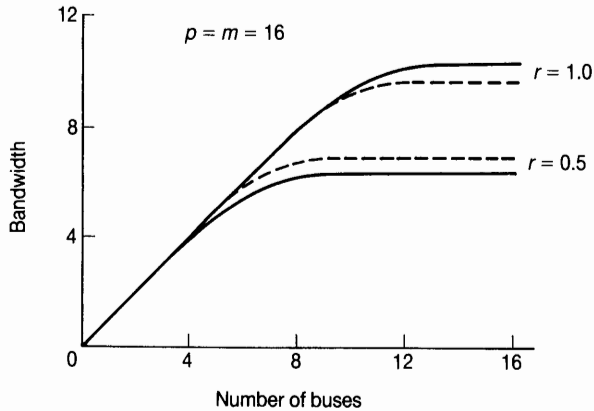


Figure 8.6 Bandwidth of multiple bus system (— analysis, - - - simulation)

but for request rates of less than 1, the analytical values are less than the simulation values, as then there is more opportunity for rejected requests to be accepted in later cycles.

In the analysis for the cross-bar switch and for the multiple bus system, we assume that rejected requests are discarded and do not influence the bandwidth. In Chapter 7, Section 7.3.1, we presented a method of computing the effect of rejected requests being resubmitted by adjusting the request rate. This method can be applied to multiple bus and cross-bar switch networks to obtain a more accurate value for the bandwidth. However, the method assumes that the rejected requests will be resubmitted to a memory module selected at random rather than to the same memory module as would normally happen. This does not matter in the case of the single bus system with a single path to all memory modules, but has an effect in the case of multiple buses and cross-bar switches. However, the method does bring the results closer to actual values from simulation.

Some work has been done to incorporate priority into the arbitration function (see Liu and Jou, 1987) and to have a “favorite” memory module for each processor which is more likely to be selected (see Bhuyan, 1985) and to characterize the reliability (see Das and Bhuyan, 1985). An early example of the use of Markov chain model is given by Bhandarkar (1975). Markov models are used by Irani and Önyüksel (1984) and Holliday and Vernon (1987). Actual measurements and simulation are used to compare analytical models, for example as in Baskett and Smith (1976).

8.4 Dynamic interconnection networks

In this section, we will describe various schemes for interconnecting processing elements (processors with memory) or interconnecting processors to memories, apart from using buses. The schemes are applicable to both MIMD and SIMD computer systems, though particular network characteristics might better suit one type of computer system. Our emphasis is on general purpose MIMD computer systems.

8.4.1 General

In a *dynamic interconnection network*, the connection between two nodes (processors, processor/memory) is made by electronic switches such that some (or all) of the switch settings are changed to make different node to node connections. For ease of discussion, we will refer to inputs and outputs, implying that the transfer is unidirectional; in practice most networks can be made bidirectional. (Of course, the whole network could be replicated, with input and outputs transposed.)

Networks sometimes allow simultaneous connections between all combinations of input and outputs; such networks are called *non-blocking networks*. Non-blocking networks are *strictly non-blocking* if a new interconnection between an arbitrary unused input and an arbitrary unused output can always be made, irrespective of existing connections, without disturbing the existing paths. Some non-blocking networks may require paths to be routed according to a prescribed routing algorithm to allow new input/output connections to be made without disturbing existing interconnections; such non-blocking networks are called *wide-sense non-blocking networks*. Many networks are formulated to reduce the number of switches and do not allow all combinations of input/output connections simultaneously; such networks are called *blocking networks*. A network is *rearrangeable* if any blocked input/output connection path can be re-established by altering the internal switches to reroute paths and free the blockage.

In general, the switches are grouped into switching stages which may have one (or more) input capable of connecting to one (or more) output. Dynamic networks can be classified as:

1. Single stage.
2. Multistage.

In a single stage network, information being routed passes through one switching stage from input to output. In a multistage network, information is routed through more than one switching stage from input to output. Multistage networks generally have fewer internal switches, but are often blocking. Some networks have non-blocking characteristics for certain input/output combinations, which may be useful in particular applications.

8.4.2 Single stage networks

A fundamental example of a dynamic single stage network is the cross-bar switch network analyzed previously, in which the stage consists of $n \times m$ switches (n input nodes, m output nodes) and each switch allows one node to node connection. This network is non-blocking and has the minimum delay through the network compared to other networks, as only one switch is involved in any path. The number of switches increases as $O(nm)$ (or $O(n^2)$ for a square network) and becomes impractical for large systems. We shall see that the non-blocking nature of the cross-bar switch network can be preserved in the multistage Clos network and with substantially fewer switches for large systems. However, the single stage cross-bar switch network is still a reasonable choice for a small system. The complete connectivity and flexibility of the cross-bar is a distinct advantage over multistage blocking networks for small systems. The term “cross-bar” stems from the historical use of mechanical switches in old telephone exchanges.

8.4.3 Multistage networks

Multistage networks can be divided into two types:

1. Cross-bar switch-based networks.
2. Cell-based networks.

Cross-bar switch-based networks use switching elements consisting of cross-bar switches, and hence multistage cross-bar switch networks employ more than one cross-bar switch network within a larger network. Cell-based networks usually employ switching elements with only two inputs and two outputs, and hence could be regarded as a subset of the cross-bar switch network, though the 2×2 switching elements in some cell-based networks are not full cross-bar switches. Instead they have limited interconnections.

Multistage cross-bar switch-based networks – Clos networks

In 1953 Clos showed that a multistage cross-bar switch network using three or more stages could give the full non-blocking characteristic of a single stage cross-bar switch with fewer switches for larger networks. This work was done originally in the context of telephone exchanges, but has direct application to computer networks, especially when the non-blocking characteristic is particularly important.

A general *three-stage Clos network* is shown in Figure 8.7, having r_1 input stage cross-bar switches, m middle stage cross-bar switches and r_2 output stage cross-bar switches. Each cross-bar switch in the first stage has n_1 inputs and m outputs, with one output to each middle stage cross-bar switch. The cross-bar switches in the middle stage have r_1 inputs, matching the number of input stage cross-bar switches, and r_2 outputs, with one output to each output stage cross-bar switch. The cross-bar switches in the final stage have m inputs, matching the number of middle stage

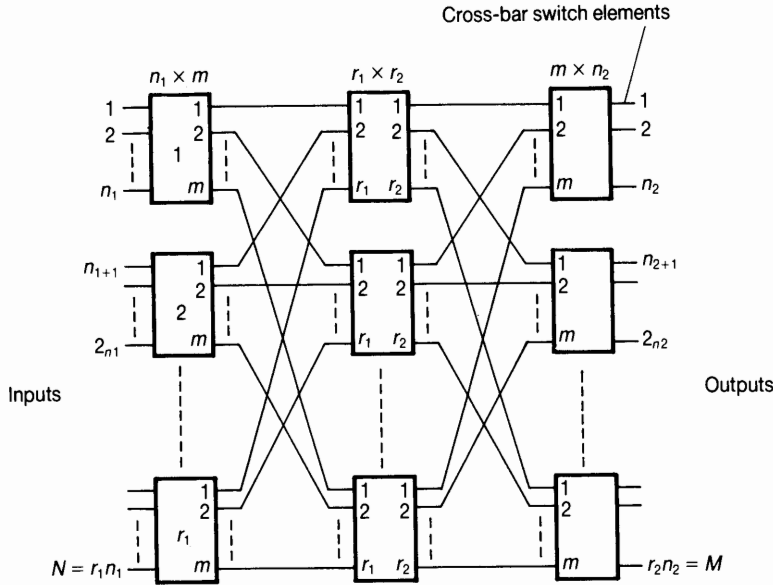


Figure 8.7 Three-stage Clos network

cross-bar switches, and n_2 outputs. Hence the numbers n_1 , n_2 , r_1 , r_2 and m completely define the network. The number of inputs, N , is given by $r_1 n_1$ and the number of outputs, M , is given by $r_2 n_2$.

Clearly, any one network input has a path to any network output. Whether the network is non-blocking will depend upon the number of middle stages. Clos showed that the network is non-blocking if the number of cross-bar elements in the middle stage, m , satisfies:

$$m \geq n_2 + n_1 - 1$$

For a network with the same number of inputs as outputs, the number of input/outputs = $r_1 n_1 = r_2 n_2$. If $n_1 = n_2$, the middle stages are square cross-bar switches and the non-blocking criterion reduces to:

$$m \geq 2n - 1$$

Clos derived the number of switches in a square three-stage Clos network with input and output networks of the same size, as:

$$\text{Number of switches} = (2n - 1) 2N + \frac{N^2}{n^2}$$

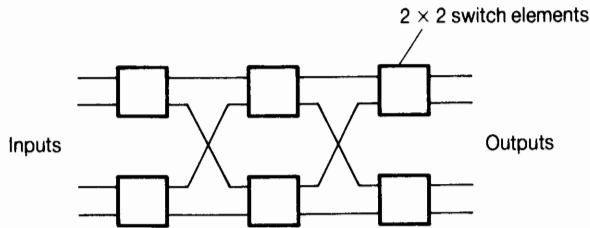


Figure 8.8 Three-stage Benes network

resulting in fewer switches than a single stage cross-bar switch when N is greater than about twenty-five for a square network (Broomell and Heath, 1983). It has been shown that a Clos network is rearrangeable if $m \geq n_2$, otherwise the network becomes blocking.

Clos networks can be created with five stages by replacing each switching element in the middle row with a three-stage Clos network. Similarly seven, nine, eleven stages, etc. can be created by further replacement. The *Benes network* is a special case of the Clos network with 2×2 cross-bar switch elements. A three-stage Benes network is shown in Figure 8.8. Benes networks could also be classified as cell-based networks.

Cell-based networks

The switching element (or cell) in cell-based networks typically has two inputs and two outputs. A full cross-bar switch 2×2 network cell has twelve different useful input/output connections (states). Three further 2×2 network patterns exist, one connecting the inputs together, leaving the outputs free, one connecting the outputs together, leaving the inputs free, and one connecting the inputs together and the outputs together; there is no input/output connection. A final state has no interconnections. Four binary control signals would be necessary to specify the states of a 2×2 network.

Some, if not most, cell-based networks employ 2×2 cells which do not have all possible states. The two state (straight through or exchange) 2×2 network is the most common. In practice, once a path is chosen for one of the inputs – either the upper or the lower output – there is only one possible path allowed for the other input (which will be the upper output if the lower output has been taken, or the lower output if the upper output has been taken). Hence, the straight through/exchange states are sufficient and only one binary signal need be present to select which state should exist at any instant.

Most cell-based networks are highly blocking, which can be evidenced by the fact that if there are s switching cells, each with two states, there are only 2^s different states in the complete network. However, with, say, p input/outputs, there are $p!$ different combinations of input/output connections and usually $p!$ is much larger than 2^s .

Each stage of cells can be interconnected in various ways. The *baseline network* (Feng, 1981) shown in Figure 8.9, is one example of a network with a very convenient self-routing algorithm (*destination tag algorithm*) in which successive bits of the destination address control successive stages of the network. Each stage of the baseline network divides the routing range into two. The first stage splits the route into two paths, one to the lower half of the network outputs and one to the upper half. Hence, the most significant bit of the destination address can be used to route the inputs to either the upper half of the second stage, when the bit is 0, or to the lower half if the bit is 1. The second stage splits the route into the upper quarter or second quarter if the upper half of the outputs has been selected, or to the third quarter or lower quarter if the lower half has been selected. The second most significant bit is used to select which quarter, once the most significant bit selection has been made. This process is repeated for subsequent stages if present. For eight inputs and outputs, there would be three stages, for sixteen inputs and outputs there would be four stages, and so on. The least significant bit controls the last stage. Such *self-routing networks* suggest packet switching data transmission.

Shuffle interconnection pattern

The *perfect shuffle* pattern finds wide application in multistage networks, and can also lead to destination tag self-routing networks. Originally, the perfect shuffle was developed by Pease, in 1968, for calculating the fast Fourier transform (Broomell and Heath, 1983), and was later developed for other interconnection applications by Stone and others. The input to output permutation of the (2-) perfect shuffle network is based upon shuffling a pack of cards by dividing the pack into two equal parts which are slid together with the cards from each half of the pack interleaved. The perfect shuffle network takes the first half of the inputs and interleaves the second half such that the first half of inputs pass to odd numbered outputs and the second

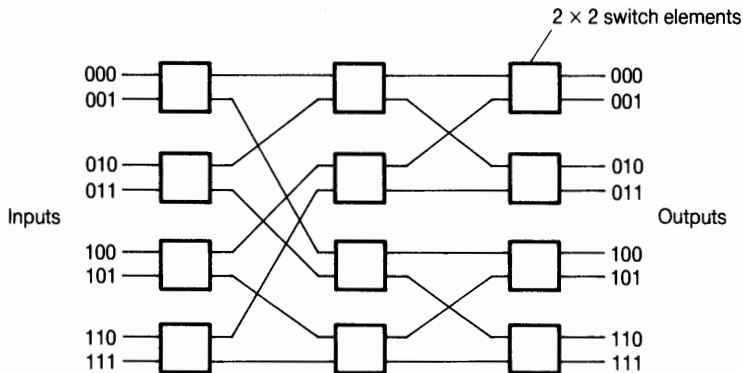


Figure 8.9 8 x 8 baseline network

half to even numbered outputs. For example, with eight inputs, the first half of the inputs consists of 0, 1, 2 and 3 and the second half of 4, 5, 6 and 7. Input 0 passes to output 0, input 1 to output 2, input 2 to output 4, input 3 to output 6, input 4 to output 1, input 5 to output 3, input 6 to output 5 and input 7 to output 7.

Given that the input/output addresses have the form $a_{n-1}a_{n-2} \dots a_1a_0$, the perfect shuffle performs the following transformation:

$$\text{Shuffle } (a_{n-1}a_{n-2} \dots a_1a_0) = a_{n-2} \dots a_1a_0a_{n-1}$$

i.e. the address bits are cyclically shifted one place left. The inverse perfect shuffle cyclically shifts the address bits one place right.

To make all possible interconnections with the shuffle pattern, a *recirculating network* can be created by recirculating the outputs back to the inputs until the required connection is made. Exchange “boxes” are introduced; these selectively swap pairs of inputs, as shown in Figure 8.10 (*shuffle exchange network*). Each exchange box has two inputs and two outputs. There are two selectable transfer patterns, one when both inputs pass to the two corresponding outputs, and one when each input passes to the other output (i.e. the inputs are transposed). The exchange boxes transform the address bits by complementing the least significant bit, i.e.:

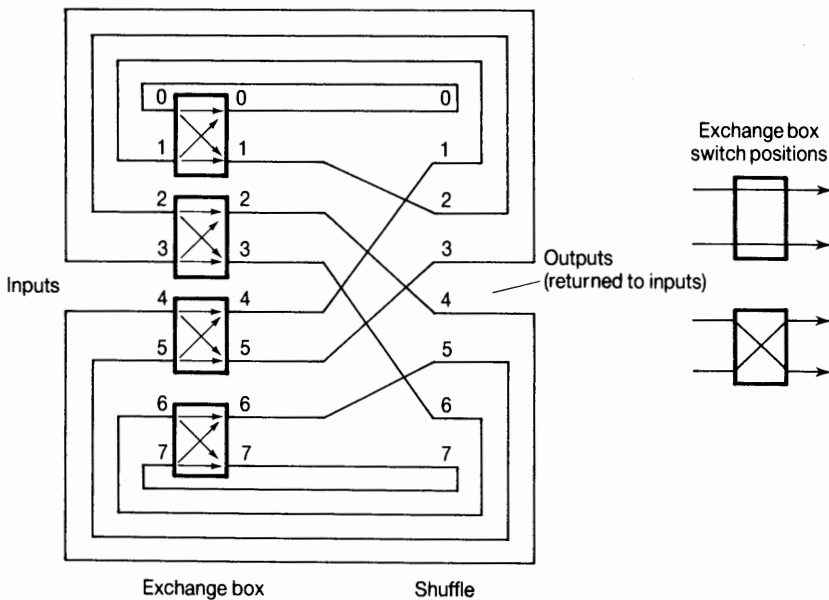


Figure 8.10 Shuffle exchange network

$$\text{Exchange } (a_{n-1}a_{n-2} \dots a_1a_0) = a_{n-1}\overline{a_{n-2}} \dots a_1\overline{a_0}$$

For example, 6 (110) passes over to 7 (111) and 7 passes over to 6. The interconnection function is given by a number of shuffle exchange functions. Any input can be transferred to any output by repeated passes through the network. For example, to make a connection from 0 (000) to 6 (110) would require two passes, one pass to exchange to 1 (001) and shuffle to 2 (010), and one pass to exchange to 3 (011) and shuffle to 6 (110). A maximum of n recirculations are necessary to obtain all permutations.

Multistage perfect shuffle networks – Omega network

Rather than recirculate the paths, perfect shuffle exchange networks can be cascaded to become the *Omega network*, as shown in Figure 8.11. The network (like the baseline network) has the particular feature of the very simple destination tag self-routing algorithm. Each switching cell requires one control signal to select either the upper cell output or the lower cell output (0 specifying the upper output and 1 specifying the lower). The most significant bit of the address of the required destination is used to control the first stage cell; if this is 0 the upper output is selected, and if it is 1, the lower output is selected. The next most significant bit of the destination address is used to select the cell output of the next stage, and so on until the final output has been selected.

The cells used need to be able to select either the upper or the lower output and a 2×2 straight through/exchange cell is sufficient. The Omega network was proposed for array processing applications with four-state cells (straight through/exchange/broadcast upper/broadcast lower). The Omega network is highly blocking, though

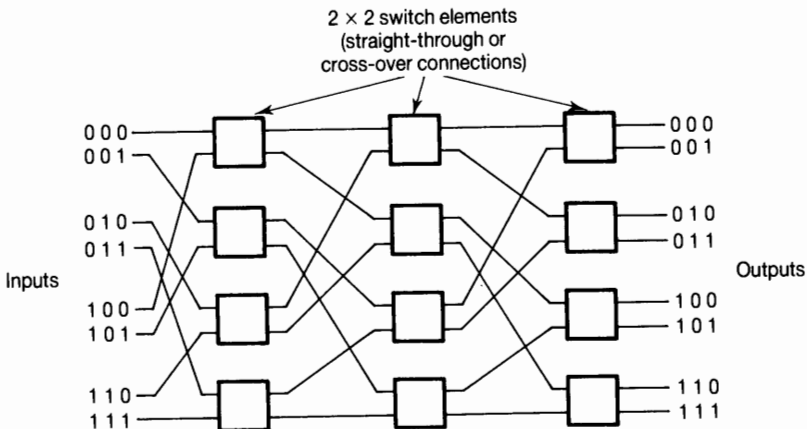


Figure 8.11 Omega network

one path can always be made from any input to any output in a free network. The *indirect binary n -cube network*, which is similar to the Omega network, was proposed for processor to processor interconnections using only two-state cells. (The *direct binary n -cube* has links between particular nodes and is also called a hypercube, see page 286.) The indirect binary n -cube and Omega networks were found to be functionally equivalent by a simple address translation. Switching networks are deemed equivalent if they produce the same permutations of input/output connections irrespective of their internal connections or actual input/output address numbering system.

Generalized self-routing networks

The self-routing networks such as Omega, baseline and indirect binary n -cube networks can be extended to use numbering system bases other than two and a generalized q -shuffle. In terms of cards, the q -shuffle takes qr cards and divides the cards into q piles of r cards. Then one card from each pile is taken in turn to create a shuffled pile.

The *Delta network* (Patel, 1981) is a generalization using a numbering base which can be other than 2 throughout. This network connects a^n inputs to b^n outputs through n stages of $a \times b$ cross-bar switches. (Omega, baseline and indirect N -cube networks use $a = b = 2$.) The destination address is specified in base b numbers and the destination tag self-routing algorithm applies. Each destination digit has a value from 0 to $b - 1$ and selects one of b outputs of the $a \times b$ cross-bar element. An example of a Delta network is shown in Figure 8.12.

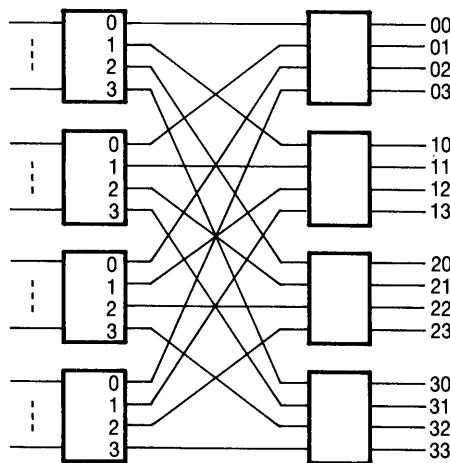


Figure 8.12 Delta network (base-4)

The stage to stage link pattern is a four-shuffle in this example. The destination tag self-routing networks have been further generalized into the *generalized shuffle network* (GSN) (Bhuyan and Agrawal, 1983). The GSN uses a shuffle network pattern constructed from arbitrary number system radices. An example is shown in Figure 8.13. Different radices can be used at each stage.

Note that now the basic 2×2 cell is not necessarily employed. Some studies have indicated that better performance/cost might be achieved by, for example, using 4×4 networks. In all destination tag routing networks (baseline, Omega, n -cube, and all networks that come under generalized networks) there can be only one route from each input to each output. Hence the networks are not resilient to cell failures. Extra stages can be introduced, as shown in Figure 8.14 to provide more than one path from input to output. This method has been studied by Raghavendra and Varma (1986).

8.4.4 Bandwidth of multistage networks

We derived the bandwidth of a single cross-bar switch as:

$$BW = m(1 - (1 - r/m)^p)$$

It follows that for a multistage network composed of stages of $a \times b$ cross-bar switches (Delta, GSN etc.) the number of requests that are accepted and passed on to the next stage is given by:

$$b(1 - (1 - r_0/b)^a)$$

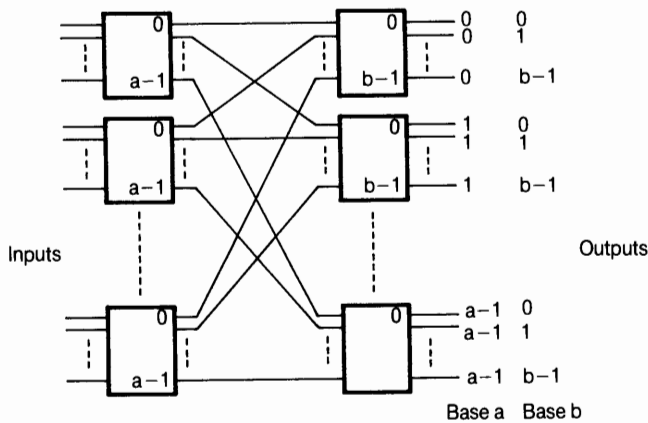


Figure 8.13 Generalized shuffle network stage

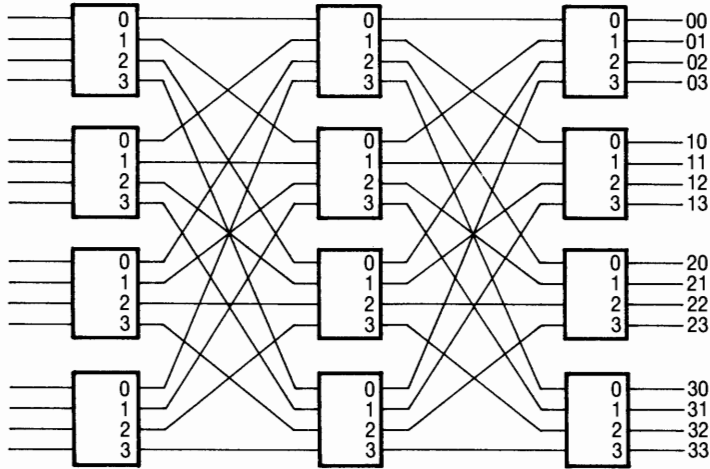


Figure 8.14 Extra stage Delta network

where r_0 is the request rate at the input of the first stage. The number of requests on any one of the b output lines of the first stage is given by:

$$r_1 = 1 - (1 - r_0/b)^a$$

These requests become the input to the next stage, and hence the number of requests at the output of the second stage is given by:

$$r_2 = 1 - (1 - r_1/b)^a$$

Hence the number of requests passed on to the output of the final stage can be found by recursively evaluating the function:

$$r_i = 1 - (1 - r_{i-1}/b)^a$$

for $i = 1$ to n , where n is the number of stages in the network, and $r_0 = r$. The bandwidth is given by:

$$\text{BW} = b^n r_n$$

as there are b^n outputs in all; there are a^n inputs. The probability that a request will be accepted is given by:

$$P_A = \frac{b^n r_n}{a^n r}$$

The derivation given is due to Patel (1981) in connection with Delta networks. Figure 8.15 shows the bandwidth and probability of acceptance of Omega networks (Delta network with $a = b = 2$) compared to single stage $N \times N$ cross-bar switch networks, where $N = 2^n$. Note that the number of stages in the $2^n \times 2^n$ multistage network is $\log_2 N$ and this can be significant, i.e. for $N = 4096$, there are twelve stages.

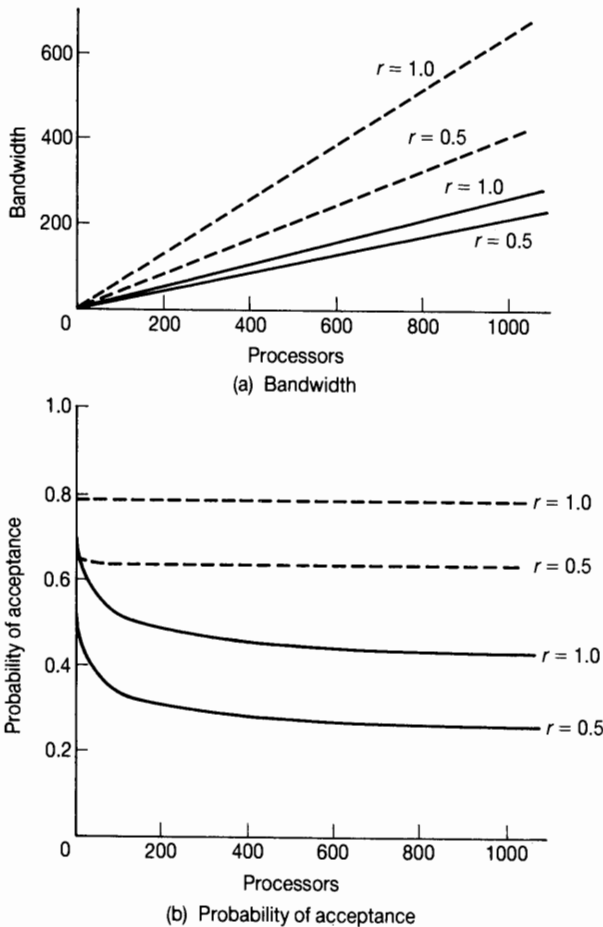


Figure 8.15 Performance of multistage networks (— Omega, --- full cross-bar switch) (a) Bandwidth (b) Probability of acceptance

8.4.5 Hot spots

Though memory references in a shared memory multiprocessor might be spread across a number of memory locations, some locations may experience a disproportionate number of references, especially when used to store locks and synchronization variables. These shared locations have been called *hot spots* by Pfister and Norton (1985). When a multistage interconnection network is used between the memory and processors, some paths between processors and memories are shared. Accesses to shared information can cause widespread contention in the network, as the contention at one stage of the network can affect previous stages. Consider a multistage network with request queues at the input of each stage. A hot spot in memory occurs and the last stage request queue fills up. Next, requests entering the inputs of the stage become blocked and the queues at this stage fill up. Then requests at the inputs of previous stages become blocked and the queues fill up, and so on, if there are more stages. This effect is known as *tree saturation* and also blocks requests not even aimed for the hot spot. The whole network can be affected.

Pfister and Norton (1985) present the following analysis to highlight the effect of hot spots. Suppose there are N processors and N shared memory modules, and the memory request rate is r . Let the fraction of these requests which are for hot spots be h . Then the number of hot-spot requests directed to the hot-spot memory is Nrh . The number of remaining non-hot-spot requests directed to the memory module is $Nr(1-h)/N = r(1-h)$ assuming that these requests are uniformly distributed among all memory modules. The total number of requests for the memory module is $Nrh + r(1-h) = r(h(N-1) + 1)$. The asymptotically maximum number of requests that can be accepted by one memory module is 1. Hence the asymptotically maximum number of accepted requests is $r/r(h(N-1) + 1) = 1/(h(N-1) + 1)$. Hence the maximum bandwidth is given by:

$$BW = N/(h(N-1) + 1)$$

This equation is plotted in Figure 8.16. We see that even a small fraction of hot-spot requests can have a profound effect on the bandwidth. For example, with $h = 0.1$ per cent, the bandwidth is reduced to 500 with 1000 processors. The request rate, r , has no effect on the bandwidth, and for large numbers of processors (P), the bandwidth tends to $1/h$. For example, when $h = 1$ per cent, the bandwidth is limited to 100 irrespective of the number of processors.

Two approaches have been suggested to alleviate the effects of hot spots, namely:

1. Software combining trees.
2. Hardware combining circuits.

In the software approach, operations on a single variable are broken down into operations which are performed separately so as to distribute and reduce the hot spots. The operations are arranged in a tree-like manner and results are passed along

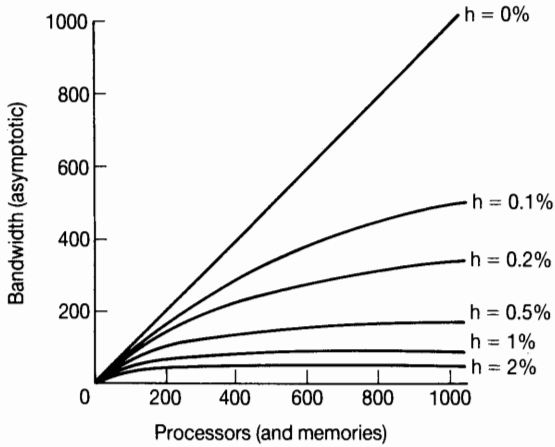


Figure 8.16 Asymptotic bandwidth in presence of hot spots

the tree to the root. Further information on the software approach can be found in Yew, Tzeng and Lawrie (1987).

In the hardware approach, circuits are incorporated into the network to recognize requests for shared locations and to combine the data access. In one relatively simple hardware combining network, read accesses to the same shared memory location are recognized at the switching cells and the requests combined to produce one read request to the memory module. The returning data is directed through the network to all required destinations.

Since shared variables are often used as synchronization variables, synchronization operations can be combined. The *fetch-and-add operation* suggested by Gottlieb *et al.* (1983) for combining networks returns the original value of a stored variable and adds a constant to the variable as specified as an operand. The addition is performed by a cell within the network. When more than one such operation is presented to the network, the network recognizes the operations and performs additions, leaving the memory to return the original value through the network and be presented with one final store operation. The network will modify the value returned to give each processor a value it would have received if the operations were performed serially.

An example of fetch-and-add operations in a multistage network is shown in Figure 8.17. Three fetch-and-add operations are received from three processors to operate upon memory location M:

Processor 1	$f\text{-}\&\text{-}a\ M, +x$
Processor 2	$f\text{-}\&\text{-}a\ M, +y$
Processor 3	$f\text{-}\&\text{-}a\ M, +z$

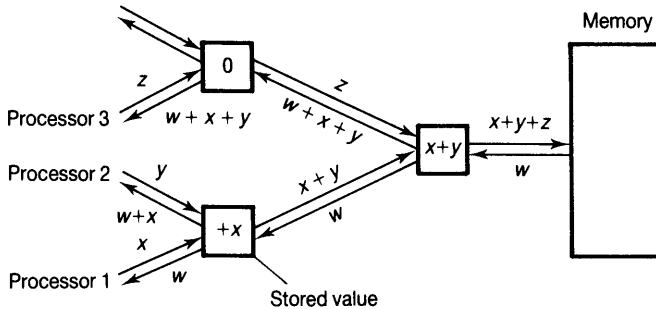


Figure 8.17 Fetch-and-add operations in a multistage network

Suppose the original value stored in M is w . As requests are routed through the network, individual cells perform additions and store one of the increments internally. In Figure 8.17, the first two requests are routed through the same cell and x and y are added together to be passed forward, to be added to the z from the third operation. The result, $x + y + z$, is presented to the memory and added to the stored value, giving $w + x + y + z$ stored in the memory. The original value, w , is passed back through the network. At the first cell encountered, $x + y$ had been stored and this is added to the w to give $w + x + y$, which is routed towards processor 3, and w is routed towards processors 1 and 2. In this cell, x had been stored, and is added to the w to give $w + x$, which is routed to processor 2, and w is routed to processor 1. Hence the three processors receive w , $w + x$ and $w + x + y$ respectively, which are the values they might have received had the operations been performed separately (actually the values if the operations were in the order: first processor 1, then processor 2 and then processor 3).

8.5 Overlapping connectivity networks

In this section we will introduce a class of networks called *overlapping connectivity networks* (Wilkinson, 1989). These networks have the characteristic that each processor can connect directly to a group of memory modules and processors, and to other groups through intermediate processors. Adjacent interconnection groups include some of the same memories and processors. The networks are attractive, especially for a very large number of processors which cannot be provided with full connectivity but need to operate with simultaneous requests to locally shared memory or by communication between processors. Applications for cascaded/overlapping connectivity networks include image processing, neural computers and dataflow computers.

8.5.1 Overlapping cross-bar switch networks

Two forms of an overlapping connectivity “rhombic” cross-bar switch scheme are shown in Figure 8.18. In Figure 8.18(a) each memory module has two ports, and processors can access whichever side the processor buses connect. The buses form rings by connecting one edge in the figure to the opposite edge, and the scheme expands to any size. With four buses, as shown in the figure, processor P_i can connect to memory modules $M_{i-3}, M_{i-2}, M_{i-1}, M_i, M_{i+1}, M_{i+2}, M_{i+3}$ and M_{i+4} using one of the two ports on each memory, for all i where M_i is the memory to the immediate left of processor P_i . Hence, each processor has an overlapping range of eight memory modules. In the general case of b vertical and b horizontal buses in each group, processor P_i can connect to memory modules, $M_{i-b+1} \dots M_{i-1}, M_i, M_{i+1} \dots M_{i+b}$, i.e. $2b$ memory modules. Connections from processor to memory modules are

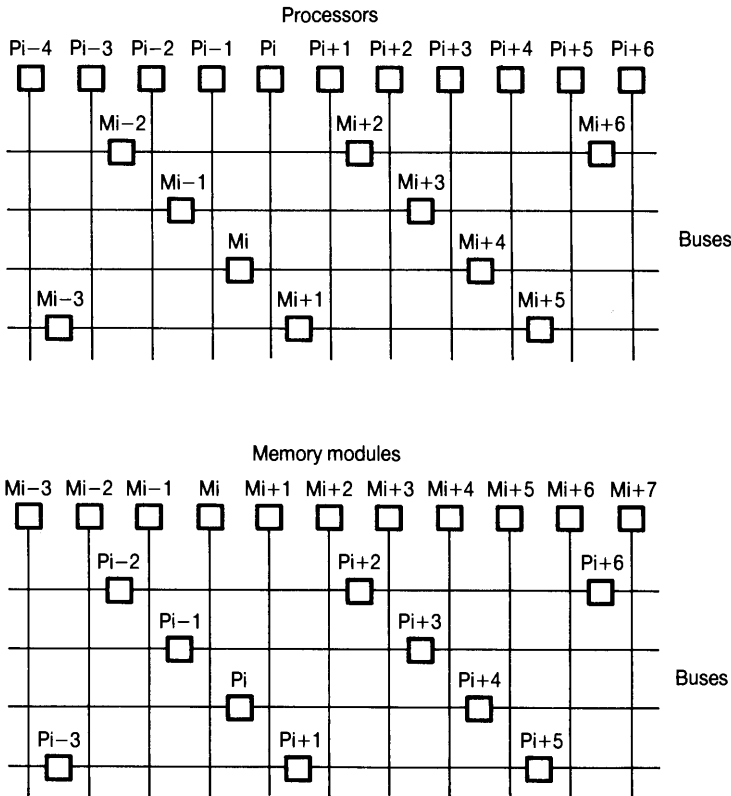


Figure 8.18 Cross-bar switch with overlapping connectivity (a) With two-port memory (b) With two-port processors

made with one cross-bar switch. Since two memories are accessed via each bus, there will be bus contention as well as memory contention. The bus contention could be removed by providing separate buses for the memory modules to the right and left of processors, but this would double the number of switches and buses. We shall assume only one bus providing access to two memory modules and separate memory addresses used to differentiate between the memory modules.

Let the total number of processors and memory modules in the system be P and M respectively, and the number of processors and memory modules in each section be p and m respectively. Then, Mb switches are needed in the cascaded networks compared to MP in a cross-bar switch (M^2 in a square switch).

In Figure 8.18(b), single port memory modules are used, together with processors having access to two buses. With four buses, as shown in the figure, all processors can connect to four memory modules on each side of the processor, or to $2b$ memory modules when there are b buses. There are $2b-2$ memory modules common to two adjacent sets of reachable elements, as in Figure 8.18(a). Note that not all requests can be honored because the corresponding bus may be used to honor another request to a different memory module, i.e. the system has bus contention because two processors share each bus. The bus arbitration might operate by memory module arbiters independently choosing a request from those pending, and when two requests which require the same bus are chosen, only one is accepted. Ideally, the arbitration circuitry should consider all requests pending before making any selection of requests, so that alternative selections can be made to avoid bus contention when possible.

Bandwidth

The bandwidth of the networks in Figure 8.18 with one stage (i.e. a single stage "rhombic" cross-bar switch network with circular bus connections), can be derived in a similar fashion to a full cross-bar switch network and leads to:

$$BW = M(1 - (1 - r/m)^m)$$

where M is the total number of memory modules, m is the number of memory modules reached by each processor, and there are the same number of processors as memories. Figure 8.19 shows the bandwidth function plotted against a range of requests for a single stage network, and simulation results when rejected requests are resubmitted until satisfied.

The bandwidth of the unidirectional cascaded rhombic cross-bar network can be derived by deducing the probability that a request has been issued for a memory in an immediate group, r_m say, and by substituting r_m for r in the previous equation. Suppose that each processor generates an output request from either an internal program queue or from an input buffer holding requests passed on from adjacent processors, and that the program queue has priority over the input buffer. As a first approximation, we can consider the program queue from the nearest processor, and then if no program requests present, the program queue from the next processor in

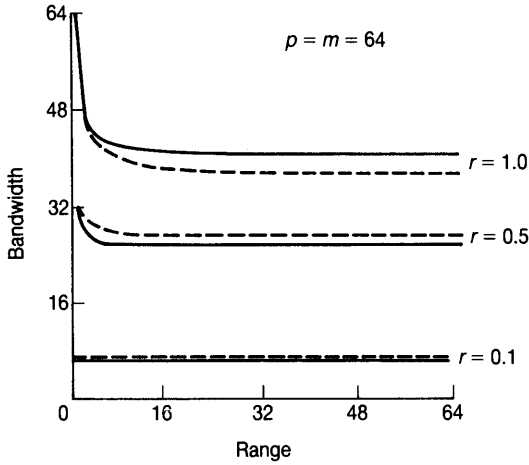


Figure 8.19 Bandwidth of single stage rhombic cross-bar network (--- analysis, — simulation)

the previous cycle is passed forward, then the next processor in the cycle before. This leads to:

$$r_m = r/g + (1-r)r/g + (1-r)^2r/g \cdots (1-r)^{g-1}r/g$$

$$= (1 - (1-r)^g)/g$$

and hence:

$$BW = M(1 - (1 - (1-r)^g)/gm)^m$$

where requests from each processor extend over g groups of memories. This equation ignores queuing, but has been found to compare reasonably well with simulation results of the network. Figures 8.20(a) and (b) show the bandwidth of the cascaded network against range and against number of buses respectively. Simulation results are also shown.

The overlapping connectivity cross-bar switch network can be expanded into two or higher dimensional networks. A two-dimensional network is shown in Figure 8.21. The processors (or processing elements) are identified by the tuple (i, j) along the two diagonals. Each processor in Figure 8.21 can reach twelve other processors with an overlapping connectivity. P_{ij} can reach $P_{i-1, j-1}$, $P_{i, j-1}$, $P_{i+1, j-1}$, $P_{i+2, j-1}$, $P_{i-2, j}$, $P_{i-1, j}$, $P_{i+1, j}$, $P_{i+2, j}$, $P_{i-2, j+1}$, $P_{i-1, j+1}$, $P_{i, j+1}$, $P_{i+1, j+1}$, via horizontal and vertical buses. The scheme can be expanded to provide more processors within each group. In the general case, if each bus has c switching elements, $4c+4$ processors can be reached

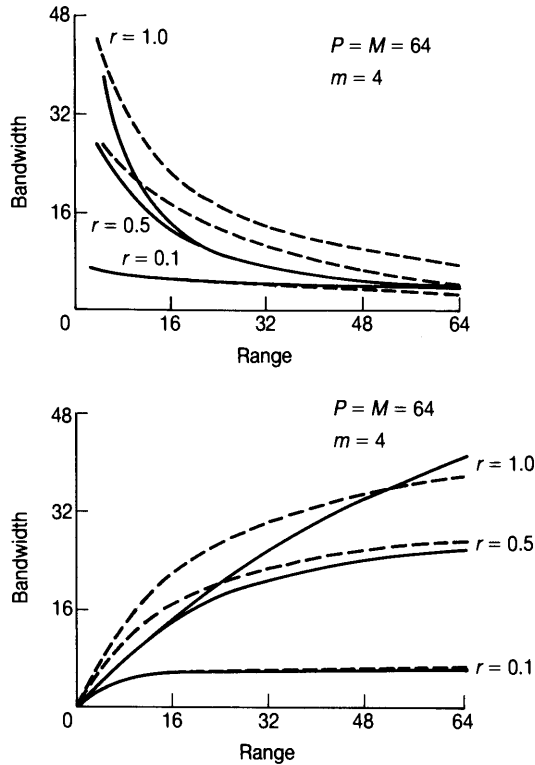


Figure 8.20 Bandwidth of cascaded rhombic cross-bar network (— simulation, --- analysis) (a) Bandwidth against range of requests (b) Bandwidth against number of buses

by any processor (with edges wrapping round). The switch points could be three-state switches providing left, right and cross-over paths. However, two-state switches providing cross-over and either right or left turn are sufficient. By always crossing over or making one a right turn (say), a path will be established between two processors.

8.5.2 Overlapping multiple bus networks

Figure 8.22 shows two overlapping bus configurations. In Figure 8.22(a) there are four buses with four processors connecting to the buses. As in all multiple bus systems, two connections need to be made for each path. Under these circumstances, with four buses, processor P_i can connect to a group of processing elements to the

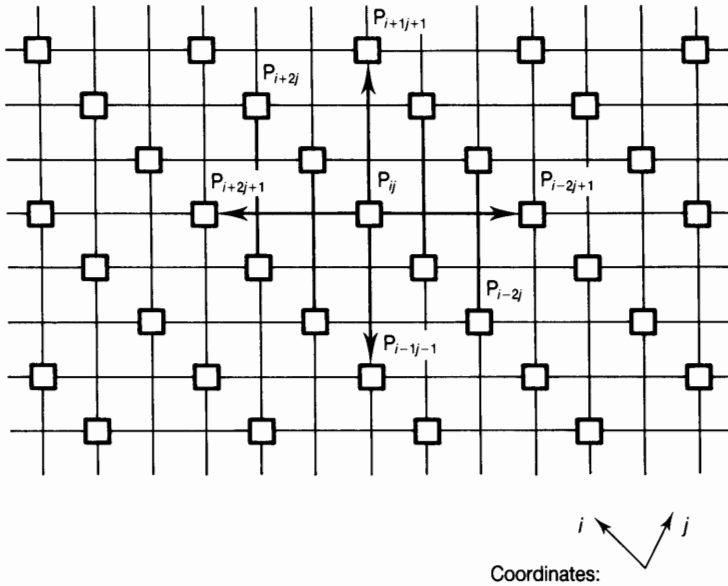


Figure 8.21 Two-dimensional scheme with overlapping connectivity

immediate left, P_{i-3} , P_{i-2} and P_{i-1} , and to the immediate right, P_{i+1} , P_{i+2} and P_{i+3} , for all i . P_{i-3} can be reached through one bus, P_{i-2} can be reached through two buses, P_{i-1} through three buses, P_{i+1} through three buses, P_{i+2} through two buses and P_{i+3} through one bus, for all i . As the processor to be reached is further away, there are fewer buses available and consequently less likelihood of reaching the processor. In the general case, processor P_i can connect to processors $P_{i-b+1} \dots P_{i-1}$, $P_{i+1} \dots P_{i+b-1}$, or $2(b-1)$ other processors. There are $b - 1$ buses available to connect processors P_{i-1} and P_{i+1} and a linearly decreasing number of buses for more distant processors, which is an appropriate characteristic. The scheme as described is appropriate to interconnect processors with local memory. Global memory could be incorporated into the interconnection network by replacing some processors with memory modules.

An overlapping connectivity multiple bus scheme with fewer buses than elements in each group and both processors and memory modules is presented in Figure 8.22(b). The processors are fully connected to the buses and the memory is partially connected to the buses. (Memory modules fully connected and processors partially connected is also possible.) Since each group of memory modules connect to two adjacent sets of buses, these modules can be shared between adjacent groups of processors. The

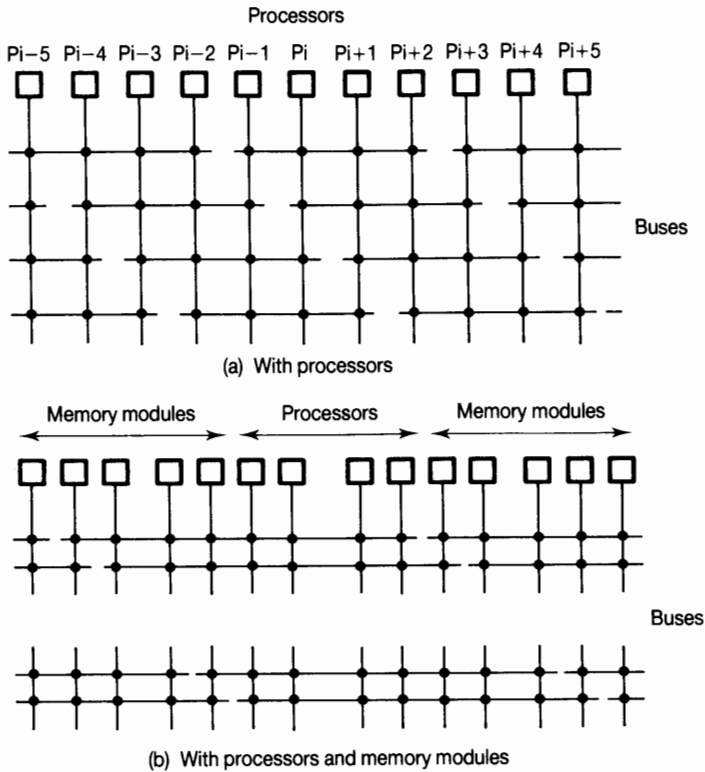


Figure 8.22 Multiple bus scheme with overlapping connectivity
 (a) With processors (b) With processors and memory modules

scheme can be considered as composed of a number of rhombic cross-bar switches, cascaded together, similar to Lang's simplification (Section 8.1). A suitable rhombic configuration would be eight processors completely connected to eight buses and sixteen memory modules connecting to the buses in a rhombic pattern.

In Figure 8.22(b), the memory modules form the Lang rhombic pattern but are divided by processors which are fully connected to the buses. Hence, the same connectivity is possible between the processors and memory modules on both sides of the processors (given suitable b and m to satisfy Lang's conditions). If we ignore contention arising when requests from adjacent rhombic groups are made to the same shared memory module, the bandwidth can be derived from the bandwidth of a fully connected multiple bus system.

8.6 Static interconnection networks

8.6.1 General

Static interconnection networks are those which allow only direct fixed paths between two processing elements (nodes). Each path could be unidirectional or bidirectional. In the following, we will generally assume links capable of bidirectional transfers when counting the number of links. The number of links would, of course, be double if separate links were needed for each direction of transfer. Static interconnection networks would be particularly suitable for regular processor–processor interconnections, i.e. in which all the nodes are processors and processors could process incoming data or pass the data on to other processors. We will find that static networks are used in multiple processor VLSI structures described in Chapter 9.

In general, the number of links in a static interconnection network when each element has the same number of links is given by $(\text{number of nodes}) \times (\text{number of links of a node}) / 2$, the factor of $1/2$ due to each path being used in two nodes.

8.6.2 Exhaustive static interconnections

In *exhaustive* or *completely connected networks*, all nodes have paths to every other node. Hence n nodes could be exhaustively interconnected with $n - 1$ paths from each node to the other $n - 1$ node. There are $n(n - 1)/2$ paths in all. If each direction of transfer involves a separate path, there are $n(n - 1)$ paths. Exhaustive interconnection has application for small n . For example, a set of four microprocessors could reasonably be exhaustively interconnected using three parallel or serial ports attached to each microprocessor. All four processors could send information simultaneously to other processors without contention. The absence of contention makes static exhaustive interconnections particularly attractive, when compared to the non-exhaustive shared path connection schemes to be described. However, as n increases, the number of interconnections clearly becomes impractical for economic and engineering reasons.

8.6.3 Limited static interconnections

Interconnections could be limited to, say, a group of the neighboring nodes; there are numerous possibilities. Here we will give some common examples.

Linear array and ring structures

A one-dimensional *linear array* has connections limited to the nearest two neighbors and can be formed into a *ring* structure by connecting the free ends as shown in Figure 8.23. The interconnection might be unidirectional, in which case the former creates a linear pipeline structure; alternatively the links might be bidirectional. In

either case, such arrays might be applicable to certain computations. Each node requires two links, one to each neighboring node, and hence an n node array requires n links. In the *chordal ring* network, shown in dotted lines, each node connects to its neighbors as in the ring, but also to one node three nodes apart. There are now three links on each node and $3n/2$ paths in all.

Two-dimensional arrays

A two-dimensional array or *near-neighbor mesh* can be created by having each node in a two-dimensional array connect to all its four nearest neighbors, as shown in Figure 8.24. The free ends might circulate back to the opposite sides. Now each node has four links and there are $2n$ links in all. This particular network was used in the Iliac IV computer with an 8×8 array, and is popular with VLSI structure because of the ease of layout and expandability.

The two-dimensional array can be given extra diagonal links. For example, one, two, three or all four diagonal links can be put in place, allowing connections to diagonally adjacent nodes. Each node has eight links and the network has $4n$ links.

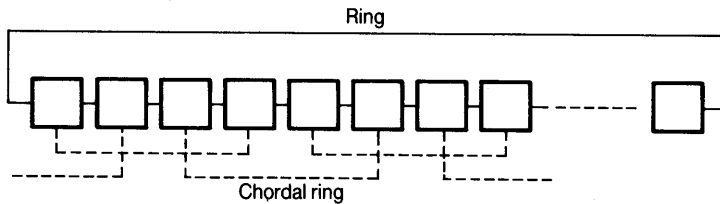


Figure 8.23 Linear array

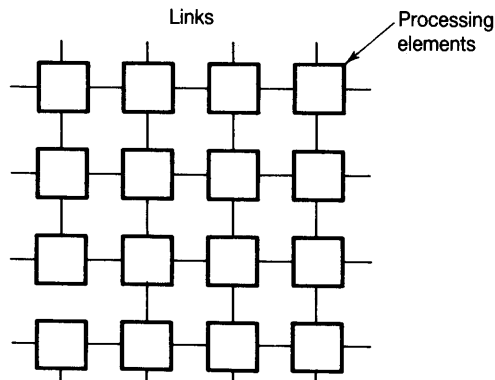


Figure 8.24 Two-dimensional array

In Figure 8.25, each node has six links and there are $3n$ links in the network. This network is also called a *systolic array*, as it can be used in systolic multiprocessors.

Star network

The *star* connection has one node into which all other nodes connect. There are $n - 1$ links in all, i.e. the number of links grows proportional to n , which is generally the best one could hope for, and any two nodes can be reached in two paths. However, the central node must pass on all transfers to required destinations and substantial contention or bottleneck might occur in high traffic. Also, should the central node fail, the whole system would fail. This might be the case in other networks if additional mechanisms were not incorporated into the system to route around faulty nodes but, given alternative routes, fault tolerance should be possible. Duplicated star networks would give additional routes.

Tree networks

The *binary tree* network is shown in Figure 8.26. Apart from the root node, each node has three links and the network fans out from the root node. At the first level below the root node there are two nodes. At the next level there are four nodes, and at the j th level below the root node there are 2^{j-1} nodes (counting the root node as level 0). The number of nodes in the system down to the j th level is:

$$\begin{aligned}
 n = N(j) &= 1 + 2 + 2^2 + 2^3 \dots 2^{j-1} \\
 &= \frac{(2^j - 1)}{(2 - 1)} \\
 &= 2^j - 1
 \end{aligned}$$

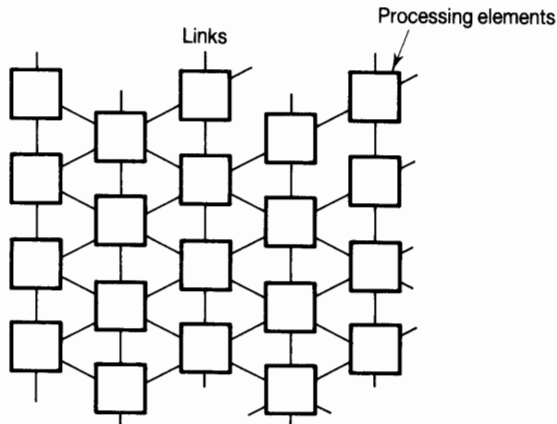


Figure 8.25 Hexagonal configuration

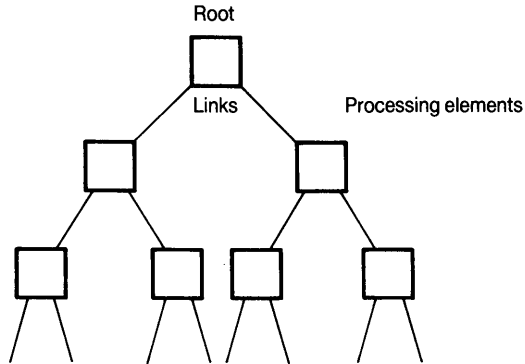


Figure 8.26 Tree structure

and the number of levels $j + 1 = \log_2(n + 1) + 1$. This network requires $n - 1$ links. (The easiest way to prove this expression is to note that every additional node except the root node adds one link.)

The tree network need not be based upon the base two. In an m -ary tree, each node connects to m nodes beneath it and one from above. The number of nodes in this system down to the j th level is:

$$\begin{aligned} n = N(j) &= 1 + m + m^2 + m^3 \dots m^{j-1} \\ &= \frac{(m^j - 1)}{(m - 1)} \end{aligned}$$

and the number of levels $j + 1 = \log_m(n+1) + 1$. Again, the network requires $n - 1$ links, but fewer intermediate nodes are needed to connect nodes as the value of m is increased.

The binary and general m -ary tree networks are somewhat similar to the star network in terms of routing through combining nodes. The root node is needed to route from one side of the tree to the other. Intermediate nodes are needed to route between nodes which are not directly connected. This usually means travelling from the source node up the tree until a common node in both paths from the route node is reached and then down to the destination node.

The networks so far described are generally regular in that the structure is symmetrical. In irregular networks, the symmetry is lost in either the horizontal or vertical directions, or in both directions. An irregular network can be formed, for example, by removing existing links from a regular network or inserting extra links. The binary tree network is only regular if all nodal sites are occupied, i.e. the tree has 1 node, 3 nodes, 7 nodes, 15 nodes, 31 nodes, etc.

Hypertree networks

In the *hypertree network* (Goodman and Séquin, 1981) specific additional links are put in place directly between nodes to reduce the “average distance” between nodes. (The average distance is the average number of links that must be used to connect two nodes, see page 287.) Each node is given a binary address starting at the root node as node 1, the two nodes below it as nodes 2 and 3, with nodes 4, 5, 6 and 7 immediately below nodes 2 and 3. Node 2 connects to nodes 4 and 5. Node 3 connects to nodes 6 and 7, and so on. The additional links of the hypertree connect nodes whose binary addresses differ by only one bit (a Hamming distant of one). Notice that the hypertree network is not regular.

Cube networks

In the *3-cube network*, each node connects to its neighbors in three dimensions, as shown in Figure 8.27. Each node can be assigned an address which differs from adjacent nodes by one bit. This characteristic can be extended for higher dimension *n-cubes*, with each node connecting to all nodes whose addresses differ in one bit position for each dimension. For example, in a 5-cube, node number 11101 connects to 11100, 11111, 11001, 10101 and 01101. The number of bits in the nodal address is the same as the number of dimensions. *N-cube* structures, particularly higher dimensional *n-cubes*, are commonly called *hypercube networks*. The *generalized hypercube* (Bhuyan and Agrawal, 1984) can use nodal address radices other than 2, but still uses the characteristic that addresses of interconnected nodes differ in each digit position. The (binary) hypercube is an important interconnection network; it has been shown to be suitable for a very wide range of applications. Meshes can be embedded into a hypercube by numbering the edges of the mesh in Gray code. In Chapter 9, we will describe message-passing multiprocessor systems using hypercube networks.

Numerous other networks have been proposed, though in most cases they have not been used to a significant extent. In the *cube connected cycles network*, 2^k nodes divided into $2^{k-r} \times 2^r$ nodes are connected such that 2^r nodes form a group at the vertices of a (2^{k-r}) -cube network. Each group of 2^r nodes is connected in a loop,

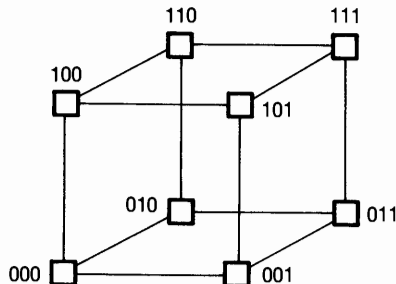


Figure 8.27 Three-dimensional hypercube

with one connected to each of the two neighboring nodes and also one link to a corresponding node in another dimension.

Though we have described direct link static networks in terms of communicating nodes, some networks could be used for shared memory systems. For example, the nodes in the network could contain shared memory which can be reached by processors in other nodes using the links that operate as buses. A possibility is to have multiple buses which can extend through to other nodes. This can, for example, lead to an *overlapping connectivity mesh network*. In a *spanning bus hypercube network*, each node connects to one bus in each dimension of the network. For a two-dimensional network, nodes connect to two buses or two sets of buses that stretch in each of the two dimensions. For a three-dimensional network, each node connects to three buses.

8.6.4 Evaluation of static networks

Clearly, there are numerous variations in limited interconnections, some of which suit particular computations. With limited interconnections, some transfers will require data to pass through intermediate nodes to reach the destination node. Whatever the limited connection network devised, there must be a means of locating the shortest route from the source to the destination. A routing algorithm which is easy and fast to implement is preferable.

Request paths

A critical factor in evaluating any interconnection network is the number of links between two nodes. The number of intermediate nodes/links is of interest because this gives the overall delay and the collision potential. The *average distance* is defined as (Agrawal *et al.*, 1986):

$$\text{Average distance} = \frac{\sum_{d=0}^{\text{Max}} dN_d}{N-1}$$

where N_d is the number of nodes separated by d links. Max is the maximum distance necessary to interconnect two nodes (not the maximum distance as this would be infinity) and N is the number of nodes. For any particular network, interconnection paths for all combinations of nodal connections would need to be computed, which is not always an easy task. Notice that the average distance formulae may not be the actual average distance in an application.

Number of links

Another factor of interest is the number of links emanating from each node, as this gives the node complexity. The number of links is usually fairly obvious from the

288 Shared memory multiprocessor systems

network definition. With an increased number of links, the average distance is shorter; the two are interrelated. A *normalized average distance* is defined as:

$$\text{Normalized average distance} = \text{average distance} \times \text{links/node}$$

which gives an indication of network performance taking into account its complexity. The *message density* has been defined as:

$$\text{Message density} = \frac{\text{Average distance} \times \text{number of nodes}}{\text{Total number of links}}$$

In a limited static interconnection network, distant nodes can be reached by passing requests from a source node (processor) through intermediate nodes (called "levels"). Links to four neighbours reach $4(2i-1)$ nodes at the i th level from the node. For hexagonal groups (Figure 8.25), there are $6i$ nodes at the i th level, i.e. the number of nodes at each level increases proportionally, and the number of nodes that can be reached, n , is given by:

$$n = \sum_{i=1}^L 6i = 3L(L+1)$$

where L is the number of levels. In the hexagonal configuration, every node at each level can be reached by one path from the previous level (this is not true for the square configuration). The average number of levels to reach a node, and hence the average number of requests in the system for each initial nodal request, is given by:

$$av = \sum_{i=1}^L (6i^2)/n$$

To place an upper bound on the number of simultaneous requests in the system, requests from one processor to another can be passed on through a fixed number of nodes.

Bandwidth of static networks

We have seen that the performance of dynamic networks is often characterized by their bandwidth and also probability of acceptance. The bandwidth and the probability of acceptance metric can be carried over to static networks, though this is rarely done. One example of a direct binary n -cube (hypercube) is given in (Abraham and Padmanabhan (1989). We can make the following general analysis for any static network.

Suppose that each node has *input requests* and can generate *output requests* either by passing input requests onwards or from some internal program (*internal*

requests). Let the probability that a node can generate an internal request for another node be r . The requested node might be one directly connected to it or it might be one which can be reached through intermediate nodes. In the latter case, the request must be presented to the intermediate nodes as external requests, but these nodes might also have internally generated requests and only one request can be generated from a node, irrespective of how many requests are present. There could be at most one internal request and as many external requests as there are links into the node. Let r_{out} be the probability that a node generates a request (either internally or passes on an external request) and r_{in} be the probability that a node receives an external request. Some external requests will be for the node and only a percentage, say A , will be passed onwards to another node. Incorporating A , we get:

$$r_{\text{out}} = r + Ar_{\text{in}}(1 - r)$$

and the bandwidth given by:

$$\text{BW} = (1 - A)r_{\text{in}}N$$

where there are N nodes. The value for A will depend upon the network.

The probability that an external request is received by node i from a node j will depend upon the number of nodes that node j can request, i.e. the number of nodes connected directly to node j , and the probability is given by r_{out}/n , where n nodes connect directly to node j and all links are used. We shall assume that all nodes have the same number of links to other nodes and, for now, all are used. The probability that node j has not requested node i is given by $(1 - r_{\text{out}}/n)$. The probability that no node has requested node i is given by $(1 - r_{\text{out}}/n)^n$. The probability that node i has one or more external requests at its inputs is given by:

$$r_{\text{in}} = 1 - (1 - r_{\text{out}}/n)^n$$

The probability that a node generates a request in terms of the probability of an internal request and the number of nodes directly connected (and communicating) to the node is given by:

$$r_{\text{out}} = r + A(1 - r)(1 - (1 - r_{\text{out}}/n)^n)$$

which is a recursive formula which converges by repeated application. A suitable initial value for r_{out} is r , r_{out} being some value in excess of r .

The derivation assumes that an external request from node j to node i could be sent through node i and back to node j , which generally does not occur, i.e. an external request passing through node i can only be sent to $n-1$ nodes at most, and more likely only to nodes at the next level in the sphere of influence (up to two nodes in the hexagonal configuration) whereas internal requests will generally have an equal probability of requesting any of the nodes connected.

PROBLEMS

8.1 Suggest relative advantages of the cross-bar switch system with central control and the cross-bar switch system without central control.

8.2 Design a 16×16 cross-bar switch multiprocessor system using microprocessors (any type) for the master–slave mode of operation. Give details at the block diagram level of the major components.

8.3 Repeat the design in Problem 8.2 for a system without central control.

8.4 Derive an expression for the probability that i requests are made for a particular memory, given that the probability that a request made by one processor is r and there are m memories. (Clue: look at the Bernouli formula.) Using this expression, derive the general expression for the bandwidth of a $p \times m$ cross-bar switch system.

8.5 Derive an expression for the bandwidth of a cross-bar switch system, given that each processor has an equal probability of making a request for any memory or of not making a request at all.

8.6 Design an 8-bus multiple bus multiprocessor system using microprocessors (any type) for a system without a master processor. Give details at the block diagram level of the major components.

8.7 Suggest how a multiple bus system could be designed for a master–slave operation. Are there any advantages of such systems?

8.8 Derive an expression for a multiple bus system in which the bus arbitration is performed before the memory arbitration. Show that this arrangement leads to a lower bandwidth than the normal method of having memory arbitration before the bus arbitration.

8.9 Figure 8.28 shows a combined cross-bar switch/shared bus system without central control. There are P processors and M memory modules in the system with p processors sharing each horizontal bus. Show that the bandwidth of the system is given by:

$$\text{BW} = M \left(1 - \left[1 - \frac{(1-r)^p}{M} \right]^{P/p} \right)$$

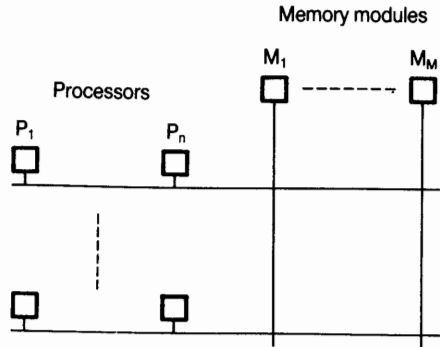


Figure 8.28 System for Problem 8.9

8.10 Design a non-blocking Clos network for sixty-four processors and sixty-four memories.

8.11 Identify relative advantages of multistage networks and single stage networks.

8.12 Ascertain all input/output combinations in an 8×8 single stage recirculating shuffle exchange network which require the maximum number of passes through the network.

8.13 How many stages of a multistage Omega network are necessary to interconnect 900 processors and 800 memories? What is the bandwidth when the request rate is 40 per cent? Make a comparison with a single stage cross-bar switch network.

8.14 Design the logic necessary with each cell in an 8×8 Omega network for self-routing.

8.15 Determine whether it is possible to connect input i to output i in an 8×8 Omega network for all i simultaneously.

8.16 Show that a three-stage indirect binary n -cube network and a three-stage Omega network are functionally equivalent.

8.17 Illustrate the flow of information in a three-stage multistage network with fetch-and-add operations, given that four processors execute the following:

292 Shared memory multiprocessor systems

Processor 1	f-&-a 120, 9
Processor 2	f-&-a 120, 8
Processor 3	f-&-a 120, 7
Processor 4	f-&-a 120, 6

8.18 Derive the average distance between two nodes in a three-dimensional hypercube.

8.19 Demonstrate how each of the following structures can be implemented on a hypercube network:

1. Binary tree structure.
2. Mesh network.

8.20 Derive an expression for the number of nodes that can be reached in a north-south-east-west nearest neighbor mesh network at the L th level from the node.

PART



***Multiprocessor
systems without
shared memory***

Message-passing multiprocessor systems

This chapter concentrates upon the design of multiprocessor systems which do not use global memory; instead each processor has local memory and will communicate with other processors via messages, usually through direct links between processors. Such systems are called *message-passing multiprocessors* and are particularly suitable when there is a large number of processors.

9.1 General

9.1.1 Architecture

The shared memory multiprocessors described in the previous chapters have some distinct disadvantages, notably:

1. They do not easily expand to accommodate large numbers of processors.
2. Synchronization techniques are necessary to control access to shared variables.
3. Memory contention can significantly reduce the speed of the system.

Other difficulties can arise in shared memory systems. For example, data coherence must be maintained between caches holding shared variables. Shared memory is, however, a natural extension of a single processor system. Code and data can be placed in the shared memory to be accessed by individual processors.

One alternative multiprocessor system to the shared memory system, which totally eliminates the problems cited, is to have only local memory and remove all shared memory from the system. Code for each processor is loaded into the local memory and any required data is stored locally. Programs are still partitioned into separate parts, as in a shared memory system, and these parts are executed concurrently by individual processors. When processors need to access information from other processors, or to send information to other processors, they communicate by sending messages, usually along direct communication links. Data words are not stored

globally in the system; if more than one processor requires the data, it must be duplicated and sent to all requesting processors.

The basic architecture of the message-passing multiprocessor system is shown in Figure 9.1. The message-passing multiprocessor consists of nodes, which are normally connected by direct links to a few other nodes. Each node consists of an instruction processor with local memory and input/output communication channels. The system is usually controlled by a host computer, which loads the local memories and accepts results from the nodes. For communication purposes, the host can be considered simply as another node, though the communication between the instruction processor nodes and the host will be slower if it uses a single globally shared channel (for example an Ethernet channel). There are no global memory locations. The local memory of each nodal processor can only be accessed by that processor and the local memory addresses only refer to the specific local memory. Each local memory may use the same addresses. Since each node is a self-contained computer, message-passing multiprocessors are sometimes called *message-passing multi-computers*.

The number of nodes could be as small as sixteen (or less), or as large as several thousand (or more). However, the message-passing architecture gains its greatest advantage over shared memory systems for large numbers of processors. For small multiprocessor systems, the shared memory system probably has better performance and greater flexibility. The number of physical links between nodes is usually between four and eight. A principal advantage of the message-passing architecture is that it is readily scalable and has low cost for large systems. It suits VLSI construction, with one or more nodes fabricated on one chip, or a few chips, depending upon the amount of local memory provided.

Each node executes one or more *processes*. A process often consists of sequential

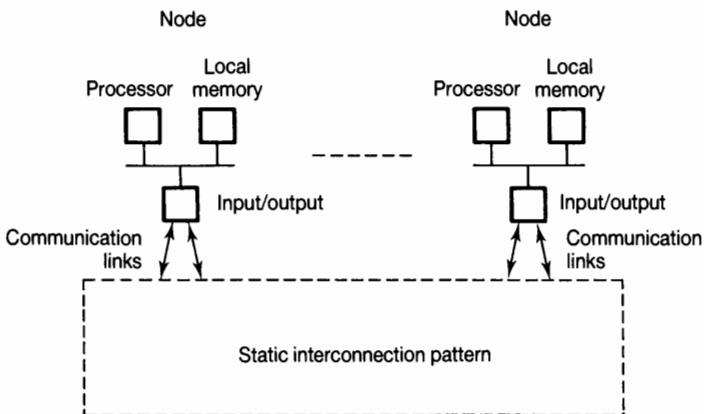


Figure 9.1 Message-passing multiprocessor architecture

code, as would be found on a normal von Neumann computer. If there is more than one process mapped onto one nodal processor, one process is executed at a time. A process may be descheduled when it is waiting for messages to be sent or received, and in the meantime another process started. Messages can be passed between processes on one processor using internal channels. Messages between processes in different processors are passed through external channels using physical communication links between processors. We will use the term *link* to refer to a physical communication path between a pair of processors. *Channel* refers to a named communication path either between processes in one processor or between processes on different processors.

Ideally, the process and the processor which will execute the process are regarded as completely separate entities, even at this level. The application problem is described as a set of communicating processes which is then mapped onto the physical structure of processors. A knowledge of the physical structure and composition of the nodes is necessary to plan an efficient computation.

The size of a process is determined by the programmer and can be described by its *granularity*:

1. Coarse granularity.
2. Medium granularity.
3. Fine granularity.

In coarse granularity, each process contains a large number of sequential instructions and takes a substantial time to execute. In fine granularity, a process might consist of a few instructions, even one instruction; medium granularity describes the middle ground. As the granularity is reduced, the process communication overhead usually increases. It is particularly desirable to reduce the communication overhead because of the significant time taken by a nodal communication. Message-passing multiprocessors usually employ medium/coarse granularity; fine granularity is possible and is found in dataflow systems. (Dataflow is described in Chapter 10). A fine grain message-passing system has been developed by Athas and Seitz (1988) after pioneering work by Seitz on medium grain message-passing designs, which will be described later. For fine grain computing, the overhead of message passing can be reduced by mapping several processes onto one node and switching from one process to another when a process is held up by message passing. The process granularity is sometimes related to the amount of memory provided at each node. Medium granularity may require megabytes of local memory whereas fine granularity may require tens of kilobytes of local memory. Fine grain systems may have a much larger number of nodes than medium grain systems.

Process scheduling is usually *reactive* – processes are allowed to proceed until halted by message communication. Then the process is descheduled and another process is executed, i.e. processes are message-driven in their execution. Processes do not commonly migrate from one node to another at run time; they will be assigned to particular nodes statically before the program is executed. The

programmer makes the selection of nodes. A disadvantage of static assignment is that the proper load sharing, in which work is fairly distributed among available processors, may be unclear before the programs are executed. Consideration has to be given to spreading code/data across available local memory given limited local memory.

Each node in a message-passing system typically has a copy of an operating system kernel held in read-only memory. This will schedule processes within a node and perform the message-passing operations at run time. The message-passing routing operations should have hardware support, and should preferably be done completely in hardware. Hardware support for scheduling operations is also desirable. The whole system would normally be controlled by a host computer system.

However, there are disadvantages to message-passing multiprocessors. Code and data have to be physically transferred to the local memory of each node prior to execution, and this action can constitute a significant overhead. Similarly, results need to be transferred from nodes to the host system. Clearly the computation to be performed needs to be reasonably long to lessen the loading overhead. Similarly, the application program should be computational intensive, not input/output or message-passing intensive. Code cannot be shared. If processes are to execute the same code, which often happens, the code has to be replicated in each node and sufficient local memory has to be provided for this purpose. Data words are difficult to share; the data would need to be passed to all requesting nodes, which would give problems of incoherence. Message-passing architectures are generally less flexible than shared memory architectures. For example, shared memory multiprocessors could emulate message passing by using shared locations to hold messages, whereas message-passing multiprocessors are very inefficient in emulating shared memory multiprocessor operations. Both shared memory and message-passing architectures could in theory perform single instruction stream–multiple data stream (SIMD) computing, though the message-passing architecture would be least suitable and would normally be limited to multiple instruction stream–multiple data stream (MIMD) computing.

9.1.2 Communication paths

Regular static direct link networks, which give local or nearest neighbor connections (as described in Section 8.6, page 283), are generally used for large message-passing systems, rather than indirect dynamic multistage networks. Some small dedicated or embedded applications might use direct links to certain nodes chosen to suit the message transfers of the application. Routing a message to a destination not directly connected requires the message to be routed through intermediate nodes.

A network which has received particular attention for message-passing multiprocessors is the direct binary hypercube, described in Section 8.6.3 (page 286). The direct binary hypercube network has good interconnection patterns suitable for a wide range of applications, and expands reasonably well. The interconnection pattern for binary hypercubes is defined by giving each node a binary address. Each

node connects to those nodes whose binary addresses differ by one bit only. Hence each node in an n -dimensional hypercube requires n links to other nodes. A six-dimensional hypercube is shown in Figure 9.2 laid out in one plane. Hypercube connections could be made in one backplane, as shown in Figure 9.3 for a

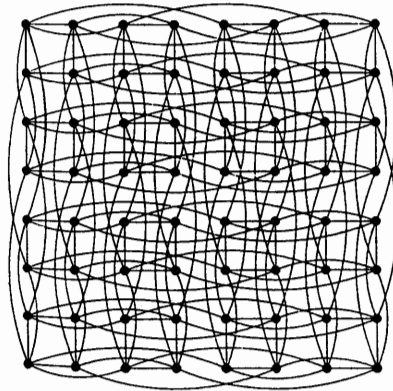


Figure 9.2 Six-dimensional hypercube laid out in one plane

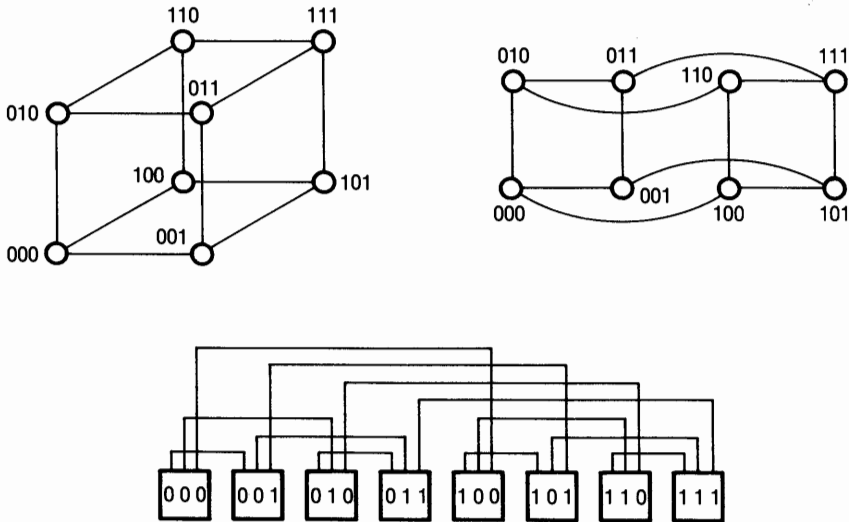


Figure 9.3 Three-dimensional hypercube (a) Interconnection pattern (b) Laid out in one plane (c) Connections along a backplane

three-dimensional hypercube. Nearest neighbor two-dimensional mesh networks are also candidates for message-passing systems, especially large systems.

The nodal links are bidirectional. The links could transfer the information one bit at a time (bit-serial) or several bits at a time. Complete words could be transmitted simultaneously. However, bit-serial lines are often used, especially in large systems, to reduce the number of lines in each link. For coarse grain computations, message passing should be infrequent and the bit-serial transmission may have sufficient bandwidth. The *network latency*, the time to complete a message transfer, has two components; first there is a path set-up time, which is proportional to the number of nodes in the path; second is the actual transmission time, which is proportional to the size of the message for a fixed link bandwidth. The link bandwidth should be about the same as memory bandwidth; a greater bandwidth cannot be utilized by the node. Since the message data can be more than one word, the links require DMA (direct memory access) capabilities.

Each process is given an identification number (process ID) which is used in the message-passing scheme. Message passing can use a similar format to computer network message passing. For example, messages consist of a header and the data; Figure 9.4 shows the format of a message. Because more than one process might be mapped onto a node, the process ID has two parts, one part identifying the node and one part the process within the node. The nodal part (physical address) of the ID is used to route the message to the destination node. The message type enables different messages along the same link to be identified.

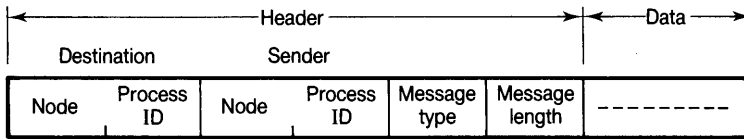


Figure 9.4 Message format

Messages may need to pass through intermediate nodes to reach their destination. Queues inside the nodes are used to hold pending messages not yet accepted. However, the messages may be blocked from proceeding by previous messages not being accepted, and would then become queued, until the queues become full and eventually the blockage would extend back to the source process. The order in which messages are sent to a particular process should normally be maintained, even when messages are allowed to take different routes to the destination. Of course, constraining the route to be the same for all messages between two processes simplifies maintaining message order.

Messages can be routed in hypercube networks according to the following algorithm, which minimizes the path distance. Suppose the current nodal address is $P = p_{n-1}p_{n-2} \dots p_1p_0$ and the destination address is $D = d_{n-1}d_{n-2} \dots d_1d_0$. The exclusive-OR function $R = P \oplus D$ is performed operating on pairs of bits, to obtain $R = r_{n-1}r_{n-2} \dots r_1r_0$.

Let r_i be the i th bit of R . The hypercube dimensions to use in the routing are given by those values of i for which $r_i = 1$. At each node in the path, the exclusive function $R = P \oplus D$ is performed. One of the bits in R which is 1, say r_k , identifies the k th dimension to select in passing the message forward until none of the bits are 1, and then the destination node has been found. The bits of R are usually scanned from most significant bit to least significant bit until a 1 is found. For example, suppose routing from node 5 (000101) to node 34 (100010) is sought in a six-dimensional hypercube. The route taken would be node 5 (000101) to node 21 (100101) to node 17 (100001) to node 19 (100011) to node 34 (100010). This hypercube routing algorithm is sometimes called the *e-cube routing algorithm*, or *left-to-right routing*.

Deadlock is a potential problem. Deadlock occurs when messages cannot be forwarded to the next node because the message buffers are filled with messages waiting to be forwarded and these messages are blocked by other messages waiting to be forwarded. Dally and Seitz (1987) developed the following solution to deadlock.

The interconnections of processing nodes can be shown by a directed graph, called an *interconnection graph*, depicting the communication paths. A *channel dependency graph* is a directed graph showing the route taken by a message for a particular routing function. In the channel dependency graph, the channels are depicted by the vertices of the graph and the connections of channels are depicted by the edges. A network is deadlock-free if there are no cycles in the channel dependency graph. Given a set of nodes $n_0, n_1 \dots n_{n-1}, n_n$ and corresponding channels $c_0, c_1 \dots c_{n-1}, c_n$, there are no cycles if messages are routed in decreasing order (subscript) of channel. Dally and Seitz introduced the concept of *virtual channels*. Each channel, c_i , is split into two channels, a low channel, c_{0i} , and a high channel, c_{1i} . For example, with four channels, c_0, c_1, c_2 and c_3 , we have the low virtual channels, c_{00}, c_{01}, c_{02} , and c_{03} , and the high channels c_{10}, c_{11}, c_{12} , and c_{13} . If a message is routed on high channels from a node numbered less than the destination node and to low channels from a node numbered greater than the destination node, there are no cycles and hence no deadlock.

Routing messages according to a decreasing order of dimension in a hypercube (left-to-right routing) is naturally deadlock-free as it satisfies the conditions without virtual channels.

9.2 Programming

9.2.1 Message-passing constructs and routines

Message-passing multiprocessor systems can be programmed in conventional sequential programming languages such as FORTRAN, PASCAL, or C, augmented with mechanisms for passing messages between processes. In this case, message-passing

is usually implemented using external procedure calls or routines, though statement extensions could be made to the language. Alternatively, special programming languages can be developed which enable the message passing to be expressed. Message-passing programming is not limited to message-passing architectures or even multiprocessor systems; it is done on single processor systems, for example between UNIX processes, and many high level languages for concurrent programming have forms of message passing (see Gehani and McGettrick (1988) for examples).

Message-passing language constructs

Programming with specially developed languages with message-passing facilities is usually at a much higher level than using standard sequential languages with message-passing routines. The source and destination processes may only need to be identified. For example, the construct:

```
SEND expression_list TO destination_identifier
```

causes a message containing the values in `expression_list` to be sent to the destination specified. The construct:

```
RECEIVE variable_list FROM source_identifier
```

causes a message to be received from the specified source and the values of the message assigned to the variables in `variable_list`. Sources and destination can be given *direct names*. We might, for example, have three processes – keyboard, process1 and display – communicating via messages:

```
PROGRAM Comprocess
PROCESS keyboard
    VAR key_value, ret_code:INTEGER;
    REPEAT
        BEGIN
            read keyboard information
            SEND key_value TO process1;
        END
    UNTIL key_value = ret_code
END
PROCESS process1
    VAR key_value, ret_code, disp_value:INTEGER;
    REPEAT
        BEGIN
            RECEIVE key_value FROM keyboard;
            compute dis_value from key_value
            SEND disp_value TO display;
        END
```

```

        UNTIL key_value = ret_code
    END
    PROCESS display
        VAR ret_code, disp_value:INTEGER;
        REPEAT
            BEGIN
                RECEIVE dis_value FROM process1;
                display dis_value
            END
        UNTIL dis_value = ret_code
    END
END

```

It is also possible to have statements causing message-passing operations to occur under specific conditions, for example the statement:

```

    WHEN Boolean_expression RECEIVE variable_list FROM
        source_identifier

```

or alternatively, the “guarded” command:

```

    IF Boolean_expression RECEIVE variable_list FROM
        source_identifier

```

which will accept a message only when/if the Boolean expression is TRUE.

Sequential programming languages with message-passing routines

To send and receive message-passing routines attached to standard sequential programming languages may be more laborious in specification and would only implement the basic message-passing operations. For example, message-passing send and receive routines with the format:

```

    send(channel ID,type,buffer,buffer_length,node,process_ID)
    recv(channel ID,type,buffer,buffer_length,message_byte_
        count,node,process_ID)

```

might be provided for FORTRAN programming. Such routines are usually found on prototype and early message-passing multiprocessor systems and need further routines to handle the message memory.

9.2.2 Synchronization and process structure

Message-passing send/receive routines can be divided into two types:

1. Synchronous or blocking.
2. Asynchronous or non-blocking.

Synchronous or *blocking* routines do not allow the process to proceed until the operation has been completed. *Asynchronous* or *non-blocking* routines allow the process to proceed even though the operation may not have been completed, i.e. statements after a routine are executed even though the routine may need further time to complete.

A blocking send routine will wait until the complete message has been transmitted and accepted by the receiving process. A blocking receive routine will wait until the message it is expecting is received. A pair of processes, one with a blocking send operation and one with a matching blocking receive operation, will be synchronized with neither the source process nor the destination process being able to proceed until the message has been passed from the source process to the destination process. Hence, blocking routines intrinsically perform two actions; they transfer data and they synchronize processes. The term *rendezvous* is used to describe the meeting and synchronization of two processes through blocking send/receive operations.

A non-blocking message-passing send routine allows a process to continue immediately after the message has been constructed without waiting for it to be accepted or even received. A non-blocking receive routine will not wait for the message and will allow the process to proceed. This is not a common requirement as the process cannot usually do any more computation until the required message has been received. It could be used to test for blocking and to schedule another process while waiting for a message. The non-blocking routines generally decrease the process execution time. Both blocking and non-blocking variants may be available for programmer choice in systems that use routines to perform the message passing.

Non-blocking message passing implies that the routines have buffers to hold messages. In practice, buffers can only be of finite length and a point could be reached when a non-blocking routine is blocked because all the buffer space has been exhausted. Memory space needs to be allocated and deallocated and the messages and routines should be provided for this purpose; the send routine might automatically deallocate memory space. For low level message passing, it is necessary to provide an additional primitive routine to check whether a message buffer space is reavailable.

Process structure

The basic programming technique for the system is to divide the problem into concurrent communicating processes. We can identify two possible methods of generating processes, namely:

1. Static process structure.
2. Dynamic process structure.

In the *static process structure*, the processes are specified before the program is executed, and the system will execute a fixed number of processes. The programmer usually explicitly identifies the processes. It might be possible for a compiler to assist in the creation of concurrent message-passing processes, but this seems to be an open research problem. In the *dynamic process structure*, processes can be created during the execution of the program using process creation constructs; processes can also be destroyed. Process creation and destruction might be done conditionally. The number of processes may vary during execution.

Process structure is independent of the message-passing types, and hence we have the following potential combinations in a language or system:

- Synchronous communication with static process structure.
- Synchronous communication with dynamic process structure.
- Asynchronous communication with static process structure.
- Asynchronous communication with dynamic process structure.

Language examples include Ada (having synchronous communication with static process structure), CSP (having asynchronous communication with static process structure) and MP (having synchronous communication with dynamic process structure) (Liskov, Herlihy and Gilbert, 1988). Asynchronous communication with dynamic process structure is used in message-passing systems using procedure call additions to standard sequential programming languages (e.g. Intel iPSC, see Section 9.3.2). The combination is not known together in specially designed languages, though it would give all possible features. Liskov, Herlihy and Gilbert suggest that either synchronous communication or static process structure should be abandoned but suggest that it is reasonable to retain one of them in a language. The particular advantage of asynchronous communication is that processes need not be delayed by messages, and static process structure may then be sufficient. Dynamic process structure can reduce the effects of delays incurred with synchronous communication by giving the facility to create a new process while a communication delay is in progress. The combination, synchronous communication with dynamic process structure, seems a good choice.

Program example

Suppose the integral of a function $f(x)$ is required. The integration can be performed numerically by dividing the area under the curve $f(x)$ into very small sections which are approximated to rectangles (or trapeziums). Then the area of each section is computed and added together to obtain the total area. One obvious parallel solution is to use one process for each area or group of areas, as shown in Figure 9.5. A single process is shown accepting the results generated by the other processes.

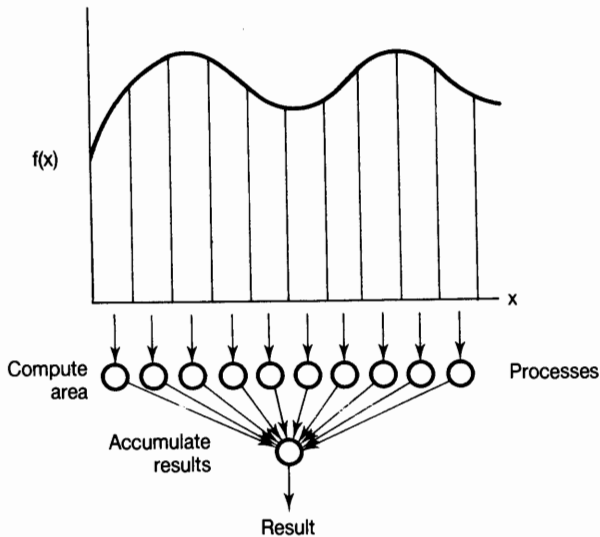


Figure 9.5 Integration using message-passing processes

Let the basic blocking message-passing primitives in the system be `send(message, destination_process)` and `receive(message, source_process)`. With the integral processes numbered from 0 to $n-1$ and the accumulation process numbered n , we have two basic programs, one for the processes performing the integrals and one for the process performing the accumulation, i.e.:

Integral process j

```
PROGRAM Integral
  VAR area, n: INTEGER;
  compute  $j$ th area
  send(area, n)
END
```

Accumulation process

```
PROGRAM Accumulate
  VAR area, i, n, acc: INTEGER;
  FOR i = 0 TO n-1
  BEGIN
    receive(area, i);
    acc := acc + area
  END
  WRITE ('Integral is', acc)
END
```

Variables are local and need to be declared in each process. The same names in different processes refer to different objects. Note that processes are referenced directly by number. The integral process requires information to compute the areas, namely the function, the interval size and number of intervals to be computed in each process. This information is passed to the integral processes perhaps via an

initiation process prior to the integral processes starting their computation.

The accumulation process could also perform one integration while waiting for the results to be generated. A single program could be written for all processes using conditional statements to select the actions a particular process should take, and this program copied to all processes. This would be particularly advantageous if there is a global host-node broadcast mode in which all nodes can receive the same communication simultaneously. In this case, we have:

Composite process

```

PROGRAM Comprocess
VAR  mynode, area, i, n, acc: INTEGER;
    read input parameters
    identify nodal address, mynode
    IF mynode = n THEN
        BEGIN
            compute nth area
            FOR i = 0 TO n-1
                BEGIN
                    receive(area, i);
                    acc := acc + area
                END
            WRITE ('Integral is', acc)
            END
        ELSE
            BEGIN
                compute jth area
                send(area, n)
            END
    END
END

```

Various enhancements can be made to improve the performance. For example, since the last accumulation is in fact a series of steps, it could be divided into groups of accumulations which are performed on separate processors. The number of areas computed by each process defines the process granularity and would be chosen to gain the greatest throughput taking into account the individual integration time and the communication time. In some cases, reducing the number of nodes involved has been found to decrease the computation time (see Pase and Larrabee, 1988).

Host-node communication is usually much slower than node-node communication. If separate transactions need to be performed for each node loaded (i.e. there is no broadcast mode) the time to load the nodal program could be decreased by arranging the program to be sent to the first node, which then passes a copy on to the next node and so on. The most effective method to reduce the communication time is to

arrange for each node to transmit its information according to minimal spanning tree. Results could be collected in a pipeline or tree fashion with the results passed from one node to the next. Each node adds its contribution before passing the accumulation onwards. Pipeline structures are useful, especially if the computation is to be repeated several times, perhaps with different initial values.

9.3 Message-passing system examples

9.3.1 Cosmic Cube

The Cosmic Cube is a research vehicle designed and constructed at Caltech (California Institute of Technology) under the direction of Seitz during the period 1981–5 (Seitz, 1985; Athas and Seitz, 1988) and is credited with being the first working hypercube multiprocessor system (Hayes, 1988) though the potential of hypercubes had been known for many years prior to its development. The Cosmic Cube significantly influenced subsequent commercial hypercube systems, notably the Intel iPSC hypercube system. Sixty-four-node and smaller Cosmic Cube systems have been constructed. The Intel 8086 processor is used as the nodal instruction processor with an Intel 8087 floating point coprocessor. Each node has 128 Kbytes of dynamic RAM, chosen as a balance between increasing the memory or increasing the number of nodes within given cost constraints. The memory has parity checking but not error correction. (A parity error was reported on the system once every several days!) Each node has 8 Kbytes of read-only memory to store the initialization and bootstrap loader programs. The kernel in each node occupies 9 Kbytes of code and 4 Kbytes of tables. The interconnection links operate in asynchronous full-duplex mode at a relatively slow rate of 2 Mb/s. The basic packet size is sixty-four bits with queues at each node. Transmission is started with send and receive calls. These calls can be non-blocking, i.e. the calls return after the request is put in place. The request becomes “pending” until it can be completed. Hence a program can continue even though the message request may not have been completed.

The nodal kernel, called the Reactive Kernel, RK, has been divided into an inner kernel (written in assembly language) and an outer kernel. The inner kernel performs the send and receive message handling and queues messages. Local communication between processes in one node and between non-local processes is treated in a similar fashion, though of course local communication is through memory buffers and is much faster. The inner kernel also schedules processes in a node using a round robin selection. Each process executes for a fixed time period or until it is delayed by a system call. The outer kernel contains a set of processes for communication between user processes using messages. These outer kernel processes include processes to create, copy and stop processes.

The host run-time system, called the Cosmic Environment, CE, has routines to establish the set of processes for a computation and other routines for managing the

whole system. The processes of a computation are called the process group. The system can be used by more than one user but is not time-shared; each user can specify the size of a hypercube required using a CE routine and will be allocated a part of the whole system not used by other users – this method has been called *space-shared*. In a similar manner to a virtual memory system, users reference logical nodal addresses, which have corresponding physical nodal addresses. The logical nodal addresses for a requested n -cube could be numbered from 0 to $n-1$.

Dynamic process structure with reactive process scheduling is employed. Programming is done in the C language, with support routines provided for both message passing and for process creation/destruction. The dynamic process creation function – `spawn (parameters)` – creates a process consisting of a compiled program in a node and process, all specified as function parameters. Specifying the node/process as function parameters rather than letting the operating system make this choice, enables predefined structures to be built up and allows changes to be made while the program is being executed. The send routine is `xsend (parameters)` where the parameters specify the node/process and a pointer to a message block. The `xsend` routine deallocates message space. Other functions available include blocking receive message, `xrecvb`, returning a pointer to the message block, allocating message memory space, `xmalloc`, and freeing message space, `xfree`. Later development of the system incorporated higher level message-passing mechanisms and fine grain programming. Statements such as:

```
IF i = 10 THEN SEND(i+1) TO self ELSE EXIT Fi
```

can be found in programs in the programming environment Cantor (see Athas and Seitz (1988) for further details).

Seitz introduced *wormhole* routing (Dally and Seitz, 1987) as an alternative to normal *store-and-forward routing* used in distributed computer systems. In store-and-forward routing, a packet is stored in a node and transmitted as a whole to the next node when a free path can be established. In wormhole routing, only the head of the packet is initially transmitted from the input to the output channel of a node. Subsequent parts of the packet are transmitted when the path is available. The term *flit* (flow control bits) has been coined to describe the smallest unit that can be accepted or blocked in the transmission path. It is necessary to ensure that the flits are received in the same order that they are transmitted and hence channels need to be reserved for the flits until the packet has been transmitted. Other packets cannot be interleaved with the flits along the same channels.

9.3.2 Intel iPSC system

The Intel Personal Supercomputer (iPSC) is a commercial hypercube system developed after the Cosmic Cube. The iPSC/1 system uses Intel 80286 processors with 80287 floating point coprocessors. The architecture of each node is shown in

Figure 9.6. Each node consists of a single board computer, having two buses, a processor bus and an input/output bus. The PROM (programmable read-only memory) has 64 Kbytes and the dual port memory has 512 Kbytes. The nodes are controlled by a host computer system called a cube manager. The cube manager has 2–4 Mbytes of main memory, Winchester and floppy disk memory, and operates under the XENIX operating system. As with the Cosmic Cube, each node has a small operating system (called NX). Eight communication channels are provided at each node, seven for links to other nodes in the hypercube and one used as a global Ethernet channel for communication with the cube manager. Typical systems have thirty-two nodes using five internode communication links. Internode communication takes between 1 and 2.2 ms for messages between 0 and 1024 bytes. Cube manager to node communication takes 18 ms for a 1 Kbyte message (Pase and Larrabee, 1988). The iPSC/2, an upgrade to the iPSC/1, uses Intel 80386 processors and hardware wormhole routing. Additional vector features can be provided at each node or selected nodes on one separate board per node.

FORTRAN message-passing routines are provided, including `send` and `recv`, having the format given previously. `sendw` and `recvw` are blocking versions of `send` and `recv`. If non-blocking message passing is done, the routine status can be used to

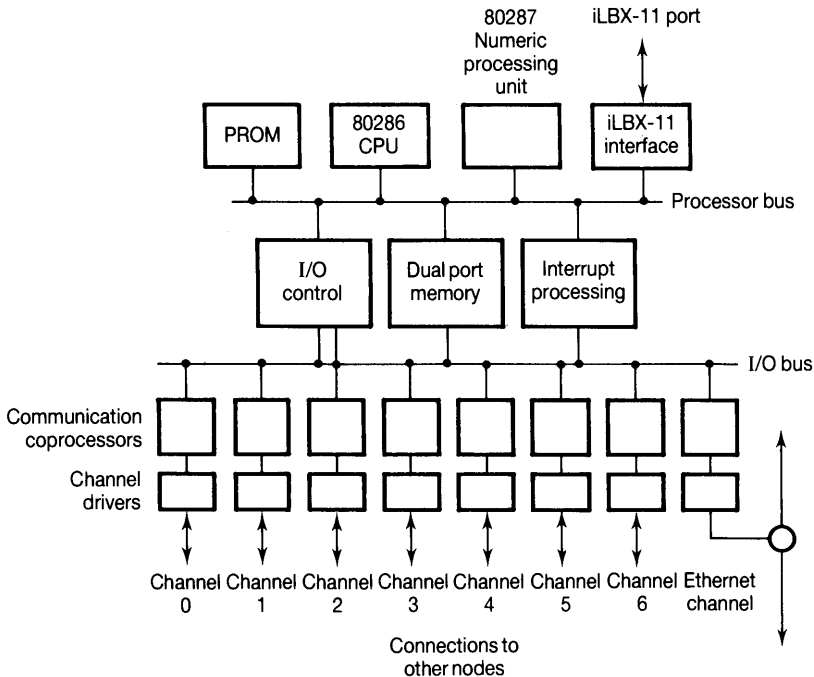


Figure 9.6 Intel iPSC node

ascertain whether a message buffer area is reavailable for use. Messages sent and received from the host use the commands `sendmsg` and `recmsg` and operate with blocked messages without type selection.

9.4 Transputer

In this section, we will present the details of the *transputer*, the first single chip computer designed for message-passing multiprocessor systems. A special high level programming language called *occam* has been developed as an integral part of the transputer development. Occam has a static process structure and synchronous communication, and is presented in Section 9.5.

9.4.1 Philosophy

The transputer is a VLSI processor produced by Inmos (Inmos, 1986) in 16- and 32-bit versions with high speed internal memory and serial interfaces. The device has a RISC type of instruction set (Section 5.2, page 151) though programming in machine instructions is not expected, as *occam* should be used.

Each transputer is provided with a processor, internal memory and originally four high-speed DMA channels which enable it to connect to other transputers directly using synchronous send/receive types of commands. A link consists of two serial lines for bidirectional transfers. Data is transmitted as a single item or as a vector. When one serial line is used for a data package, the other is used for an acknowledgement package, which is generated as soon as a data package reaches the destination.

Various arrays of transputers can be constructed easily. Four links allow for a two-dimensional array with each transputer connecting to its four nearest neighbors. Other configurations are possible. For example, transputers can be formed into groups and linked to other groups. Two transputers could be interconnected and provide six free links, as shown in Figure 9.7(a). Similarly, a group of three transputers could be fully interconnected and have six links free for connecting to other groups, as shown in Figure 9.7(b). A group of four transputers could be fully interconnected and have four links to other groups, as shown in Figure 9.7(c). A group of five transputers, each having four links, could be fully interconnected but with no free links to other systems.

A key feature of the transputer is the programming language, *occam*, which was designed specifically for the transputer. The name *occam* comes from the fourteenth century philosopher, William of Occam, who presented the concept of Occam's Razor: "Entia non sunt multiplicanda praeter necessitatem", i.e. "Entities should not be multiplied beyond necessity" (May and Taylor, 1984). The language has been designed for simplicity and provides the necessary primitive operations for point-to-point data transfers and to specify explicit parallelism. The central concept in an

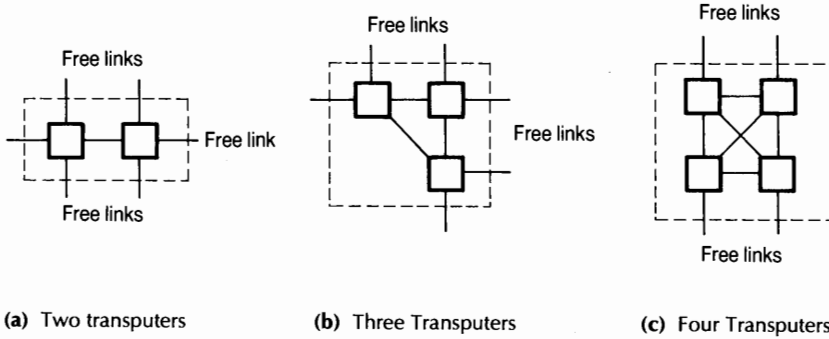


Figure 9.7 Groups of transputers fully interconnected (a) Two transputers (b) Three transputers (c) Four transputers

occam program is the *process* consisting of one or more program statements, which can be executed in sequence or in parallel. Processes can be executed concurrently and one or more processes are allocated to each transputer in the system. There is hardware support for sharing one transputer among more than one process. The statements of one process are executed until a termination statement is reached or a point-to-point data transfer is held up by another process. Then, the process is descheduled and another process started automatically.

9.4.2 Processor architecture

The internal architecture of the processor is shown in Figure 9.8 and has the following subparts:

1. Processor.
2. Link interfaces.
3. Internal RAM.
4. Memory interface for external memory.
5. Event interface.
6. System services logic.

The first transputer product, the T212, announced in 1983, contained a 16-bit integer arithmetic processor. Subsequent products included a 32-bit integer arithmetic processor part (T414, announced in 1985) and a floating point version (the T800, announced in 1988). The floating point version has an internal floating point arithmetic processor attached to the integer processor and the data bus, such that both processors can operate simultaneously. Though the processor itself is a RISC type, it is microprogrammed internally and instructions take one or more processor

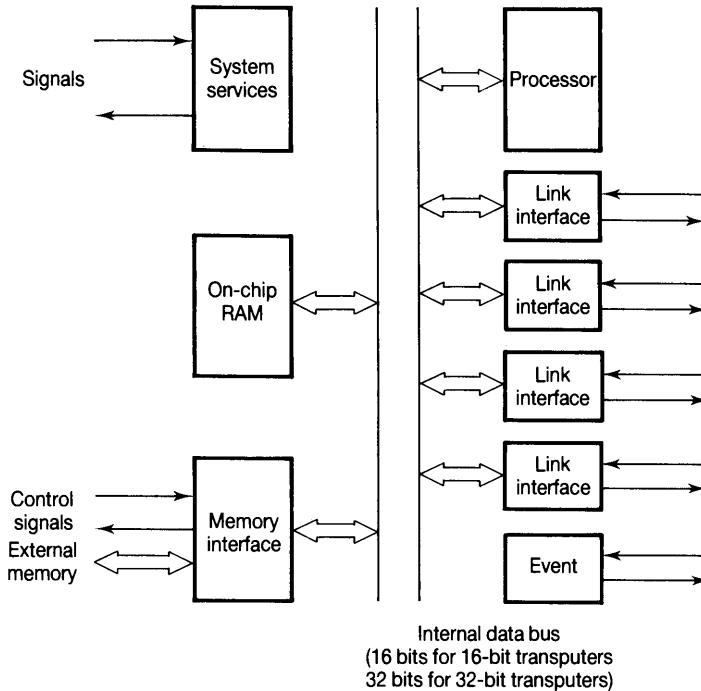


Figure 9.8 Internal architecture of transputer (without floating point unit)

cycles. Simple operations such as add or subtract take one cycle but complex operations can take several cycles. One cycle can be 50 ns (for 20 MHz, but this depends upon the clock frequency). Instructions are one or more bytes. Thirty-two-bit transputers have memory organized in 32-bit words and four bytes are fetched together. There is a two word prefetch buffer. The first products have four serial links, 10 Mbits/sec serial link transfer rate and 2 Kbytes internal static RAM. Subsequent products have increased transfer rates and increased memory. External memory can also be attached through the memory interface, though access to external memory is generally slower than to internal memory. The memory interface circuitry can be programmed to generate various signals to match external memory chips.

The transmission is synchronous and a common clock signal, or a separate clock at the same frequency, is applied to devices. Information is transmitted in the form of packets. Every time a data packet is sent, the destination responds by returning an acknowledge packet, as shown in Figure 9.9. Data packets consist of two 1s followed by an 8-bit data and terminated with a 0 (i.e. eleven bits in all). Acknowledge packets consist of a 1 followed by a 0 (i.e. two bits in all) and can be returned as soon as the data package can be identified by the internal link interface. The links

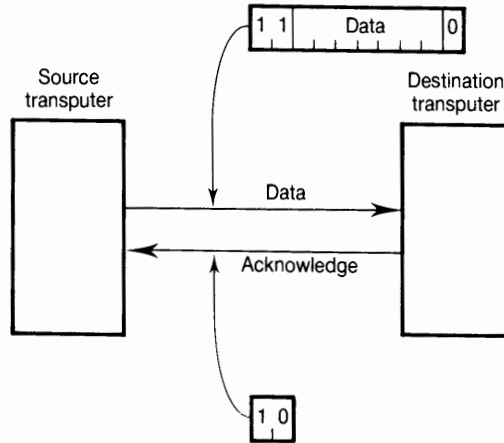


Figure 9.9 Transputer input/output handshaking technique

provide the hardware for the input and output statements in occam (see later) and operate as DMA channels, i.e. data packets can be sent one after the other for vectors. Input/output statements use internal channels rather than external links for communication between processes within a transputer.

The event interface enables an external device to signal attention and to receive an acknowledgement. The event interface operates as an input channel and is programmed in a similar way.

9.5 Occam

This section reviews the occam language, giving the main features, though not minor variations provided in the syntax. The reader is referred to May (1987) for a complete definition of the language.

9.5.1 Structure

Like all high level language programs, occam programs consist of statements, though only a limited number of different statements are provided. The first version of occam kept the facilities to a minimum on the principle that simple is best, but further facilities were added (and slight changes made in the syntax) in the later version called occam 2. We will describe the occam 2 syntax.

Occam is a block-structured language using indentation rather than brackets or BEGIN-END to show a compound structure. Each level of indentation consists of two spaces and each statement is normally placed on a separate line. Statements can be specified as being executed sequentially, in parallel or dependent upon some condition (including waiting for input or output). Explicit input and output statements are provided for passing data between processes. Communication between processes is dependent upon both the source and destination being ready, i.e. both the input and output statements on the respective processes must be encountered before the transfer of data commences. Occam uses prefix operators, i.e. operators are added to the head of a process to specify some condition or action or to declare variables. Comments can be included in the program by starting the comment with `--`. The comment can occupy a single line at the same level of indentation as the next statement or it can be added to the end of a statement. Termination symbols are not used at the end of statements on separate lines (i.e. redundant features are removed from the language).

9.5.2 Data types

The type of a variable is declared, as in other languages such as PASCAL or C, but using a colon to show that it is prefixing a process. The original reserved word was VAR which was subsequently changed to INT. The declaration:

```
INT x:
```

declares the variable `x` of a type appropriate for the transputer (16-bit 2's complement signed integer in the range -32768 to $+32767$ for the 16-bit T212, 32-bit 2's complement signed integer for 32-bit transputers). More explicit types are provided. `INT16 x`: declares `x` as a 16-bit integer; `INT32 x`: declares `x` as a 32-bit integer; `INT64 x`: declares `x` as a 64-bit integer; `REAL32 x`: declares `x` as a 32-bit floating point number using the 32-bit IEEE format; `REAL64 x`: declares `x` as a floating point number using the 64-bit IEEE format. Other types include `BYTE` (a number between 0 and +256) and `BOOL` (true or false).

Variables are always local variables; the concept of global variables has no meaning in a system without global memory. Variables are declared prior to processes or "subprocesses" and not only at the beginning of the complete program. They then have the scope given by the level of indentation.

Arrays in occam 2 can be declared by prefixing the declaration with the number of elements in the array (counting from zero). For example:

```
[10]INT x:
```

declares a one-dimensional array, `x`, having the elements `x[0]`, `x[1]`, ..., `x[9]`. Multidimensional arrays can be declared by having more than one `[]` prefix.

Channels are also declared in the process before being used, originally using the reserved word `CHAN` and subsequently using the construct `CHAN OF protocol` where `protocol` can specify the data type of the data being sent across the channel. For example:

```
CHAN OF INT output:
```

declares a channel called `output` which will carry integer values. A set of channels can be formed into a vector of channels with the same name. We will omit the channel declaration in some simple programs, assuming that the declaration has been made in some higher level process.

9.5.3 Data transfer statements

Five *primitive processes* exist in occam for data transfer, three of which are *actions* concerned with data transfer, assignment, input and output.

The *assignment action* is conventional and allows the value of an expression to be assigned to a variable and has the general form:

```
variable := expression
```

where `:=` means the usual “becomes equal to”. (The type of the expression and the variable must be the same; type cohesion is not performed automatically in occam/occam 2, though occam 2 does have cohesion operations.) Multiple sequential assignments are possible in a single statement, by separating the variables and expressions with commas.

Transfer of data between processes through internal or external channels is achieved by having the *output action*:

```
channel ! expression
```

in the source process, where `!` is the symbol for output (an output exclamation), and the *input action*:

```
channel ? variable
```

in the destination process, where `?` is the symbol for input (a query for input). When both input and output have been encountered, the data transfer takes place. This *rendezvous* technique is illustrated in Figure 9.10. Processes can be synchronized using the *rendezvous* and, if the data value is unimportant, the variable and expression can be replaced with the occam word `ANY`, i.e. `c?ANY` and `c!ANY` have the effect of synchronizing processes. Multiple input/output can be specified by separating the variables and expressions with semicolons, or using a vector of variables, i.e.

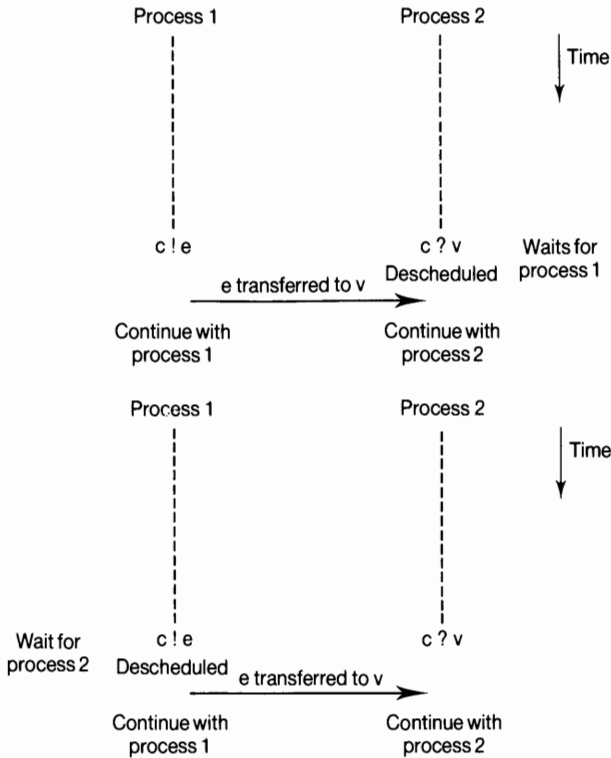


Figure 9.10 Rendezvous technique (a) Input encountered before output (b) Output encountered before input

vect [] describes a one-dimensional array of variables with the name vect.

The two remaining primitive processes are STOP which “starts but never proceeds, and never terminates”, and SKIP which “starts, performs no action, and terminates” (May, 1987). SKIP has a part to play in constructs which must have a terminating process.

9.5.4 Sequential, parallel and alternative processes

In most programming languages, it is assumed that statements are executed one after another in the sequence written unless control statements are used. In occam, the sequential nature of processes is not assumed; in fact processes can be specified as

318 Multiprocessor systems without shared memory

executed concurrently or sequentially. Concurrent processes on separate transputers can execute truly concurrently. Processes on one transputer simulate concurrent operation.

Sequential operation is specified with the sequence (SEQ) process:

```
SEQ
  process1
  process2
  .
  .
```

Each component process is executed after the previous process has finished. For example, the following sequence process, as illustrated in Figure 9.11, takes data from an input channel c1 and sends it to an output channel c2:

```
INT x:
SEQ
  c1?x
  c2!x
```

Notice that the declaration is at the same level of indentation as the SEQ process, but the component processes are at an extra level of indentation. Also, the input and output actions are performed in sequence.

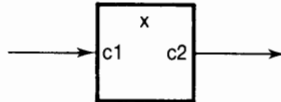


Figure 9.11 Single stage buffer

Concurrent operation is specified with the parallel (PAR) process:

```
PAR
  process1
  process2
  .
  .
```

All component processes are executed simultaneously (conceptually if only one transputer is available). For example, the following parallel process, as illustrated in Figure 9.12, accepts two inputs and transfers the values from each input to a different output channel:

```

PAR
  INT x:
  SEQ
    c1?x
    c3!x
  INT y:
  SEQ
    c2?y
    c4!y
    
```

In both SEQ and PAR processes, the syntax does allow no processes to be specified and, in this unlikely case, they behave as SKIPS.

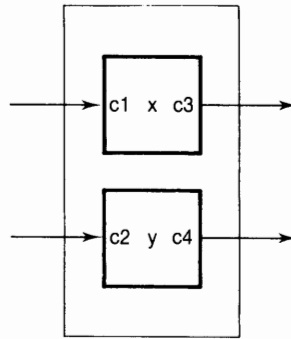


Figure 9.12 Parallel buffers

In the alternation process (ALT), a component process can be executed dependent upon an input proceeding. The first input to proceed enables the associated process to be executed; the others are discarded. The construction of the alternate process with input is:

```

ALT
  input
  process
  input
  process
  .
  .
    
```

320 Multiprocessor systems without shared memory

Notice in particular the levels of indentation. The following alternate process, as illustrated in Figure 9.13, has three input channels and a single output channel. The first input to proceed passes the input value to the output:

```
INT x:
ALT
  c1?x
    cout!x
  c2?x
    cout!x
  c3?x
    cout!x
```

The general ALT syntax describes a “guard” (an input statement in the previous statement) where guard and process could be another alternate process. The guard is an input, or Boolean expression & (and) input, or Boolean expression & SKIP (effectively a Boolean expression alone).

9.5.5 Repetitive processes

One construct, the WHILE construct, is provided for program loops and has the format:

```
WHILE Boolean expression
  process
```

While the Boolean expression yields a true value, the process is repeated.

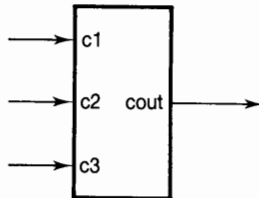


Figure 9.13 Alternate process

Example

The following sequence will repeatedly accept data from channel c1 and transfer the value received to channel c2:

```

WHILE TRUE
  INT x:
  SEQ
    c1?x
    c2!x

```

Notice the two levels of indentation. To accept values between 0 and 9, we might try:

```

INT x:
SEQ
  x := 0
  WHILE (x >= 0) AND (x <= 9)
    SEQ
      c1?x
      c2!x

```

However, the first transfer from input to output would take place whether or not the input value is within the desired range. To remedy this fault, we could have:

```

INT x:
SEQ
  c1?x
  WHILE (x >= 0) AND (x <= 9)
    SEQ
      c2!x
      c1?x

```

Now we also leave the process with one more input than output action. Whether this is a problem depends upon the communicating processes.

9.5.6 Conditional processes

The conditional IF construct allows one of a series of component processes to be executed dependent upon a Boolean expression. The IF construct has the format:

322 Multiprocessor systems without shared memory

```
IF
  Boolean expression
  process
  Boolean expression
  process
  :
```

where the “Boolean expression – process” construct can be another IF construct. Notice that the Boolean expression is indented by two spaces and the component process is indented by a further two spaces. Each Boolean expression is evaluated in turn and the first found to be TRUE causes the associated process to be executed and the IF construct to terminate. If no Boolean expression is found to be true, the IF construct behaves as a STOP statement. Hence, in these cases, one would need to include a NOT Boolean expression SKIP construct within the IF statement if the next statement is always to be executed or, more conveniently, simply TRUE SKIP at the end.

Example:

The following program sequence causes x to be output on channel c1 if an input is equal to 1, or on channel c2 if an input is equal to 2. Nothing is to happen if it is not equal to 1 or 2:

```
INT i, x:
SEQ
  input?i
  IF
    i = 1
    c1!x
    i = 2
    c2!x
  TRUE
  SKIP
```

Occam 2 has a CASE statement which allows a process to be selected according to an expression being evaluated to a particular value given in the statement.

The transputer has an internal real-time clock which enables variables to be declared as being of type TIMER and to be accessed by parallel processes. Timers are incremented at regular intervals dependent upon the external clock frequency applied. In the T212, for example, there are two timers – a high priority timer incrementing every 1 μ s and a low priority timer incrementing every 64 ms if the applied frequency is 5 MHz. Both timers cycle after 2^{32} increments and are of type INT. Timers are set to specific values by using input statements with literals. The timer will be incremented automatically after loading. Processes can be delayed until a specific value occurs.

9.5.7 Replicators

A replicator can be used in the SEQ construct, PAR construct, ALT construct or IF construct to specify that the component process is duplicated a number of times. The basic formats are:

```

SEQ  name = base FOR count
PAR  name = base FOR count
ALT  name = base FOR count
IF   name = base FOR count
    
```

where name is an integer variable and base and count are integer expressions. In each case, the component process is repeated count times and the variable, name, is incremented by one each time, starting at base.

Examples

Suppose that there are ten input channels and that each is examined in turn so that each value received is transferred to one output channel. We could have:

```

[10]CHAN OF INT c:  -- declares 10 channels c[0] to c[9]
CHAN OF INT cout:  -- declares channel cout
SEQ i = 0 FOR n
  INT x:
  SEQ
    c[i]?x
    cout!x
    
```

In Figure 9.14, there are four processes, each continually accepting an input from the previous process (except the first) and multiplying the value input by two. We could have:

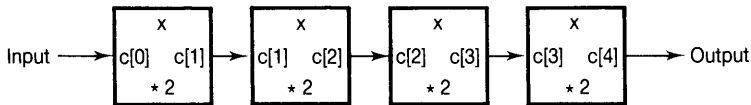


Figure 9.14 Four-stage buffer accepting continual input

324 Multiprocessor systems without shared memory

```
[5]CHAN OF INT c: -- declares five channels c[0] to c[4]
PAR i = 0 FOR 4
  WHILE TRUE
    INT x:
    SEQ
      c[i]?x
      x := 2*x -- or use more efficient shift operator
      c[i+1]!x
```

To implement an eight-to-one line multiplexer, as shown in Figure 9.15, we could have:

```
[8]CHAN OF INT c:
CHAN OF INT cout, select:
INT x:
SEQ
  select?x
  IF i = 0 FOR 8
    x = i
    SEQ
      c[i]?x
      cout!x
```

The program does not handle the situation $x < 0$ or $x > 7$. It is left as an exercise to incorporate this possibility.

9.5.8 Other features

Occam has the usual form of procedures using the construct PROC name (parameter list), and a number of other constructs (see Burns (1988) for a good description).

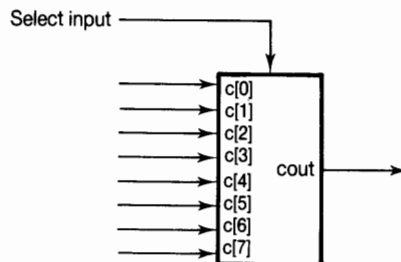


Figure 9.15 Eight-input multiplexer

A feature of occam and the transputer is that the program can be written irrespective of the number of transputers; the number of transputers and their actual interconnections via channels can be specified later. The program could be tested on a single transputer system before being put onto a multiprocessor system. Processes are allocated to processors using the `PLACED PAR - PROCESSOR` construct. For example:

```
PLACED PAR
  PROCESSOR 1
    Process1
  PROCESSOR 2
    Process2
```

allocates process1 to processor 1 and process2 to processor 2 and the processes are executed in parallel. Constructs are available to signify that particular processes are to be executed in preference to others, i.e. a priority is assigned to processes (`PRI PAR` construct).

The semantics of the language have been studied by Roscoe and Hoare (1986) leading to a set of algebraic laws. Also, it may be that complex programs can be transformed into other versions using formal mathematical transformations which can lead to improved programs. Occam transformations, though not necessarily difficult, are beyond the scope of this book (see May and Taylor, 1984).

We should mention that occam is a simple language; hence we are able to give the major details of the language in a section of a chapter. It lacks some features found in conventional high level languages, maybe purposely. The data structures are limited. Recursion is not allowed.

PROBLEMS

9.1 Design a multiprocessor architecture having both direct link message-passing characteristics and shared memory characteristics and make an evaluation of the design.

9.2 Deduce an equation for the network latency of a message-passing system, given that the bandwidth is B , the message length is L and the path set-up time per node is given by $t_{\text{set-up}}$. Plot the equation for fixed message length and for fixed path length. Under what conditions does the network latency approach a constant? Suggest an advantage of a constant network latency.

9.3 Produce a logic design for hardware routing in a hypercube according to the algorithm given in Section 9.1.2, page 300.

9.4 Write a program to perform the numerical integration of an arbitrary function $f(x)$, as given in Section 9.2.2, (page 305), but using a tree structure to accumulate the results.

9.5 Deduce the actions of the following occam terminal screen handler program:

```

CHAN OF INT key.out, prog.out, screen:
INT ch, running, alarm.time, term.char:
SEQ
  running := TRUE
  term.char := 0
  WHILE running
    SEQ
      TIME ? alarm.time
      alarm.time := alarm.time + 5
    ALT
      key.out ? ch
      Screen ! ch
      prog.out ? ch
    IF
      ch = term.char
      running := FALSE
      ch <> term.char
      Screen ! ch

```

The period within a name is regarded as a normal symbol without any special meaning. It is used in the same way as the underscore in C programs, to clarify the name.

9.6 The declarations in the following occam program are incomplete. Add the required symbols and explain the operation of the program:

```

CHAN
PAR
  INT
  SEQ
    y := 1
    WHILE TRUE
      ALT
        datainput ? x
        output1 ! x*y
        scaleinput ? y
      SKIP

```

```

INT
WHILE TRUE
  SEQ
    output1 ? x
    IF
      x = 0
        output2 ! x
      x <> 0
        output2 ! -x
INT
SEQ
  y := 0
  WHILE TRUE
    ALT
      output2 ? x
      result ! x + y
    offsetinput ? y
  SKIP

```

9.7 Write a program in occam for each node in a three-dimensional hypercube to route a message from a source node to a destination node using the hypercube routing algorithm described in Section 9.1.2, page 300.

9.8 Design a message-passing routing algorithm for a mesh network which broadcasts a host message to all nodes in the mesh at the greatest speed. Further, design a message-passing routing algorithm which broadcasts a message from a node in the mesh to all other nodes in the mesh. Show how these algorithms can be implemented with message-passing routines.

9.9 Write a program in occam to broadcast a message from the host node to all other nodes in a hypercube.

9.10 Show how the statements:

```

IF Boolean_expression RECEIVE variable_list FROM
  source_identifier
WHEN Boolean_expression RECEIVE variable_list FROM
  source_identifier

```

could be implemented in occam.

328 Multiprocessor systems without shared memory

9.11 Show how non-blocking send and receive routines could be constructed in occam.

9.12 Incorporate the possibility that $x < 0$ or $x > 7$ into the last program in Section 9.5.7, page 324.

Multiprocessor systems using the dataflow mechanism

This chapter will consider alternative computer designs to the (von Neumann) stored program systems described in previous chapters, concentrating upon the dataflow technique. The dataflow technique is a multiprocessor technique which enables parallelism to be found without being explicitly declared. The chapter concludes with a summary of the book.

10.1 General

The computer designs so far considered execute instructions, which are stored in a memory, in a particular sequence; the multiprocessor designs presented have more than one such sequence executed simultaneously to increase the execution speed, but do not change the basic mode of operation. A program counter is necessary within each processor to guide each sequence. The instruction execution within each processor is serial and hence inherently slow. To gain an advantage, the programmer, or a compiler, has to spot the independent instructions which can be passed to separate processors; the communication overhead also has to be sufficiently low.

The traditional “program counter controlled” stored program (von Neumann) computer system is sometimes called a *control flow* computer, especially when we wish to differentiate this computer system from alternative types in which the program execution sequence may not be controlled by a central control unit having a program counter. In alternative designs, the sequence is defined by some other mechanism. There are several alternative mechanisms that could be applied. If there is a program of instructions held in a memory, the following possibilities for execution can be identified:

1. An instruction is executed when the previous instruction in a defined sequence has been executed.
2. An instruction is executed when the operands required become available.

3. An instruction is executed when results of the instruction are required by other instructions.
4. An instruction is executed when particular data patterns appear.

The first method is the traditional control flow computer mechanism; the second method is known as *data driven* or *dataflow*; the third is *demand driven* and the fourth is *pattern driven*.

Computers do not necessarily need to perform operations specified by stored instructions at all. Neural networks, which attempt to model the human brain to some extent, do not have a stored programme of instructions. Neural networks have a long history; the idea of copying the brain can be traced back many years to the 1940s, when Turing contemplated neural computers (Hodges, 1983). However, putting the concept into practice has eluded research workers until recently. Practical implementations of neural computers could arguably be classified as type four, though perhaps a separate category should be made for neural computers.

We will concentrate upon dataflow computers in this final chapter. The dataflow technique was originally developed in the 1960s by Karp and Miller (1966) as a graphical means of representing computations. In the early 1970s, Dennis (1974) and later others began to develop computer architectures based upon the dataflow computational model. Let us first examine the basic dataflow computational model; and then more recent dataflow architectures.

10.2 Dataflow computational model

The dataflow computational model uses a directed graph, sometimes called a *data dependence graph* or *dataflow graph*, to describe a computation. This graph consists of nodes, which indicate operations, and arcs from one node to another node, which indicate the flow of data between them. Nodal operations are executed when all required information has been received from the arcs into the node. Typically, a nodal operation requires one or two operands and, for conditional operations, a Boolean input value, and produces one or two results. Hence one, two or three arcs enter a node and one or two arcs leave it. Once a node has been activated and the nodal operation performed (i.e. the node has *fired*) results are passed along the arcs to waiting nodes. This process is repeated until all of the nodes have fired and the final result has been created. More than one node can fire simultaneously, and generally any parallelism in the computational model will be found automatically.

Figure 10.1 shows a simple dataflow graph of the computation $f = A/B + B \times C$. The inputs to the computation are the variables A , B and C , shown entering at the top. The paths between the nodes indicate the route taken by the results of the nodal operations. The general flow of data is from top to bottom. There are three computational operations (add, multiply and divide). We notice that B is required by two nodes. An explicit COPY node is used to generate an additional copy of B .

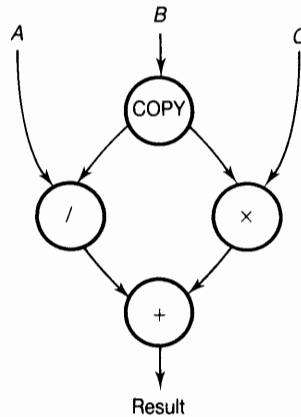


Figure 10.1 Dataflow graph of computation $A/B + B \times C$

Data operands/results move along the arcs contained in *tokens*. Figure 10.2 shows the movement of tokens between nodes. After the inputs are applied, the token containing the A operand is applied to the division node, the token containing the B operand is applied to the COPY node and the token containing the C operand is applied to the multiply node. Only the COPY node can fire, as its single-input token is present at the node. The other nodes require a result token of the COPY node. Once the COPY tokens have become available, both the multiply and divide nodes have all their tokens and can fire. The final node waits for both the multiplication and division nodes to complete before it can commence. It may be that the multiplication operation is completed before the division operation but in any event both operations must produce a token before the final addition node can fire.

Clearly, one can build up in-line computations of this sort, but practical computations usually require additional features. For example, in control-flow computations, conditional instructions provide decision-making power. Similarly, conditional operations are provided in dataflow computations. Conditional operation nodes generally take one or two operands and one conditional input. They pass forward one or two results, which depend upon the value of the data input and the value of the condition. Condition nodes can be regarded as switches, passing a data input token to an output.

Two forms of switch or condition nodes are shown in Figure 10.3. In the MERGE node, there are two data inputs and one output. If the condition is true, the left-hand side input token is passed to the output; if the condition is false, the right-hand side input token is passed to the output. In the BRANCH instruction, the single-input token is passed to the left output path if the Boolean condition is true or to the right output path if the condition is false. It is not necessary to use both outputs in the BRANCH node. MERGE and BRANCH are not both strictly necessary, as MERGE can be

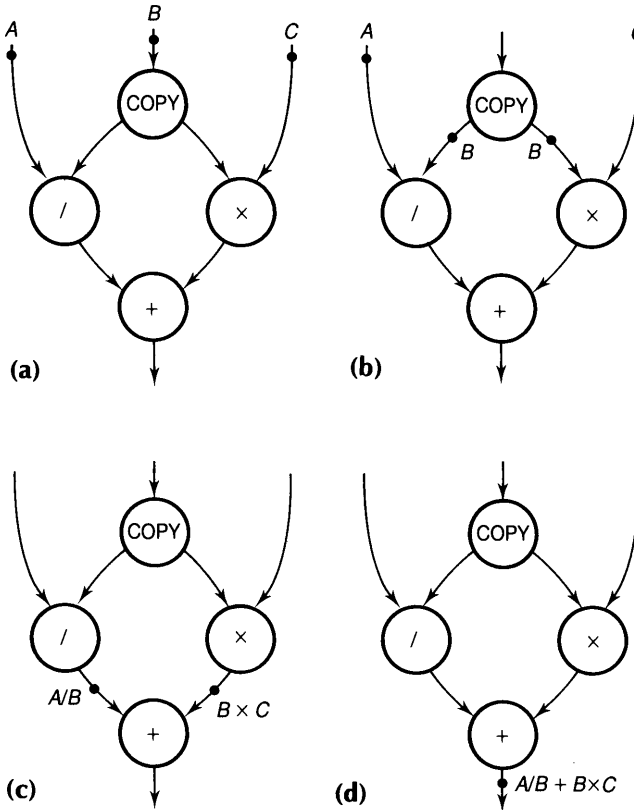


Figure 10.2 Movement of data tokens in computation $A/B + B \times C$ (a) After inputs applied (b) After copy instruction executed (c) After both divide and multiply instructions executed (d) After addition instruction executed

achieved with two BRANCH nodes and BRANCH can be achieved with two MERGE nodes (Problem 10.2).

Once the first set of inputs has been used, a second set could be applied and processed behind the first in a pipeline fashion. However, it is important that partial results (tokens) of the first computation are not processed with partial results of the second or any other subsequent computation. The results are usually required in the sequence that the inputs are applied.

There are numerous instances in which a particular computation must be repeated with different data, notably with program loops. Program iteration loops of conventional programming languages can be reproduced by feeding results back to input nodes and tokens must be used only with those tokens of the same iteration. Loops are often formed in conventional languages using loop variables which are incremented each time the body of the computation is computed. The loop is

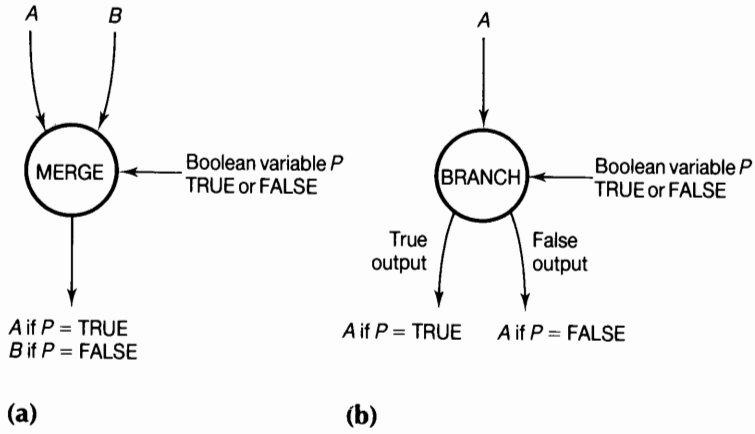


Figure 10.3 Conditional dataflow instructions
 (a) MERGE instruction (b) BRANCH instruction

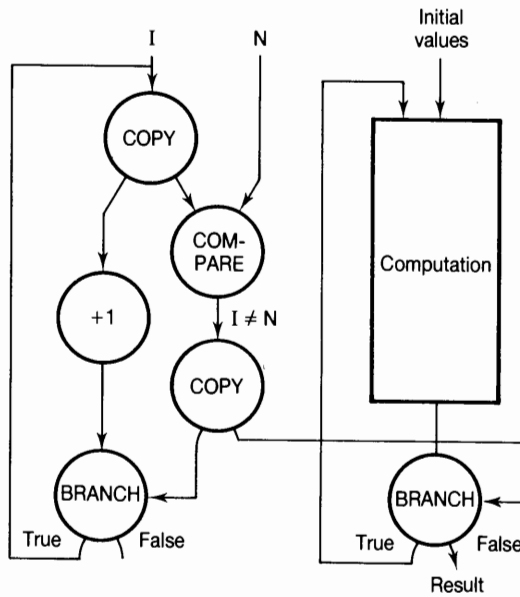


Figure 10.4 Dataflow loops

terminated when the loop variable reaches a defined value. This method can be carried over to dataflow computations, as shown in Figure 10.4. It is important that a mechanism is in place to keep the two loops in step. Mechanisms should also be provided for function calls and handling data-structures such as arrays.

There are two general dataflow schemes:

1. Static dataflow.
2. Dynamic dataflow.

which result in two classes of dataflow system. In static dataflow, there can be only one instance of a particular node firing at a time, whereas in dynamic dataflow it is possible to have multiple instances of a node firing at run-time. This might occur in code sharing, multiple function calls and recursion. Hence, such operations are not possible in static dataflow. Let us first examine static dataflow.

10.3 Dataflow systems

10.3.1 Static dataflow

The static dataflow architecture has the *firing rules*:

1. Nodes fire when all input tokens appear and the previous output tokens have been consumed.
2. Input tokens are then removed and new output tokens are generated.

These rules allow for pipeline computations and loops but not recursion or code sharing. The machine generally requires a handshaking acknowledgement mechanism, as shown in Figure 10.5, to indicate to a node that the output token has been consumed. The acknowledgement mechanism can take the form of special control tokens sent from processors once they respond to a fired node.

The static mechanism was the first dataflow mechanism to receive attention for hardware realization at MIT (Dennis, 1974) though a system was not constructed then. Dennis conceived a packet-driven ring architecture, shown in Figure 10.6, consisting of a ring of processor and memory elements interconnected via routing networks. Processing elements would receive operation packets of the form:

op-code operands destinations

where the op-code (operand code) specifies the operation to be performed, the operands specify the numbers to be used in the operation, and the destinations specify where the result of the operation is to be sent.

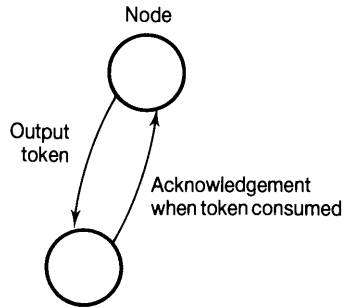


Figure 10.5 Static dataflow token-passing

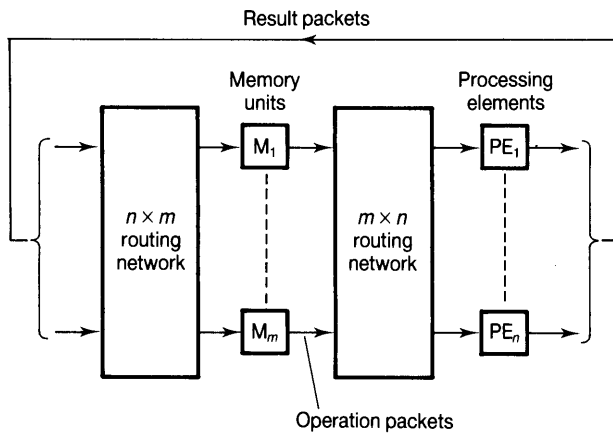


Figure 10.6 An early dataflow architecture

Notice that the numbers to be used are literals and are always carried within the instruction. Numbers are never referred to by addresses in memory, as they are not stored in a globally accessible memory; this has advantages and disadvantages. It is an advantage that operands can only be affected by one selected node at a time. It is a disadvantage in that complex data structures, or even simple vectors or arrays, could not reasonably be carried in the instruction and hence cannot be handled in the mechanism (unless the mechanism is modified).

The result packet takes the form:

value destination

where value is the value obtained after the operation has been executed. The result packets pass through a routing network to “instruction cells” in the memory unit, as identified by the destination address in the result packet. An instruction cell generates an operation packet when all of the input packets (tokens) have been received. Typically, two such packets are required in the instruction cell to generate an operation packet. The operation packet is then routed to a processing element. If all processing elements are identical (i.e. a homogeneous system) any free processing elements could be chosen. In a non-homogeneous system with specialized processing elements, each capable of performing particular functions, the op-code in the operation packet is used to select the processing element.

Dennis noted that the scheme, as he proposed it, was impractical if each instruction cell needed to be fabricated individually with a large number of instruction cells. Therefore, he proposed that instruction cells should be formed into groups (instruction blocks) each having a single input and a single output. The dataflow system can be further simplified by associating the instruction cells with processing elements, leading to the architecture shown in Figure 10.7. Here the processing elements (PEs) generate and receive packets and only one routing network is necessary to forward packets from PE outputs to PEs inputs. The destination PE is specified in the result packet.

Dataflow architectures generally use some form of routing network to transmit result packets on to a token matching mechanism. The packets usually consist of several bytes and could be sent one byte at a time, to reduce the interconnections. In the basic Dennis architecture, the matching mechanism is incorporated into the instruction cells or processing elements. The routing network must be capable of receiving byte-serial packets, with destinations encoded within the packets, and of forwarding the packets on to the required cells or processing elements. There are a number of possible candidates for the routing networks, as described in Chapter 8

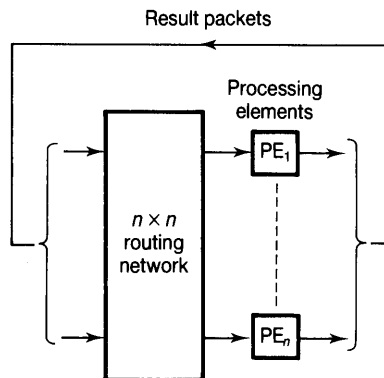


Figure 10.7 Simplified dataflow architecture

(see especially Section 8.4.3, page 263 on dynamic multistage networks). The principal criterion is throughput of packets rather than the propagation time from the input of the routing network to the output, as the system is pipelined. Self-routing networks are particularly attractive for this application.

10.3.2 Dynamic dataflow

Dynamic dataflow describes a dataflow system in which the dataflow graph being executed is not fixed, but can be altered (dependent upon the executing code) through such actions as recursion and code sharing. Recursion and code sharing cannot be done in the static architecture as described but can be accomplished by simply copying the code every time a call is made. This is known as *dynamic code copying*. Alternatively, tags could be attached to the packets to identify tokens with particular computations (instances of shared code, recursive code, loops, functions, etc.); this is known as *dynamic token tagging*, and is the method normally associated with dynamic dataflow. Dynamic token tagging dataflow uses the firing rules:

1. A node fires when all input tokens with the same tag appear.
2. More than one token is allowed on each arc and previous output tokens need not be consumed before the node can fire again.

The dynamic token tagging dataflow system needs storage for unmatched tokens, hardware for matching the tokens and a hardware mechanism for generating the tags. Tokens may not be taken strictly in the order generated and a first-in first-out token queue for storing the tokens is not suitable. However, no acknowledgement mechanism is required. The term *coloring* has been used for the token labeling operation, and tokens with the same color belong together (Dennis, 1974). Token tagging can be extended to cover coloring elements of arrays so that the correct elements are processed. It implies that all tokens must now have a coloring tag and that machine operations must be provided to operate upon the tags.

Token tags

To cover the three situations identified – functions and recursion, loops and array elements – each token tag could have three fields (Glauert, Gurd and Kirkham, 1985):

iteration level activation name index

Each field will hold a number, say, from zero onwards. Iteration level identifies the particular activation of a loop body, activation name identifies the particular function call and index identifies the particular element of an array. Iteration level and activation name might be combined into one field, activation name, referring to a loop body or function activation.

Individual fields of the tag need to be treated as data values and passed from one node to another. As a minimum requirement, each field requires operations to extract the field value and to set the field to a value. For example, a “read token field” operation accepts a single token and produces a token with the value of the input token field. The “set token field” operation produces an output token corresponding to the input token except with its token field set to a value given as a literal in the operation. Increment and decrement operations can also be provided, particularly for iteration level. Often, after a particular routine has been completed using the coloring facility, the result is delabeled by setting the token field(s) to 0. We note that it is possible for function call, a loop, and array index to occur in combination, for example, for a function call to occur with a different array index. It is also possible for nested function calls, loops, etc. to occur. Figure 10.8 shows one general arrangement for a loop with tagging of parameters before each activation of the loop, and retagging of the results.

Different instances of a function call can be colored with unique activation level numbers, but the mechanism needs to take into account that each call will be from a different place in the program and that it is necessary to return from these different places. In a normal von Neumann computer, a stack is used to hold the return address. In a dynamic token tagging dataflow system, a return operation can be provided to produce a token corresponding to one of the input tokens but having a specific destination address as given by a second input token, as shown in Figure 10.9. Further operations are necessary to allow complete generality of the function call in all situations (see Glauert, Gurd and Kirkham, 1985).

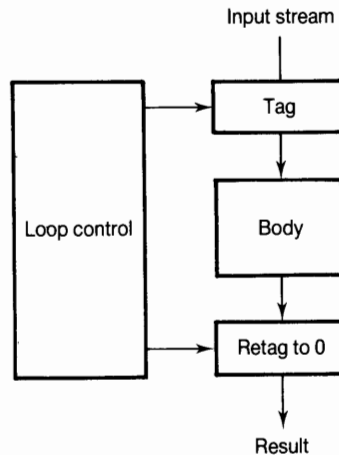


Figure 10.8 Token tagging in loops

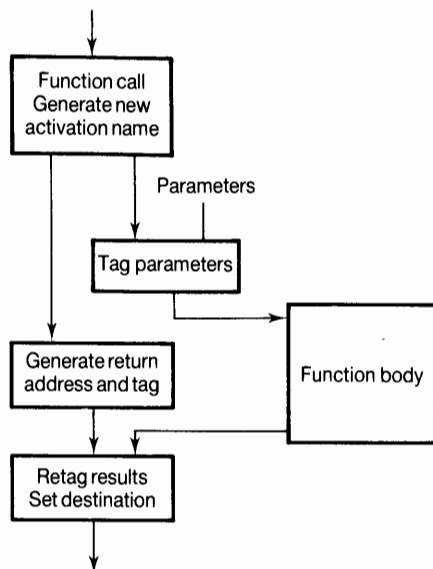


Figure 10.9 Function call using tags

A simple approach to using the index field to access elements of an array would be to sequence through the elements until the specific element having the required index value has been found. Suppose the index field operations include “read index”, RIX, and “set index”, SIX. Read index, RIX, can be used on each element to find its index value, which can then be compared to the required value using a normal arithmetic compare operation. When the required value has been found, a set index operation, SIX, could be used to set the index to 0 and extract the token. However, this procedure is somewhat inefficient. Large data structures require alternative approaches, normally returning to globally stored data referenced by pointers in the tokens. Then, the pure form of dataflow with all data passed within the tokens is abandoned for large data structures (arrays, records, etc.). Performing arithmetic operations on elements from two arrays can be done efficiently using coloring by simply passing the pairs of elements through a normal arithmetic operation node. Only pairs of elements with the same index will be processed together.

System examples

The dynamic dataflow machine designed at MIT by Arvind in the late 1970s (Arvind and Nikhil, 1990) uses an array of processing elements interconnected the same way as the static dataflow architecture shown in Figure 10.7. The processing elements incorporate a program memory and instruction queue connected to an arithmetic and

logic unit. A “waiting match” memory holds tokens as received at the input module of the processing element. When a full complement of tokens has been received, the appropriate instruction is fetched from the program memory and this instruction is executed with operands within the matched tokens. The result token is output to the routing network and directed to the destination processing element. Each processing element in the MIT dynamic architecture incorporates a memory called an *I-structure memory*, to hold array data structures which cannot reasonably be transmitted within the tokens, thus alleviating the restriction on array data structures. Additional tags are stored with the data structures to control access to the data structures.

Figure 10.10 shows the architecture of the dynamic dataflow computer system constructed at Manchester University by Gurd and Watson (1980). This system also uses a pipelined ring architecture, but separates the computational functions from the token/tag matching functions. Each processor executes 166-bit packages consisting of two data operands, a tag, an op-code and one or two destinations (i.e. the address of the next instruction or instructions). A system/computation flag is associated with all packets to differentiate between a computational packet to be executed by a processor and a packet carrying a system message (such as to load an instruction into memory). Data operands can be 32-bit integers or floating point numbers.

Once an executable packet has been executed, a processor generates one or two result token packets. A result token packet contains one data operand, a tag and the destination address(es), 96 bits in all. Each result token packet enters a two-way switch which enables data to be input from an external source or output to an external destination (peripheral device). Assuming the result token packet is to pass to another node, the packet is directed to a first-in first-out token queue and on to a matching unit. Here, a search is made to identify another token to complete the tokens necessary to fire a node (if the node requires two tokens). The search is done in hardware by comparing the destination and tag of the incoming token with all the stored tokens. If a match is found, a token-pair packet is formed, otherwise the incoming token is stored in the matching unit awaiting its matching token. Token-pairs and single operand tokens are passed on to the program store which holds the nodal instructions, and the full executable packet is formed. Executable packets are sent to free processors in an array of fifteen processors when possible.

On average, packets into the program node store and executable packets are expected to be formed every 300 ns, and token packets produced by processors at an average rate of one every 200 ns, but this will depend on the program. These figures indicate that three tokens must enter the matching unit in the period that two token packets are generated, or that one in every three tokens is for a single input node. Similarly, as executable packets enter the processor array at a rate of one every 300 ns, every two executable tokens must generate three token packets if the generation rate is one every 200 ns. In fact, the processors are microprogrammable devices requiring 4.5 μ s per machine operation and a throughput of one executable packet every 300 ns is achieved if all fifteen processors are active.

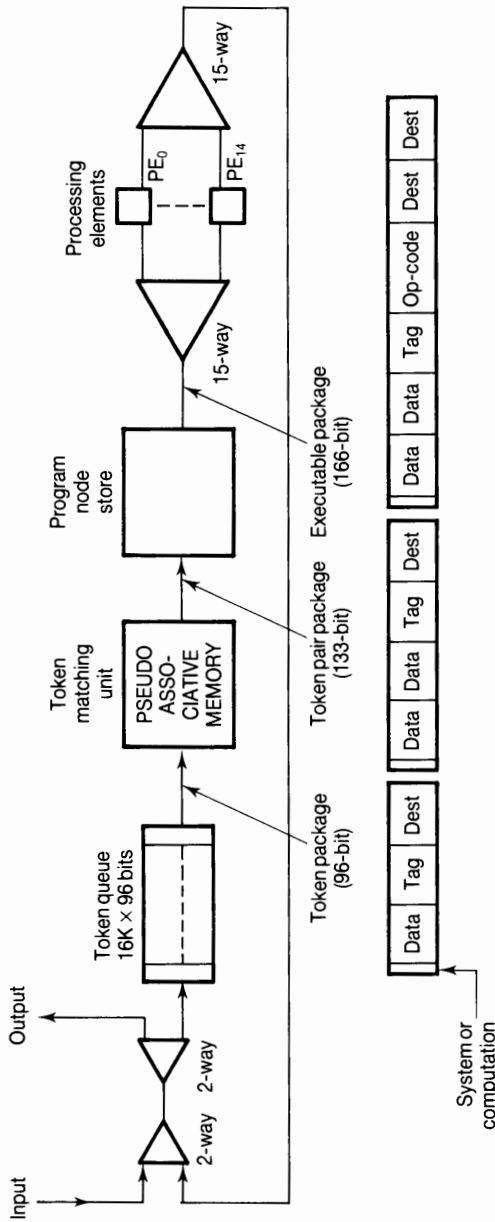


Figure 10.10 Manchester dataflow computer architecture

Hardware matching of a large number of tokens is problematical and is solved in the Manchester machine by employing a hardware hashing technique rather than by using a true associative memory, because such memory would be uneconomic.

10.3.3 VLSI dataflow structures

The dataflow technique can be applied to VLSI (very large scale integration) arrays of interconnected cells. Each cell performs primitive dataflow operations on data received via direct links from neighboring cells, and consists of a processor, a small stored program and input/output communication. The array is connected to a host system which downloads the programs into the cells. A cell will fire when all the operands for one of its stored instructions are received, and internal logic is necessary to detect when such enabling conditions occur. Each cell will perform the operations of one or more nodes in a dataflow graph. A typical operation might require one or two input operands from neighboring cells and produce one output operand to a neighboring cell. Figure 10.11 illustrates how a dataflow graph might be mapped onto a hexagonal dataflow array. In this case, each cell in the array executes one primitive operation of the graph.

A VLSI dataflow array has been demonstrated by Koren *et al.* (1988). Their dataflow architecture uses a static hexagonal interconnection cell pattern. The array is connected to a host system, and communication buses, which are connected to the host, are provided within the array for loading the program memories and extracting the results. Each processing element has all the main features of a normal stored program computer, including familiar instructions within its instruction set, such as arithmetic and logical operations and input/output. In addition, instructions are provided for cell initialization. Instructions can operate upon six periphery registers, R1 through to R6, which are provided for communicating to adjacent cells. Up to six instructions can be stored in each cell at any instant in a small stored instruction memory. There is one instruction that can produce an output for each periphery register. The contents of these registers are automatically transferred to the neighboring cell when this can accept the data in the corresponding periphery register. Each cell operates on 8-bit numbers. Sixteen- and 32-bit numbers can be processed using carry propagation techniques. Carry values are passed through the periphery registers defined by initialization instructions.

Instructions have 16 bits and the format shown in Figure 10.12. A 9-bit opcode is provided. Each of the input register bits, B_6 through to B_1 , can be set to 1 to indicate that the corresponding register holds one of the source operands. In fact, only up to three input registers can be specified in one instruction, two registers holding operands and one a carry-in value. Bit B_7 , labeled Frst, is used when the order of the input operands is important, such as in subtraction and division, to specify which of two input registers holds the first operand. For example, subtraction could be $(A - B)$ or reverse subtraction $(B - A)$.

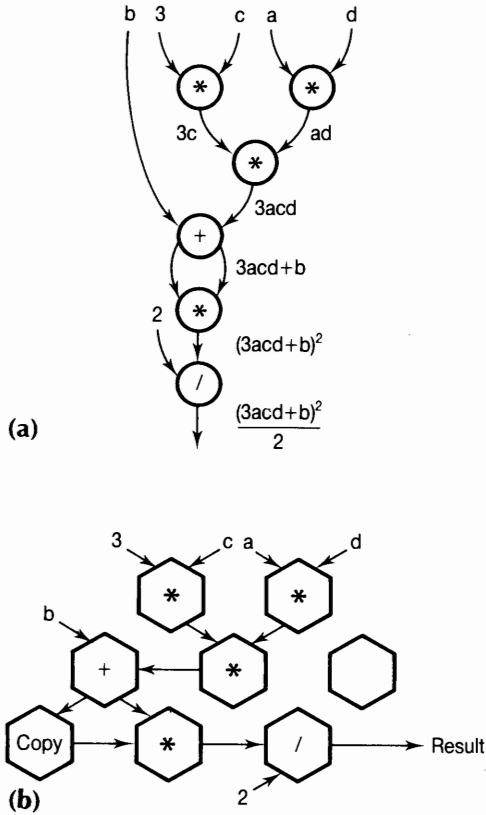


Figure 10.11 Dataflow graph mapped onto hexagonal array
 (a) Dataflow graph (b) Mapped on to hexagonal array

Operation code									Frst	Input registers					
B ₁₆	B ₁₅	B ₁₄	B ₁₃	B ₁₂	B ₁₁	B ₁₀	B ₉	B ₈	B ₇	B ₆	B ₅	B ₄	B ₃	B ₂	B ₁

Figure 10.12 Hexagonal array processor instruction format

As in all the dataflow systems described so far, instructions are not necessarily executed in the program sequence but when the required operands become available. As soon as a new operand is received via one of its periphery registers, internal logic establishes whether any waiting instructions require the operand. This is achieved using a set of *six static flags* in the instruction memory, one for each instruction, and a two-dimensional array of *dynamic flags* with seven rows and six columns. The static flags indicate that the contents of the corresponding register have to be transferred to the corresponding register in the neighboring cell, i.e. that the instruction is present.

Each column in the dynamic flag array is assigned to one of the six registers. Each of the first six rows is assigned to one of the six instructions that might be present in the instruction memory. The seventh row in the dynamic array is used by the intercell register transfer logic. The flags in the j th column are set to 1 when a new operand arrives in the j th register and the corresponding instruction requires the operand. This is achieved by logically ANDing the operand arrival set signal with the static flags. Each time an instruction is executed, the flags in the corresponding row of the flag array are reset to 0. Hence, when all bits in a column have been reset to 0, the input operand has been used by all waiting instructions. The input register becomes "empty" and ready for new data to be loaded from adjacent cells. The conditions for an instruction to be executed ("fire") are as follows:

1. Input register dynamic flags values in instruction row = static flags values indicating that all input operands are present.
2. Required output register is empty.

The dataflow graph from the application problem needs to be mapped onto the array; heuristics, as described by Koren *et al.* (1988), may be necessary to perform this mapping. In general, some cells are used simply to route operands towards their final destination. The PE (processing element) cell utilization percentage describes the percentage of cells that perform computations; Koren reports utilization percentages of between 7.4 and 75 per cent, depending on the method of mapping for a range of problems.

The array dataflow system is very similar to the message-passing systems described in Chapter 9, in that programs are loaded into independent nodes prior to execution and information passes from node to node. Indeed, a general purpose message-passing system could be programmed to operate on dataflow principles, though without operand matching facilities it would need to be at a reasonably coarse grain level.

10.3.4 Dataflow languages

Traditional procedural high level languages such as Pascal and FORTRAN could be pressed into service for dataflow computers, with some modifications or additions to

the language. However, procedural languages grew from the development of sequential computations/computers and are not ideal as dataflow languages, particularly for the “fine grain” dataflow computing we have described so far.

Some key attributes of dataflow to be incorporated into dataflow languages are:

1. Parallelism constrained by data dependencies.
2. Locality of effect characteristic and freedom from side effects.
3. Lack of variables that can be altered.
4. Lack of stored values that can affect subsequent computations.

Locality of effect refers to operations only affecting values within a defined area. The phrase *freedom from side effects* has been coined to describe the general situation when an action in a program does not have unfortunate far-reaching consequences. These consequences might be unexpected and lead to errors. An example of side effects is a procedure which alters global variables in the calling program. The variables may be parameters passed to the procedure during the call, or worse, global variables directly referenced in the body of the procedure. Clearly, this effect could be avoided by not using global variables at all and by passing parameters by reference (name). Side effects cause difficulties in establishing program correctness and data dependencies for parallel computations. Problems can occur, particularly with data structures such as arrays and records, in which a change in one element of the structure at one part of a program might have significant effects in another part of the program.

Prime candidates for dataflow programming languages are those within the class of languages called *functional languages*. A program written in a functional language consists only of functions where a function is called by name, supplied with parameters and returns a value (or values), as in conventional high level languages. However, a functional language program is written entirely of functions. The program usually consists of a main function, which calls other functions, which may in turn call other functions, and so on, until a bottom level is reached in which the functions are composed of language primitives. There are no assignment statements, except that a program variable can be given a value once. The values of variables never change, i.e. variables are more like defined constants.

There is research interest in functional languages because they might help write better structured programs, particularly parallel programs, and not necessarily for dataflow computers as the machine architecture. The reader is directed to Peyton Jones (1989) for details. We can make some general observations regarding functional languages.

Functional languages have several potential advantages, notably that a function cannot have any effect on the program except to return a value (or values, in the case of some function languages). They are free from side effects. Apart from eliminating a source of errors, freedom from side effects also helps parallel computations. As side effects are not present to alter expressions, the order of evaluation of expressions does not matter, and variables and their value can be interchanged at

any time. This is known as *referential transparency*. Clearly, programmers could restrain themselves from using those features in a programming language which lead to possible side effects. A program could be written in a normal programming language as a series of functions. Hughes (1989) makes the point in his defence of functional languages that simply omitting features in a language cannot make the language more powerful; this is not sufficient. In addition to prohibiting programming styles which lead to side effects, functional languages also provide special program constructs to put together complex functions from simpler functions, and deal with lists (as in LISP, the first major functional language).

Functional languages have at the most one assignment for each program variable. Such languages are known as *single assignment languages*, and form the development of most, if not all, dataflow languages. There is a direct relationship between computations in dataflow graphs and single assignment. A program value is generated only at a dataflow node and cannot be altered elsewhere, except as a generated new data token. The single assignment convention carries over this concept that data operands can only be altered once, creating a new value. That is, variable names can only appear on the left hand side of an expression once. This implies that the program sequence:

A := 1;		A := 1;
A := A + 1;	would be rewritten as	A' := A + 1;
A := A + B + C;		A'' := A' + B + C;

where A' and A'' are new variables introduced to maintain the single assignment rule. Fortunately, by using language constructs, such transformations can be avoided in many instances when they occur in loops in single assignment languages.

Iterations in single assignment languages employ special notations so as not to break the single assignment rule. In normal languages, one makes use of a loop variable to count the number iterations of a loop and perhaps also within expressions in the loop body. To take a very simple example, to generate factorial 10!, we might write in Pascal:

```
fact := 1;
FOR i := 2 to 10 DO fact := fact*i;
```

However, this would be disallowed in a single assignment language because fact is reassigned a new value on each iteration through the loop. One "solution" would be to unfold the iteration into separate computations, i.e. fact := 1; fact := fact*2; fact := fact*3; etc., and rename fact for each successive statement. We have seen that dynamic dataflow architectures can handle iterations by coloring tokens with an iteration tag, and hence there is a simple graphical transformation for the factorial loop. Some dataflow languages have reserved words such as NEW or OLD to identify successive values, for example:

```
i := OLD i + 1;
fact := OLD fact*i;
```

Several dataflow languages have been developed, often in connection with dataflow work on machine architectures. For example, the dataflow language VAL (Value-oriented Algorithmic Language) was developed in the late 1970s and early 1980s at MIT under the direction of Dennis, and also at Lawrence Livermore National Laboratory. A detailed description and analysis of VAL is given by McGraw (1982).

Subsequently, SISAL (Streams and Iterations in a Single Assignment Language) was developed, though still as a research language. Details of the SISAL language are given by Allan and Oldehoeft (1985). SISAL was used in the development of the Manchester dataflow computer (Böhm and Sargeant, 1989). Streams and iterations are fundamental aspects of dataflow programming. A stream of tokens are used in dataflow to describe a sequence of data items like an array, except that the elements can be processed as they are generated rather than having to wait for the complete set of elements to be generated, as would be necessary with an array. The possibility of parallel computations on stream elements exists. We will briefly mention some of the unusual features of SISAL, which are characteristics of dataflow languages and are therefore representative. Of course, other dataflow languages may have different reserved words and syntax.

Functions can return more than one value, which are assigned to successive “variables”. For example:

```
a,b,c := funct0(par1)
```

evaluates the function `funct0`, using the value of the parameter `par1`, returning three results which are assigned to `a`, `b` and `c` respectively. The function could return a stream.

Changing one element of an array implies that a completely new array is created in a functional language. In SISAL:

```
B := A[i:v]
```

creates a new array, `B`, which is the same as array `A` except that the i th element is changed to the value v . In a dataflow system which stores arrays, an alternative to copying the array is to modify the stored array, a decision that could be made by the compiler, dependent upon the use of the arrays.

Sequential expressions, which are similar to conventional programming languages except that each must return a value, exist in dataflow languages. For example, in SISAL, the IF THEN ELSE expression could be used in the statement:

```
a := IF b = c THEN funct1 ELSE funct2 END IF
```

If the Boolean expression $b = c$ is TRUE, `funct1` is evaluated and its result is returned to be assigned to `a`, otherwise `funct2` is evaluated and its result is

returned and assigned to a . This construct is very similar to that allowed in ALGOL 68 and C.

Iterative sequential expressions, in which each evaluation within a loop has to be executed in strict sequence because of data dependencies, can use the FOR-INITIAL expression. The FOR-INITIAL expression consists of four parts, initialization with the reserved word INITIAL to specify the initial values, the body of the loop specifying the repetitive computation, loop termination conditions with WHILE or UNTIL placed before the loop body if tested before the loop begins, or after the loop body if tested at the end of each loop executed, and finally RETURNS followed by a return expression to specify the values returned. The return expression can be VALUE OF, ARRAY OF or STREAMS OF. VALUE OF can also be extended to include VALUE OF SUM and other "reduction" operators. VALUE OF SUM var returns the value of the arithmetic summation of var. For example:

```
FOR INITIAL
  i := 1;
  j := 1
WHILE i < 10
REPEAT
  j := OLD j * i;
  i := OLD i + 1
RETURNS STREAM OF j
END FOR
```

returns a stream of factorial values 1, 2, 6, 24

Iterations that can be performed in parallel (i.e. without any data dependencies) have a separate "for-all" FOR expression, for example:

```
FOR i IN 1,n
  Sum := A[i]*i
  RETURNS VALUE OF SUM sum
END FOR
```

returns the summation of the elements after each is multiplied by i . The construct IN 1, n specifies that there will be n activations of the loop, each with a successive value of i from 1 to n . Each activation can proceed in parallel. The construct VALUE OF SUM sum specifies that each value of sum computed is added together.

For-all iteration expressions are available to access multidimensional arrays, either with the inner (dot) product range of indices or the outer product range of indices, using DOT and CROSS respectively. For example i IN 1, 4 DOT j IN 8, 11 generates the indices [1,8], [2,9], [3,10] and [4,11] and i IN 1, 5 CROSS j IN 11, 15 generates the indices [1,1], [2,1], [3,1], [4,1], [1,2], [2,2], [3,2], [4,2], [1,3], [2,3], [3,3], [4,3], [1,4], [2,4], [3,4] and [4,4]. There is more than one method by which the system implements the VAL OF SUM summation (and other reduction

operators). The method used may result in different values producing overflow conditions and the language definition fails to be sufficiently precise if the actual method is not specified, as in VAL (McGraw, 1982). For example, the summation could be done by adding successive values to an accumulating sum, the most likely method for a single processor system. Alternatively, a tree could be formed with pairs of values added at each node of the tree and partial results passed upwards towards the root of the tree. This method would allow each node to be done on a separate processor concurrently. It is possible that one method may produce an overflow not occurring in the other method. In SISAL, the associativity order can be specified using LEFT, RIGHT and TREE.

Internal error detection and recovery in a traditional computer often relies on errors being detected by processor hardware during the execution of an instruction. An interrupt mechanism is activated to make the processor execute an error routine. This method may be hard to do in a dataflow computer with substantial concurrency. In VAL and SISAL, error handling is performed in a novel way by introducing error values within each data type. When an error is produced at a node, the appropriate error value is generated in place of a valid numeric or Boolean data value. For example, in VAL, positive and negative overflow are indicated with the error values POS_OVER, and NEG_OVER respectively. Underflow is indicated by POS_UNDER and NEG_UNDER. An error condition not identified specifically has the error value UNDEF. Boolean values now become three-valued, TRUE, FALSE and UNDEF. SISAL has the values UNDEF, to cover all arithmetic errors, and BROKEN, to indicate some form of control flow error preventing the desired result being generated. Error values are propagated through successive nodes. Both languages have methods of testing for errors.

There are other languages for dataflow, for example the dataflow language Lucid (Wadge and Ashcroft, 1985).

10.4 Macrodataflow

10.4.1 General

The dataflow mechanism as described operates at the instruction level, i.e. so-called *fine grain dataflow*. However, there is a high communications overhead in passing operands. A mechanism to reduce this communications overhead is to apply dataflow at the procedural level, so-called *coarse grain dataflow*. It is also possible to have variable grain dataflow in which nodes might represent simple operations through to complete sequential procedures. A number of phrases have been invented in the literature to describe the variations in dataflow. For example, coarse grain/variable grain dataflow can be described as *combined dataflow/control flow*. We shall use the term *macrodataflow* (a term also used by Gajski *et al.* (1983) and others) to cover dataflow in which each node can represent a complex serial function. Macrodataflow

graphs assume the same overall construction as fine grain dataflow graphs, in that each node is interconnected to other nodes by arcs carrying tokens. The tokens can carry single data items. Given that the nodes represent procedures/functions, the input tokens carry procedure/function parameters, and the output tokens carry procedure/function results.

Macrodataflow node firing rules can be:

1. Standard firing rule – node fires only when all of the operands are received.
2. Non-standard firing rule – node fires when certain specified operands are received. Each nodal operation is completed when all necessary operands are received. (The necessary operands may not be all of the operands.)

The second firing rule, a variation of the first, allows part of a procedure/function to be executed while waiting for additional parameters to complete the procedure/function. The firing rule would be inappropriate for a fine grain dataflow system because the fired operations of fine grain dataflow are of a simple type and would normally require all the inputs to perform any meaningful processing. The non-standard firing rule includes the possibility that one or more of the input tokens need not ever be received for the node to fire. These tokens become “don’t care” conditions. The tokens that must arrive to start the node firing, those which must arrive eventually and the “don’t care” conditions can be specified in the nodal enabling conditions, for example, for each input arc, 1 = token must arrive to start, 0 = token can arrive later and X = “don’t care” condition.

10.4.2 Macrodataflow architectures

The original fine grain dataflow systems, operating either as static or dynamic dataflow, employed ring structure architectures, the VLSI array dataflow systems being an exception. Macrodataflow (coarse grain) computing tends to suggest more conventional architectures containing a number of processing elements centrally controlled with access to a common memory. An example of a suitable system is the Cedar computer system (Gajski *et al.*, 1983) shown in Figure 10.13. A number of processor clusters connect to a global memory through a global routing network. Each processor cluster consists of a number of processing elements connecting to a number of local memory units through a local routing network and controlled by a cluster control unit within the processor cluster. The processor clusters are controlled by a global control unit. The program is a directed graph which is loaded into the global control unit which controls the execution of the program.

A macrodataflow architecture presented by Ayyad and Wilkinson (1987) employs a shared memory system but with the unusual characteristic that the memory provides access to specific locations, rather than the processor using bus arbitration to access the memory. Access to specific memory locations is offered and identified to the processors by broadcasting the memory addresses in a repeating sequence.

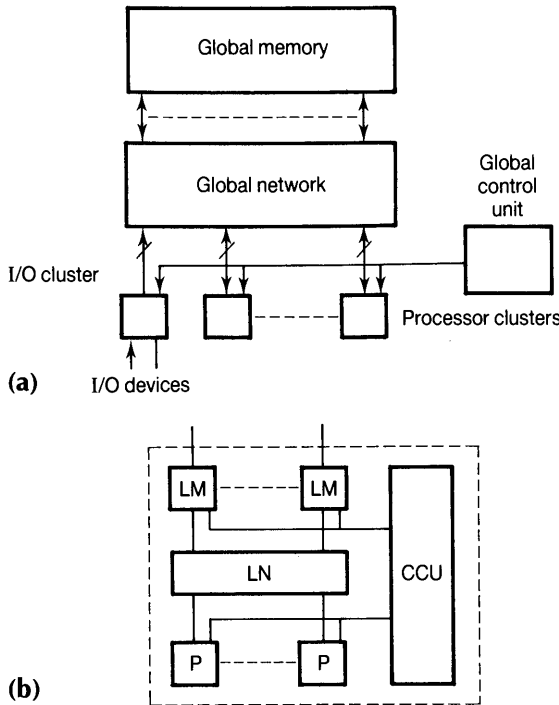


Figure 10.13 Cedar computer system (a) Architecture (b) Processor cluster (LM, local memory; LN, local network; P, processor; CCU, cluster control unit)

The sequence of locations offered can be found from knowledge of those nodes not yet satisfied. Figure 10.14(a) shows an implementation using a common bus interconnection system. A number of processors, each with local memory, are connected to a common memory through a single bus. Logic in each processor compares the incoming address from the common memory module with the address required by the processor. When a match is found, a data path is established between the common memory and the processor. Though a common bus is used here, no bus contention occurs and no arbitration logic is required. The cycling can operate at very high speed and is only limited by the address generation logic and the comparators; the address sequencing is only temporarily halted for memory accesses when necessary. Simultaneous writing to the same location in the common memory by more than one processor must be inhibited in the scheme. In macrodataflow, simultaneous writing of parameters into a common memory location should never occur.

An implementation for a cross-bar switch architecture is shown in Figure 10.14(b) and requires each memory module to be provided with address generation logic but, significantly, no additional traditional arbitration logic is required. Typically, the

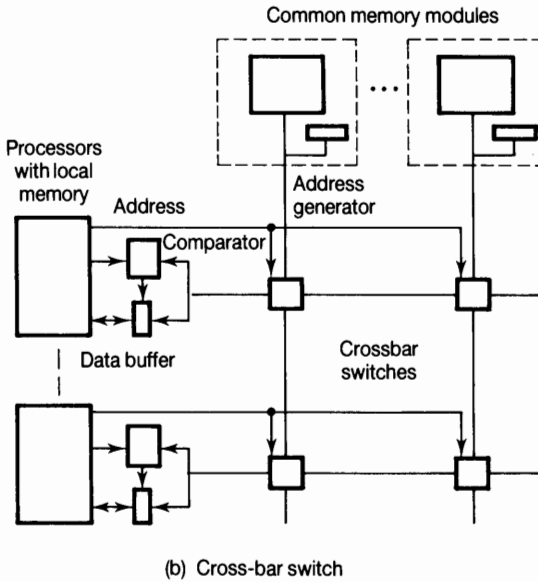
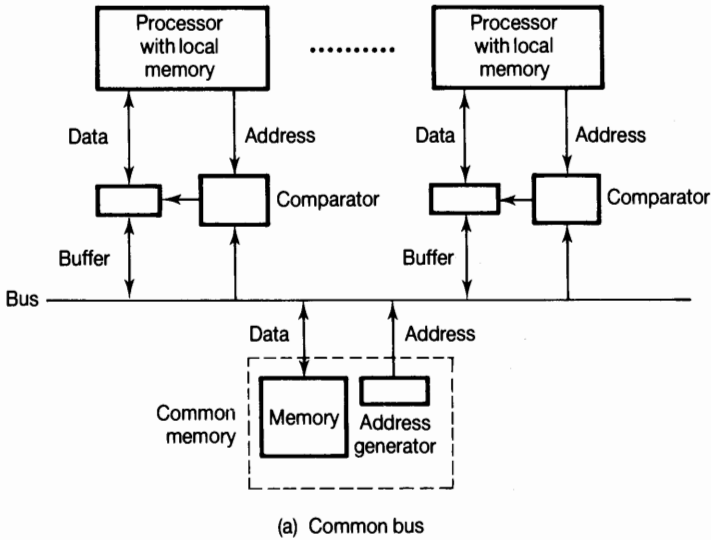


Figure 10.14 Macrodata flow architecture (a) Common bus (b) Cross-bar switch

upper address bits from the processing elements select the memory module. In the cross-bar switch implementation it is possible for more than one processor to access the same memory module during the same address generation cycle. In a normal cross-bar switch system this is prevented by the arbitration logic.

In dataflow, and in macrodataflow computations in particular, we would expect a processor to be waiting for more than one parameter on occasion. To accommodate multiple requests, the system can be provided with additional comparator circuits. As an alternative to using discrete comparators, particularly for multiple requests, content addressable memories (CAMs) could be employed, which compare all stored values with an applied value simultaneously. In this application, the addresses of locations required by the processor are loaded into a CAM as they are generated. Addresses generated by the cycling device are compared simultaneously with all stored addresses by the internal logic of the CAM. If any match is found, the processor is informed, perhaps by an interrupt signal. A signal is also passed to the cycling device to prevent it continuing with the next address and a data path is established.

10.5 Summary and other directions

Dataflow has had a fairly long development time, with a few research groups studying the technique in earnest since the mid-1970s without it becoming widespread in commercial use. Admittedly, the pressure of market forces to maintain compatibility with existing systems greatly inhibit the introduction of a radically different computer system requiring a different style of programming and different programming languages. The dataflow idea of operations only being performed when the operands are available was applied to a traditional stored program computer in the IBM 360/91 in the form of internal forwarding (Section 4.2.4, page 122) in the later 1960s and predates dataflow architectural developments. However, this early form of dataflow was abandoned after the introduction of cache memories, which achieved better performance with less complicated hardware. Subsequently, commercial computer manufacturers studied the technique for a while, for example Digital Equipment Corporation, with their initial involvement with the Manchester dataflow project. It would seem that dataflow will become a general computer technique just as pipelining is a general technique and will find some applications. Hence it is included in this text.

We have developed the designs of computers, starting from enhancements to the single processor stored program computer (Part I) through to the application of more than one processor sharing a memory (Part II) and finally to systems which do not share globally stored data (Part III). There are some other possible computer designs not dealt with, such as neural computers, computers based upon optical technology and architectures for implementing functional languages. There have been a few experimental systems for implementing functional languages, especially in England,

including the ALICE machine at Imperial College, London and the GRIP (Graph Reduction in Parallel) machine at University College, London. For the most part, these experimental projects have used architectures that do not look significantly different to shared/local memory multiprocessor systems, except for the support given to the computational model. The specialized ring architectures such as the early dataflow computer architectures have not been used. The reader is referred to Peyton Jones (1989) and the references contained in this paper for further information on research directions on systems for functional programming.

PROBLEMS

10.1 Draw a dataflow graph to compute the function:

$$T = (x + y)(x - y)/(x^2 - y^2)$$

Show the movement of tokens through the graph.

10.2 Draw a dataflow graph to achieve a BRANCH operation using two MERGE operations and a dataflow graph to achieve a MERGE operation using two BRANCH operations.

10.3 Identify the situations in which both fields of a two-field tag, activation name and index, would need to change.

10.4 Determine the computation performed by the SISAL code segment:

```
FOR x IN data
  RETURNS
    VALUE OF SUM x
  END FOR
```

where data is a stream of integers.

10.5 The following is a code segment written in the VAL language:

```
FORALL i IN [1, n]
  x := IF i = 2 THEN x
  EVAL PLUS x*i
  PLUS x
ENDALL
```

which is similar to SISAL. Deduce the computation performed.

10.6 Draw a dataflow graph for the SISAL program:

```
FOR i IN 1,n
  x := IF (i/2)*2 = i THEN x
  RETURNS SUM OF x
END FOR
```

10.7 Write a program in SISAL to perform matrix multiplication of two $n \times n$ matrices.

10.8 Write a SISAL program to compute the numerical integration of a function $f(x)$ by dividing the integral into small areas and computing each area using trapezoidal approximation (see Section 9.2.2, page 305).

10.9 Draw dataflow graphs for the C programs:

```
(a) if ((a == b) && (c < d)) c = c - a;
    else c = c + a;
(b) for (i = 1; i < m; i++) {
    c[i] = 0;
    for (j = 1; j < n; j++) c[i] = c[i] + a[i]*b[i];
}
```

Rewrite (a) in SISAL.

10.10 Design a VLSI array dataflow system using a north–south–east–west mesh interconnection pattern. Give details of the instruction memory and matching logic.

10.11 Make a study of the viability of a variable grain dataflow system on a shared memory multiprocessor system. What types of hardware support would you envisage being required?

10.12 Discuss how a message-passing multiprocessor system could operate on dataflow principles.

References and further reading

- Abraham, S. and K. Padmanabhan (1989), "Performance of the direct binary n -cube network for multiprocessors", *IEEE Trans. Comput.*, **38**, no. 7, 1000–11.
- Advanced Micro Devices (1988), *32-bit Microprogrammable Products Am29C300/29300 Data Book*, Sunnyvale, California.
- Agarwal, A., M. Horowitz and J. Hennessy (1989), "An analytical cache model", *ACM Trans. Comp. Syst.*, **7**, no. 2, 184–215.
- Agrawal, P. D., V. K. Janakiram and G. C. Pathak (1986), "Evaluating the performance of multicomputer configurations", *IEEE Computer*, **19**, no. 5, 23–7.
- Allan, S. J. and R. R. Oldehoeft (1985), "HEP SISAL: Parallel functional programming", in *MIMD Computation: The HEP supercomputer and its applications*, J. S. Kowalik (ed.), MIT Press: Cambridge, Massachusetts, 123–50.
- Andrews, G. R. and F. B. Schneider (1983), "Concepts and notations for concurrent programming", *Computer Surveys*, **5**, no. 1, 3–43.
- Arvind and R. S. Nikhil (1990), "Executing a program on the MIT tagged-token dataflow architecture", *IEEE Trans. Comput.*, **39**, no. 3, 300–18.
- Athas, W. C. and C. L. Seitz (1988), "Multicomputers: Message-passing concurrent computers", *IEEE Computer*, **21**, no. 8, 9–24.
- August, M. C., G. M. Brost, C. C. Hsiung and A. J. Schiffler (1989), "Cray X-MP: The birth of a supercomputer", *IEEE Computer*, **22**, no. 1, 45–52.
- Ayyad, A. and B. Wilkinson (1987), "Multiprocessor scheme with application to macro-dataflow", *Microprocessors Microsyst.*, **11**, no. 5, 255–63.
- Babb II, R. G. (1984), "Parallel processing with large-grain data flow techniques", *IEEE Computer*, **17**, no. 7, 55–61.
- Babb II, R. G. (1985), "Programming the HEP with large-grain data flow techniques", in *MIMD Computation: The HEP supercomputer and its applications*, J. S. Kowalik (ed.), MIT Press: Cambridge, Massachusetts, 203–27.
- Babb II, R. G. (ed.) (1988), *Programming Parallel Processors*, Addison-Wesley: Reading, Massachusetts.
- Baer, J.-L. (1980), *Computer Systems Architecture*, Computer Science Press: Rockville, Maryland.
- Banning, J. (1979), "Z-bus and peripheral support packages tie distributed computer systems together", *Electronic Design*, **27**, no. 24.
- Barron, I., P. Cavill and D. May (1983), "Transputer does 10 or more MIPS even when not used in parallel", *Electronics*, November, 109–15.

358 References and further reading

- Baskett, F. and A. J. Smith (1976), "Interference in multiprocessor computer systems with interleaved memory", *Comm. ACM*, **19**, 327–34.
- Batcher, K. E. (1980), "Design of a massively parallel processor" *IEEE Trans. Comput.*, **C-29**, 836–40.
- Beetem, J., M. Denneau and D. Weingarten (1987), "The GF11 parallel computer", in *Experimental Parallel Computer Architectures*, J. J. Dongarra (ed.), North-Holland: Amsterdam, 255–98.
- Beims, B. (1984), "Multiprocessing capabilities of the MC68020 32-bit microprocessor", Application Note AR220, Motorola Inc.
- Belady, L. (1966), "A study of replacement algorithms for a virtual-store computer", *IBM Systems Journal*, **5**, no. 2, 78–101.
- Bell, J., D. Casasent and C. G. Bell (1974), "An investigation of alternative cache organizations", *IEEE Trans. Comput.*, **C-23**, no. 4, 346–51.
- Benes, V. E. (1965), *Mathematical Theory of Connecting Networks and Telephone Traffic*, Academic Press: New York.
- Bernstein, A. J. (1966), "Analysis of programs for parallel processing", *IEEE Trans. Elec. Comput.*, **E-15**, 746–57.
- Bhandarkar, D. P. (1975), "Analysis of memory interference in multiprocessors", *IEEE Trans. Comput.*, **C-24**, no. 9, 897–908.
- Bhuyan, L. N. (1985), "An analysis of processor–memory interconnection networks", *IEEE Trans. Comput.*, **C-34**, no. 3, 279–83.
- Bhuyan, L. N. (1987), "Interconnection networks for parallel and distributed processing", *IEEE Computer*, **20**, no. 5, 9–12.
- Bhuyan, L. N. and D. P. Agrawal (1983), "Design and performance of generalized interconnection networks", *IEEE Trans. Comput.*, **C-32**, no. 12, 1081–9.
- Bhuyan, L. N. and D. P. Agrawal (1984), "Generalized hypercube and hyperbus structures for a computer network", *IEEE Trans. Comput.*, **C-33**, no. 1, 323–33.
- Blevins, D. W., E. W. Davis, R. A. Heaton and J. H. Reif (1988), "Blitzen: A highly integrated massively parallel machine", *Proc. 2nd Sym. Frontiers of Massively Parallel Computations*, October, Fairfax, Virginia.
- Böhm, A. P. W. and J. Sargeant (1989), "Code optimization for tagged-token dataflow machines", *IEEE Trans. Comput.*, **38**, no. 1, 4–14.
- Bouknight, W. J., S. A. Denenberg, D. E. McIntyre, J. M. Randall, A. H. Sameh and D. L. Slotnick (1972), "The Illiac IV system", *Proc. IEEE*, April, 369–88. Reprinted (1982), in *Computer Structures Principles and Examples*, D. P. Siewiorek, C. G. Bell and A. Newell, McGraw-Hill: New York.
- Broomell, G. and J. R. Heath (1983), "Classification categories and historical development of circuit switching topologies", *Computing Surveys*, **15**, no. 2, 95–133.
- Buehrer, R. and K. Ekanadham (1987), "Incorporating data flow ideas into von Neumann processors for parallel execution", *IEEE Trans. Comput.*, **C-36**, no. 12, 1515–22.
- Burns, A. (188), *Programming in Occam 2*, Addison-Wesley: Wokingham, England.
- Buzbee, B. (1984), "Parallel processing makes tough demands", *Computer Design*, September, 137–40.
- Chaitin, G. J., M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins and P. W. Markstein (1981), "Register allocation via coloring", *Computer Languages*, **6**, 47–57.
- Chang, D. Y., D. J. Kuck and D. H. Lawrie (1977), "On the effective bandwidth of parallel memories", *IEEE Trans. Comput.*, **C-26**, no. 5, 480–90.

- Chen, T. C. (1980), "Overlap and pipeline processing", in *Introduction to Computer Architecture*, H. S. Stone *et al.*, 2nd ed., SRA: Chicago, 427–85.
- Cheng, H. (1989), "Vector pipelining, chaining and speed on the IBM 3090 and Cray X-MP", *IEEE Computer*, **22**, no. 9, 31–46.
- Chu, W. W. and H. Opderbeck (1976), "Program behavior and the page-fault-frequency replacement algorithm", *IEEE Computer*, **9**, no. 11, 29–38.
- Clark, D. W. and J. S. Emer (1985), "Performance of the VAX-11/780 translation buffer: Simulation and measurement", *ACM Trans. Comput. Syst.*, **3**, no. 2, 31–62.
- Conti, C. J., D. H. Gibson and S. H. Pikowsky (1968), "Structural aspects of the system 360/85: General organization", *IBM Systems Journal*, 2–14.
- Conway, M. E. (1963), "A multiprocessor system design", *Proc. AFIPS Fall Joint Computer Conf.*, **4**, 139–46, Spartan Books: Baltimore, Maryland.
- Dally, W. and C. L. Seitz (1987), "Deadlock-free message routing in multiprocessor interconnection networks", *IEEE Trans. Comput.*, **C-36**, no. 5, 547–53.
- Das, C. R. and L. N. Bhuyan (1985), "Bandwidth availability of multiple-bus multiprocessors", *IEEE Trans. Comput.*, **C-34**, no. 10, 918–26.
- Dasgupta, S. (1990), "A hierarchical taxonomic system for computer architectures", *IEEE Computer*, **23**, no. 3, 64–74.
- Davidson, E. S. (1971), "The design and control of pipelined function generators", *Proc. 1971 Int. Conf. on Systems, Networks and Computers*, Oaxtepec, Mexico, 19–21.
- Denning, P. J. (1968), "The working set model for program behavior", *Comm. ACM*, **11**, no. 11, 323–33.
- Denning, P. J. (1970), "Virtual memory", *Computing Surveys*, **2**, no. 3, 153–89.
- Denning, P. J. (1980), "Working sets past and present", *IEEE Trans. Soft. Eng.*, **SE-6**, no. 1, 64–84.
- Denning, P. J. and D. R. Slutz (1978), "Generalized working sets for segmented reference strings", *Comm. ACM*, **21**, no. 9, 750–9.
- Dennis, J. B. (1974), "First version of a data flow procedure language", *Lecture Notes in Computer Science*, **19**, 362.
- Detmer, R. (1985), "Chip architecture for parallel processing", *Electronics and Power*, March, 227–31.
- Dijkstra, E. W. (1968), "Cooperating sequential processes", in *Programming Languages*, F. Genuys (ed.), Academic Press: New York, 43–112.
- Dubois, M., C. Scheurich and F. A. Briggs (1988), "Synchronization, coherence and event ordering in multiprocessors", *IEEE Computer*, **21**, no. 2, 9–21.
- Duncan, R. (1986), *Advanced MSDOS*, Microsoft Press: Redmond, Washington.
- Easton, M. E. and P. A. Franaszek (1979), "Use bit scanning in replacement decisions", *IEEE Trans. Comput.*, **C-28**, no. 2, 133–41.
- Feng, T.-Y. (1972), "Some characteristics of associative/parallel processing", *Proc. 1972 Sagamore Computing Conf.*, 5–16.
- Feng, T.-Y. (1981), "A survey of interconnection networks", *IEEE Computer*, **14**, no. 12, 12–27.
- Fernbach, S. (ed.) (1986), *Supercomputers Class VI Systems, Hardware and Software*, North-Holland: Amsterdam.
- Fisher, J. A. (1984), "The VLIW machine: a multiprocessor for compiling scientific code", *IEEE Computer*, **17**, no. 7, 45–54.
- Flynn, M. J. (1966), "Very high speed computing systems", *Proc. IEEE*, **12**, 1901–9.

360 References and further reading

- Gabriel, J., T. Lindholm, E. L. Lusk and R. A. Overbeek (1985), "Logic programming on the HEP", in *MIMD Computation: The HEP supercomputer and its applications*, J. S. Kowalik (ed.), MIT Press: Cambridge, Massachusetts, 181–202.
- Gajski, D., D. Kuck, D. Lawrie and A. Sameh (1983), "Cedar: A large scale multiprocessor", *Proc. 1983 Int. Conf. on Parallel Processing*, IEEE, 524–9.
- Gajski, D. and J.-K. Peir (1985), "Essential issues in multiprocessor systems", *IEEE Computer*, **18**, no. 6, 9–27.
- Gehani, N. and A. D. McGettrick (eds.) (1988), *Concurrent Programming*, Addison-Wesley: Wokingham, England.
- Gimarc, C. E. and V. M. Milutinović (1987), "A survey of RISC processors and computers of the mid 1980s", *IEEE Computer*, **20**, no. 9, 59–69.
- Glauert, J. R. W., J. R. Gurd and C. C. Kirkham (1985), "Evolution of a dataflow architecture", in *Concurrent Languages in Distributed Systems*, G. L. Reijns and E. L. Daglass (eds.), Elsevier Science Publishers B. V.: North-Holland, 1–15.
- Goodman, J. R. and C. H. Séquin (1981), "Hypertree: A multiprocessor interconnection topology", *IEEE Trans. Comput.*, **C-30**, no. 12, 923–33.
- Gottlieb, A., R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph and M. Snir (1983), "The NYU ultracomputer: designing a MIMD shared memory parallel computer", *IEEE Trans. Comput.*, **C-32**, no. 2, 175–89.
- Goyal, A. and T. Agerwala (1984), "Performance analysis of future shared storage systems", *IBM J. Res. Develop.*, **28**, no. 1, 95–107.
- Grimsdale, R. L. (1984), "Programming languages", in *Distributed Computing*, F. B. Chambers, D. A. Duce and G. P. Jones (eds.), Academic Press: London.
- Grossman, C. P. (1985), "Cache-DASD storage design for improving system performance", *IBM Systems Journal*, **24**, no. 314, 316–34.
- Gupta, A. (ed.) (1987), *Multi-Microprocessors*, IEEE Press: New York.
- Gupta, A. and H.-M. D. Toong (1985), "Increasing throughput of multiprocessor systems", *IEEE Trans. Ind. Electronics*, **IE-32**, no. 3, 260–7.
- Gurd, J. R., C. C. Kirkham and I. Watson (1985), "The Manchester prototype dataflow computer", *Comm. ACM*, **28**, no. 1, 34–52.
- Gurd, J. R. and I. Watson (1980), "Data driven system for high speed parallel computing: Part 2—Hardware design", *Computer Design*, July, 97–106.
- Hallin, T. G. and M. J. Flynn (1972), "Pipelining of arithmetic functions", *IEEE Trans. Comput.*, August, 880–6.
- Händler, W. (1977), "The impact of classification schemes on computer architecture", *Proc. Int. Conf. Parallel Processing*, August, 7–15.
- Hayes, J. P. (1988), *Computer Architecture and Organization*, 2nd ed., McGraw-Hill: New York.
- Hellerman, H. (1966), "On the average speed of a multiple-module storage system", *IEEE Trans. Electronic Comput.*, August, 670.
- Hennessy, J. L. (1984), "VLSI processor architecture", *IEEE Trans. Comput.*, **C-33**, no. 12, 1221–46.
- Hennessy, J. L. and D. A. Patterson (1990), *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann: San Mateo, California.
- Hill, M. D. (1988), "A case for direct-mapped caches", *IEEE Computer*, **21**, no. 12, 25–40.
- Hillis, W. D. (1985), *The Connection Machine*, MIT Press: Cambridge, Massachusetts.
- Hoare, C. A. R. (1974), "Monitors: An operating system structuring concept", *Comm. ACM*, **17**, no. 10, 549–57.

- Hoare, C. A. R. (1978), "Communicating sequential processes", *Comm. ACM*, **21**, no. 8, 666–77.
- Hodges, A. (1983), *Allan Turing the Enigma*, Simon and Schuster: New York.
- Holliday, M. A. and M. K. Vernon (1987), "Exact performance estimates for multiprocessor memory and bus interference", *IEEE Trans. Comput.*, **C-36**, no. 1, 76–84.
- Hoogendoorn, C. H. (1977), "A general model for memory interference in multiprocessors", *IEEE Trans. Comput.*, **C-26**, no. 10, 998–1005.
- Howe, C. D. and B. Moxon (1987), "How to program parallel processors", *IEEE Spectrum*, **24**, no. 9, 36–41.
- Hughes, J. (1989), "Why functional programming matters", *The Computer Journal*, **32**, no. 2, 98–107.
- Hwang, K. (1979), "Global and modular two's complement cellular array multipliers", *IEEE Trans. Comput.*, **C-28**, no. 4, 300–6.
- Hwang, K. (1985), "Multiprocessor supercomputers for scientific/engineering applications", *IEEE Computer*, **18**, no. 6, 57–73.
- Hwang, K. and F. A. Briggs (1984), *Computer architecture and parallel processing*, McGraw-Hill: New York.
- Hwang, K., J. Ghosh and R. Chowkwanyun (1987), "Computer architectures for artificial intelligence processing", *IEEE Computer*, **20**, no. 1, 19–27.
- Hwang, K., P.-S. Tseng and D. Kim (1989), "An orthogonal multiprocessor for parallel scientific computations", *IEEE Trans. Comput.*, **38**, no. 1, 47–61.
- Inmos Ltd. (1984a), "IMS T424 transputer preliminary data", Bristol, England.
- Inmos Ltd. (1984b), *Occam Programming Manual*, Prentice Hall: Englewood Cliffs, New Jersey.
- Inmos Ltd. (1986), *Transputer Reference Manual*, Bristol, England.
- Inmos Ltd. (1987), *IMS T800 Architecture*, Technical Note 6, Bristol, England.
- Intel Corp. (1979), *The 8086 Family User's Manual*, Santa Clara, California.
- Intel Corp. (1982), *Component Data Catalog*, Santa Clara, California.
- Intel Corp. (1985a), *iAPX 86/88, 186/188 User's Manual Hardware Reference*, Santa Clara, California.
- Intel Corp. (1985b), *Introduction to the 80386*, Santa Clara, California.
- Intel Corp. (1987a), *80286 High Performance Microprocessor with Memory Management and Protection*, Santa Clara, California.
- Intel Corp. (1987b), *82385 High-Performance 32-bit Cache Controller Architectural Overview*, Santa Clara, California.
- Irani, K. B. and I. H. Önyüksel (1984), "A closed-form solution for the performance analysis of multiple-bus multiprocessor systems", *IEEE Trans. Comput.*, **C-33**, no. 11, 1004–12.
- Jordan, H. F. (1985), "HEP architecture programming and performance", in *MIMD Computation: The HEP supercomputer and its applications*, J. S. Kowalik (ed.), MIT Press: Cambridge, Massachusetts, 1–40.
- Jump, J. R. and S. R. Ahuja (1978), "Effective pipelining of digital systems", *IEEE Trans. Comput.*, **C-27**, no. 9, 855–65.
- Karp, A. H. (1987), "Programming for parallelism", *IEEE Computer*, **20**, no. 5, 43–51.
- Karp, R. M. and R. E. Miller (1966), "Properties of a model for parallel computations: Determinacy, termination, queueing", *SIAM Journal of Applied Mathematics*, **14**, no. 6, 1390.
- Katevenis, M. G. H. (1985), *Reduced Instruction Set Computer Architectures for VLSI*, MIT Press: Cambridge, Massachusetts.

362 References and further reading

- Kilburn, T., D. B. G. Edwards, M. J. Lanigan and F. H. Sumner (1962), "One-level storage system", *IRE Trans.*, **EC-11**, 223–35. Reprinted (1982), in *Computer Structures Principles and Examples*, D. P. Siewiorek, C. G. Bell and A. Newell, McGraw-Hill: New York.
- Kogge, P. M. (1981), *The Architecture of Pipelined Computers*, McGraw-Hill: New York.
- Koren, I., B. Mendelson, I. Peled and G. M. Silberman (1988), "A data-driven VLSI array for arbitrary algorithms", *IEEE Computer*, **21**, no. 10, 30–43.
- Kung, H. T. (1982), "Why systolic architectures", *IEEE Computer*, **15**, no. 1, 37–46.
- Kung, S.-Y., K.S. Arun, R. J. Gal-Ezer and D. V. B. Rao (1982), "Wavefront array processor: Language, architecture and applications", *IEEE Trans. Comput.*, **C-31**, no. 11, 1054–65.
- Lampert, L. (1979), "How to make a multiprocessor computer that correctly executes multiprocess programs", *IEEE Trans. Comput.*, **C-28**, no. 9, 690–1.
- Lang, G. R., M. Dharssi, F. M. Longstaff, P. S. Longstaff, P. A. S. Metford and M. T. Rimmer (1988), "An optimum parallel architecture for high-speed real-time digital signal processing", *IEEE Computer*, **21**, no. 2, 47–57.
- Lang, T., M. Valero and I. Alegre (1982), "Bandwidth of crossbar and multiple-bus connections for multiprocessors", *IEEE Trans. Comput.*, **C-31**, no. 12, 1227–34.
- Lang, T., M. Valero and M. A. Fiol (1983), "Reduction of connections for multibus organizations", *IEEE Trans. Comput.*, **C-32**, no. 8, 707–16.
- Lea, R. M. (1986), "VLSI and WSI associative string processors for cost-effective parallel processing", *The Computer Journal*, **29**, no. 6, 486–94.
- Lilja, D. J. (1988), "Reducing the branch penalty in pipelined processors", *IEEE Computer*, **21**, no. 7, 47–55.
- Lipovski, G. J. and M. Malek (1987), *Parallel Computing Theory and Comparisons*, Wiley: New York.
- Liskov, B., M. Herlihy and L. Gilbert (1988), "Limitations of synchronous communication with static process structure in languages for distributed computing", in *Concurrent Programming*, N. Gehani and A. D. McGettrick (eds.), Addison-Wesley: Wokingham, England, 545–64.
- Liu, Y.-C. and C.-J. Jou (1987), "Effective memory bandwidth and processor blocking probability in multiple-bus systems", *IEEE Trans. Comput.*, **C-36**, no. 6, 761–4.
- Mano, N. M. (1982), *Computer System Architecture*, Prentice Hall: Englewood Cliffs, New Jersey.
- Maruyama, K. (1975), "mLRU page replacement algorithm in terms of the reference matrix", *IBM Tech. Disclosure Bulletin*, **17**, no. 10, 3101–3.
- Matsen, F. A. and T. Tajima (eds.) (1986), *Supercomputers, and Scientific Computations*, University of Texas Press: Austin, Texas.
- Mattos, P. (1987a), "Applying the transputer", *Electronics and Power*, June, 397–401.
- Mattos, P. (1987b), "Program design for concurrent systems", Inmos Ltd., Technical Note 5, Bristol, England.
- May, D. (1987), *Occam 2 Language Definition*, Inmos Ltd: Bristol, England.
- May, D. and R. Taylor (1984), "OCCAM: an overview", *Microprocessors Microsyst.*, **8**, no. 2, 73–80.
- Maytal, B., S. Iacobovici, D. B. Alpert, D. Biran, J. Levy and S. Y. Tov (1989), "Design considerations for a general-purpose microprocessor", *IEEE Computer*, **22**, no. 1, 66–76.
- McGraw, J. R. (1982), "The VAL language: description and analysis", *ACM Trans. Program. Lang. Syst.*, **4**, no. 1, 44–82.
- Microsoft Inc. (1987), *Microsoft Macro Assembler 5.1 Microsoft CodeView and Utilities*, Microsoft Press: Redmond, Washington.

- Milutinović, V. M. (1989), *High Level Language Computer Architectures*, Computer Science Press, Freeman and Co: New York.
- Mokhoff, N. (1984), "Parallelism makes strong bid for next generation computers", *Computer Design*, September, 104–31.
- Motorola Inc. (1984), *MC68000 16-32-bit Microprocessor Programmer's Reference Manual*, 4th ed.
- Motorola Inc. (1985a), *MC68452 Advance Information*, Phoenix, Arizona.
- Motorola Inc. (1985b), *MC68000 16-/32-bit Microprocessor*, East Kilbride, Scotland.
- Motorola Inc. (1988a), *MC88100 RISC Microprocessor User's Manual*, Phoenix, Arizona.
- Motorola Inc. (1988b), *MC88200 Cache/Memory Management Unit User's Manual*, Phoenix, Arizona.
- Mudge, T. N., J. P. Hayes, G. D. Buzzard and D. C. Winsor (1984), "Analysis of multiple bus interconnection networks", *Proc. 1984 Int. Conf. on Parallel Processing*, IEEE, 228–32.
- National Semiconductor Inc. (1981), *COP2440/COP2441/COP2442 and COP2340/COP2341/COP2342 Single-Chip Dual CPU Microcontrollers*.
- Nelson, J. C. C. and M. K. Refai (1984), "Design of a hardware arbiter for multi-microprocessor systems", *Microprocessors Microsyst.*, **8**, no. 1, 21–4.
- Ottenstein, K. J. (1985), "A brief survey of implicit parallelism detection", in *MIMD Computation: The HEP supercomputer and its applications*, J. S. Kowalik (ed.), MIT Press: Cambridge, Massachusetts, 93–122.
- Oxley, D. (1981), "Motivation for a combined data flow-control flow processor", 25th Annual Symposium of the Society of Photo-Optical Instrumentation Engineers – Sessions on Real-Time Signal Processing IV, *San Diego, California*, 305–11.
- Padua, D. A., D. J. Kuck and D. H. Lawrie (1980), "High-speed multiprocessors and compilation techniques", *IEEE Trans. Comput.*, **C-29**, no. 9, 763–76.
- Padua, D. A. and M. J. Wolfe (1986), "Advanced compiler optimizations for supercomputers", *Comm. ACM*, **29**, no. 12, 1184–1201.
- Pase, D. M. and A. R. Larrabee (1988), "Intel iPSC Concurrent Computer", in *Programming Parallel Processors*, R. G. Babb II (ed.), Addison-Wesley: Reading, Massachusetts, 105–23.
- Patel, J. H. (1981), "Performance of processor-memory interconnections for multiprocessors", *IEEE Trans. Comput.*, **C-30**, 771–80.
- Patterson, D. A. (1985), "Reduced instruction set computers", *Comm. ACM*, **28**, no. 1, 8–21.
- Patterson, D. A. and J. L. Hennessy (1985), "Response to 'computers, complexity and controversy'", *IEEE Computer*, **18**, no. 11, 142–3.
- Patton, P. C. (1985), "Multiprocessors: architecture and applications", *IEEE Computer*, **18**, no. 5, 29–40.
- Paul, G. and G. S. Almasi (eds.) (1988), *Parallel Systems and Computations*, North-Holland: Amsterdam.
- Pease III, M. C. (1977), "The indirect binary n -cube microprocessor array", *IEEE Trans. Comput.*, **C-26**, no. 5, 458–73.
- Perrott, R. H. and A. Zarea-Aliabadi (1986), "Supercomputer languages", *Computing Surveys*, **18**, no. 1, 5–22.
- Peyton Jones, S. L. (1989), "Parallel implementation of functional programming languages", *The Computer Journal*, **32**, no. 2, 175–86.
- Pfister, G. F. and V. A. Norton (1985), "Hot spot contention and combining in multistage interconnection networks", *IEEE Trans. Comput.*, **C-34**, no. 10, 943–8.

364 References and further reading

- Pohm, A. V. and O. P. Agrawal (1983), *High-speed memory systems*, Reston: Virginia.
- Prieve, B. G. and R. S. Fabry (1976), "VMIN: an optimal variable-space page replacement algorithm", *Comm. ACM*, **19**, no. 5, 295-7.
- Radin, G. (1983), "The 801 minicomputer", *IBM J. Res. Develop.*, **27**, no. 3, 237-46.
- Raghavendra, C. S. and A. Varma (1986), "Fault-tolerant multiprocessors with redundant-path interconnection networks", *IEEE Trans. Comput.*, **C-35**, no. 4, 307-16.
- Ravi, C. V. (1972), "On the bandwidth and interference in interleaved memory systems", *IEEE Trans. Comput.*, **C-21**, no. 4, 899-901.
- Reddi, S. S. and E. A. Feurstel (1976), "A conceptual framework for computer architecture", *Computing Surveys*, **8**, no. 2, 277-300.
- Rettberg, R. and R. Thomas (1986), "Contention is no obstacle to shared-memory multiprocessing", *Comm. ACM*, **29**, no. 12, 1202-12.
- Roscoe, A. W. and C. A. R. Hoare (1986), *The Laws of Occam Programming*, Oxford University Computing Laboratory, Technical Monograph PRG-53.
- Seitz, C. L. (1984), "Concurrent VLSI architectures", *IEEE Trans. Comput.*, **C-33**, no. 12, 1247-65.
- Seitz, C. L. (1985), "The cosmic cube", *Comm. ACM*, **28**, no. 1, 22-33.
- Skillicorn, D. B. (1988), "A taxonomy for computer architectures", *IEEE Computer*, **21**, no. 11, 46-57.
- Smith, A. J. (1982), "Cache memories", *Computing Surveys*, **14**, no. 3, 473-530.
- Smith, A. J. (1985), "Disk cache: miss ratio analysis and design considerations", *ACM Trans. Comput. Systems*, **3**, no. 3, 161-203.
- Smith, A. J. (1987a), "Line (block) size selection in CPU cache memories", *IEEE Trans. Comput.*, **C-36**, no. 9, 1063-75.
- Smith, A. J. (1987b), "Cache memory design: An evolving art", *IEEE Spectrum*, **24**, no. 12, 40-4.
- Snyder, L. (1982), "Introduction to the configurable highly parallel computer", *IEEE Computer*, **15**, no. 1, 47-56.
- Srini, V. P. (1986), "An architectural comparison of dataflow systems", *IEEE Computer*, **19**, no. 3, 68-87.
- Stallings, W. (1987), *Computer Organization and Architecture*, Macmillan: New York.
- Stone, H. S. *et al* (1980), *Introduction to Computer Architecture*, 2nd ed., SRA: Chicago.
- Stone, H. S. (1987), *High Performance Computer Architecture*, Addison-Wesley: Reading, Massachusetts.
- Strecker, W. D. (1978), "VAX-11/780: A virtual address extension to the DEC PDP-11 family", *AFIPS Proc. NCC*, 967-80. Reprinted (1982), in *Computer Structures Principles and Examples*, by D. P. Siewiorek, C. G. Bell and A. Newell, McGraw-Hill: New York.
- Strecker, W. D. (1983), "Transient behavior of cache memories", *ACM Trans. Comput. Systems*, **1**, no. 4, 281-93.
- Tabak, D. (1987), *Reduced Instruction Set Computer RISC Architecture*, Research Studies Press Ltd.: Letchworth, England.
- Tanenbaum, A. S. (1984), *Structured Computer Organization* (2nd ed.), Prentice Hall: Englewood Cliffs, New Jersey.
- Tanenbaum, A. S. (1990), *Structured Computer Organization* (3rd ed.), Prentice Hall: Englewood Cliffs, New Jersey.
- Terrano, A. E., S. M. Dunn and J. E. Peters (1989), "Using an architectural knowledge base to generate code for parallel computers", *Comm. ACM*, **32**, no. 9, 1065-72.
- Texas Instruments (1984), *The TTL Data Book for Design Engineers*, Dallas, Texas.

- Thiebaut, D. and H. S. Stone (1987), "Footprints in the cache", *ACM Trans. Comput. Systems*, **5**, no. 4, 305–29.
- Thornton, J. E. (1970), *Design of a Computer: The Control Data 6600*, Scott, Foresman and Company: Glenview, Illinois.
- Tomasulo, R. M. (1967), "An efficient algorithm for exploiting multiple arithmetic units", *IBM Journal*, **11**, 25–33. Reprinted (1982), in *Computer Structures Principles and Examples*, by D. P. Siewiorek, C. G. Bell and A. Newell, McGraw-Hill: New York.
- Treleaven, P. C., D. R. Brownbridge and R. P. Hopkins (1982), "Data driven and demand driven computer architectures", *Computing Surveys*, **14**, no. 1, 93–143.
- Tucker, S. G. (1986), "The IBM 3090 System: an overview", *IBM Systems Journal*, **25**, no. 1, 4–18.
- Vegdahl, S. (1984), "A survey of proposed architectures for the execution of functional languages", *IEEE Trans. Comput.*, **C-33**, no. 12, 1050–71.
- Wadge, W. W. and E. A. Ashcroft (1985), *Lucid, the Dataflow Programming Language*, Academic Press: London.
- Watson, I. and J. R. Gurd (1979), "A prototype data flow computer with token labelling", *AFIPS National Computer Conference*, June 4–7, New York, 623–8.
- Weiss, S. and J. E. Smith (1984), "Instruction issue logic in pipelined supercomputers", *IEEE Trans. Comput.*, **C-33**, no. 11, 1013–22.
- Wilkes, M. V. (1951), "The best way to design an automatic calculating machine", *Rept. Manchester University Computer Inaugural Conf.*, 16–18. Reprinted (1976), in *Computer Design Development: Principal Papers*, E. E. Schwartzlander (ed.) Hayden: Rochelle Park, New Jersey, 266–70.
- Wilkinson, B. (1987), *Digital System Design*, Prentice Hall: London.
- Wilkinson, B. (1989), "Simulation of rhombic cross-bar switch networks for multiprocessors", *Proc. 20th Annual Pittsburgh Conf. on Modeling and Simulation*, 1213–18.
- Wilkinson, B. and H. Abachi (1983), "Cross-bar switch multiprocessor system", *Microprocessors Microsyst.*, **7**, no. 2, 75–9.
- Wittie, L. D. (1981), "Communication structures for large networks of microcomputers", *IEEE Trans. Comput.*, **C-30**, no. 4, 264–73.
- Wong, F. S. and M. R. Ito (1984), "Design and evaluation of the event-driven computer", *Proc. IEE*, **131**, no. 6, 209–22.
- Wulf, W. A. and S. P. Harbison (1978), "Reflections in a pool of processors: an experience report on C.mmp/Hydra", *Carnegie-Mellon University Dept. of Computer Science Report CMU-CS-78-103*.
- Yalamanchili, S. and J. K. Aggarwal (1985), "Reconfiguration strategies for parallel architectures", *IEEE Computer*, **18**, no. 12, 44–61.
- Yang, Q., L. N. Bhuyan and B.-C. Liu (1989), "Analysis and comparison of cache coherence protocols for a packet-switched multiprocessor", *IEEE Trans. Comput.*, **38**, no. 8, 1143–53.
- Yen, D. W. L., J. H. Patel and E. S. Davidson (1982), "Memory interference in synchronous multiprocessor systems", *IEEE Trans. Comput.*, **C-31**, no. 11, 1116–21.
- Yew, P.-C., N.-F. Tzeng and D. H. Lawrie (1987), "Distributing hot-spot addressing in large-scale multiprocessors", *IEEE Trans. Comput.*, **C-36**, no. 4, 388–95.
- Zilog Inc. (1984), *Z80000 CPU Preliminary Technical Manual*, Cambell, California.
- Zilog Inc. (1985), *Z80000 CPU Preliminary Product Specification*, Cambell, California.
- Zilog Inc. (1986), *Z280 MPU Microprocessor Unit, Advance Information*, Cambell, California.

Index

- Absolute addressing, 7
- Accessed bit, 42, 58
- Access time:
 - average, 65–8, 93–4
 - cache, 67, 77, 78, 80, 93–4
 - memory, 50, 64
 - single bus system, 240–1
- Accumulator, 6
- Acknowledge signal, 216
- Active low signal, 215
- Ada, 305
- Adder:
 - carry-look-ahead, 124
 - carry-save, 126–7
 - full, 124
 - ripple, 124
- Addition:
 - fixed point, 124–7
 - floating point, 129–30
- Address, 4:
 - lines, 14–15
 - local, 243
 - logical, 52
 - physical, 52
 - real, 27
 - virtual, 27
- Addressing:
 - absolute, 7
 - immediate, 7
 - modes, 6
 - register direct, 7
 - register indirect, 7
 - RISC, 153
- Address translation:
 - cache, 36–7
 - paging, 32–41
- Adjusted request rate, 238
- Aging registers, 82–3
- ALGOL-68, 198
- ALICE computer system, 354
- Alternation process, occam, 319–20
- ALU (arithmetic and logic unit), 8, 15
- Am2901, 145
- Amdahl 470, 38
- Amdahl's law, 190
- Applied parallelism, 192
- Arbitration:
 - bus *see* Bus arbitration
 - cross-bar switch network, 253–4
 - multiple bus network, 250, 260
- Arithmetic pipeline, 123–30
- Array computer, 173, 175–82
 - bit-organized, 180–1
 - word-organized, 175–80
- Array multiplier, 127
- Associative cache, 71–4
- Associative mapping, 33–4
- Associative memory, 33
- Associativity, 35, 73
- Asynchronous message passing routines, 304
- Asynchronous pipeline, 103–5
- AT&T WE3210, 6
- Atlas computer, 28
- Auxiliary memory, 25, *see also* Secondary memory
- Average distance, in networks, 287
- Average latency, 133
- Babbage, 3
- Backing store, 5
- Bandwidth:
 - analysis assumptions, 256–7
 - cross-bar switch, 257–60
 - multiple bus, 260–61
 - overlapping connectivity cross-bar switch, 277–9
 - single bus, 237–8
 - static networks, 288–9
- Base, floating point number, 14, 128
- Baseline network, 266
- Benchmark, 146–7
- Benes network, 265

- Bernstein's conditions, 199
- Best fit replacement algorithm, 55
- Biased exponent, 129
- Bit encoded instruction, 148
- Bit selection, 74
- BLITZEN computer, 181–2
- Block, cache, 69
- Block organization, direct mapped cache, 69–70
- Blocking network, 262
- Blocking, message passing routine, 304
- Branch bypassing, 113
- Branch folding, 114
- Branch history table, 114–15
- Branch instruction:
 - effect on pipelines, 111
 - frequency, 112
 - RISC, 153, *see also* Delayed branch instructions
 - usage, 112–23, 147
- Broadcast, hypercube, 307–8
- Broadcast instruction, 179
- Broadcast writes, 97
- BSP (Burroughs' Scientific Processor), 179
- Burroughs computers:
 - B5000, 52
 - D-825, 254
- Bus, 14–15
 - asynchronous, 216
 - clear signal, 231
 - contention, 214
 - grant signal, 215
 - local, 241–3
 - master, 214
 - request signal, 215
 - requests, multiple, 216–18
 - synchronous, 216
 - system, 241–3
 - watcher, 96
- Bus arbitration:
 - centralized, 217
 - decentralized, 217, 224, 230–31
 - equations, 219, 232–3
- Busy waiting, with locks, 204
- Butterfly multiprocessor, 193
- Byte, 15

- Cache (memory), 19, 64–99
 - access time, 65, 67–8
 - address translation, 36–7
 - associativity, 73
 - block, direct mapped, 69–71
 - coherence, 95
 - controller, 79
 - data, 76
 - DEC computers:
 - PDP-11-60, 70
 - VAX-11/780, 79
 - VAX-8800, 70
 - descriptor registers, 58
 - direct mapping, 68–71
 - disk, 94–5
 - fetch policy, 75–6
 - fully associative, 71–2
 - IBM computers:
 - 360/85, 75
 - 370/158, 70
 - 3033, 79, 97
 - instruction, 76
 - Intel 80386, 79, 97
 - miss ratio, 87–8
 - multiprocessor, 95–9
 - National Semiconductor NS32532, 76
 - performance, 86–90
 - sector organization, 74–5
 - set-associative, 73–4
 - Zilog Z-280, Z80000, 72
- CAM, (content addressable memory), 33, 71, 353
- Carry-in, 124
- Carry-look-ahead adder, 124
- Carry-out, 124
- Carry-save adder, 126–7
- Control Data computer:
 - CDC 6600, 120
- Cedar computer system, 350–1
- Cell-based network, 263
- Chaining, pipeline, 139–40
- Changed bit, 42
- Channel dependency graph, 301
- Checkerboard effect, in segmentation, 55
- Chordal ring, 283
- CISC (complex instruction set computer), 18, 144–6
- CLIP computer, 181
- Clock period, 11
- Clock replacement algorithm, 45
- Clos network, 263–5
- C.mmp, 254–5
- CMOS, 10
- COBEGIN-COEND construct, 197
- Code segment register, 57
- Code, superfluous, 39
- COEND construct, 197
- Cold start, in caches, 86
- Collision vector, 132
- Collision, in a pipeline, 132
- Coloring, in dataflow, 337
- Combining circuits, hardware, 273–4
- Common bus request signal, 235
- Compiler:
 - optimizing for RISCs, 150, 155
 - parallelizing, 202
- Completely connected network, 282
- Computer:
 - control-flow, 329
 - dataflow, 330
 - neural, 330
 - parallel, 171
 - pattern driven, 330
 - program, 6

- Computer (*continued*)
 - stored program, 3
 - von Neumann, 3
- Concurrency, 171
- Concurrentizing, 202
- Condition code register, 9
- Conditional critical section, 210
- Conditional process, occam, 321–2
- Conditional synchronization, 210
- Context switch, 86
- Conti, C. J., 65
- Control lines, 14–15
- Control memory, 9
 - alterable, 9
- Control unit, 4, 8–9, 178
- Control-flow computer, 329
- Conway, 22
- Coprocessor, 243–7
 - arithmetic, 243–6
 - input/output, 247
- Cosmic cube, 308–9
- Cost:
 - memory system, 50
 - pipeline, 105
 - multiprocessor, 172
 - trade-offs, 24
- Counters, to implement replacement algorithms, 46, 48, 82
- CP/M, 13
- CPU (central processing unit), 4, 54
- Cray computers, 11, 66, 105
 - Cray 1, 11, 138
 - Cray 2, 11, 138, 172
 - Cray 3, 11
 - Cray X-MP, 11, 138
 - Cray Y-MP, 138
- CREATE construct, 196–7
- Critical section, 203
 - conditional, 210
- Cross-bar switch:
 - multiprocessor, 184–5, 252–6
 - network, 184–8, 252–6, 263
 - overlapping, 276–9
- CSP (programming language), 305
- Cube connected cycles network, 286
- Cube network, 286
- Cyber 205, 138
- Cycle time:
 - memory, 64
 - processor, 11, 153
- Daisy chain, rotating, 231
- Daisy chained signals, 227–9
- DAP computer, 181
- Data bus, 163
- Data cache, 76
- Data dependence graph, 330
- Data dependencies, 117–21
- Data driven computer, 330
- Data lines, 14–15
- Data segment register, 57
- Dataflow, 329–55
 - architectures, 336, 339–40, 342–4, 350–3
 - branch node, 331–2
 - coarse grain, 349
 - computational model, 330–4
 - computer, 330
 - dynamic, 334, 337–42
 - error handling, 349
 - fine grain, 349
 - graph, 330–3, 343
 - instruction cell, 336
 - languages, 344–9
 - loops, 332–4
 - macro-, 349–50
 - merge node, 331–2
 - node firing rules, 330, 334, 337, 350
 - packet, 334, 340
 - routing network, 336–7
 - static, 334–7
 - streams, 347
 - systems, 329–54
 - VLSI, 342–4
- Davidson's pipeline control algorithm, 133
- Deadlock, 206, 209
- Deadly embrace, 209
- DEC (digital equipment corporation) computers:
 - PDP 8E, 14
 - PDP 11, 14, 255–6
 - PDP 11/60, 70
 - PDP 11/70, 40
 - PDP 11/780, 40
- Decode history table, 116
- Delay, adding in a pipeline, 137–8
- Delayed branch instruction, 116–17, 154
- Delayed branch with execute, 154
- Delayed memory load instruction, 154
- Delta network, 269
 - with extra stage, 270–1
- Demand driven computer, 330
- Demand fetch, 76
- Demand paging, 43
- Denning, P. J., 42, 43
- Dennis, J. B., 334, 347
- Descriptor cache registers, 58
- Destination tag algorithm, 266
- Dhrystone 146–7
- Dijkstra, E. W., 207
- Direct mapped cache, 68–71
 - advantages, 70
- Direct mapping, 32–33, 68–71
- Directory method, for cache coherence, 97–8
- Dirty bit, 42
- Disk caches, 94–5
- Disk controller, microprocessor, 14
- Disk memory, 14, 18, 25, 30
 - access time, 67–8
- Displacement, in segmentation, 51

- DLT (directory look-aside buffer) *see* TLB
- DMA (direct memory access), 26, 41, 54, 92, 96, 235, 243, 311, 314
- DOPAR-PAREND constructs, 197
- Dual port memory, 137
- Dynamic binding, 57
- Dynamic code copying, dataflow, 337
- Dynamic dataflow, 334, 337–42
- Dynamic interconnection networks, 262
- Dynamic pipeline, 124
- Dynamic process structure, 304–5
- Dynamic token tagging, dataflow, 337

- E*-cube routing algorithm, 301
- ECL, 10, 153, 172, 256
- EDVAC, 6
- Efficiency, 106, 188
- Emulation, 9
- Ethernet, 310
- Execute cycle, 7–8, 16
- Exhaustive networks, 282
- Exponent, 14, 128
 - biased, 129
- External fragmentation, 55
- Extra segment register, 57
- Extra stage networks, 270

- Failure rate, 210 *see also* Reliability
- Fault tolerance, 172–73
- Fault tolerant systems, 187
- Feedback pipeline, 127
- Fetch-and-add operation, 274–5
- Fetch cycle, 7–8, 16
- Fetch/execute overlap, 16, 140
- First fit replacement algorithm, 55
- First-in first-out replacement algorithm, 44, 82
- First-in not-used first-out replacement algorithm, 45
- Fixed partition replacement algorithm, 42, 81
- Fixed-point numbers, 124
- Fixed priority:
 - access time, 241
 - boolean equations, 219, 231–4
 - parallel arbiter, 219–20
 - serial arbiter, 227–9
- Flit, 309
- Floating point numbers, 13–14, 127–30
- Floating point unit, 163
- Flynn's classifications, 173–4
- Footprint, 90
- Forbidden latency, 133
- FORK-JOIN construct, 195–6
- Forwarding, 121
- Freedom from side effects, 345
- Fully associative cache, 71–2
- Fully associative mapping, cache, 71–2
- Functional languages, 345
- Functional units, multiple, 139

- GaAs 149
- Generalized shuffle network, 270
- GF-11, 180
- Global algorithm, 42
- Grant acknowledge signal, 216
- Granularity, process, 297
- Greedy strategy, in a pipeline, 133
- GRIP computer system, 354

- Handshaking, 104, 215
- Harvard architecture, 5, 153, 160
- Hashing, page, 38
- Hexagonal array, 284
- High order interleaving, 20
- Hit ratio, 50, 67–8 *see also* miss ratio
- Hot spots, 273
- Hypercube:
 - iPSC, 308
 - network, 286, 299–300
 - routing, 300–1
 - spanning bus, 287
- Hypertree, 286

- IBM computers:
 - 360 146–7
 - 360/85, 65, 75, 122
 - 360/91, 122, 353
 - 360/195, 122
 - 370/158, 70
 - 370/168, 85, 205
 - 801, 153–6, 165
 - 3033, 38, 79, 81, 97
 - 3080, 11, 12
 - 3090, 12
 - 3880, 95
 - MVS operating system, 92
- IEEE 796 bus, 235
- Illiac IV, 177–9, 283
- Immediate addressing, 7
- Index (cache), 68
- Indirect addressing, multilevel, 145
- Indirect binary *n*-cube network, 269
- Indivisible instruction, 204
- Initial collision vector, 132
- Initiation, in a pipeline, 132
- Inmos processors:
 - T414, 312
 - T800, 312
 - T212, 312, 322
- Input/output coprocessor, 247
- Input/output interface, microprocessor, 14
- Instruction:
 - bit-encoded, 148
 - broadcast, 179
 - BSET (test a bit and set), 205
 - buffer, 113–14
 - bus, 163
 - cache, 76
 - CAS (compare and swap), 205

370 Index

Instruction (*continued*)

- encoding, 6–7, 146
- execution time, 65, 77
- fetch/execute overlap, 16–17, 107–11
- four-address, 6
- indivisible, 204
- lock prefix, 204–5
- one-address, 6
- one-and-a-half-address, 6
- pointer, 5
- register, 8
- set, 5
- TAS (test and set), 205
- three-address, 6, 151, 160
- transputer format, 165
- two-address, 6
- useage, 147
- zero-address, 6

Intel:

- 432, 148
- 4004, 13
- 8080, 13
- 8086, 13, 57, 117, 145, 148, 204–5, 244, 308
- 8087, 244–5, 308
- 80186, 13
- 80286, 13, 57–9, 113, 309–10
- 80287, 244, 309
- 80386, 13, 41, 60, 79, 97, 113, 146, 310
- 80486, 13
- 82385, 79
- 82385, 97
- iPSC, 308, 309–11

Interconnection graph, 301

Interleaved memory, 20, 65–6, 109, 259

Internal forwarding, 122, 152–3, 154, 159

Internal fragmentation, 39

Inverse translation buffer, 92

IP-1, 254

I-structure memory, in dataflow, 340

JOIN constructs, 195–6

Jump instruction *see* Branch instruction

Katevenis, M. G. H., 149, 156

Kilburn, T., 27–30, 33

Koren, I., 342–4

Lampport's conditions, 206

Lang, T, 251, 259

Last-in first-out replacement algorithm, 45

Latency, 133

Left-to-right routing, 301

Line (cache), 69

Linear addressing, 41, 60

Linear array, 282–3

Linear pipeline, 130

Linear segmentation, 41

Local algorithm, 42

Local bus, 241–3

Locality of effect, 345

Lock, 203–7

Logical address, 52

Low order interleaving, 20

LRU (least recently used):

- replacement algorithm, 45–7, 82–5
- implementation, 46–7, 82–5
- priority, 226

LSI, 12

Machine instructions, 5

Macrodataflow, 349–50

- architectures, 350–3
- firing rules, 350

Main memory, 4

MAL (minimum average latency), 133

Mantissa, 14, 128

MASM 146–7

Master-slave operation, cross-bar switch system, 253

Matrix, storage of elements, 179

Maximum rate pipeline, 105

Memory:

- cache *see* cache memory
- contention, cross-bar switch network, 257–9
- hierarchy, 18
- interleaving *see* Interleaved memory
- management, 25–63
- Atlas, 28, 33
 - 8086 family, 57–60
 - MC88100, 37, 39
 - VAX-11/780, 37, 38, 40
 - VAX 8600, 37
- multiport, 184–5
- pollution, 88
- protection, 27, 54
- two-port, 276

Mesh network, 178, 283, 287

Message-passing:

- communication paths, 298–301
- message format, 300
- multicomputers, 296
- multiprocessor systems, 24, 295–325
- programming, 301–8
- routines, 304

MFLOPS, 11

Microcode, 9, 149

Microinstruction, 9

Microprocessor, 12–15

- architecture, 14–15
- 32-bit, 13, 15

Microprogram, 9

MIMD (multiple instruction stream-multiple data stream) computer, 173–4, 182–7

MIN (minimum replacement algorithm), 49

MIN (multistage interconnection network) *see* Multistage network

Minsky's conjecture, 192

MIPS (millions of instructions per second), 11

- MIPS processor, 156
- MISD (multiple instruction stream-single data stream) computer, 173–4
- Miss ratio 67, 87–90
- MIT, 26, 339, 347
- MMU (memory management unit), 54
- Modified bit, 42, 98
- Modified state diagram, pipeline, 134
- Monitor, 210
- MOPS (millions of operations per second), 11
- Motorola:
 - MC6800, 13, 214
 - MC68000, 13, 110, 145, 148, 205, 222–3, 229–30
 - MC68020, 13, 146, 205, 245–6
 - MC68030, 13, 41
 - MC68452, 222–3
 - MC68881, 246
 - MC88100, 120, 160–4
 - MC88200, 39, 163
- MP programming language, 305
- MPP (massively parallel processor), 181–2
- MS-DOS, 26, 146–47
- MSI, 12, 153
- MSISD (multiple single instruction stream-single data stream) computer, 173–4
- Multibus I, 235
- Multics, 57
- Multifunction pipelines *see* Pipeline, Multifunction
- Multiple bus:
 - multiprocessor, 183–4, 250–2
 - network, 183–4, 250–2
 - overlapping connectivity, 279–81
 - partial, 251–2
- Multiple prefetching, instruction, 113
- Multiprocessor systems, 22–4, 171–87
 - classifications, 173–5
 - efficiency, 106
 - message-passing, 24, 185–6
 - programming, 193–203
 - shared memory, 23
 - speed-up factor, 188–93
- Multiprogramming, 18, 47
- Multistage (interconnection) network, 184–5, 262
 - bandwidth, 270
 - Baseline, 266
 - Benes, 265
 - Clos, 263
 - Delta, 269
 - fault tolerant, 270
 - generalized shuffle, 270
 - hot spots, 273–5
 - indirect binary n -cube, 269
 - Omega, 268
 - recirculating, 267
 - shuffle, 266
- Multistreaming, 122–3
- Mutual exclusion, 203
- MVS operating system, 92
- National Semiconductor NS32532, 76
- Natural parallelism, 192
- Near(est) neighbor network, 178, 283
- Network:
 - average distance, 287
 - cross-bar switch, 184–5, 252–6, 263, 276–9
 - latency, 300
 - blocking, 262
 - chordal ring, 283
 - completely connected, 282
 - cube, 28
 - cube connected cycles, 286
 - exhaustive, 282
 - hexagonal array, 284
 - hypercube, 286, 299–300
 - spanning bus, 287
 - hypertree, 286
 - linear array, 282–3
 - mesh, 283
 - multiple bus, 183–4, 250–2, 279–81
 - near(est) neighbor, 283
 - non-blocking, 262
 - number of links, 282, 287–8
 - overlapping connectivity mesh, 287
 - shuffle exchange, 267
 - single bus, 183–4, 213–14
 - single stage, 262–3
 - star network, 284
 - static, 282–9
 - systolic array, 284
 - three-cube, 286
 - tree, 284–5
 - binary, 284–5
 - m -ary, 285
- Neural computer, 330
- Nibble, 13
- Non-blocking message passing routine, 304
- Non-blocking network, 262
- Non-cacheable items, 76
- Non-usage-based algorithms, 41
- No-op, 117
- Normal mode, 54
- Normalized number, 129
- Nova computer, 145
- Nullification method, for branch instructions 117
- Occam, 314–25
 - allocation of transputers/processes, 325
 - process:
 - alternation, 319–20
 - conditional, 321–2
 - parallel, 317–19
 - primitive, 316
 - repetitive, 320–1
 - replicator, 323–4
- Occam's razor, 311
- Offset, in segmentation, 51

372 Index

- Omega network, 268
- Omnibus, 14
- One level store, 27, 33
- Operating system, 12, 26, 30, 37, 54, 59,
 - cross-bar switch, 253-4
 - message-passing, 310
- Optimal replacement algorithm, 49
- Orthogonal multiprocessor, 256
- Overlap, 102
- Overlapping connectivity networks, 275-81
 - cross-bar switch, 276-9
 - mesh network, 287
 - multiple bus, 279-81
- Overlaying, 26
- P** (operation), 207
- Page fault frequency algorithm, 49
- Page fault, 34, 41
- Page mapping, multilevel, 39-41
- Page replacement algorithm, 30
- Page size, 38-9
- Page table, 28
- Paged segmentation, 55-7
 - address translation, 56
- Paging, 27-41
- Parallel adder, 124-5
- Parallel arbiter, 220-5, 232-4
- Parallel computer, 171
- Parallel constructs:
 - ALGOL-68, 198
 - block structured languages, 197-9
 - COBEGIN-COEND, 197
 - FORK-JOIN, 195-6
 - FORTRAN-like languages, 195-7
 - occam, 317-19
 - PARBEGIN-PAREND, 197-9
- Parallel priority arbitration schemes, 216, 218-27
- Parallel process, occam, 317-19
- Parallel programming, 171
- Parallelism:
 - explicit, 194-9
 - implicit, 194, 199-202
 - in loops, 201-2
- PARBEGIN-PAREND constructs, 197-9
- PAREND construct, 197-9
- Partitions, fixed, 42
- Partitions, variable, 42
- Pattern driven computer, 330
- PDP computers *see* DEC computers
- Pease, M. C., 266
- Perfect shuffle, 266
- Performance/cost:
 - general, 24
 - memory management, 49-51
 - pipelines, 105-7
 - trade-offs, 24
- Physical address, 52
- Pipeline, 17
 - data dependencies, 117-21
 - effects of branch instructions, 111-13
 - efficiency, 106
 - feedback, 127
 - five-stage, 110, 111, 141, 163
 - four-stage, 151-2, 154
 - hazard detection, 119-20
 - information transfer between stages, 103-5
 - linear, 130
 - multifunction, 123-4, 131
 - scheduling, 133-8
 - six-stage, 163
 - speed-up factor, 105-6
 - stage, 105
 - three-stage, 17, 108, 159, 163
 - two-stage, 16, 107, 116, 151-2 *see also* fetch/execute overlap)
- Pipelining, 102
- PL.8, 154
- PL/1, 154
- Placement algorithm, 55
- Polling schemes, for bus arbitration, 235-7
- Posted write-through, 79
- Prefetch, in cache 76
- Prefetching on a miss, 76
- Primary memory *see* main memory
- Primitive process, occam, 316
- Principle of locality, 31
- Principle of optimality, 49
- Priority:
 - acceptance dependent, 225
 - dynamic, 216, 225-7
 - equal, 225-6
 - first-come first-served, 227
 - fixed, 216, 219
 - fixed time slice, 227
 - LRU, 226
 - processor, 216
 - random, 225
 - rotating, 225-7
- Private bit, 98
- Probability of acceptance, request, 238-9
- Processes, 188
 - concurrent, 193-4
 - equal duration, 188-9
 - granularity, 296
 - message-passing, 296, 311-12
 - occam, 317-22
 - optimum division, 191-2
 - parallel with serial section, 189-91
 - reactive, 297
 - sequential, 317-19
- Process structure:
 - dynamic, 304-5
 - static, 304-5
- Program:
 - computer, 6
 - counter, 5
 - page references, 31
- Programmer, 6

- Protected virtual address mode (80286), 57
- Queue:
 - dataflow, 337, 339–41
 - in page replacement algorithms, 44, 47
 - Intel 80x86 instruction, 113, 245
 - monitor, 210
- Queueing algorithm, 227
- Radin, G., 149
- RAM (random access memory), 5
- Random number generator, for simulation, 259
- Random replacement algorithm, 43, 82
- Reactive processes, 297
- Read-after-read hazard, 118–19
- Read-after-write hazard, 118–19
- Read-through, 68
- Real address, 27
 - number of bits, 30
- Real-time clock, in transputer, 322
- Rearrangeable networks, 262
- Recirculating network, 267
- Reduced state diagram, 134
- Redundancy, system, 187
- Reference matrix, 83–5
- Referential transparency, 346
- Register:
 - direct addressing, 7
 - file, three-port, 153
 - indirect addressing, 7
 - lifetime, 155–56
 - stack, 83
 - window, 156–60
 - pointer, 156
- Relative addressing, 7
- Reliability, 187
- Relocation, program, 30
- Rendezvous, 304, 316
- Repetitive process, occam, 320–1
- Replacement algorithm:
 - best fit, 55
 - cache, 81–5
 - clock, 45
 - first fit, 55
 - first-in first-out, 44, 82
 - fixed, 42, 81
 - fixed partition, 42
 - global, 42
 - last-in first-out, 45
 - local, 42
 - LRU, 45–7, 82–5
 - MIN, 49
 - non-useage based, 41, 81
 - optimal, 49
 - page fault frequency, 49
 - paging, 41–51
 - random, 43, 82
 - segmentation, 55
 - stack, 49
 - useage based, 41, 81
 - variable partition, 42
 - variable, 42, 81
 - VMIN, 49
 - working set, 47–9
 - worst fit, 55
- Replicated units, 20, 105–6
- Replicators, occam, 323–4
- Reservation table, 130–1
- Return address, procedure, 5, 145
- Reverse translation buffer, 92
- RISC (reduced instruction set computer), 18, 117, 120, 122, 144, 148–67
- RISC I/II, 156–60
- Rotating priority, 225–7
- Routing:
 - hypercube, 300–1
 - store-and-forward, 309
 - wormhole, 309
- S1 computer, 256
- Scoreboard bits/technique, 120, 152
- Secondary memory, 5, 64
- Sector mapping, cache, 74–5
- Segment:
 - length, 53
 - number, 51
 - protected, 54
 - selector, 58
- Segmentation, 51–5
 - Intel:
 - 8086, 57
 - 80286, 57–9
 - with paging, 55–60
 - replacement algorithms, 55
 - symbolic, 56
- Segmented name space, 56
- Seitz, C. L., 297, 308
- Selective fetch, in cache, 76
- Self-routing network, 266
- Semantic gap, 144
- Semaphores, 207–10
 - binary, 207
 - counting, 208
- Sequential locality, 31
- Sequential process. occam, 317–19
- Serial arbiter, 227–31
- Serial priority arbitration schemes, 216, 227–34
 - see also* Daisy chain
- Set associative cache, 73–74
- Set size:
 - set-associative cache, 73
 - set associative page mapping, 35
- Shared caches, 95–96
- Shared memory multiprocessors, 22, 182–5
 - disadvantages, 295–6
- Shuffle exchange network, 267
- SIMD (single instruction stream-multiple data stream) computer, 173–4

374 Index

- Simple cycle, pipeline, 134
- Simulation:
 - cache, 86
 - cross-bar switch network, 259
 - multiple bus network, 261
 - overlapping connectivity networks, 278–9
 - random number generator, 259
 - trace-driven, 88
- Single assignment languages, 346
- Single bus network, 183–4, 213–14
- Single stage network, 262–3
- SISAL programming language, 347–9
- SISD (single instruction stream-single data stream) computer, 173–4
- Sloop bus, 96
 - controller, 97
- Smith, A. J., 87–9, 95, 97–8
- Software combining trees, 273–4
- Space-shared, 309
- Space-time product, 50
- Spanning bus hypercube network, 287
- Spatial locality, 31
- Speed-up factor:
 - estimates, multiprocessor, 192–3
 - multiprocessor, 188–92
 - pipeline, 105–6
 - single bus system, 239–40
- Spin locks, 204
- SSI, 12, 153
- Stack, 5, 6, 31, 53
 - algorithm, 49
 - pointer, 5
 - segment register, 57
- Staging latch, in pipelines, 104–5
- Star network, 284
- State diagram, for a pipeline, 133
- Static binding, 57
- Static dataflow, 334–7
- Static interconnection networks, 282–89
- Static process structure, 304–5
- Status vector, 133
- Stone, H. S., 49, 87, 89, 90
- Store-and-forward routing, 309
- Stored program computer, 3
- Streams, dataflow, 347
- Supercomputer, 11, 138
- Symbolic segmentation, 56
- Synchronous message passing routines, 304
- Synchronous pipeline, 103–5, 131
- Synonym, 92
- System bus, 183–4, 241–3
- System mode, 54
- Systolic array, 284
- TAS instruction, 205
- Temporal locality, 31
- Thrashing, 42
- Three-cube network, 286
- Three-level overlap, 108–9
- Time-shared bus, 213–15 *see also* Single bus network
- TLB (translation look-aside buffer), 36–8, 41, 91–4
- Token tagging, in dataflow, 337–9
- Token, in dataflow, 331
- Tomasulo, R. M., 122
- Trace-driven simulation, 86
- Transient behavior, in caches, 89
- Translation buffer *see* TLB
- Transputer, 165–66, 311–14
 - real-time clock, 322
- Tree network:
 - binary, 284–5
 - hyper, 286
 - m-ary, 285
 - saturation, 273
- TTL, 10
- Unibus, 14
- UNIX, 94, 195, 302
- Unused bit, 42
- Usage-based algorithm, 41, 81
- Use bit, 42
 - scanning technique, 42, 47, 48
- V (operation), 207
- VAL, programming language, 347, 349
- Valid bit:
 - cache, 72
 - pipeline, 120
- Variable partition:
 - paging, 42
 - replacement algorithm, 42, 81
- VAX-11/780, 38, 40, 43, 79, 145–6, 148
- VAX-8800, 70
- Vector computer, 123, 138–40, 173
- Vector instruction, 175–7
- Vector processor, 123, 138
- Virtual address, 27
 - number of bits, 30
- Virtual memory, 19, 27
 - with cache memory, 90–2
- VLSI, 12, 127, 149, 156, 165, 172, 185, 296, 311, 342
- VMIN replacement algorithm, 49
- Von Neumann computer, 3, 173, 329
- Voter, 187
- Warm start, in cache, 86
- Whetstone, 147
- Wide data bus, 21
- Wide word length memory, 20, 66
- Working set, 36, 47

- Working set (*continued*)
 - replacement algorithm, 47–9
- Workspace pointer, 166
- Wormhole routing, 309
- Worst fit replacement algorithm, 55
- Write-after-read hazard, 118–19
- Write-after-write hazard, 118–19
- Write back, 80
- Write once, 97
- Write-through, 68, 77–9
- Written bit *see* Modified bit

- X-network, 181–2

- X-MP *see* Cray computers
- Y-MP *see* Cray computers

- Zero address instruction format, 6
- Zilog:
 - Z-80, 13, 229
 - Z-280, 72, 243
 - Z8000, 230
 - Z80000, 72
- Zuse, 3