OXFORD APPLIED MATHEMATICS AND COMPUTING SCIENCE SERIES

The Architecture of Digital Computers

R.G. GARSIDE

R. G. GARSIDE
University of Lancaster

The architecture of digital computers

CLARENDON PRESS · OXFORD
1980

Oxford University Press, Walton Street, Oxford OX2 6DP

OXFORD LONDON GLASGOW
NEW YORK TORONTO MELBOURNE WELLINGTON
KUALA LUMPUR SINGAPORE JAKARTA HONG KONG TOKYO
DELHI BOMBAY CALCUTTA MADRAS KARACHI
NAIROBI DAR ES SALAAM CAPE TOWN

Published in the United States by Oxford University Press, New York

British Library Cataloguing in Publication Data

Garside, R G
The architecture of digital computers.-(Oxford applied mathematics and computing science series).
1. Electronic digital computers — Design and construction
I. Title II. Series
621.3819′58′2 TK7888.3 79-41148

ISBN 0-19-859627-8
ISBN 0-19-859638-3 Pbk

Printed and bound in Great Britain by Morrison & Gibb Ltd, Edinburgh.

Preface

When I learnt my *second* machine-level language, I became fascinated by the possible variations in computers at this level. My fascination resulted in this book, which is designed to give a comprehensive survey of computers from 1946 to the present, and which should be of interest to anyone involved in computing. In particular it should appeal to those who are interested in the differences between computers at an architectural level, for example programmers who would like to know more than is revealed in their assembler language manuals. It is suitable as a textbook for courses on computer architecture for second- and third-year undergraduate students, who have done some programming at the machine level.

The book describes the different features seen by a machine-level programmer, after all the supportive software is stripped away. It does not attempt to describe this software itself. Nor does it describe the technology and design of the combinatorial and sequential circuits out of which the hardware is built. A number of excellent books are available which describe these two levels. The computers discussed are always general-purpose, stored-program digital computers, except for a brief excursion to analog and hybrid computers in Chapter 7.

An attempt has been made to adhere to a common terminology throughout, despite the use by computer manufacturers of different terms for the same thing (or the same term for different things). In several cases, of course, terms introduced by IBM have become de facto standards.

I would like to thank the many computer manufacturers and others who have supplied information, my colleagues and students at the University of Lancaster on whom I tried out a number of these ideas, and my wife for continual encouragement.

Lancaster, October 1979 *R.G.G.*

Acknowledgements

Details of the Data General Nova are taken from *How to use the Nova computers* (Copyright 1970, 1971 by Data General Corporation). Figure 3.22(a) is reprinted from *User's manual: programmer's reference: S/130 micro programming WCS feature* (Copyright 1977 by Data General Corporation. Reproduced with permission of Data General Corporation.)

Details of the DEC PDP-8 are taken from *PDP8/e and PDP8/m small computer handbook* (Copyright 1971 by Digital Equipment Corporation). Details of the DEC PDP-10 are taken from *DECsystem 10 assembly language handbook* (Copyright 1967, 1968, 1969, 1970, 1971, 1972 by Digital Equipment Corporation). Details of the PDP-11 are taken from *PDP11/45 processor handbook* (Copyright 1971 by Digital Equipment Corporation. Reproduced with permission of Digital Equipment Corporation.)

Details of the IBM 360 and 370 ranges are taken from *IBM system/360 principles of operation* (Copyright 1964 by IBM Corporation) and *IBM system/370 principles of operation* (Copyright 1970, 1972, 1973 by IBM Corporation. Reproduced with permission of IBM Corporation.)

Figure 1.5 is derived from *John Von Neumann: collected works,* Volume 5 edited by A. H. Taub. (Copyright 1961 by Pergamon Press Ltd. Reproduced with permission of the publisher.)

Figure 2.19 is reprinted from *A guide to IBM 1401 programming* by D. D. McCracken. (Copyright 1961 by IBM Corporation. Reproduced with permission of IBM Corporation.)

Figure 2.21 is reprinted from *Communication networks for computers* by D. W. Davies and D. L. A. Barber. (Copyright 1973 by John Wiley & Sons Ltd. Reproduced with permission of the publisher.)

Figure 3.22(b) is reprinted with some revision from *Burroughs B1700 systems reference manual.* (Copyright 1972 by Burroughs Corporation. Reproduced with permission of Burroughs Corporation.)

Figure 5.10 is reprinted from *Operating systems* by S. E. Madnick and J. J. Donovan. (Copyright 1974 by McGraw-Hill Book Company. Reproduced with permission of the publisher.)

Figure 6.15 is reprinted with some revision from *Control Data 6000 series computer systems reference manual.* (Copyright 1966, 1967, 1968, 1969, 1970, 1971, 1972, 1973 by Control Data Corporation. Reproduced with permission of Control Data Corporation.)

Figure 6.19 is reprinted with some revision from 'Distributed computer networks' by P. T. Kirstein in *Nature* for 16th October 1975. (Copyright 1975 by MacMillan Journals Ltd. Reproduced with permission of the author and publisher.)

Contents

1 Introduction

1.1. The Classic Von Neumann Computer

THE TYPICAL programmer sees a modern computer in terms of one or more high-level languages, together with a command language which he must use to communicate with the operating system. His only contact with its hardware may be with an interactive terminal (if he is lucky), or with a series of error messages couched in terms of the internal structure of the computer (if he is unlucky). Even the assembler-language programmer is protected from some of the worst idiosyncrasies of the computer's hardware by the ministrations of the operating system (typically in the control of peripheral devices). The aim of this book, however, is to look beneath this superstructure, and to describe computers and their variation at a basic hardware level, below the operating system but above the detailed electronics; we discuss further the nature of this level in § 1.3.

We first consider a very simple computer, with which we can introduce the basic concepts and terminology required for the remainder of the book. We choose to do this in terms of a simplified version of the computer described in the paper 'Preliminary discussion of the logical design of an electronic computing instrument' by Burks, Goldstine, and Von Neumann (1946), for reasons which will be mentioned later. This computer, shown in Figure 1.1, consists of four sections: the store; the data manipulation unit; the input and output units; and the control unit. We review the salient points of these sections in turn.

(a) *The store*

The *store* is a collection of store locations or *words,* into each of which the computer can place a piece of information, to be retained for later extraction and use. A word is a group of electronic components, each of which can be set into either of two states. Thus a word can store any information that can be coded in the form of an appropriate number of binary bits: such a piece of information might be a numeric value, a group of one or more characters, or (as we shall see later) an instruction for controlling the computer.

The number of bits of information which a store word can hold is its

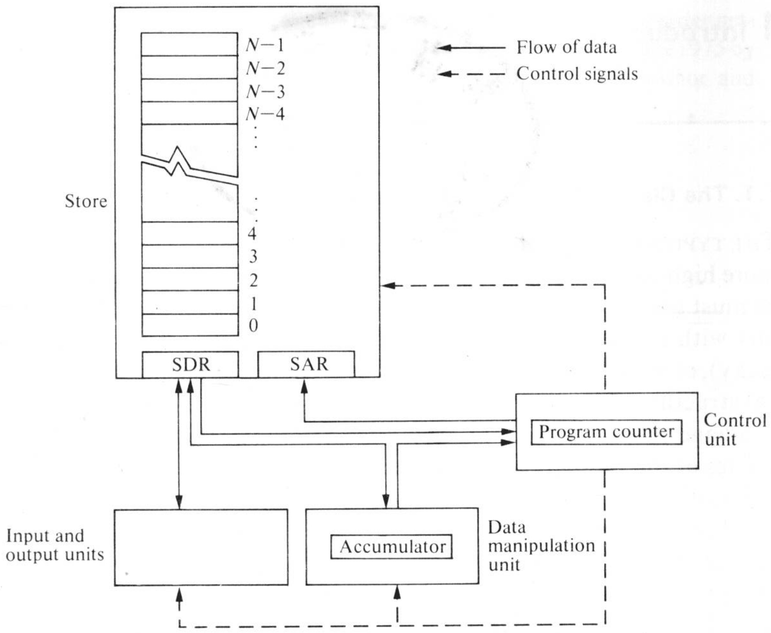

FIG. 1.1

word length, and is the same for all words in the store. The word length for the store of a particular computer is chosen to hold a maximal numeric value or number of characters appropriate to the application area and cost to which the computer is designed. Word lengths vary from 12 or less to 64 or more bits, typical lengths being 16 and 32 bits.

In order to store or retrieve information we must have some means by which the computer can refer to any location. Since a store is constructed as a vector or linear sequence of words, we specify each one by its position in the sequence. Thus if we have a set of N words or store locations, each has an associated numeric value in the range zero to N–1, the *store address*, by which it is uniquely specified. Such a store address is a new type of information that we might want to deposit as the contents of a word.

Two special storage devices or registers form part of the store, the *store address register* (SAR) and the *store data register* (SDR), or store buffer register. In order to store a piece of information, it is placed in the SDR; the address at which it is to be stored is placed in the SAR, and the control unit of the computer sends a 'write' signal to the store. After some delay depending on the technology involved, the operation is complete; the specified location contains the new information, its previous contents having been overwritten and lost. In order to retrieve a piece of information, its address is placed in the SAR and a 'read' signal sent to the store; after some delay, the information is available in the SDR, for routing to some other part of the computer. Note that the store location accessed still contains a copy of the information read.

We assume that the store is randomly accessible; that is, the time taken to access a store location in order to store or retrieve information is constant and (in particular) is independent of the particular location being accessed and of the location previously accessed.

Notice that we avoid the (anthropomorphic) term memory, and the term core store, which presupposes a particular store technology (although it is often used generically); when we wish to distinguish among several levels of storage on a computer, we will refer to this basic level as *main* or *primary* store. Furthermore the term *register* is sometimes used to refer to any device which holds a group of one or more bits of information, and which is capable of being accessed at electronic speeds: in this book we use it only for storage devices provided for some special purpose (such as the SAR), and do not apply it to a general store location.

(b) *The data manipulation unit*

The *data manipulation unit* is capable of performing any of a fixed set of *operations* as signalled by the control unit. A typical operation requires two pieces of data or *operands* upon which to operate; one is extracted from an appropriate location in store, and one is found in a special storage device or register within the data manipulation unit, the *accumulator*. The result of the operation is placed in the accumulator, destroying its previous contents. In a computer oriented towards numerical calculation, the operations must include provision for addition and subtraction, and probably multiplication and division as well; on a computer oriented towards character manipulation there would be various operations for moving and scanning character strings.

We have chosen a name for this unit which does not presuppose numerical calculation; other terms for this unit are the arithmetic unit, the arithmetic and logical unit (ALU), and the mill (a term introduced by Babbage, and used in some British computers).

(c) *Input and output units*

Any computer has a number of input/output or peripheral devices attached to it, both for communication with the outside world (card readers, line printers, terminals, etc.) and for augmenting the computer's information storage capacity (magnetic discs, magnetic tape, etc.). As with the data manipulation unit, these units are able to carry out any one of an appropriate set of operations when signalled by the control unit; for example, to read a digit punched on paper tape and store it in a specified store location or to retrieve a digit from a specified store location and print it on an electric typewriter.

At present we assume that the computer does only one thing at a time. If a read or write operation is started on a peripheral device, the computer awaits the completion of this operation before continuing with the next. When we consider that a typical input/output operation is thousands or millions of times slower than a typical internal operation (such as the addition of two numbers), and that there may be other tasks to which the computer could turn its attention while such an operation is going on, then we can see scope for redesigning the computer in this area.

We introduce the term *transput* from the Algol 68 language to cover both input and output: thus for instance we will talk of a transput device instead of a peripheral device, and a transput operation instead of an input/output operation.

(d) *The control unit*

The *control unit* supervises the operation of the other three sections of the computer discussed above. It does this by initiating the transfer of data between units, and by sending appropriate control signals, in accordance with a schedule or *program* of *instructions*. Each instruction is encoded in such a way that it can be held in a store location, so that the store contains both the data being operated on and the program of instructions specifying the operations to be performed. The control unit contains a special storage device or register, the *program counter,* which contains the address of the store location holding the next instruction to be executed or obeyed. Other names in common use for this register are the instruction counter, next instruction address register (or NIAR), sequence register, and current order register.

While the computer is running, the control unit is executing a basic cycle divided into two phases, *fetch* and *execute*. The *fetch* phase involves extracting the contents of the store location referred to by the program counter, and decoding it into an operation code portion, specifying an operation to be carried out, and an operand portion, specifying a store address (whose numeric value we will indicate as X). The program counter is then incremented by one, to point to the next store location in sequence, since this normally holds the next instruction to be executed. The control unit then enters the *execute* phase, to carry out the operation decoded in the fetch phase. There are three cases.

(a) An operation of the data manipulation unit is called for; for example, transfer the numeric value held at store address X to the data manipulation unit, and add it to the value in the accumulator, or transfer the value in the accumulator to the store and deposit it at address X.

(b) An input/output operation is called for; for example, transfer the character at store address X to the typewriter and print it, or read the next character punched on a piece of paper tape and transfer it to the store, to be deposited at address X.

(c) The sequence of instructions being executed is to be changed, so that the next instruction to be executed is not the one stored immediately after the instruction currently being executed. The store address X extracted from the current instruction is placed by the control unit in the program counter (destroying its previous contents), so that the next fetch extracts an instruction from store address X (and then from store addresses $X+1$, $X+2$, etc., until a further such 'jump' instruction is encountered).

We require two types of jump instruction; one (the *unconditional jump)* which always changes the program counter, and one (the *conditional jump)* which changes the program counter only if a certain condition is true (such as that the accumulator contains a non-negative value); if the condition is false, the next instruction to be executed is the one stored immediately after the jump instruction. Thus the computer selects a course of action dependent on the data values it encounters.

To illustrate these concepts, Figure 1.2 gives a section of program to compare the contents of locations 100 and 101, and store the larger value in location 102.

```
Load into accumulator contents of location 100
Subtract from accumulator contents of location 101
If contents of accumulator non-negative, jump to
Load into accumulator contents of location 101
Jump to
Load into accumulator contents of location 100
Store accumulator in location 102
•
•
•
•
```

FIG. 1.2

The stored program concept

One of the consequences of holding instructions in store locations is that they can be treated as data, and manipulated as such by the computer. This allows us to write programs which incorporate instruction modification. Suppose, for example, that we wish to write a program in which the computer has to sum N values held in successive store locations, perhaps with store addresses 100, 101, 102, and so on. N may be too large for there to be room for that number of add instructions to be held economically in the store, or the value of N may not be known when the program is being prepared (for example it might be read in as a piece of data). On the simple computer described above we would have to do something like the following:

We write an add instruction which initially refers to store address 99, and arrange to have it executed N times by appropriate use of a conditional jump instruction. Then, before each execution of the add instruction, we arrange to manipulate it as data in such a way that one is added to the address portion of the instruction; thus the add instruction refers successively to store addresses 100, 101, 102, and so on. A skeleton program for this is shown in Figure 1.3.

```
Set accumulator to zero
Set loop counter to N
Add one to operand field of 'add' instruction below
Add into accumulator contents of location 99
Subtract one from loop counter
If loop counter > 0, jump to
```

FIG. 1.3

There are major difficulties with the use of this technique, since it is likely to result in programs whose structure is extremely opaque, and which are therefore very difficult to understand and debug. Furthermore the lack of any rigid demarcation between the (unchanging) program and the (changing) data means that the computer hardware cannot be used to protect the program from corrupting itself, nor can a single copy of a program be shared simultaneously by several users. For these reasons the basic form of instruction modification described above is no longer used; instead its effect is achieved by the use of index registers, as described in the next section.

The stored program concept, the realization that instructions can be encoded and held in the store of the computer together with the data being operated on, is an important one for two reasons. First, as mentioned above, it introduced the idea of instruction modification, for which better mechanisms could then be found. Second, a program as a whole could be treated as data by a supervisory program; a loader, an assembler or compiler, or an operating system. Since this is such a vital concept, all the computers discussed in this book are stored program computers. We are thus ignoring two classes of externally programmed device, common in the early days of computing:

(a) Computers whose programs were held on such external media as paper tape or punched cards. Examples of this class are the Harvard Mark I (Aiken and Hopper 1946) and Babbage's analytical engine (Bowden 1953, Appendix 1).
(b) Devices where the program resides in a plugboard, such as the early ENIAC computer (Goldstine and Goldstine 1946).

We could envisage computers with two main stores, one for the program and one for data, but these would be very inflexible. Instead we assume that each store location in main store can hold either an instruction or a piece of data; this does not, of course, preclude the possibility of distinguishing these two cases at any particular time, either by segregating instruction areas from data areas, or by marking store locations in some way.

The general-purpose electronic digital computer

Let us now try and define rather more precisely the type of device that we are considering. We have introduced the term *computer,* and by this we mean a device which can process substantial quantities of data without detailed human intervention.

We have limited the field to stored-program computers, and we further limit the field to *general-purpose* computers; that is, to computers with an instruction set rich enough to perform a wide variety of tasks. This concept is rather vague because it can be shown that a computer with a very basic instruction set can simulate a Turing machine and therefore can, in theory, perform any 'computable' task, so that such a computer is (again in theory) completely general-purpose (see for example Minsky 1967).

We will not normally discuss the technology out of which the architectures that we describe are to be implemented. However, we will from time to time assume that we are discussing *electronic* computers, implemented by the routing and gating of electrical signals. This does not preclude the theoretical possibility of implementation in other technologies, such as fluidics (see for example Gluskin, Jacoby, and Reader 1964), although historically this has not been the case.

Finally, we will be considering *digital* computers, where data is stored and manipulated by devices that can assume one of only a finite number of states, rather than by devices which can assume any state in a prescribed continuous range (as in analog computers).

Von Neumann's computer

There are many possible computers which could have been used to exemplify the terms we have introduced. For example we could have used a simple computer such as the DEC PDP-8, or a suitable hypothetical computer such as that presented for didactic purposes by Knuth (1968). However, these have assimilated some of the developments in computer design which we will discuss later. Instead, we choose to use the Von Neumann, Princeton, or IAS (Institute of Advanced Studies) computer introduced in Von Neumann's paper. We take this paper to mark the beginning of the era of modern computing, since it was the 'first widely circulated document about high speed computers' (Knuth 1970).

The paper proposed a computer with a store of 4096 40-bit words stored on the faces of a number of 'Selectrons' or electrostatic storage tubes, this being a device able to provide random-access storage before the introduction of the ferrite core store (first used on the Whirlwind computer at MIT in 1953). Although such a device matches the modern conception of main store, most early computers (such as EDVAC (Knuth 1970) and EDSAC (Wiles and Renwick 1949)) had a serial store, implemented as a magnetic drum or set of delay lines (as discussed in §5.6).

The word length of 40 bits was chosen to give suitable accuracy for the type of fixed-point binary calculations for which the computer was designed (hardware floating-point arithmetic having been rejected). Instructions of such a length would have been wasteful of storage, so instructions were 20 bits long (6 for an operation code, 12 for a store address, and 2 unused) and held two to a store word, as shown in Figure 1.4. The control unit executed first the left-hand instruction, then the right-instruction, of each word in the program. Pairs of jump instructions were provided to transfer control to the left- or right-hand instruction of a specified store location.

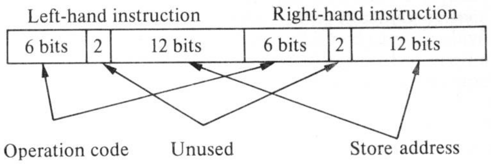

FIG. 1.4

The internal operations proposed in this paper are shown in Figure 1.5. Notice the ‘partial substitution’ orders (18 and 19), provided for instruction modification and subsequently altered to provide certain shifting facilities as well. Apart from the storage of two instructions to a word, the only extension to our rudimentary computer is in the provision of a second storage device or register in the data manipulation unit, the arithmetic register AR, used in conjunction with the accumulator for multiplication and division; the need for such a register is discussed in §2.1. Figure 1.6 shows how the section of program in Figure 1.2 would appear encoded for the Von Neumann computer, to occupy locations 50 to 53.

Input/output operations were not specified in detail in this paper, but three devices were proposed: an electric typewriter for the transfer of small quantities of data, a display unit for graphical presentation of results, and several magnetic wire or tape units to provide a secondary storage medium and for all normal input and output. It was expected that input data would be transcribed to the magnetic wire by a process which did not involve the computer, and similarly for output.

Word-oriented single-address binary computers

Some basic features of the Von Neumann computer do not carry over to all the other designs we will study, for it can be characterized as a word-oriented, single-address binary computer.