Computer Architecture

Design and performance

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

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

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

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

Contents

PART

Computer design techniques

CHAPTER

1 Computer systems

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

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

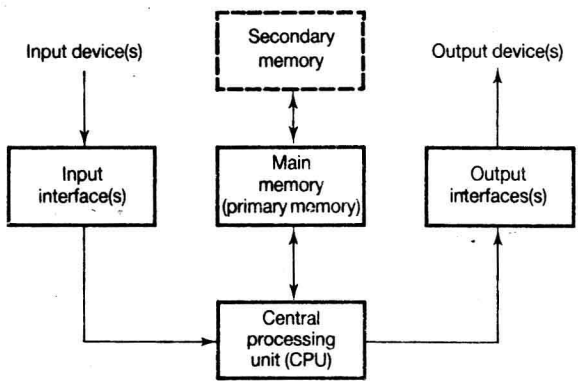

Figure 1.1 Stored program digital computer

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 truely 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.

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: