

Mark Priestley

Routines of Substitution

John von Neumann's work on software
development, 1945-1948

April 5, 2018

Springer

Preface

This book is a historical and philosophical study of the programming work carried out by John von Neumann in the period 1945-8. The project was inspired by the earliest known surviving example of von Neumann's coding, a routine written in 1945 to 'mesh' two sequences of data and intended to be part of a larger program implementing the algorithm now known as mergesort. These programs had a certain longevity, versions of them appearing in a 1948 report on programming technique that von Neumann wrote with his collaborator Herman Goldstine. The publication of that report marked the end of von Neumann's active interest in programming theory. He continued to design and write programs as opportunities presented themselves, but computer programming was no longer at the forefront of his creative work.

It would be hard to overstate the importance of this passage of work. Among other things, it encompassed the elaboration of the model misleadingly known as the 'von Neumann architecture', which became the foundation of the design of the overwhelming majority of computers built since 1945, the design of the first machine code for those computers, the basis of the overwhelming majority of all programming carried out since 1945, and the promulgation of a highly influential methodology of program development, including the flow diagram notation.

For all the importance of this work, its history is less well known. Many of the primary texts are unpublished or available only in expensive and hard to obtain editions. As a result, von Neumann's work is often (mis)interpreted in the light of subsequent understandings of computers and programming. Von Neumann himself is a controversial figure, and in recent years his role in early computing has been eclipsed in popular representations by that of Turing. It is a good time to carry out a clear-eyed re-examination and re-evaluation of von Neumann's work.

At the heart of the book is an examination of the 1945 manuscript in which von Neumann developed the meshing routine. The text of the manuscript itself, along with a preliminary document describing the code he used to write this program, are reproduced as appendices. The program is approached in three chapters describing the historical background to von Neumann's work, the significance of the sorting application itself, and the development of the EDVAC, the machine for which the program was written. The subsequent chapters widen the focus again, discussing

the subsequent evolution of the program and the crucial topic of subroutines, before concluding by situating von Neumann's work in a number of wider contexts.

The book also offers a unifying philosophical interpretation of von Neumann's approach to coding. Even when its historical importance is recognized, the technical content of his work is routinely disparaged. It is said to be too rooted in the details of computer hardware, or too complex and/or mathematical, or to lack the elegance of other approaches to automatic computation. Nevertheless, there is a clear and unifying vision running through it which we should at least try to understand before moving to criticism.

This project has benefited greatly from archival work done in connection with the *ENIAC in Action* project, and I would like to thank my collaborators on that project, Tom Haigh and Crispin Rope, and to gratefully acknowledge the financial support of the Mrs. I. D. Rope Second Charitable Foundation, which made much of the earlier research possible. The manuscripts transcribed in Appendices A and B are reproduced by kind permission of the American Philosophical Society, and I would also like to thank the staff of the APS Library for the enthusiastic assistance that made working there such a pleasure.

Mark Priestley
London, 2018

Contents

1	Introduction	1
1.1	Programs as texts and artefacts	2
1.2	The background to the 1945 manuscript	4
2	Sorting and collating	7
2.1	Punched card machines	7
2.2	The unification of sorting and computing	9
2.3	Internal and external sorting	13
2.4	Algorithms for internal sorting	15
3	EDVAC and its codes	19
3.1	The Bell Labs relay computer	19
3.2	The first EDVAC	21
3.3	The second EDVAC	27
3.4	Substitution	32
4	The 1945 meshing routine	35
4.1	Defining the problem	36
4.2	An algorithm for meshing	37
4.3	Sequence programming	38
4.4	Coding with variables	40
4.5	Allocating the short tanks	42
4.6	Making a single code sequence	43
4.7	Loading and calling the meshing routine	45
4.8	Process overview	47
5	Planning and coding	49
5.1	The IAS machine and its code	50
5.2	Programming with diagrams	53
5.3	The meshing routine	57
5.4	The sorting routine	60

5.5	Planning and coding	62
5.6	The final versions of the sorting routines	66
6	Subroutines	69
6.1	Master and subsidiary routines	70
6.2	General program sequences	71
6.3	Preparatory routines	73
6.4	Diagramming subroutines	76
6.5	Second-class citizens	79
7	Contexts and conclusions	81
7.1	The genesis of the general purpose computer	81
7.2	The history of programming	83
7.3	Planning	85
7.4	Coding	86
A	Von Neumann's second EDVAC code	89
B	Von Neumann's meshing routine manuscript	95
	References	109
	Index	115

Chapter 1

Introduction

In the summer of 1944 the mathematician John von Neumann encountered the ENIAC, then under construction at the University of Pennsylvania's Moore School of Electrical Engineering in Philadelphia. ENIAC was being built for the Ballistic Research Laboratory (BRL) of the US Army's Ordnance Department, based at the Aberdeen Proving Ground on the Maryland coast some 70 miles from Philadelphia. In 1943, faced with a backlog of routine calculations, BRL had agreed to support John Mauchly's proposal to build an electronic computer capable of solving the differential equations involved in producing firing tables for the new weapons they were developing.

Despite being a member of BRL's Scientific Advisory Committee and engaged in a nation-wide search for computing facilities that could be deployed on behalf of the Manhattan Project laboratory at Los Alamos, it appears that von Neumann first heard about ENIAC in a casual conversation with Herman Goldstine at Aberdeen railway station. Goldstine, a University of Michigan mathematician, had been called up into the Army and posted to Aberdeen in August, 1942. He was given the job of running BRL's satellite computing unit at the Moore School, a group of human computers performing ballistic computations with the aid of a differential analyzer. When Mauchly and Presper Eckert appeared with proposals for a giant electronic calculator to automate these calculations, Goldstine was immediately interested, and his support was instrumental in getting the project approved and funded by the Ordnance Department. He became BRL's official representative on the ENIAC project and played a crucial role in bringing it to a successful conclusion.

In mid-1944 ENIAC was still more than a year away from completion, but its design was more or less fixed and the team were eager to pitch ideas for a follow-up project before the generous war-time funding arrangements dried up. Von Neumann immediately saw ENIAC's potential and helped bring it about that the first problem it ran, in the winter of 1945/6, was a series of calculations for Los Alamos. However, his collaboration with the ENIAC team focused on the design of the new machine written up in the famous *First Draft of a Report on the EDVAC*, now regarded as the first definitive statement of the principles underlying the modern computer.

The *First Draft* concluded with a brief and rather incomplete discussion of the instructions that would be used to control EDVAC. This might give the impression that programming was an afterthought in the development of the new machine, but in fact von Neumann devoted a significant amount of time and effort to the topic in 1945; in particular, he developed a number of sets of instruction and tested them by writing programs. One example of this work has been preserved in the archive of Goldstine's papers in the American Philosophical Society. A 23-page manuscript, transcribed in Appendix B of this book, contains a detailed and systematic account of the development of a program for what von Neumann called the 'meshing' of two sequences of data, a routine that he planned to use as part of an implementation of the algorithm now known as mergesort.

This data-processing application was rather different from the mathematical problems that were expected to form the normal workload of machines like ENIAC and EDVAC, but proved to be surprisingly central to the evolution of von Neumann's thinking about programming. The sorting application thus provides a lens through which to examine the origins of the software development methodology presented in the reports entitled *Planning and Coding Problems for an Electronic Computing Instrument* that he and Goldstine issued in 1947 and 1948.

1.1 Programs as texts and artefacts

This approach assumes that historical and philosophical insight can be gained by the detailed examination of individual programs, in this case von Neumann's evolving meshing and sorting routines. The study of individual programs is uncommon in the literature, and exactly what it might involve is a more complex question than appears at first sight. The answer will depend on what programs are taken to be.

At first, it seemed natural to think of programs as texts. When computer time was a scarce and expensive resource, most programming was a paper exercise seen as a novel form of mathematical or logical activity. Programs were taken to be sets of orders or instructions given to slave-like machines. Naturally, human programmers and mechanical computers spoke different languages, and the notion of translation was invoked to describe the transformation of these orders into a form the machine could 'understand' (Nofre et al. 2014).

If programs are texts, the study of programs is a form of reading. The activity of reading programs has occasionally been emphasized by computer scientists, but the systematic study of program texts has never formed a central part of computer science education. More commonly, programs are used as a textual resource for the craft practice of programming where existing code is copied and then altered for a new purpose. Reading programs for their own sake has instead been adopted by scholars with a more humanist orientation, in the fields of software studies and critical code studies, applying the techniques of literary analysis and close reading to program texts (Montfort et al. 2013, for example).

Von Neumann's manuscript was subjected to a close reading by Donald Knuth (1970), who adopted the perspective of a computer scientist or programmer rather than that of a literary scholar. Knuth translated von Neumann's code into a typical late-1960s assembly language, a notation now almost as arcane as von Neumann's original symbolism. The details of Knuth's analysis are still valuable, but this book hopes to inspire and make possible a reading of von Neumann's original texts.

The textual analogy is not the only way to think of computer programs, however. Turing pointed to a general equivalence between programs and machines, and in practical contexts this has been thought of as a distinction between static code and dynamic processes running on computers. The historian Michael Mahoney placed the emphasis firmly on the latter:

Ultimately it is [programs'] behaviour rather than their structure [...] that interests us. We do not interact with computers by reading programs; we interact with programs running on computers. The primary source for the history of software is the dynamic process, and, where it is still available, it requires special tools of analysis. Programs and processes are artefacts, and we must learn to read them as such. (Mahoney 2005, p. 129)

As Mahoney pointed out, however, this implies that a prerequisite for studying a program is a machine to run it on, along with all the necessary system software, operating systems, database systems and the like. For many, if not most, historical programs, this infrastructure no longer exists, reducing the attempt to study software to a never-ending exercise in emulation and reconstruction.

Even this attempt would be rather artificial in the case of von Neumann's 1945 program, written for a computer that was never built, never run, then rewritten for a different machine and—perhaps—run five years later when that machine finally came into service. It is not clear that it ever was an artefact, or that it is useful to study it as one. It would be straightforward to build a simulator on which to run the program, but it was only intended to perform a small and rather simple data manipulation and there are limits to what would be learned from this exercise.

Another issue with Mahoney's proposal arises from the fact that, as he argued elsewhere in his article, software is embedded in its social context. Even in the situation where programs have survived and can be run on a historical machine, the 'dynamic process' only captures a small part of the significance of the software to its original users. Mahoney's proposal seems to edge us towards the second of the two modes of history identified by Jon Agar (1998), 'one centred on texts, the other on recapturing spirit'.

Historians do not have to make a sharp choice between treating software as text or artefact, however. At different stages in the passage from conception to use, one aspect or the other will dominate, and the historian's interest can legitimately be directed to either. In an influential article, McClung Fleming (1974) enumerated the similarities in approach to studying artefacts and texts, and this book loosely follows the schema that he proposed, beginning with the identification of the artefact in question.

1.2 The background to the 1945 manuscript

By the summer of 1944, the ENIAC team had identified two major shortcomings of the machine they had not yet completed: it could only store and manipulate a few numbers, and the process of setting it up to run a new problem was cumbersome and time-consuming. Since the beginning of the year, they had been exploring ideas and technologies to address these problems but had not articulated a compelling case for embarking on a new development project. Von Neumann brought a much clearer sense of computational priorities: drawing on his wartime consulting experience, he had identified the solution of non-linear partial differential equations as a critical obstacle to the development of various fields of applied mathematics. These equations could not be solved analytically, and numerical solutions were beyond the scope of all existing and planned computational devices.

A machine with ENIAC's speed would be needed to solve such equations, but it would also need a large fast store to hold the mass of numeric data required. This fitted well with the ENIAC team's inchoate plans, and BRL were soon persuaded that a new machine would address parts of their computational workload that even ENIAC could not reach. By September, a contract had been agreed authorizing the Moore School to begin work on a new machine and soon afterwards von Neumann was appointed as a consultant to the new project. In the early months of the collaboration he contributed to discussions on a wide range of issues, but it was understood that the primary focus of his work was the logical control of the new machine.

This logical control includes methods for translating a problem into a form which can be inserted into the EDVAC; means for telling the EDVAC by coded signals which mathematical operations are to be performed on these data; and means for abstracting from the machine the solutions which are desired. (Eckert et al. 1945)

As he travelled back and forth across the country, von Neumann continued to think about EDVAC, keeping in touch with the Philadelphia group by writing to Goldstine. He was in Aberdeen briefly at the beginning of February 1945, but soon headed west to Los Alamos. Nevertheless, as he reported:

I am continuing working on the control scheme for the EDVAC, and will definitely have a complete writeup when I return. (von Neumann 1945d)

By the middle of March, he was back in Princeton and met the rest of the team for a series of meetings set up to discuss the problems of logical control. At the end of March, it was reported that:

Dr. von Neumann plans to submit within the next few weeks a summary of these analyses of the logical control of the EDVAC together with examples showing how certain problems can be set up. (Eckert et al. 1945).

The details of the problems von Neumann was going to consider were not recorded. Sorting was a topic of interest, the meeting on March 23 noting that 'two switches give an appreciable gain only in systematic sorting problems', but there is no evidence that von Neumann took any steps to code a sort routine until April when, as discussed in Section 2.2, sorting became a high-priority topic for the team.

Towards the end of April, von Neumann sent the manuscript of what became the *First Draft* to the EDVAC team. The *First Draft* only mentions sorting in a couple of places, as one of the applications that should be taken into account when estimating the size of EDVAC's memory. At the beginning of May, however, he sent Goldstine a letter with new material to be incorporated into the manuscript, commenting that he had also 'worked on sorting questions' and noting rather cryptically that:

My present EDVAC sorting scheme requires ~130 minor cycles ~4 tanks for the logical instructions (~1.5% memory capacity), and uses the code specified without any changes. (von Neumann 1945e)

These estimates suggest that the sorting scheme had been worked out in detail but unfortunately the letter did not include any more information about it.

However, the contents of this letter were not included in the text of the report that was reproduced and circulated in June. As a result, the prominence of the *First Draft* has had the effect of concealing the programming work that was part of its preparation. The code is presented on its last page, with the important feature of address modification only mentioned in passing in the final remark of the report. The details are incomplete, even though von Neumann had sent a complete version to Goldstine in May. There is no mention of the detailed coding work that von Neumann had done which must, in turn, have affected the development of the code itself.

As work progressed, von Neumann carried on coding. By September, a major change had complicated the structure of EDVAC's memory, as explained in Section 3.3. The new plans were described in a progress report at the end of September in which Eckert and Mauchly characterized von Neumann's contribution as follows:

Dr. von Neumann [...] has contributed to many discussions on the logical controls of the EDVAC, has proposed certain instruction codes, and has tested these proposed systems by writing out the coded instructions for specific problems. (Eckert and Mauchly 1945, 3)

Von Neumann's manuscript is associated with this period in EDVAC's history. A significant side-effect of the memory upgrade was to substantially change the way the machine was programmed. In a second manuscript, reproduced in Appendix A of this book, he gave a rather abstract description of EDVAC's new architecture and a set of instructions very similar to those used in the meshing program. Eckert and Mauchly (1945, 75-7) reproduced a version of this 'order code' in their report, describing it as 'essentially that which von Neumann has proposed after trying out various coding methods on typical problems'.

Von Neumann's manuscripts are undated, but they were most likely written in the late summer or early autumn of 1945. They then had to be typed and copied for circulation to interested parties. In November, Goldstine sent some reports to Calvin Mooers, who was working on a computer project led by John Atanasoff at the Naval Ordnance Laboratories. There was close collaboration between this project and the Moore School: von Neumann advised at the start of the project, and Mauchly made regular consultancy visits. Although the project was abandoned in 1946, Mooers himself developed an interest in coding and was one of the few outsiders invited to give a talk at the Moore School lectures in the summer of 1946 (Campbell-Kelly

and Williams 1985). It appears that Goldstine had intended to send von Neumann's program to Mooers:

I am sorry we have not as yet sent you the von Neumann sorting problem. My secretary has, however, been extremely busy with some other work and will get that report to you as soon as she possibly can. (Goldstine 1945a)

The technologies of document reproduction available in 1945 hindered the dissemination of research in ways that are now hard to imagine. Von Neumann's manuscript was typed up on at least two separate occasions: Goldstine's archive preserves one copy of one typescript and five copies of a second. In each of these six documents, all the mathematical material as well as the code of the program itself was inserted by hand. This work was not all carried out by Goldstine's hard-pressed secretary, Akrevoe Kondopria¹; he received a note from a friend who signed himself 'Jimmy' and wrote, perhaps reproachfully:

Here's that MS with formulas filled in – the first 9 pp I left with you. I'm in opticians having bifocals fitted, alas! (Jimmy nd)

It is difficult to identify who filled in all the formulas in the typescripts of von Neumann's program, but certainly more than one person was involved, including von Neumann himself, and handwriting similar to Jimmy's is found on one of the typescripts. The date January 17, 1946, is pencilled in the margin of one typescript, suggesting that work was still going on two months after Goldstine had written to Mooers. But it all may have been for nothing: there are no letters in Goldstine's papers acknowledging receipt of the program, and the presence of six copies of the document suggests it may never in fact have been dispatched.

Arthur Burks (1989) came to believe that the surviving manuscript contained the detailed version of the sorting program that von Neumann had, in May, promised to send to Goldstine, and that there was no other code waiting to be discovered. However, while von Neumann had referred to a 'sorting' program, the manuscript distinguishes sorting and meshing as separate tasks and only gives code for the meshing problem. More significantly, von Neumann wrote that his program used the *First Draft* code 'unchanged', but the manuscript uses the later, rather different code. It seems quite possible that there were earlier versions of von Neumann's program that have not been found, and that he had also worked on the sorting problem.

¹ Kondopria is identifiable from the 'bureaunym' (Barany 2018) 'AK' appearing at the foot of Goldstine's letter. She later worked with Goldstine and von Neumann on the Electronic Computer Project at the Institute for Advanced Study.

Chapter 2

Sorting and collating

Von Neumann's manuscript describes a routine to merge two sorted sequences into one, intended to be used as part of a larger program to sort a sequence of data. These applications were, and continued to be, of importance to the EDVAC team, despite seeming very different from the normal workload of an automatic mathematical computer. This chapter describes the migration into the new world of electronic computation of concepts and practices that were for the most part familiar from the domain of data processing.

2.1 Punched card machines

The fundamental principle of the IBM accounting method and similar systems was the representation of information by means of perforations in cards. Standard IBM cards contained 80 columns, each of which could be punched in one of 12 places to represent the digits 0–9 or the special characters X and Y. Punching a single hole allowed a column to hold a single digit; later schemes allowed letters to be coded by punching multiple holes in a single column. Cards were known as *unit records* and could hold various pieces of information about a business entity such as a customer or a transaction. Subsets of the card's columns that held discrete pieces of information such as a social security number were known as *fields*.

Once punched, unit records were grouped into *files*, or decks of cards, such as the file containing the daily payroll records for all of a factory's employees. These files were processed by a wide variety of special-purpose machines. The original punched card machine was the tabulator, developed for the US Census of 1890. Reflecting the needs of this application, the arithmetical capability of tabulators was limited to making counts, but they were soon joined by more powerful machines capable of computing and printing totals and sub-totals of the numerical information punched on the cards. By the 1930s, these simple machines were being increasingly replaced by more general 'Electric Accounting Machines'.

For an accounting machine to perform its required function, the selection and order of the cards presented to it was crucial. Different files, and different orderings of the cards within them, would be required for different purposes. An important part of the accounting method, therefore, was to perform physical operations on decks of cards to bring about the required logical relationships between the information stored on them. Machine support was also available for this purpose:

When the punching has been completed, the cards are usually in miscellaneous order. The next step is to arrange them in sequence by some desired classification—that is, to group them according to some information which is punched in them. The Electric Card-Operated Sorting Machine is used for this purpose. (IBM 1936, 2–6)

A sorter examined one column of a card and depending on the perforation in that column sent the card to one of thirteen output pockets, one for each punch position and one for rejects. It therefore split an input deck into multiple decks which had to be reassembled by the operator to create a single sorted deck.

This simple operation formed the basis of semi-automated procedures to perform more complex tasks, such as rearranging a deck of cards so that the cards were in ascending order of some field, such as employee number. To sort on a field of more than one column, multiple sorts had to be performed, one for each column in the sort field, starting with the least significant column. After each sort, the subdecks had to be reassembled and reintroduced into the machine by hand. Thus sorting a deck of cards on a 5-digit employee identification number, for example, would require five passes of the deck through a sorter, with a manual collection of the cards being performed after each pass.

While sorters split card files into multiple decks, collators performed the opposite task of combining two files into one. The collator was developed in 1937 in response to the data processing demands created by the Social Security Act introduced by Roosevelt in 1935.

The ‘world’s biggest bookkeeping job’ was done in a Baltimore brick loft building, chosen because it had 120,000 square feet of floor space and was structurally strong enough to bear the weight of 415 punching and accounting machines. A production line was set up to punch, sort, check, and file half a million cards a day. The collator became a widely used device in government and business generally.

H. J. McDonald, who sold the account, recalls that IBM President Watson ordered the development of the collator because ‘The Social Security agency punched cards from records sent in by employers all over the country. There were millions and millions of them, and if we hadn’t had some way of putting them together we would have been lost; we just couldn’t have done it.’ (Eames and Eames 1973, 109)

The collator could perform a range of functions, but its most important job was:

to file two sets of cards together. This operation is referred to as *merging*; that is, two sets of cards are filed or merged together according to a control field. For example, in a payroll application, at the end of a pay period, the Daily Payroll cards must be in sequence by Employee Number. [...] At the end of each day, the Daily Payroll cards are sorted and then filed by Employee Number behind the cards of the preceding day. This will eliminate sorting all the Payroll cards together by Employee Number at the end of the pay period. (IBM 1945, 3)

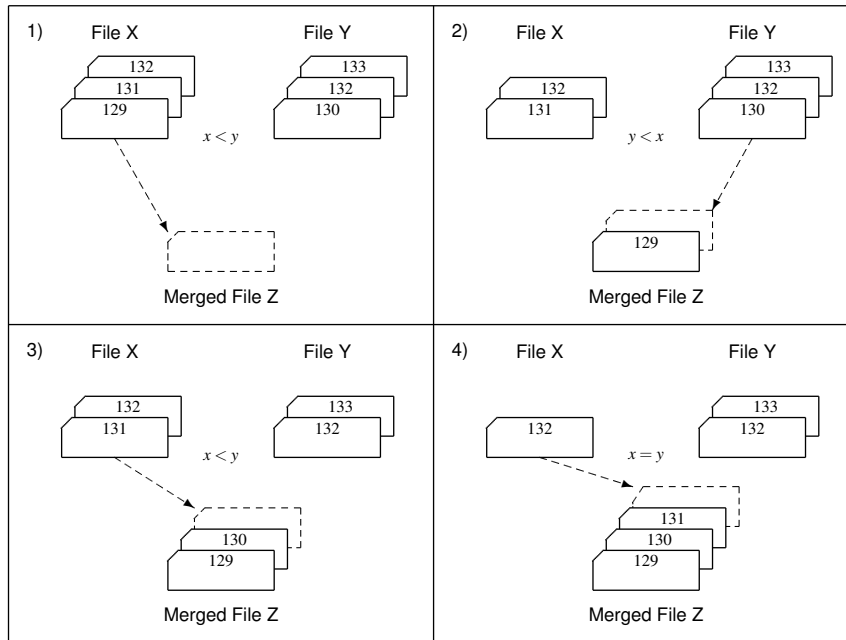


Fig. 2.1 Merging two card files on a collator. Adapted from (IBM 1945, 4).

This operation relied on a *comparing unit* which examined two numbers x and y and triggered different operations depending on whether $x < y$, $x = y$, or $x > y$. The numbers to be compared could both be read from cards; alternatively, a number read from a card could be compared with one set up on the machine. This gave the collator a wider range of functionality than simply merging two card decks. The development of the collator therefore represented an introduction of conditional control into IBM's tribe of punched card machines.

The control panel then becomes the medium through which the comparing units are controlled, and the medium through which the results of these comparisons are in turn directed to control the movement of the cards in accordance with the requirements of the specific application. (IBM 1945, 6)

The documentation for the collator included the depiction of merging shown in Figure 2.1. This is the procedure that von Neumann coded in his 1945 manuscript, but he consistently preferred to call the operation *meshing* rather than merging.

2.2 The unification of sorting and computing

The EDVAC team discussed sorting and collating intensively in the spring of 1945 and frequently returned to the topic thereafter. Nevertheless, as Eckert and Mauchly

(1945, 54) observed, ‘sorting and collating are not always thought of as computing operations’, raising the question of why these data-processing operations were so important to a project dedicated to the design of an automatic scientific calculator.

Large punched-card operations were an integral part of scientific computation. Harvard astronomer Leland Cunningham joined BRL during the war and worked closely with the ENIAC team to plan the machine’s mathematical capabilities. He was a member of the Ballistic Computations Committee that planned how ENIAC would be used when it was delivered to BRL, and in April, 1945, he considered the punched card support that would be needed (Cunningham 1945). Punched cards were ENIAC’s input and output medium, and Cunningham realized that there would be a lot of them—he estimated that 750,000 cards would be required in the first three months of operation—and that they would require a lot of processing between runs. He drew up four diagrams, which he labelled ‘flow-charts’, showing how ENIAC’s input data would be prepared and its results processed by a menagerie of punched card machines including punches, verifiers, reproducers, sorters, collators, summary punches, printers and tabulators.

Experience showed that Cunningham’s foresight was well grounded. Records of ENIAC’s early work, such as the Los Alamos computation and Douglas Hartree’s program, give exhaustive details of the different card decks that had to be prepared for problems which, mathematically, involved only the numerical solution of some differential equations (Haigh et al. 2016). In Cunningham’s flowcharts and in these early applications, ENIAC appears not so much as a revolutionary electronic computer as a novel piece of equipment in a traditional punched card installation. Just as IBM’s sorters automated a single process but required manual support to turn them into effective tools, so ENIAC’s automation of the processes of computation relied on support from the machines of the previous technological generation.

As Cunningham’s flowcharts vividly show, ENIAC was part of a computational ecology in which computing and sorting were carried out by different devices and the team seems to have assumed that the same would hold true for EDVAC. Mauchly was particularly interested in the question of sorting and collating, and in late 1944 he interviewed several heavy users of punched card machinery, including William Madow of the US Census Bureau (Norberg 2005, 79-80).

The Census Bureau had been involved with punched card equipment ever since enlisting Herman Hollerith’s assistance to process the results of the 1890 census (Truesdell 1965). Madow outlined some of the data-processing bottlenecks that the Bureau faced, and suggested a number of special-purpose devices to carry out tasks such as ‘forming the sum of the products of pairs of numbers’. Mauchly’s notes of the meeting concluded with a rather mundane proposal for a ‘device which would simultaneously provide the sums of squares of a variable, x , and the sums of squares of a variable, y , and the sums of products, xy , as well as forming the sum of the x values and the sum of the y values’ (Mauchly 1944a).

Another of Mauchly’s interviewees was Solomon Kullback, head of the US Army Signal Corps’ Cryptanalytical and Cryptographic Department. Kullback had been one of the first US liaison officers at Bletchley Park and the report he wrote at the end of his posting described the IBM installation at Bletchley in very complimentary

terms, recommending that ‘our IBM room follow insofar as is consistent with our needs and organization a procedure similar to the British IBM setup’ (Kullback 1942).

In wartime, the Signal Corps were using around a million punched cards per day, more than even the Census Bureau, and Kullback emphasized the need for faster and more flexible machines. Mauchly noted that ‘high-speed computing devices can obviously be adapted to use as high-speed ciphering devices’, while Kullback hinted at the open-ended nature of cryptographic work, observing that ‘the interest of the Signal Corps would best be served by the development of extremely flexible devices which need not be specialized to the point where they would be considered simply as ciphering machines’ (Mauchly 1944b).

At a second meeting the following April, Kullback described several ‘sorting and hunting’ problems; Mauchly classified these as ‘function table’ problems that would be carried out ‘in the same way that one would hunt reciprocals or other functions using IBM equipment’ (Mauchly 1945c). Another problem involved searching through a large volume of coded messages to locate small statistical irregularities in the occurrence of particular textual patterns, and Mauchly observed that ‘it would be advantageous to store the data in a high-speed memory device before beginning the frequency counts’.

In October, then, Mauchly appears to have been thinking of developing high-speed devices to replicate, accelerate, or slightly extend the performance of existing special-purpose machines. By April, plans for EDVAC had matured significantly, and this is reflected in the later discussions. He contemplated the use of high-speed memory in cryptanalysis, and after discussion of another problem Kullback ‘saw no reason why the same method could not be employed when using a device like the EDVAC’ (Mauchly 1945c). At the same time, he was discussing the advantages of more conventional high-speed computation in meteorology with a group of civilian and military weather forecasters (Mauchly 1945a).

In mid-April, Mauchly noted that at the time of the earlier meetings it had been ‘taken for granted that that sorting and computing problems were best handled by quite separate and distinct machines’ and that it had taken a particularly demanding application to force a reappraisal of this view:

the need for reconsideration of this point was evident when Dr. Wilks supplied an example of a problem in which an enormous amount of sorting was required, and it became evident that the solution of this problem by an automatic machine process was not possible in a ‘reasonable’ time unless both computing and sorting were done at speed characteristic of electronic devices.

It is therefore prudent to enquire more closely into sorting problems, and to consider carefully the extent to which it is expedient to adapt the EDVAC to handle sorting and collating problems (a) within its high-speed system, and (b) by simple adaptation of the input and output devices. (Mauchly 1945b)

Samuel Wilks was a Princeton statistician well-known to the EDVAC group: along with Goldstine and Cunningham, he attended the proto-cybernetics meeting organized in January by von Neumann, Norbert Wiener, and Howard Aiken, and in the division of labour that von Neumann proposed after the meeting he was assigned

to a group investigating the ‘application of fast, mechanized computing methods to statistical problems’ (von Neumann 1945f). Mauchly did not record any details of Wilks’ problem, but the middle of April marks a sea-change in team’s attitude to sorting. On April 13, von Neumann was in Washington where he was due to deliver a lecture on the theory of games to Kullback’s Statistical Seminar, when he received a plea from Goldstine:

We should be very pleased to have you come either on Tuesday or Wednesday, which ever is more convenient for you. At the present we are thinking about the sorting and printing problems associated with the Edvac and are anxious to get your opinion and advice on many points. (Goldstine 1945c)

According to contemporary notes taken by Arthur Burks, the EDVAC group had two meetings with von Neumann in April, and the second meeting

opened with a discussion of sorters. Mr. Eckert pointed out the advantages of the magnetic tape, namely that it can stand handling, that it can be erased, and that being in a continuous loop, its contents cannot get out of order. [...] There [was] some discussion of whether or not several magnetic tape sorters should be made, some of which could be detached from the machine and used separately for simple sorting problems. This question was left unsettled since the answer depends upon the speed of the tape and the extent to which the EDVAC relies on magnetic storage. (Burks 1945)

The topic of sorting evidently seized von Neumann’s attention. At the beginning of May, as we have seen, he reported to Goldstine that he had been working on sorting questions and had coded a sorting scheme. He concluded that

the EDVAC, with the logical controls as planned for ‘mathematical problems’, and without any modifications for ‘sorting’ problems, is definitely faster than the IBM’s on sorting. (von Neumann 1945e)

An explicit link between sorting and statistical problems was made in a number of texts. The *First Draft*, for example, completed by the end of April, referred to ‘sorting problems and certain statistical experiments’ as placing significant demands on EDVAC’s memory, and in 1948 Goldstine and von Neumann summarized the connection as follows:

The sorting operations can be combined, and alternated, with properly mathematical operations. [...] This circumstance is likely to be of great importance in statistical problems. It represents a fundamental departure from the characteristics of existing sorting devices, which are very limited in their properly mathematical capabilities. (Goldstine and von Neumann 1947-8, vol. 2)

It has been suggested that von Neumann’s comparisons of EDVAC with punched card machines reflect a recognition of the potential role of electronic computers in data processing. For example, Charles and Ray Eames annotated a reproduction of the first page of von Neumann’s meshing routine manuscript with the following comment:

Von Neumann had no doubts that the new machine would be effective for scientific computing, so he chose to test its versatility by coding an operation central to business applications—sorting. (Eames and Eames 1973, 138)

There seems to be no evidence that business applications were ever considered as possible uses of EDVAC, however. Mauchly was contemplating new machines that could sort faster than the IBMs, but it was Wilks' statistical problem that forced the issue and led to EDVAC being reconceptualized as a combined computing and sorting machine. Once that connection had been made, it would no doubt have been natural for Mauchly to begin to think of EDVAC-type machines as a technology that could be sold to organizations like the Census Bureau to address their data-processing needs, a marketing trajectory that was indeed followed by the startup company he formed with Eckert in 1946 (Norberg 2005, 81).

Sorting and collating were not just test procedures borrowed from the world of data processing, then, but were intrinsic to many of the applications that EDVAC was intended to handle. As the new project got under way, Mauchly was simultaneously surveying intensive users of punched-card machinery in a variety of application areas, apparently with the intention of developing high-speed sorting devices that could be used either independently or alongside EDVAC.

The idea that EDVAC itself could be used to carry out sorting and collating seems to have emerged rather gradually. The reference to 'systematic sorting problems' in the notes of the team's March meetings (Burks 1945) indicates that the topic had received some attention, but the flurry of activity following Mauchly's April memo suggests that it was only at this point that the idea that EDVAC could be a device which unified the apparently distinct activities of sorting and computing really seized the imagination of the group. Importantly, this realization was the result of an extended period of investigation of and reflection on potential uses for the new machine.

The details of how this unification was achieved are of more than local interest, as they mark the first step in the evolution of the computer, originally conceived simply as a scientific instrument, into a general-purpose information processing machine.

2.3 Internal and external sorting

Mauchly's memo distinguished two ways in which an EDVAC-like machine could perform sorting. The first was 'within its high-speed system', with all the data to be sorted held in the machine's high-speed memory and the sort being performed under the control of coded instructions, while the second involved 'simple adaptation of the input and output devices' which played an integral role in the sorting procedure. He later distinguished these as 'the "internal" and "external" memory sorting techniques' (Mauchly 1946a), terms that are still in use today. Von Neumann's meshing and sorting routines are all purely internal processes.

Although von Neumann evidently found the challenge of coding these routines rewarding and insightful, internal sorting had an obvious limitation, namely the amount of information that could be held in memory. In the *First Draft* he hinted at this issue, estimating that EDVAC's internal memory would be able to hold the equivalent of '700 fully used cards' and concluding that

the device has a non negligible, but certainly not impressive sorting capacity. It is probably only worth using on sorting problems of more than usual mathematical complexity. (von Neumann 1945a, 63)

It wasn't immediately obvious what adaptations should be made to the input and output devices to provide an alternative to purely internal sorting. At the April meeting, the focus was first on the external storage medium to be used and Eckert offered powerful reasons for favouring magnetic tape over punched cards. At the end of the meeting there was

some discussion of whether or not several magnetic tape sorters should be made, some of which could be detached from the machine and used separately for simple sorting problems. (Burks 1945)

This suggests that the group was thinking in terms of special-purpose devices to perform external sorting rather than having EDVAC's central control simply make use of the tape units. Internal sort routines could be considered in isolation from the details of input and output, but the problem of sorting large data sets was more complex and, as von Neumann noted in the *First Draft*, presupposed a more detailed consideration of EDVAC's external storage medium and associated input and output devices than had yet been carried out.

Eckert and Mauchly came back to external sorting in their September report. They noted that to take advantage of high processing speeds, sorting and collating should be carried out within the machine whenever possible. However:

This 'solution' is in fact quite inadequate for for large sorting jobs, since the memory capacity of the electronic machine is necessarily limited, while the number of cards (representing the memory) which may be processed by a card sorter is essentially unlimited. The magnetic wire or tape is the corresponding unlimited memory for the electronic machine, and sorting and collating processes must be devised which will utilize this memory without being restricted in any way by the limitations of the internal high-speed memory. (Eckert and Mauchly 1945, 55)

They then considered how the familiar procedures used by punched card sorters and collators could be translated onto EDVAC. They started off by commenting on an effect of the change of media:

The essence of sorting and collating is to take information which is arranged in one order (which may be random) and rearrange it in some other order. When the information is punched into cards, the information is 'moved' from one place to another by moving the card that carries that information. When paper tapes are used, this method is no longer feasible, and the information itself must be moved by perforating the same information into a new tape (Eckert and Mauchly 1945, 55)

Every time the information is reordered, of course, previously punched paper tapes became useless and to avoid the issue of waste Eckert and Mauchly proposed the use of erasable and rewritable magnetic media, such as tape or wire.

They then discussed how a device which could sort data into only two categories, as opposed to the 13-way split that punched-card sorters provided, would still be able to sort data using a digit-by-digit algorithm. They concluded that four tape mechanisms would be required to do this efficiently, and went on to show how the

basic binary collating procedure could also be carried out on such a set-up. As far as performance went,

any sort of collating, systematic or monotonic, can be carried on at the speed with which the magnetic wire device can transfer information [...] It should be emphasized again that, when the data to be sorted or collated exist entirely within [the] high-speed memory, then much higher rates are available. (Eckert and Mauchly 1945, 62-3)

The difference between this analysis and von Neumann's routines is striking. While acknowledging the need for an analysis of the interaction between internal and external memory, in practice von Neumann deferred this as long as possible. As late as 1948, he and Goldstine wrote that they would postpone consideration of a more practical sorting scheme until the details of the input and output devices and the instructions to control them were fixed.

Conversely, Eckert and Mauchly offered little explanation of the code that would control a sorting procedure using external tape drives, and in passages such as the following still seemed to be considering special-purpose devices:

The collator must compare the key words from the two inputs and determine which of the two units of input data is to be recorded next on the output wire. A comparing circuit is able, within one or two minor cycles, to determine which of two numbers is the larger. This time is negligible relative to the time required for reading or recording one word on a wire. (Eckert and Mauchly 1945, 62)

In November, 1945, von Neumann gave an outline of sorting and collating to members of the newly formed Electronic Computer Project (ECP) at the Institute for Advanced Study. His account of the relationship between internal and external sorting suggested that both would be controlled by the central computer, the only difference being the source of the data to be sorted.

The binary meshing of two monotone sequences into a single monotone sequence can be done by the comparison operation in the computer. If the lengths of sequences and the amount of related information exceed the memory capacity of the computer then the tapes have to be used for storage, so that the limiting speed of the meshing operation is the speed at which information can be recorded or read on the tapes. [...] The memory is equivalent only to about two stacks of 500 cards each, hence larger operations will certainly require streaming into the tape and back. (IAS 1945, #3, 9)

2.4 Algorithms for internal sorting

Both Arthur Burks (1998) and Donald Knuth (1973) have credited von Neumann with inventing the internal sorting algorithm now known as 'mergesort'. In a draft for an unpublished book, Burks located the moment of invention at an April 1945 meeting of the EDVACteam.

The input would consist of a series of records, each in turn consisting of a key number and a series of associated data numbers. The task was to produce an alphabetized series of records. Johnny sketched the following procedure: begin by treating each record as a unit sequence of records and compare the keys of the records pairwise to form ordered sequences of two

records, then form sequences of four records in a similar manner, form sequences of eight records, . . . , until there is a single ordered sequence of all the records. (Burks 1998)

However, this eureka moment does not appear in the notes that Burks took of the meeting at the time, and von Neumann's manuscript gives no details of the sorting algorithm that would use the meshing routine he had coded. Von Neumann told Goldstine that he had written a sorting routine, but Burks himself came to believe that the meshing routine was in fact the only code that von Neumann had written.

In September, Eckert and Mauchly (1945, 54) commented that it was 'relatively simple to provide the controls and instruction code' to perform an internal sort, but gave no details. However, von Neumann mentioned the mergesort procedure briefly at the ECP meeting in November:

General sorting can be accomplished by iteration of the meshing process in $n \log_2 n$ steps, by meshing $\frac{n}{2}$ pairs then $\frac{n}{4}$ pairs with 2 elements each, etc. (IAS 1945, #3, 9)

Mauchly gave a lengthier description of the procedure in April 1946, in a manuscript that served as the basis for a lecture he gave at the Moore School in July. He began with a systematic classification of sorting procedures, largely dropped from the lecture, and identified three 'methods for random sorting', by which he meant those which made no assumptions about the initial ordering of the data to be sorted. Two of these were based on the digit-by-digit technique used on punched card sorters, but the third was based on 'collation', defined as follows:

By the process of collation, items from two different sequences are interspersed in accordance with some rule. Thus, two monotonic sequences can be blended into one monotonic sequence. (Mauchly 1946a)

He spelled out the details of the 'collation method of sorting' only when he turned to a description of the relative efficiency of the different techniques.

It is clear that if $T/2$ items were already arranged in proper order on one tape or in registers, and the $T/2$ remaining items were ordered on a second tape (or in registers), then by [repeatedly comparing two data items and moving one to a new position] a single sequence could be prepared by collating these two given sequences, so that the desired final order was attained. But $T/2$ items could have been prepared from two sequences of $T/4$ items each, and so forth.

By this method, then, it is possible to start with a number of very small sequences and collate these until all the data has been combined into one large sequence. The total number of operations required [...] can be estimated as of the order of $T \log_2 T$. (Mauchly 1946a)

Mauchly presented sorting as an adaptation and extension of familiar punched-card techniques to the new environment of high-speed automatic computing. Von Neumann's meshing routine implemented the procedure illustrated in Figure 2.1, and the added ingredient necessary to arrive at the mergesort algorithm was the insight that repeated application of this process on longer and longer sequences could provide a way of sorting a sequence of data. Mauchly did not discuss the details of how this informally stated procedure would be coded, however.

In the lecture delivered in July, however, Mauchly (1946b) framed his description of mergesort rather differently. He explicitly discussed what 'efficiency' might

mean when applied to algorithms, and described an insertion sort procedure. He then introduced mergesort as a more efficient alternative to an insertion sort, rather downplaying its organic connection with the earlier world of mechanical collators.

It is striking that von Neumann and Mauchly explain meshing in as much detail as its application to sorting, and highlight the use of the ‘comparison operator’ to discriminate between two data items. The automation of conditional control was a new and untested technology in 1945, and although the meshing procedure might appear trivial to us, we should not underestimate its novelty. As Knuth (1973, 384) suggested, implementing these non-numerical procedures did provide reassurance that, as von Neumann (1945e) put it, ‘the present principles for the logical controls are sound’.

It is possible that further archival material will surface to put more flesh on the bare bones of this story, but while it is certainly plausible that von Neumann coded the mergesort procedure in the spring of 1945 using the code described in the *First Draft*, the only EDVAC code that survives from 1945 appears to be that contained in the meshing routine manuscript. In 1946, Goldstine and von Neumann produced versions of the meshing and sorting routines using the IAS machine code, and the latter seems to be the earliest surviving implementation of mergesort. Both routines were further modified and published in 1948, in the last of the *Planning and Coding* reports.

The following two chapters describe the EDVAC code and von Neumann’s 1945 meshing routine in detail, and the later versions of the meshing and sorting routines are discussed in Chapter 5.

Chapter 3

EDVAC and its codes

The *First Draft* is celebrated as the first text to clearly articulate the key principles of the architecture of the modern computer. However, its foundational role in modern programming is equally significant. Von Neumann's proposals, described by Haigh et al. (2014) as constituting a *modern code paradigm*, were fundamental to the style of programming adopted as the new machines became operational.

An integral part of von Neumann's work was to test his ideas by writing code to solve actual problems. The manuscript describing the meshing routine contains the only example of this coding known to survive and provides a unique insight into the way von Neumann used the EDVAC code and the style of programming that he envisaged. The manuscript uses a different code from the one presented in the *First Draft*, however, a development linked to changes in EDVAC's hardware design. This chapter describes the evolution of EDVAC and its code during 1945 and summarizes the code used to write the meshing routine.

3.1 The Bell Labs relay computer

On 26 January, 1944, BRL hosted a meeting with representatives of Bell Telephone Laboratories, hoping to obtain assistance with a number of computing-related projects. Bell were developing computing machines employing standard telephone relays as the basic switching element. These included the Relay Interpolator, which became operational in late 1943, and a 'ballistic computer system' planned for the US Navy's ordnance department.

Bell's George Stibitz (1944) noted after the meeting that BRL were interested in 'the possibility of using relay computer equipment in connection with the general ballistic problem'. As an example of a particularly time-consuming problem, Leland Cunningham had described the preparation of plane-to-plane firing tables requiring the calculation of between 3,000 and 20,000 trajectories, a task that consumed up to three months of computation time rather than the two-week timescale that would be needed to keep up with the rapid development of weapons technology.

BRL thus seemed to be envisaging a machine with the same range of application as ENIAC. Whether this reflected the anticipated volume of work or a deeper anxiety about the timely completion of the ENIAC project is not recorded, but Goldstine was careful to assign the proposed relay machine a subaltern role in relation to ENIAC:

Since the needs of the Ballistic Research Laboratory are so great for reliable and efficient computing equipment, it was felt desirable to canvass with Drs. Stibitz and Kane the possibility of building relay-type devices for performing numerical calculations which might not be feasible to carry out on the new electronic machine because of a conflict of priorities in the Laboratory or because the computations were not quite suited to the characteristics of this machine. (Goldstine 1944)

Stibitz imagined a machine that was a larger version of the ballistic computer being built for the Navy, estimating that it would contain about twenty times as much equipment, and he reported that Bell were prepared to begin an investigation into the design of a machine that would be suitable for the needs of both services. This machine became known as the ‘Bell relay computer’ or ‘Model V’, and in the end two copies were built, one for the National Advisory Committee on Aeronautics and the other for BRL. Despite Stibitz’s optimism that a machine could be completed by the end of 1944, however, the BRL machine was not delivered until August 1947, just as ENIAC was being installed in its permanent home in Aberdeen.

The January meeting discussed some high-level design principles for the new machine, Stibitz noting that ‘what is required is a system consisting of fundamental units which will store numbers, multiply, divide, read data, hunt functions and print the results in usable form’. Bell set to work with some eagerness. Engineer Samuel Williams produced an outline description of a ‘calculating system’ by February 11 and a more comprehensive report by the end of March. Noting Stibitz’s comment that a machine developed to solve the immediate requirements of the Army and Navy would easily be applicable to a more general range of problems, Williams summarized the new design as follows:

Regardless of the kind of problem to be solved, it appears that equipment capable of receiving a number from a tape and transferring that number to a register where it may be held for future use; of transferring numbers from registers to a calculator where the desired calculation is effected; of holding the result of a calculation within the calculator when the result is to be used for the next calculation or transferring the result to a register or both; and of transferring a number from calculators or registers or both to a printer register for printing; all of these functions being orderly controlled by a tape, will provide the flexibility required for a universal system. (Williams 1944, 1)

The Bell proposal described a flexible arrangement in which problem-specific configurations of the fundamental units would be selected and operated by control circuits known as ‘computers’, and the report included a diagram showing a system containing three computers, each comprising a number of registers, a calculator, and a number of special purpose control units. This design differed from Mark I and ENIAC in completely separating the storage of numbers from their use in calculation, with the consequence that the transfer of numbers between registers and calculator became a characteristic feature of the machine’s operation.

At the beginning of August, von Neumann wrote to Los Alamos director Robert Oppenheimer, reporting on the availability of ‘calculating machines’. He had met Stibitz and Williams, and his letter included a comprehensive summary of the Bell proposals. In particular, he noted that:

The operations \pm are not done as in I.B.M’s, as part of transfers, but in a separate computing device, the “adder”, just like multiplication, division, or square rooting.

An instruction on the control-tape therefore looks like this: “Take the contents of register a, also the contents of register b, add (or subtract, or multiply, etc.), and put the result into register c.” At the same time it must be specified, whether the content of a (or b) must be held or cleared after this step. (von Neumann 1944)

In fact, rather than a single control tape, the machine had multiple tapes. Williams proposed to divide the programming information for a problem between a ‘routine tape’ and a ‘problem tape’ which would contain problem-specific information such as numerical parameters. In contrast, routine tapes would control the operations to be carried out, but make no reference to specific numerical information. Williams characterized this by saying that a ‘routine tape contains the algebraic formulas required for solving the problem’ and was ‘not specific to any given problem but may be used for problems of the same nature. These tapes may be filed and properly catalogued so that the operator may obtain the tapes required for the particular problem from such a file’ (Williams 1944, 2-4).

3.2 The first EDVAC

At the most general level, the *First Draft* described EDVAC as a collection of five functionally distinct units that von Neumann termed ‘organs’. The central part of the machine consisted of the memory \mathcal{M} , a dedicated unit \mathcal{A} to carry out all the arithmetic operations, and a central control unit \mathcal{C} . Input and output organs \mathcal{I} and \mathcal{O} connected the central units of the machine with an external recording medium \mathcal{R} .¹ These units and the communicative relationships between them clearly echo the structure that Williams had outlined for the Bell machine.

The two proposals differed significantly in their approach to storage, however. The Bell machine used distinct media for different kinds of information: special-purpose registers built out of relays held the numbers being calculated, but tabulated numerical information was stored on paper tape, and for commonly used functions ‘permanent tables [...] built into relays or crossbar switches’ (Williams 1944, 2) were proposed as an alternative. The machine’s instructions were also stored on paper tape.

The *First Draft* subsumed these different media in a more abstract and uniform model of memory. Von Neumann’s classification of the material that EDVAC would have to remember largely reproduced that of the Bell machine: numerical results would have to be stored, as would the instructions governing particular problems,

¹ In the *First Draft*, \mathcal{C} and \mathcal{A} were denoted by the abbreviations *CC* and *CA*.

and he also noted that in many cases it would be necessary or convenient to store tabulated function values. But function was no longer to determine structure:

While it appeared, that various parts of this memory have to perform functions which differ somewhat in their nature and considerably in their purpose, it is nevertheless tempting to treat the entire memory as one organ, and to have its parts even as interchangeable as possible for the various functions enumerated above. (von Neumann 1945a, 6)

The unification of memory had a knock-on effect on control. The Bell machine had multiple control units: each tape reader had a control circuit, and in addition there was to be a ‘progress control’ circuit to exercise control over the problem and routine tape control circuits. In EDVAC, where the multiple sources of control information had been replaced by a single memory, these circuits were replaced by a single control organ \mathcal{C} responsible for sequencing the machine’s operations.

In the *First Draft*, the basic unit of memory was characterized as ‘the ability to retain the value of one binary digit’ (von Neumann 1945a, 57). This capability was to be provided by long tubes full of mercury known as *delay lines*. Units, or *bits* as they were later dubbed, were represented by acoustic pulses travelling down the length of a line. Electronic circuitry at the end of the tube detected the pulses and, under the control of \mathcal{C} , transmitted them to the arithmetic unit if necessary, recirculating either the original pulses or new data back into the other end of the tube for continued storage.

Von Neumann estimated that 32 units would enable numbers to be stored to a sufficient degree of precision, and a group of 32 units was called a *minor cycle*. As well as its physical meaning, this term was used to denote the length of time taken for 32 pulses to emerge from the end of a delay line. A reasonable capacity for a delay line seemed to be 32 minor cycles, and this grouping—a total of 1,024 binary digits—was termed a *major cycle*. The curious and rather Ptolomaic terminology of cycles was presumably chosen to evoke the repeated circulation of the pulses through the delay lines.

The organization of EDVAC’s memory and its relationship to the control organ \mathcal{C} is depicted in Figure 3.1. For identification purposes, numbers were assigned to the major and minor cycles. If major cycles are indexed by x and the minor cycles within a major cycle by y , each minor cycle in \mathcal{M} can be identified by a pair of numbers yx .²

Finding a suitable term to describe the coordinates of a minor cycle was rather problematic, perhaps because the position denoted by the y coordinate varies as a minor cycle moves through the delay line x , making familiar spatial metaphors rather misleading. In February, von Neumann (1945d) described x and y as ‘house numbers’, but immediately crossed the phrase out and replaced it with the more neutral ‘filing numbers’. Neither term appeared in the *First Draft*, however, and nor did the now-familiar terminology of memory addresses.

² In the *First Draft*, major and minor cycles were represented by μ and ρ , and $\mu\rho$ was often confusingly typed as ‘up’. Following (von Neumann 1945e), I use x , y , and the compound form yx throughout. The typewritten ‘w’ used to represent an unspecified operation symbol has been replaced by the ω used by von Neumann in handwritten manuscripts.

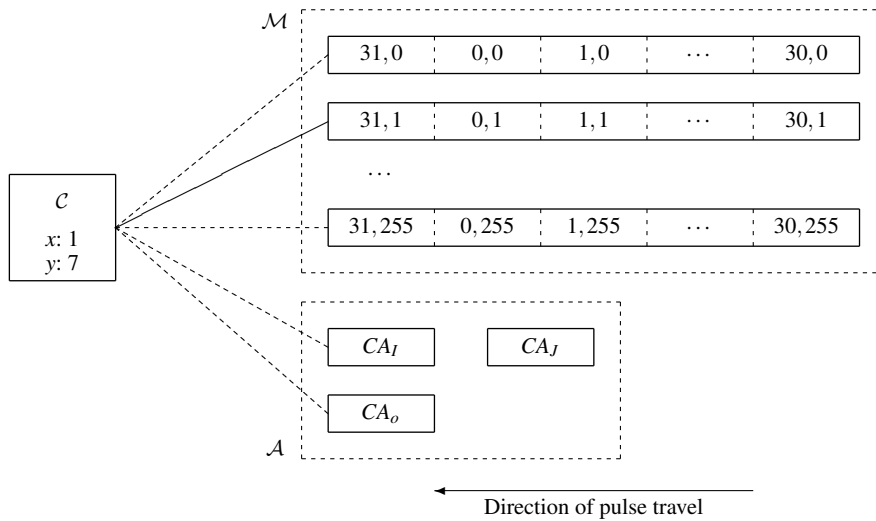


Fig. 3.1 The control and memory of the first EDVAC. \mathcal{M} contains 256 delay lines, or major cycles, each divided into 32 minor cycles. The physical location of a minor cycle would change as the pulses travelled through the delay line. The diagram shows the situation after 31 (or 63, 95, ...) minor cycles have elapsed. \mathcal{C} is ‘connected to’ the minor cycle 7, 1, but will have to wait 8 minor cycles for the code word to appear at the end of delay line 1.

The arithmetic organ \mathcal{A} contained a small amount of special-purpose memory, namely three 32-bit delay lines CA_I , CA_J and CA_O holding the two operands and result of an arithmetic operation, respectively. The control organ \mathcal{C} could store the coordinates yx of a minor cycle and, in the words of the *First Draft*, ‘connect itself’ to this minor cycle. It could only physically be connected to the major cycle x , however, and in general would have to wait for the minor cycle yx to become available.

After defining the structure of EDVAC’s memory, von Neumann turned to the provision of a ‘complete classification of [its] contents’. This would, he wrote, ‘put us into the position to formulate the code which effects the logical control of \mathcal{C} and hence of the entire device’ (von Neumann 1945a, 91).

This use of the word ‘code’ most likely derives ultimately from telegraphy, where the letters making up the messages to be transmitted were encoded as patterns of perforations punched in paper tape. Various standards were in use, such as the well-known Baudot code. Williams’ proposal for the Bell machine described the tape control units as ‘translating the tape code into the computer code’, and included a chart showing a ‘5 unit teletypewriter code’ (Williams 1944, appendix I), and similar terminology was used on the Harvard/IBM automatic calculator project. This machine, known as Mark I, read instructions from paper tape and the process of translating a human-readable version of these instructions into the machine-readable tape form was referred to as ‘coding’. Von Neumann worked with Mark I in 1944 and was no doubt familiar with this usage.

Type	First Draft	Revised Code	Description
0	$\mathcal{N}\xi^\dagger$	$\mathcal{N}\xi$	Storage for the number ξ . Interpreted by \mathcal{C} as the order to send ξ to CA_I . † Not executed following a $\omega \rightarrow f$ order.
Arithmetic orders			
1	ωh	ωh	Perform the operation ω and hold the result in CA_O .
2	$\omega \rightarrow \mathcal{A}$ $\omega h \rightarrow \mathcal{A}$	$\omega \rightarrow \mathcal{A}$ $[\omega h \rightarrow \mathcal{A}]$	Perform the operation ω and send the result to CA_I .
Combined arithmetic/substitution orders			
3	$\omega \rightarrow f$ $\omega h \rightarrow f$	$\omega \rightarrow f$ $[\omega h \rightarrow f]$	Perform the operation ω and transmit the result to the minor cycle following the current position of \mathcal{C} .
3'		$\omega \rightarrow f'$ $[\omega h \rightarrow f']$	Same as type 3, but \mathcal{C} skips the minor cycle following its current position.
4	$\omega \rightarrow yx$ $\omega h \rightarrow yx$	$\omega \rightarrow yx$ $[\omega h \rightarrow yx]$	Perform the operation ω and transmit the result to the minor cycle yx .
Extraction order			
5	$\mathcal{A} \leftarrow yx$	$\mathcal{A} \leftarrow yx^\ddagger$	Transmit the number (‡ or the yx fields of an order) in minor cycle yx to CA_I .
Transfer of control order (unconditional jump)			
6	$\mathcal{C} \leftarrow yx$	$\mathcal{C} \leftarrow yx$	Connect \mathcal{C} to minor cycle yx .

Table 3.1 A summary of the code described in the *First Draft* and the revisions to it proposed by von Neumann (1945e). In orders of types 1–4, ω is +, −, ×, ÷, i, j, s, db, or bd, and CA_O is cleared after transfer unless ωh (‘hold’) is specified. (Von Neumann did not explicitly state that the ωh variants of types 2–4 were included in the revised code.) Sending a number to CA_I has the side-effect of replacing the contents of CA_I with the original contents of CA_I .

Table 3.1 summarizes von Neumann’s first EDVAC code. In the *First Draft* he represented each minor cycle by a ‘*short symbol*, to be used in verbal or written discussions of the code [...] and when setting up problems for the device’ (von Neumann 1945a, 99), and used the word ‘code’ somewhat ambiguously to refer to the different types of minor cycle in \mathcal{M} as well as to their detailed encoding:

It is therefore our immediate task to provide a list of the orders which control the device, i.e. to describe the *code* to be used in the device, and to define the mathematical and logical meaning and the operational significance of its *code words*. (von Neumann 1945a, 85)

It is worth emphasizing that von Neumann treats numbers and orders alike as symbols that need to be encoded before they can be stored, and his codes encompass both types of data. They were distinguished by setting the least significant bit in a code word to 0 if the word contained a number and 1 if it contained an order. This does not imply a semantic difference between the two types of word, however: if \mathcal{C} found a minor cycle containing a coded number when it was expecting an order, it would interpret the minor cycle as an instruction to move that number to \mathcal{A} .

The orders in the code offer an interpretation of EDVAC's purpose as a scientific calculator. To give a simple example, if the numbers a and b were stored in the minor cycles $(12, 3)$ and $(13, 3)$, the following orders would compute $a + b$ and store the result in the minor cycle $(0, 5)$, while also holding it in CA_O for further use.

$n) \mathcal{A} \leftarrow (13, 3)$	$CA_I) \mathcal{N}b$	
$n + 1) \mathcal{A} \leftarrow (12, 3)$	$CA_I) \mathcal{N}a$	
$n + 2) +h \rightarrow (0, 5)$	$CA_J) \mathcal{N}b$	value transferred from CA_I
	$CA_O) \mathcal{N}a + b$	
	$(0, 5)) \mathcal{N}a + b$	

This example illustrates the way von Neumann laid out the code of the meshing routine. Orders are written on the left of the vertical line, the first column identifying the minor cycle in which the order in the second column is placed. In this example, the three instructions are placed in consecutive but arbitrarily located minor cycles. The annotations to the right of the vertical line list the minor cycles whose contents are changed by each order; the new value is described using the appropriate short symbol. The final column contains informal comments.

Orders of types 4 and 5 gave EDVAC the capabilities of the Bell machine that von Neumann had described to Oppenheimer, including the ability to clear or hold the result of an operation. Orders of types 1 and 2 were presumably included so that intermediate results could be reused without being moved to \mathcal{M} and back again. By default, orders would be read from successive minor cycles, mimicking the effect of reading orders from a paper tape. The jump instruction was provided in order to break away from this sequence when required. The way in which this code supported conditional branching is discussed below. A number could be moved between minor cycles in \mathcal{M} by routing it through \mathcal{A} , using the 'i' operation to move the contents of CA_I directly to CA_O . Input and output orders were not described.

The distinction between numbers and orders was significant to the working of the code. \mathcal{A} could only hold numbers, but \mathcal{M} held a mixture of numbers and orders, and the effect of transferring a number from CA_O to \mathcal{M} depended on whether the minor cycle receiving the number itself contained a number or an order, as a short comment on an unnumbered page at the end of the *First Draft* explained :

Remark: Orders ω (or ωh) $\rightarrow yx$ (or f) transfer a standard number ξ' from \mathcal{A} into a minor cycle. If this minor cycle is of the type $\mathcal{N}\xi$ (i.e. $i_0 = 0$), then it should clear its 31 digits representing ξ' , and accept the 31 digits of ξ . If it is a minor cycle ending in yx (i.e. $i_0 = 1$, order $\omega \rightarrow yx$ or $\omega h \rightarrow yx$ or $\mathcal{A} \leftarrow yx$ or $\mathcal{C} \leftarrow yx$), then it should clear only its 13 digits representing yx , and accept the last 13 digits of ξ ! ³

Shortly after writing the *First Draft*, von Neumann began to refer to the operation carried out when a code word was moved from one place to another as *substitution*. In logic, substitution is the operation of rewriting a formula by replacing some of the variables it contains with other terms. The short symbols in Table 3.1 contain a

³ This quotation has been edited slightly to be consistent with the notation used in this chapter. The apparent confusion between ξ' and ξ is in the original text.

mixture of constant and variable symbols. The variables available for substitution were ξ , x and y , and von Neumann's remark spells out that it is precisely these parts of a code word that are replaced when a number is transferred to a minor cycle.

Substitution in EDVAC's code was not purely symbolic, however, but a physical operation taking place in \mathcal{M} as a program ran. The layout of the coded number and order words was as follows:

0	ξ			\pm
	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31			
1	Code for order type and operation	y	x	0

If a number w was transferred from CA_O to a minor cycle containing a number, the 31 bits of w containing the digits and the sign of the number would replace the field ξ in the minor cycle; if on the other hand the minor cycle contained an order, bits 18–30 of w would replace the fields y and x , leaving the remainder of the minor cycle unchanged. This is presumably why the first bit of each code word encoded the distinction between numbers and orders: without this, there would be no way of distinguishing the two types of word in \mathcal{M} and performing the correct substitutions.

These rules determine how memory would be used by a program: once a program was loaded, \mathcal{M} would effectively be partitioned between minor cycles containing numbers and those containing orders. The code provided no way of changing the contents of a minor cycle from a number to an order or vice versa, and no way of changing an order to one of a different type.

To modern eyes, a striking feature of the code is the omission of an order to perform a conditional branch. Von Neumann wrote that the ability to select one of two numbers depending on the sign of a previously calculated result, was 'quite adequate to mediate the choice between any two given alternative courses of action' (von Neumann 1945a, 54). The 's' operation, which selected either the number in CA_I or CA_J depending on the sign of the number in CA_O , provided this capability and when used in conjunction with substitution could support conditional branching. For example, the following code, modelled on code from the meshing routine, has the same effect as the pseudocode instruction *if $x \geq y$ then goto α_1 else goto α_2* :

n) $\mathcal{A} \leftarrow (i', j')$	CA_I) $\mathcal{N}y$	Assume y stored in minor cycle (i', j') .
$n+1$) $\mathcal{A} \leftarrow (i, j)$	CA_I) $\mathcal{N}x$	Assume x stored in minor cycle (i, j) .
	CA_J) $\mathcal{N}y$	
$n+2$) –h	CA_O) $\mathcal{N}x-y$	The value to be tested.
$n+3$) $\mathcal{N}\alpha_2$	CA_I) $\mathcal{N}\alpha_2$	
$n+4$) $\mathcal{N}\alpha_1$	CA_I) $\mathcal{N}\alpha_1$	
	CA_J) $\mathcal{N}\alpha_2$	
$n+5$) s \rightarrow f	$n+6$) $\mathcal{C} \leftarrow \begin{matrix} \alpha_1 \\ \alpha_2 \end{matrix}$ for $x-y \begin{matrix} \geq \\ < \end{matrix} 0$, i.e. $x \begin{matrix} \geq \\ < \end{matrix} y$.	
	CA_O) $\mathcal{N}0$	
$n+6$) $\mathcal{C} \leftarrow \dots$		Unconditional jump to α_1 or α_2 .

The sign of $x - y$ can be used as a proxy for the truth value of the test $x \geq y$; the first three orders compute this value and hold it in CA_O . The next two orders, of type 0, copy the addresses α_1 and α_2 into CA_I and CA_J and the ‘s’ operator then selects one of these addresses to be substituted into the x and y fields of the following minor cycle, which contains an incomplete unconditional jump order with ellipses replacing those parts of the order that will be substituted before it is executed. This example also illustrates the ‘stacking’ convention von Neumann adopted to show alternative outcomes.

The *First Draft* did not specify what would happen if an order $\mathcal{A} \leftarrow yx$ of type 5 attempted to transfer a code word containing an order to CA_I . In his May letter to Goldstine, von Neumann specified that in this case the yx fields would be extracted from the code word and held in CA_I as a number. The letter also suggested some other changes to the code, summarized in Table 3.1. Von Neumann did not explain the reasons for these changes, but he did report that he had coded a sort routine and it is tempting to suppose that the need for the modifications became apparent during the coding process. In particular, the ability to extract the yx fields from an order and modify them would be useful when moving a sequence of minor cycles from one place in \mathcal{M} to another, as his sort problem required. However, changes to EDVAC’s memory structure meant that by the time the meshing routine manuscript came to be written, this operation was coded in a very different way.

3.3 The second EDVAC

The *First Draft* code made no allowances for the temporal properties of delay-line storage. Each minor cycle was assigned a fixed index yx , but in reality code words moved through the delay lines continuously and were only available for processing when they reached the end of a line. The code ignored this detail, with the result that a naively written program would spend a lot of time waiting for the desired minor cycles to appear. On average, an arbitrary memory access would incur a time penalty of half the transit time of the delay line.

One way of addressing this problem would be to use short delay lines holding only one word, but the economics of storage meant that it was not feasible to build a sufficiently large memory solely out of short delay lines. However, as Eckert and Mauchly noted:

A compromise is certainly possible, wherein both long tanks and short tanks are used. The long tanks can then provide most of the desired capacity, and short tanks can be used in such a way as to reduce the average waiting time. (Eckert and Mauchly 1945, 48)

The new terminology was defined as follows:

A delay line memory unit will be called a *tank*. Each pulse pattern representing a number or an order will be called a *word*. [...] A tank designed for the storage of many words will be called a *long tank*, and one which is designed to store only one word will be called a

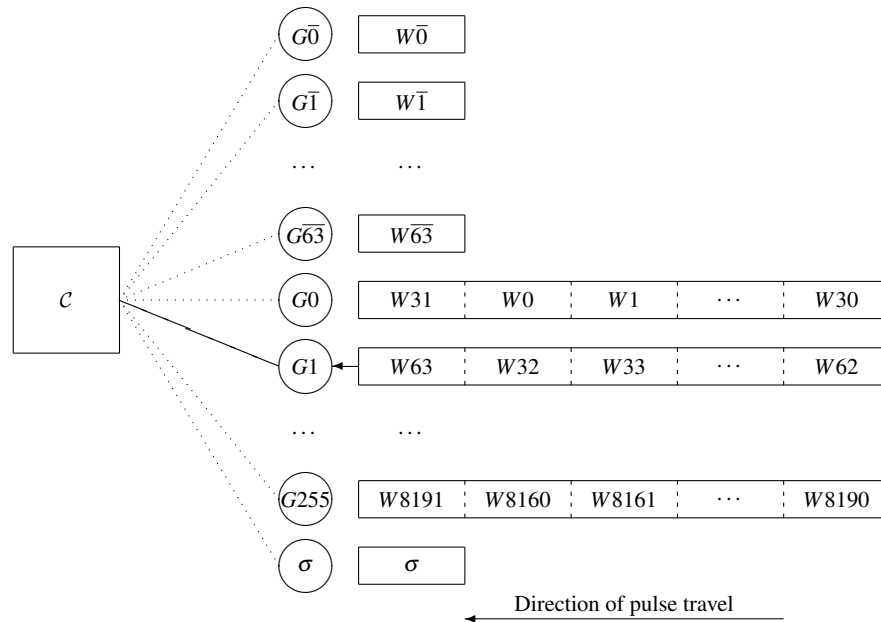


Fig. 3.2 The second EDVAC architecture as described by von Neumann, with 64 short delay lines and 256 long delay lines holding 32 words each. The diagram shows the position of the words in the delay lines at any time t where $t \bmod 32 = 31$, i.e. $t = 31, 63, 95, \dots$. Each line is accessed through a gate; in the illustration above, C is connected to long tank 1 through gate G_1 and is about to read word W_{63} .

short tank. [...] The time from the beginning of one word to the beginning of the next will be called a *minor cycle*.⁴ (Eckert and Mauchly 1945, 46)

Rather like registers in later computer designs, short tanks would allow selected data to be immediately accessible and could be used to reduce the time a program would spend waiting for numbers and orders to emerge from long tanks. However, the use of both long and short tanks meant that both the machine's code and the design of programs using it would become more complex. Von Neumann set to work, and in August reported to Haskell Curry that 'I have thought lately a good deal about the use of short delay organs—of one minor cycle length—and I think I know how to organize them' (von Neumann 1945c). He documented some of the results of this thinking, namely the memory model illustrated in Figure 3.2 and an associated code, in the text reproduced in Appendix A. The meshing program is written in a slightly modified and extended version of this code, summarized in Table 3.2. These documents give a vivid impression of von Neumann's work evaluating and revising his theoretical proposals in the light of experience gained from writing code for real-life problems.

⁴ Eckert and Mauchly credited von Neumann with introducing the terms *word* and *minor cycle*. The *First Draft* refers to *code words*, and the term *word* was formally defined by von Neumann in the text reproduced in Appendix A.

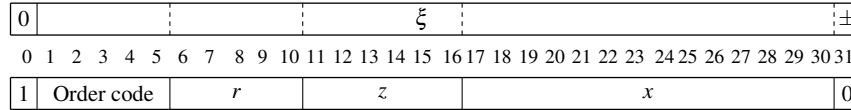
Type	Short symbol	Description
0	0	Empty word. \mathcal{C} advances to the next word.
1	$\mathcal{N}\xi$	The number ξ . \mathcal{C} advances to the next word.
Arithmetic order		
2	$\bar{z}_1 \ \omega \ \bar{z}_2$	σ receives the result of the operation $\bar{z}_1 \ \omega \ \bar{z}_2$.
Substitution orders		
3	$\sigma \rightarrow \bar{z}$	σ is substituted into $W\bar{z}$.
4	$\bar{z}_1 \rightarrow \bar{z}_2$	$W\bar{z}_1$ is substituted into $W\bar{z}_2$.
5	$\bar{z} \rightarrow x \mid r$	$W\bar{z}, \dots, W\overline{\bar{z}+r}$ are substituted into $Wx, \dots, W(x+r)$.
6	$\bar{z} \rightarrow \mid r$	$W\bar{z}, \dots, W\overline{\bar{z}+r}$ are substituted into the next $r+1$ words.
7	$x \rightarrow \bar{z} \mid r$	$Wx, \dots, W(x+r)$ are substituted into $W\bar{z}, \dots, W\overline{\bar{z}+r}$.
8	$\rightarrow \bar{z} \mid r$	The next $r+1$ words are substituted into $W\bar{z}, \dots, W\overline{\bar{z}+r}$.
Replace orders (added in the second version of the code)		
9	$x \ \dashv \bar{z} \mid r$	$Wx, \dots, W(x+r)$ replace $W\bar{z}, \dots, W\overline{\bar{z}+r}$.
10	$\dashv \bar{z} \mid r$	The next $r+1$ words replace $W\bar{z}, \dots, W\overline{\bar{z}+r}$.
Transfer of control orders (unconditional jumps)		
11	$x \rightarrow \mathcal{C}$	\mathcal{C} connects to long tank x .
12	$\bar{z} \rightarrow \mathcal{C}$	\mathcal{C} connects to short tank \bar{z} .

Table 3.2 A summary of the version of EDVAC's code used in the meshing routine. In orders of types 5–10, the suffix $\mid r$ can be omitted if $r = 0$. ‘The next $r+1$ words’ are those following the order being executed.

The complexity introduced by the short tanks is reflected in the code, much of which consists of orders to move data from one place to another, including moving multiple words in one operation. The short tank words $\bar{z} = 0, 1, \dots, 63$ inherit the numbering of the short tanks holding them. The long tank words are numbered $x = 0, 1, \dots, 8191$ and von Neumann's manuscript explains how the control organ \mathcal{C} can work out which long tank it should connect to to access a given word, and how long to wait before the word emerges. The coding of the arithmetic operations differs from the earlier code: operands are taken from two short tanks specified in the arithmetic order, and the result is made available in a special short tank σ . Despite these differences, however, the capabilities and style of the code are similar to those of the *First Draft* code.

Von Neumann now explicitly used the term *substitution* to describe the semantics of orders of types 3 to 8, which move a word w' to a location holding a word w'' . As in the earlier code, the interesting case is the partial substitution that takes place when a number word is moved to a minor cycle containing an order; in all other cases von Neumann specified that ‘ w'' is replaced by w' in its entirety’.

In this code, most order words contained three variables: a 5-bit field holding the value of r , a 6-bit field holding a short tank number z , and a 14-bit field holding a long tank number x . Number and order words were formatted as follows:



Orders of types 2 and 4 contained two short tank numbers, and must have had a different layout from the other orders.

The effect of a partial substitution of a number word into a minor cycle depended on the contents of the receiving minor cycle. Von Neumann specified that for words of types 2, 3, and 4, no substitution would take place. For words of all other types, the applicable ξ , r , z , and x fields would be replaced by the corresponding fields of the substituting number word.

Substitution in this code is more powerful than in the *First Draft*: transferring an order will overwrite whatever is in the target location, making it possible for number words be overwritten by orders. However, once a word holds an order it can never be overwritten by a number, as any attempt to transfer a number to the word will invoke the substitution mechanism, leaving the original order in place with modified variable fields.

This arrangement is too inflexible to allow effective use of the short tanks. In the meshing routine, von Neumann used these to hold a rather ad hoc mixture of numbers and orders. In a larger program, and in particular one that had multiple subroutines, short tanks would be reused in unpredictable ways. In particular, it would be impracticable if a tank that held an order in one routine could not be reused to hold a number in another.

In the meshing routine, von Neumann got round this problem by using two new orders. These appear without explanation on a one-page code summary saved with the document reproduced in Appendix A, and Eckert and Mauchly's account of the code describes them as 'replace' orders. They are denoted by \rightarrow , as opposed to the \rightarrow used for substitution orders. The replace orders simply overwrite the contents of one word with another, without regard to the contents of those words. They are only defined to move data to short tanks, suggesting that they were in fact introduced to address the problem described in the previous paragraph. In the code of the meshing routine, the replace orders are only used in the initial sequence of instructions that sets up numbers and template instructions in the short tanks.

Like its predecessor, the second EDVAC code only had unconditional jumps and conditional branches were coded by computing the target address and substituting it into an unconditional jump order. The implementation of this idea is a little more complicated than in the earlier code. The memory location holding the template unconditional jump instruction needs to be accessed by two consecutive instructions that substitute the target address into it and then transfer control to it to execute the jump. If it was stored in a long tank, this double access would incur a substantial time penalty. In the meshing routine, therefore, von Neumann placed template jump instructions in short tanks.

Consider, for example, the two-way conditional branch *if $\bar{x} \geq \bar{y}$ then goto A else goto B*. Assume that the values in the short tanks \bar{x} and \bar{y} are X and Y , and that the alternative destination addresses A and B are held in the short tanks \bar{a} and \bar{b} . The following code, adapted from section 5(i) of von Neumann's manuscript, will carry out the required conditional branch.

$n)$	$\rightarrow \bar{\Pi}$	$\bar{\Pi}) \dots \rightarrow C$
$n+1)$	$\dots \rightarrow C$	
$n+2)$	$\bar{x} - \bar{y}$	$\sigma) \mathcal{N} X - Y$ compute the number to be tested
$n+3)$	$\bar{a} s \bar{b}$	$\sigma) \mathcal{N}_B^A$ if $X - Y \leq 0$, i.e. if $X \leq Y$
$n+4)$	$\sigma \rightarrow \bar{\Pi}$	$\bar{\Pi})_B^A \rightarrow C$ if $X \leq Y$
$n+5)$	$\bar{\Pi} \rightarrow C$	
$\bar{\Pi}) A \rightarrow C$		in the case $X \geq Y$
$A)$	\dots	

The orders for the jump are stored in long tank words n to $n+5$. These remain unchanged, while the order that will actually carry out the jump is stored in a short tank, tank $\bar{\Pi}$ in this example. The template jump instruction has to be copied into this short tank before the target address can be substituted into it. In the code above, the order in word n copies word $n+1$ to short tank $\bar{\Pi}$. To ensure that this has the required effect regardless of the existing contents of $\bar{\Pi}$, von Neumann used a replace order rather than a substitution. This replaces the contents of short tank $\bar{\Pi}$ with the order $\dots \rightarrow C$ whose address field is unspecified (presumably coded as 0). Von Neumann drew a dashed box round the word $n+1$ to show that it represents data rather than an order to be immediately executed.

Order $n+2$ computes the numerical test required to differentiate the alternatives, and order $n+3$ uses the 's' operation to select one of the two addresses A or B depending on the sign of the result of this test. At this point, σ holds either A or B , and order $n+4$ substitutes this address into the appropriate field of the template jump order held in $\bar{\Pi}$. Order $n+5$ transfers control to short tank $\bar{\Pi}$ which in turn immediately transfers control to the order at A , as required. Horizontal lines indicate breaks in the default sequential execution of orders.

This example can usefully be understood as a *design pattern* (Gamma et al. 1995), an easily reusable solution to a frequently occurring problem. The problem is the provision of conditional branching in a code with no conditional jump, and the key element of the solution is the use of substitution to set up an unconditional jump instruction to the calculated target address of the conditional transfer. The solution proposed by von Neumann to this problem could be summarized as follows:

$n)$	$\rightarrow \bar{s}$	$\bar{s}) \dots \rightarrow C$
$n+1)$	$\dots \rightarrow C$	
$\dots)$	\dots	$\sigma) A$ k orders to compute the target address A
$n+k+2)$	$\sigma \rightarrow \bar{s}$	$\bar{s}) A \rightarrow C$ Copy A to \bar{s}

$n+k+3)$ $\bar{s} \rightarrow \mathcal{C}$	Jump to \bar{s}
$\bar{s})$ $A \rightarrow \mathcal{C}$	Jump to A
$A)$...	Program continues

Interestingly, this pattern is in some ways more flexible than the *if* statement: von Neumann uses it in section 5(g) of the meshing routine manuscript to code a four-way conditional branch with considerable economy and without having recourse to the analogue of nested *if* statements.

A similar technique is used in a second ‘design pattern’ in von Neumann’s code, to carry out indirect addressing. Suppose that short tank $\bar{1}$ holds the location number a of a minor cycle in a long tank, and that the number A in a must be transferred to short tank \bar{s} . The following code, again adapted from von Neumann’s manuscript, sets up the order to perform the required transfer in short tank \bar{z} .

$n)$ $\bar{1} \rightarrow \bar{z} 1$	Set up two instructions at \bar{z}
$n+1)$... $\bar{1} \rightarrow \bar{s}$	$\bar{z})$... $\bar{1} \rightarrow \bar{s}$
$n+2)$ $n+5 \rightarrow \mathcal{C}$	$\bar{z}+1)$ $n+5 \rightarrow \mathcal{C}$
$n+3)$ $\bar{1} \rightarrow \bar{z}$	$\bar{z})$ $a \rightarrow \bar{s}$ Partial substitution into the order in \bar{z}
$n+4)$ $\bar{z} \rightarrow \mathcal{C}$	
$\bar{z})$ $a \rightarrow \bar{s}$	$\bar{s})$ A
$\bar{z}+1)$ $n+5 \rightarrow \mathcal{C}$	
$n+5)$...	Program continues

3.4 Substitution

As a way of familiarizing his abstract computing machines, Turing (1936) described how their structure and functionality resembled the activity of a human carrying out a pen-and-paper calculation, and a similar analogy can help us understand how von Neumann thought of computation. Imagine the minor cycles in the memory of the EDVAC of the *First Draft* as an array of boxes outlined on a large sheet of graph paper. When a program is loaded, some of these boxes are filled with coded words. Parts of these words—the single bit that distinguishes between orders and numbers, and the operation codes of the orders—are written in ink, but the remaining bits represent numbers and are written in pencil. The part of \mathcal{C} is played by a human equipped with a pencil and an eraser. When in the course of computation a code word is transferred to a minor cycle, \mathcal{C} erases the pencilled contents of the corresponding box and replaces them by the equivalent bits of the new code word.

This image was complicated in the second iteration of the EDVAC design by the need to allow whole words to be replaced or overwritten in certain circumstances, but its basic force remains. For von Neumann, EDVAC’s control was a machine that rewrote code words, and the commonest and typical cases of this rewriting were

understand to be an application of the familiar logical operation of substitution. Memory is a syntactic space of symbols that represent numbers and orders, and computation is the process of incrementally changing the contents of that space.

The ability to modify orders as well as numbers, even in partial and constrained ways, gives modern computers much of their power and versatility, and the vital importance of substitution as a programming technique was soon recognized. The calculations that drove the automation of computing, notably the numerical solution of total differential equations, were iterative processes that repeated sequences of operations many times on a small amount of data. Eckert and Mauchly (1945, 28, 39) noted that the cost of preparing a problem ‘in a mathematical form and logical symbolism appropriate to the machine’ was ‘immensely sensitive to the way in which repetitive and iterative processes are ordered’. Machines such as ENIAC and Mark I had developed techniques of varying degrees of sophistication for repeating sequences of operations, and the EDVAC code provided the same capability.

But EDVAC was designed to address the far more challenging task of solving partial differential equations where the iterated operations had to be carried out, not on a small fixed set of variables, but on differing components of a large array of intermediate values manipulated by the program. (One of the key reasons for EDVAC’s large memory was to store this array of data.) One way to avoid having to write these orders out repeatedly would be to vary, on each iteration, the part of the order that referred to a specific variable. As Eckert and Mauchly put it:

Essentially, the machine must be made to perform ‘substitutions’. For instance, the same sequence of arithmetic processes must be carried out over and over again, but each time on a new set of numbers. To repeat such arithmetic orders each time they are to be used would be wasteful of memory capacity as well as wasteful of coding time, and is obviously unnecessary. (Eckert and Mauchly 1945, 39).

But substitution also had some rather more specific applications. One that loomed large in 1945, perhaps surprisingly to modern eyes, was the use of function tables. It was taken for granted by computer designers that automatic machines would require some analogue of the printed volumes of mathematical tables that human computers consulted to look up function values. For example, the Harvard and Bell machines read tabulated functions from tapes through which specialized units would ‘hunt’ to find a desired value.

In the *First Draft*, von Neumann assumed that even EDVAC would have to store tabular data (another reason for the large size of the machine’s internal memory), and when he was explaining the principles of the EDVAC design to the team assembled at the beginning of the IAS Electronic Computer Project he used this application to motivate the notion of substitution.

One more operation has to be provided for, the operation of substitution. Consider, for example, the incorporation of something like a 100 position log table in the memory, with a routine for biquadratic interpolation. Then a number whose logarithm is required is produced in the machine. This corresponds to an entry point in the table which will be entered by adding the entry number to the house number which corresponds to the location of the beginning of the log table, then substituting the result in a word order. This means that provision must be made for transferring a specific 12 digit portion of a 30 digit number into a house number portion of a logical order word. (IAS 1945, #2, 5)

This description brings to mind ENIAC's function tables, units which allowed function values to be set up on switches and accessed in random-access fashion. Rather than hunting through a tape, a two-digit argument was transferred to special registers in a function table and used to access the corresponding value directly. This is precisely the use of a calculated numerical value as an 'entry number' in a table, albeit one implemented by a special-purpose device instead of being identified by a 'house number' in a multi-purpose memory.

The utility of substitution in the implementation of subroutines was also quickly recognized. Eckert and Mauchly (1945, 39) observed that substitution would allow an interpolation routine to return 'to *different* parts of the problem orders after every use', and the *Preliminary Discussion* of the IAS machine discussed a mechanism by which 'classes of problems' could be coded by separating the definition of the operations to be carried out from the provision, through substitution, of the numbers to be operated on.

As an example, consider a general code for n -th order integration of m total differential equations for p steps of independent variable t , formulated in advance. Whenever a problem requiring this rule is coded for the computer the general integration sequence can be inserted into the statement of the problem along with coded instructions for telling the sequence where it will be located in memory [...]. Whenever this sequence is to be used by the computer, it will automatically substitute the correct values of m , n , p , and Δt [...] into the general sequence. A library of such general sequences will be built up [...]. When such a scheme is used, only the unique features of a problem need be coded. (Burks et al. 1946, 46)

The ability of a program to modify its own code is sometimes seen as a natural consequence of the decision to store orders in the same device as numbers. Even if this is so, the specific way in which it was understood by the EDVAC team as a form of substitution clearly draws upon existing practices in logic, and perhaps also on practical experience in automating function tables. The resulting mechanism had many applications in coding, but more general forms of substitution also appeared in other contexts in von Neumann's work on programming, as the following chapters will describe.

Chapter 4

The 1945 meshing routine

In early 1945, sequences of coded instructions controlled the operation of machines such as Bell's Relay Interpolator and the Harvard Mark I but their use was limited to the solution of familiar mathematical problems. Von Neumann's application of EDVAC's code to sorting represented a new departure. Although some processes were automated by punched card machines, sorting a deck of cards still required considerable human intervention, such as combining the sub-decks produced by a sorter into a single deck for the next stage in the process. An internal sorting routine for EDVAC would have to be entirely automatic, however. Somehow, the mixture of mechanical and manual operations involved in sorting a deck of punched cards had to be brought into EDVAC and expressed as a sequence of simple instructions.

Von Neumann took a methodical approach to this problem, making use of current ideas about program development. An ENIAC report had proposed splitting the task of developing a set-up for a new problem into three stages (Moore School 1943): the problem was first defined mathematically by specifying the necessary sequences of elementary operations, the second stage involved working out how to set up these operations on ENIAC, and the third stage comprised the actual plugging and switching required to configure the machine and run the program. This approach assumed that the problem was given in mathematical terms, typically as a set of differential equations and boundary conditions. A similar step-wise approach had been codified by the Harvard group.

Von Neumann's account of the development of the meshing routine followed a similar trajectory. His manuscript is divided into twelve numbered sections that move from a mathematical definition of sorting, through a systematic development of the instructions needed, to a detailed consideration of how those instructions would be physically arranged in EDVAC's memory prior to execution. The actual coding of the instructions into their final binary form is omitted, and the manuscript ends with some reflections on the use of the code as a subroutine and estimates of its running time. The sections are untitled, but their contents are summarized in Table 4.1.

Section	Contents
1	Defines sorting and meshing mathematically.
2	States the purpose of the manuscript.
3	Defines the top-level structure of the meshing routine.
4	Comments on the top-level structure.
5	Defines the code sequences.
6	Allocates data to short tanks.
7	Defines the initialization sequence.
8	Summarizes the complete list of instructions.
9	Enumerates the final list of instructions, starting at e .
10	Lists the substitutions required for variables p , a and e .
11	Explains how to load and run the routine and use it as a subroutine.
12	Estimates the time taken by the routine.

Table 4.1 The contents of von Neumann's manuscript.

4.1 Defining the problem

Writing to Goldstine in May, von Neumann described the sorting problem he had coded for the EDVAC of the *First Draft* as follows:

I considered a sorting problem which consists of this: Given n groups of $k+1$ numbers each: $(x_i, y_i', y_i'', \dots, y_i^{(k)})$, $i = 1, \dots, n$. Rearrange them into a new order $(x_{i^*}, y_{i^*}', y_{i^*}'', \dots, y_{i^*}^{(k)})$, the $1^*, 2^*, \dots, n^*$ being such a permutation of the $1, 2, \dots, n$, that $x_{1^*} \leq x_{2^*} \leq \dots \leq x_{n^*}$. (von Neumann 1945e)

In the manuscript this was expanded into the definition of two distinct operations, meshing and sorting, and the relationship between them. The groups of numbers being sorted were now called *complexes*.

A $p+1$ -complex: $X^{(p)} = (x^0; x^1, \dots, x^p)$ consists of the *main number*: x^0 , and the *satellites*: x^1, \dots, x^p . Throughout what follows $p = 1, 2, \dots$ will be fixed. A complex $X^{(p)}$ *precedes* a complex $Y^{(p)}$: $X^{(p)} \leq Y^{(p)}$, if their main numbers are in this order: $x^0 \leq y^0$.

These definitions can be read as a mathematical model of a punched card. A typical IBM card had 80 columns, each holding one digit. Depending on the needs of an application, these columns would be grouped into fields holding data such as employee number, hours worked, and total pay in a payroll application. A typical sorting application might be to arrange a card deck in ascending order of employee number: in von Neumann's terminology, this would be the main number and the hours worked and total pay fields would be the satellites.

Armed with these definitions, von Neumann defined sorting as the operation of finding a 'monotone permutation' of a sequence of complexes, while meshing was defined as the operation of sorting the composite of two monotone sequences. This makes meshing subordinate to sorting, suggesting that two sorted card decks could be meshed by putting one on top of the other and then sorting the resulting larger deck. Von Neumann then stated the goal of the manuscript:

We wish to formulate code instructions for sorting and for meshing, and to see how much control-capacity they tie up and how much time they require.

This makes it clear that coding was not just a theoretical exercise, and that von Neumann also intended to explore the physical parameters of EDVAC's design. Coding the sorting problem would allow him to estimate how much memory would be required and also how fast the routine would run, and in both the May letter and section 12 of the manuscript he attempted to quantify these properties.

At this point, however, the manuscript takes an unexpected turn. Even though he had defined meshing in terms of sorting, von Neumann wrote 'it is convenient to consider meshing first and sorting afterwards', and the remainder of the manuscript described a 'natural procedure' for meshing which did not require the ability to sort, namely the mechanical process illustrated in Figure 2.1. The notion of 'convenience' that motivated this switch was later spelled out as follows:

We consider first Problem 14 [meshing], which is the simpler one, and whose solution can conveniently be used in solving Problem 15 [sorting]. (Goldstine and von Neumann, 1947-8, vol. 2.)

There is a significant transformation, in other words, between the mathematical and algorithmic representations of the problem. Although he had defined meshing in terms of sorting, von Neumann reversed this dependency in the coding and planned to use the meshing routine in the implementation of sorting. This is rather at odds with the procedures outlined in the ENIAC and Mark I reports, which assumed that the order of operations identified in the mathematical analysis of the problem would be reflected in the structure of the code.

4.2 An algorithm for meshing

Section 3 of the manuscript defines the top-level structure of the meshing routine. The 'natural procedure' for meshing two sorted sequences illustrated in Figure 2.1 is to examine the top card in each input deck, take the one with the smaller key field and add it to the output deck, and repeat this procedure until all the cards in the input decks have been transferred to the output deck.

Von Neumann modelled decks of punched cards as sequences of complexes. The input decks $X = X_0, \dots, X_{n-1}$ and $Y = Y_0, \dots, Y_{m-1}$ of lengths n and m are meshed to form an output sequence $Z = Z_0, \dots, Z_{n+m-1}$ of length $n + m$. He treated the repetitive aspect of the procedure as a kind of induction in which a variable l counts how many elements of Z have been formed at the end of each step. If n' elements of X and m' elements of Y have been transferred to the output deck at the end of a step, the variables are related by the constraint $l = n' + m'$.

Thus the progress of the procedure is tracked by the values of the three variables n' , m' and l , which also serve as indices into the sequences X , Y and Z . The inductive step is defined by a case analysis that states how these variables should be updated and defines the value of the next element of Z :

- (α) $n' < n, m' < m$:
 Determine whether $x_{n'}^0 \leq$ or $> y_{m'}^0$.
- (α_1) $x_{n'}^0 \leq y_{m'}^0$: $Z_l^{(p)} = X_{n'}^{(p)}$,
 replace l, m', n' by $l + 1, m', n' + 1$.
- (α_2) $x_{n'}^0 > y_{m'}^0$: $Z_l^{(p)} = Y_{m'}^{(p)}$,
 replace l, m', n' by $l + 1, m' + 1, n'$.
- (β) $n' < n, m' = m$:
 Same as (α_1).
- (γ) $n' = n, m' < m$:
 Same as (α_2).
- (δ) $n' = n, m' = m$:
 The process is completed.

In case (α) there are cards left on both input decks. We therefore look at the main numbers $x_{n'}^0$ and $y_{m'}^0$ of the top cards of X and Y . In case (α_1) the next card to be placed on Z comes from X and in case (α_2) it comes from Y . In case (β) deck Y is exhausted, and so the next card must come from deck X . Von Neumann treats this as equivalent to (α_1), and the converse situation in (γ) is treated as equivalent to (α_2). In case (δ) both input decks are exhausted, and so the meshing is complete.

Von Neumann did not explain the relationship between this case analysis and the preceding mathematical definitions, and nor did he argue, formally or informally, that this procedure will lead to the formation of a monotone sequence Z . The level of presentation is that of a rather explicit and detailed mathematical paper. Von Neumann presumably assumed that he had stated things in as much detail as was required to make the correctness of his argument apparent to the reader in much the same way as a conventional mathematical paper would.

Section 4 of the manuscript gives some explanatory details and comments on the case analysis, including the following brief argument for program termination. At the start of the program $l = 0$ and at the end of the program $l = n + m$. Cases (α), (β) and (γ) all increment the value of l , while case (δ) terminates the program. The program therefore terminates, von Neumann says, after $n + m + 1$ ‘steps’.

4.3 Sequence programming

The case analysis provided the structure around which von Neumann developed the code of the meshing routine. His strategy was to give a ‘set of instructions’ for each of the five cases (α_1), (α_2), (β), (γ), and (δ). These were to be coordinated by two further sets of instructions to ‘effect [the] 4-way decision between (α)–(δ)’ and the ‘2-way decision between (α_1), (α_2)’. An eighth set of instructions set up the initial conditions for the program to run.

These eight sequences of instructions, summarized in Table 4.2, are the building blocks of von Neumann’s code. They were linked together in a higher-level structure involving loops and branches. As discussed in Section 5.2, this vision of program

Name	Section	Instructions	Purpose
0	7	$1_0 - 23_0$	Initialization.
1	5(g)	$1_1 - 10_1$	The four-way branch between α , β , γ , and δ .
α	5(i)	$1_\alpha - 4_\alpha$	The two-way branch between α_1 and α_2 .
α_1	5(l)	$1_{\alpha_1} - 13_{\alpha_1}$	Case α_1 of the case analysis.
α_2	5(l)	$1_{\alpha_2} - 13_{\alpha_2}$	Case α_2 of the case analysis.
β	5(j)	$1_\beta - 2_\beta$	Case β of the case analysis.
γ	5(j)	$1_\gamma - 2_\gamma$	Case γ of the case analysis.
δ	5(j)	1_δ	Case δ of the case analysis.

Table 4.2 The code sequences of the meshing program.

structure was later expressed graphically in diagrams in which the basic sequences were represented by boxes linked together in a directed graph expressing how and when the sequences would be executed.

A similar two-level approach had been built into ENIAC's hardware. Straight-forward sequences of operations were set up by configuring and connecting the so-called 'program controls' on ENIAC's independent processing units. For a complex problem, many different sequences would be set up, of which some might need to be carried out repeatedly, while others might only be executed in certain circumstances. The task of scheduling the execution of the basic sequences was called 'sequence programming' and was one of the responsibilities of a specialized unit, the master programmer (Moore School 1944, IV-40).

The master programmer could be set up to initiate a number of sequences one after the other, with each sequence being repeated as many times as necessary. It was also possible to set up nested loops, where a sequence of possibly repeated sequences was itself repeatedly invoked. By the end of 1944, a technique had been devised to transfer numeric information, such as the sign of a number, to the master programmer, giving ENIAC the additional capability to switch from one sequence to another, depending on results obtained in the course of the calculation.¹

This two-level model of program structure was also used on Mark I. On this machine, the basic sequences were expressed as coded instructions on paper tape instead of being physically set up on program controls, and sequence programming was expressed in completely different ways. Simple loops were created by gluing the ends of a tape together, literally making a loop, and the machine's human operators were given detailed operating instructions for each program telling them when to switch tapes or terminate a computation. In effect, ENIAC automated some of the manual tasks associated with the use of Mark I, but did not change its underlying model of program structure.

Von Neumann's use of the case analysis and its subsequent transformation into sequences of orders can be understood as a translation of the sequence programming

¹ The full story of ENIAC programming is rather more complicated than this, but this captures the basic model that had evolved by the end of 1944. See (Haigh et al. 2016) for fuller details.

approach into the new environment of EDVAC programming. His analysis of the problem drew a sharp distinction between the basic sequences of operations and the structure, expressed in the case analysis, that determined when they would be invoked. The sequences (α_1) , (α_2) , (β) , (γ) , and (δ) are ‘programmed’ by the case analysis and the implicit main loop that causes it to be repeated until the input decks are exhausted.

What is novel about von Neumann’s approach is that, thanks to the properties of the EDVAC code, the sequence programming could also be expressed as sequences of instructions written in the very same code as the basic sequences themselves. As well as the five basic sequences, von Neumann described sequences to control the conditional branching between (α) , (β) , (γ) , and (δ) and between (α_1) and (α_2) . The effect is that the two levels of control visible in the earlier machines collapse into one. While Mark I had tape control mechanisms and human operators and ENIAC had linked program controls and the master programmer, EDVAC had only a single control mechanism that interpreted coded instructions. Despite this technological innovation, however, it is striking that the logical organization of von Neumann’s routine is so evocative of the techniques used on the earlier machines.

4.4 Coding with variables

Once the basic sequences required for the meshing routine had been identified, von Neumann’s next task was to devise a list of instructions for each sequence. These were presented in section 5 of the manuscript using the symbolic form of the code, extended in a very natural way to make use of variables in the instructions.

As explained in Chapter 3, the code defined the possible contents of the minor cycles in EDVAC’s memory, and the symbolic form of the code simply provided a convenient shorthand for representing the code words in \mathcal{M} . For example, the bit pattern

1001110000101100000010111011110

might, depending on the coding used for order types, encode the order $751 \rightarrow \overline{24} \mid 1$, of type 7, while the bit pattern

000000000000000000000000000010

encodes the number $\mathcal{N}1_{(-30)}$, the suffix indicating multiplication by 2^{-30} . As the binary point was assumed to lie before the most significant bit of a number word, only numbers in the range $-1 \leq x < 1$ could be stored. Location numbers and other integers were held in the least significant bits, however, meaning that an integer n was stored as the value $n \times 2^{-30}$ (the rightmost bit held the sign of the number).²

² Von Neumann (1945e) stipulated that numbers should be written with their most significant bits on the right, to reflect the ‘chronological order’ in which they emerged from the delay lines ready for sequential processing by \mathcal{A} . He did not mention this point in the description of the second EDVAC code, and for simplicity it is ignored here.

Strictly speaking, then, the code can only represent constants, the actual values of the location numbers and numeric data appearing in a program. However, when a program is being developed it is neither feasible nor desirable to allocate fixed memory locations to instructions or data. When coding the instruction sequences, therefore, von Neumann implicitly extended the symbolic form of EDVAC's code to allow the use of variables in orders.

These variables were of two kinds, corresponding to the distinction between the two kinds of numbers appearing in the code: numbers having some significance to the mathematical statement of the problem on the one hand, and location numbers on the other. The mathematical variables were of two types. Some, described by von Neumann as 'the general data of the problem', were basically parameters: the length p of the complexes, the location numbers of the long tank words b , c and d where the sequences X , Y and Z began, and the lengths n and m of the sequences X and Y . The second group of mathematical variables represented numbers generated or used in the course of the computation: the number of positions n' and m' so far examined in the sequences X and Y (from which $l = n' + m'$ could be obtained), and the main numbers $x_{n'}^0$ and $y_{m'}^0$ of the two complexes currently being compared.

As well as the mathematical variables, von Neumann defined symbolic names for location numbers; for brevity, I will refer to these as 'address variables'. The address variables were written as long or short tank addresses with a suffix identifying a group of variables that were defined together; typical examples are 1_α , 1_δ , 1_0 , or $\overline{5}_\alpha$. When reading the manuscript, therefore, it is necessary to bear in mind that while $\overline{11}$, say, is a constant referring to short tank 11, $\overline{11}_1$ is an address variable, the 11th short tank defined in instruction sequence 1. Von Neumann also introduced special address variables e and a denoting the starting location of the meshing routine in memory and the location to which control should be transferred once the routine was complete.

To illustrate the use made of these variables, consider the sequence that makes the initial choice between the four branches of the case analysis. Von Neumann observed that this sequence needed ten specific pieces of information. Six of these were identified by the mathematical variables n , m , n' , m' , $x_{n'}^0$ and $y_{m'}^0$, and four address variables 1_α , 1_β , 1_γ , and 1_δ were defined to represent the first (long tank) words of the instruction sequences that control could be transferred to.

These ten numbers, along with a template jump instruction $\dots \rightarrow \mathcal{C}$, were to be held in short tanks so that they could be efficiently accessed when needed. These short tanks were identified by the address variables $\overline{1}_1 - \overline{11}_1$, and von Neumann defined their contents by writing, for example,

$$\overline{1}_1) \mathcal{N}n'_{(-30)} \text{ and } \overline{5}_1) \mathcal{N}n_{(-30)},$$

meaning that the integer values of n' and n were stored in short tanks identified by the address variables $\overline{1}_1$ and $\overline{5}_1$.

Address variables were used for the fields within an order as well as the location where the order is stored. For example, the first line of coding for this sequence,

$$1_1) \overline{1_1} - \overline{5_1} \mid \sigma) \mathcal{N}n' - n_{(-30)},$$

states that the order $\overline{1_1} - \overline{5_1}$ is stored in a long tank location identified by the address variable 1_1 , and that the result $n' - n$, obtained by subtracting the number in short tank $\overline{5_1}$ from that in short tank $\overline{1_1}$, is stored in the special register σ .

While this strategy enabled von Neumann to develop each part of the program largely independently of the others, the resulting sequences of instructions were not formally independent. The variables representing the locations of the first order in each sequence occurred in other sequences, as the target address for a jump instruction at the end of a preceding sequence. Furthermore, the variables representing the location of shared data occurred in more than one sequence.

4.5 Allocating the short tanks

Once the basic order sequences had been written in the extended code, the rest of the development process consisted largely of substituting numbers for the variables, leaving code that could be translated into binary form. Section 6 of the manuscript begins this process by dealing with the short tank address variables. The program uses $p + 25$ short tanks in total; von Neumann rationalized the use of these and assigned actual short tank numbers to the address variables as shown in Table 4.4.

The short tanks hold a mixture of numeric constants, program variables, and template orders. The use of tanks $\overline{21}$, $\overline{22}$, and $\overline{23}$ to move complexes from X or Y to Z is slightly intricate. The first five orders in sequences α_1 and α_2 substitute the address fields in $\overline{21}$ and $\overline{22}$ with the current positions in the sequences X or Y and Z , replace $\overline{23}$ with a jump order that returns control to the sixth order in the sequence, and then execute an unconditional jump to $\overline{21}$. The orders to move the complex are then executed, and the order in $\overline{23}$ returns control to the order, in α_1 or α_2 as appropriate, that follows the jump. In effect, these three tanks contain a miniature, if rather primitive, subroutine.

While tanks $\overline{1}$ to $\overline{6}$ hold the parameters of the meshing routine, tanks $\overline{7}$ to $\overline{22}$ are ‘local variables’ that need to be initialized before the routine begins, and in section 7 of the manuscript, von Neumann defined an additional code sequence to do this. The initialization of $\overline{9}$ to $\overline{22}$ is straightforward, as these tanks contain numeric constants or code templates that can simply be copied from the initialization sequence into the appropriate tanks, but the situation with $\overline{7}$ and $\overline{8}$ is a bit more complex. The initial values x_0^0 and y_0^0 of these tanks are the main numbers of the first complexes in the sequences X and Y . They are therefore found in the first words of those sequences, i.e. at locations b and c . Transferring the contents of word b to tank $\overline{7}$ and word c to tank $\overline{8}$ therefore involves indirect addressing and von Neumann coded this using the technique discussed in section 3.3.

Table 4.5 lists all eight program sequences with the short tank address variables replaced by the actual short tank numbers allocated in Table 4.4.

Short tank	Contents	Initial value	Description
$\bar{1}$	$\bar{5}_1$	n	n Length of sequence X
$\bar{2}$	$\bar{6}_1$	m	m Length of sequence Y
$\bar{3}$	$\bar{1}_3$	$b + n'(p+1)$	b Current position in X
$\bar{4}$	$\bar{2}_3$	$c + n'(p+1)$	c Current position in Y
$\bar{5}$	$\bar{3}_3$	$d + n'(p+1)$	d Current position in Z
$\bar{6}$	$\bar{4}_3$	$p+1$	$p+1$ Length of complexes
$\bar{7}$	$\bar{3}_1$	$x_{n'}^0$	x_0^0 Main number of complex n'
$\bar{8}$	$\bar{4}_1$	$y_{m'}^0$	y_0^0 Main number of complex m'
$\bar{9}$	$\bar{1}_1$	n'	0 Number of current complex in X
$\bar{10}$	$\bar{2}_1$	m'	0 Number of current complex in Y
$\bar{11}$	$\bar{7}_1$		1_α Start location of sequence α
$\bar{12}$	$\bar{8}_1$		1_β Start location of sequence β
$\bar{13}$	$\bar{9}_1$		1_γ Start location of sequence γ
$\bar{14}$	$\bar{10}_1$		1_δ Start location of sequence δ
$\bar{15}$	$\bar{11}_1$	$\dots \rightarrow C$	Template instruction for jumps
$\bar{16}$	$\bar{1}_2$		1_{α_1} Start location of sequence α_1
$\bar{17}$	$\bar{2}_2$		1_{α_2} Start location of sequence α_2
$\bar{18}$	$\bar{3}_2$		0 Constant value
$\bar{19}$	$\bar{4}_2$		-1 Constant value
$\bar{20}$	$\bar{4}_5$		1 Constant value
$\bar{21}$	$\bar{1}_5$	$\dots \rightarrow \bar{24} \mid p+2$	Template instructions to move a complex
$\bar{22}$	$\bar{2}_5$	$\bar{24} \rightarrow \dots \mid p+1$	between short and long tanks
$\bar{23}$	$\bar{3}_5$		Holds a 'return statement'
$\bar{24}$	$\bar{1}_4$		Short tanks holding the complex being moved
\dots	\dots		
$\bar{p+25}$	$(\bar{p+2})_4$		

Table 4.4 The use of EDVAC's short tanks, identified by tank number and address variable, in the meshing program. Von Neumann overlooked the tanks $\bar{12}_1$ and $\bar{13}_1$ used as temporary storage space in the instruction sequence $1_1 - 10_1$ and these were not allocated definite tank numbers.

4.6 Making a single code sequence

After coding the eight sequences, von Neumann considered how to assemble them into a complete program. To minimize the time spent waiting for instructions to emerge from the long tanks, he arranged the sequences so that, as far as possible, the logical flow of control from one to another was reflected in the physical order of the instructions in memory, coming up with the arrangement shown in Table 4.5.

The next step towards variable-free code involves replacing the long tank address variables by location numbers, thus fixing the location of the routine in memory. However, von Neumann noted that 'it is preferable to have these instructions in

1 ₀)	$\nabla \bar{9} 2$	18 ₀)	$\mathcal{N}0$	2 _{γ})	$2\alpha \rightarrow \mathcal{C}$	13 _{α_1})	$1_1 \rightarrow \mathcal{C}$
2 ₀)	$\dots \nabla \bar{7}$	19 ₀)	$\mathcal{N}-1$	1 _{α})	$\bar{8} - \bar{7}$	1 _{α_2})	$\bar{4} \rightarrow \bar{2}\bar{1}$
3 ₀)	$\dots \nabla \bar{8}$	20 ₀)	$\mathcal{N}1_{(-30)}$	2 _{α})	$\bar{16} \text{ s } \bar{17}$	2 _{α_2})	$\bar{5} \rightarrow \bar{2}\bar{2}$
4 ₀)	$8_0 \rightarrow \mathcal{C}$	21 ₀)	$\dots \rightarrow \bar{24} p+2$	3 _{α})	$\sigma \rightarrow \bar{15}$	3 _{α_2})	$\nabla \bar{23}$
5 ₀)	$\bar{3} \rightarrow \bar{9}$	22 ₀)	$\bar{24} \rightarrow \dots p+1$	4 _{α})	$\bar{15} \rightarrow \mathcal{C}$	4 _{α_2})	$6\alpha_2 \rightarrow \mathcal{C}$
6 ₀)	$\bar{4} \rightarrow \bar{10}$	23 ₀)	$1_1 \rightarrow \mathcal{C}$	1 _{α_1})	$\bar{3} \rightarrow \bar{2}\bar{1}$	5 _{α_2})	$\bar{2}\bar{1} \rightarrow \mathcal{C}$
7 ₀)	$\bar{9} \rightarrow \mathcal{C}$	1 ₁)	$\bar{9} - \bar{1}$	2 _{α_1})	$\bar{5} \rightarrow \bar{2}\bar{2}$		
		2 ₁)	$\bar{13} \text{ s } \bar{11}$	3 _{α_1})	$\nabla \bar{23}$		
		3 ₁)	$\sigma \rightarrow \bar{12}_1$	4 _{α_1})	$6\alpha_1 \rightarrow \mathcal{C}$		
		4 ₁)	$\bar{9} - \bar{1}$	5 _{α_1})	$\bar{2}\bar{1} \rightarrow \mathcal{C}$		
		5 ₁)	$\bar{14} \text{ s } \bar{12}$			6 _{α_2})	$\bar{10} + \bar{20}$
8 ₀)	$\nabla \bar{9} 13$	6 ₁)	$\sigma \rightarrow \bar{13}_1$			7 _{α_2})	$\sigma \rightarrow \bar{10}$
9 ₀)	$\mathcal{N}0$	7 ₁)	$\bar{10} - \bar{2}$			8 _{α_2})	$(p+25) \rightarrow \bar{8}$
10 ₀)	$\mathcal{N}0$	8 ₁)	$\bar{13}_1 \text{ s } \bar{12}_1$			9 _{α_2})	$\bar{4} + \bar{6}$
11 ₀)	$\mathcal{N}1\alpha_{(-30)}$	9 ₁)	$\sigma \rightarrow \bar{15}$	6 _{α_1})	$\bar{9} + \bar{20}$	10 _{α_2})	$\sigma \rightarrow \bar{4}$
12 ₀)	$\mathcal{N}1\beta_{(-30)}$	10 ₁)	$\bar{15} \rightarrow \mathcal{C}$	7 _{α_1})	$\sigma \rightarrow \bar{9}$	11 _{α_2})	$\bar{5} + \bar{6}$
13 ₀)	$\mathcal{N}1\gamma_{(-30)}$			8 _{α_1})	$(p+25) \rightarrow \bar{7}$	12 _{α_2})	$\sigma \rightarrow \bar{5}$
14 ₀)	$\mathcal{N}1\delta_{(-30)}$			9 _{α_1})	$\bar{3} + \bar{6}$	13 _{α_2})	$1_1 \rightarrow \mathcal{C}$
15 ₀)	$\dots \rightarrow \mathcal{C}$	1 _{β})	$\bar{18} - \bar{18}$	10 _{α_1})	$\sigma \rightarrow \bar{3}$	1 _{δ})	$a \rightarrow \mathcal{C}$
16 ₀)	$\mathcal{N}1\alpha_1_{(-30)}$	2 _{β})	$2\alpha \rightarrow \mathcal{C}$	11 _{α_1})	$\bar{5} + \bar{6}$		
17 ₀)	$\mathcal{N}1\alpha_2_{(-30)}$	1 _{γ})	$\bar{19} - \bar{18}$	12 _{α_1})	$\sigma \rightarrow \bar{5}$		

Table 4.5 The meshing program of section 8 of von Neumann’s manuscript. Short tank addresses have been assigned (apart from the overlooked $\bar{12}_1$ and $\bar{13}_1$) and a single sequence of 68 orders is formed by concatenating the 8 basic sequences; gaps have been left in the tabulation to facilitate comparison with Table 4.6. Von Neumann’s slip in the position of word $\bar{20}_0$ has been corrected (see Knuth 1970, 257) and the remaining variables in the code are highlighted.

such a form that they can begin anywhere, in that their first (long tank) word can be chosen freely’, thus making it possible to place the code anywhere in memory. To provide this flexibility von Neumann introduced a new address variable e denoting the location of the first instruction in the program.

At four points in the code \mathcal{C} switches to the short tanks to read a few orders. In an attempt to ensure that the next instruction is immediately available when control returns to the long tanks, von Neumann inserted blank words into the sequence so that the following instructions would be available exactly when required.³ This increased the total length of the instruction sequence to 82 words, and the variety of long tank address variables in Table 4.5 are replaced by a single sequence of address variables $e, \dots, e + 81$. The resulting code is shown in Table 4.6.

³ Knuth (1970, 258) pointed out that von Neumann’s reasoning here was fallacious. For example, the jumps in orders 5_{α_1} and 5_{α_2} invoke orders transferring sequences of words between long and short tanks. These take an unpredictable time to execute because they make what Knuth described as ‘essentially random references to long tanks’, meaning that von Neumann’s strategy of leaving four blank words is over-simplistic.

e)	$\dagger \bar{9} 2$	$e+21)$	$\mathcal{N}0$	$e+42)$	$e+44 \rightarrow \mathcal{C}$	$e+63)$	$e+27 \rightarrow \mathcal{C}$
$e+1)$	$\dots \dagger \bar{7}$	$e+22)$	$\mathcal{N}-1$	$e+43)$	$\bar{8} - \bar{7}$	$e+64)$	$\bar{4} \rightarrow \bar{2}\bar{1}$
$e+2)$	$\dots \dagger \bar{8}$	$e+23)$	$\mathcal{N}1_{(-30)}$	$e+44)$	$\bar{1}\bar{6} \text{ s } \bar{1}\bar{7}$	$e+65)$	$\bar{5} \rightarrow \bar{2}\bar{2}$
$e+3)$	$e+11 \rightarrow \mathcal{C}$	$e+24)$	$\dots \rightarrow \bar{2}\bar{4} p+2$	$e+45)$	$\sigma \rightarrow \bar{1}\bar{5}$	$e+66)$	$\dagger \bar{2}\bar{3}$
$e+4)$	$\bar{3} \rightarrow \bar{9}$	$e+25)$	$\bar{2}\bar{4} \rightarrow \dots p+1$	$e+46)$	$\bar{1}\bar{5} \rightarrow \mathcal{C}$	$e+67)$	$e+73 \rightarrow \mathcal{C}$
$e+5)$	$\bar{4} \rightarrow \bar{1}\bar{0}$	$e+26)$	$e+27 \rightarrow \mathcal{C}$	$e+47)$	$\bar{3} \rightarrow \bar{2}\bar{1}$	$e+68)$	$\bar{2}\bar{1} \rightarrow \mathcal{C}$
$e+6)$	$\bar{9} \rightarrow \mathcal{C}$	$e+27)$	$\bar{9} - \bar{1}$	$e+48)$	$\bar{5} \rightarrow \bar{2}\bar{2}$	$e+69)$	
$e+7)$		$e+28)$	$\bar{1}\bar{3} \text{ s } \bar{1}\bar{1}$	$e+49)$	$\dagger \bar{2}\bar{3}$	$e+70)$	
$e+8)$		$e+29)$	$\sigma \rightarrow \bar{1}\bar{2}_1$	$e+50)$	$e+56 \rightarrow \mathcal{C}$	$e+71)$	
$e+9)$		$e+30)$	$\bar{9} - \bar{1}$	$e+51)$	$\bar{2}\bar{1} \rightarrow \mathcal{C}$	$e+72)$	
$e+10)$		$e+31)$	$\bar{1}\bar{4} \text{ s } \bar{1}\bar{2}$	$e+52)$		$e+73)$	$\bar{1}\bar{0} + \bar{2}\bar{0}$
$e+11)$	$\dagger \bar{9} 13$	$e+32)$	$\sigma \rightarrow \bar{1}\bar{3}_1$	$e+53)$		$e+74)$	$\sigma \rightarrow \bar{1}\bar{0}$
$e+12)$	$\mathcal{N}0$	$e+33)$	$\bar{1}\bar{0} - \bar{2}$	$e+54)$		$e+75)$	$\bar{p} + \bar{2}\bar{5} \rightarrow \bar{8}$
$e+13)$	$\mathcal{N}0$	$e+34)$	$\bar{1}\bar{3}_1 \text{ s } \bar{1}\bar{2}_1$	$e+55)$		$e+76)$	$\bar{4} + \bar{6}$
$e+14)$	$\mathcal{N}e+43_{(-30)}$	$e+35)$	$\sigma \rightarrow \bar{1}\bar{5}$	$e+56)$	$\bar{9} + \bar{2}\bar{0}$	$e+77)$	$\sigma \rightarrow \bar{4}$
$e+15)$	$\mathcal{N}e+39_{(-30)}$	$e+36)$	$\bar{1}\bar{5} \rightarrow \mathcal{C}$	$e+57)$	$\sigma \rightarrow \bar{9}$	$e+78)$	$\bar{5} + \bar{6}$
$e+16)$	$\mathcal{N}e+41_{(-30)}$	$e+37)$		$e+58)$	$\bar{p} + \bar{2}\bar{5} \rightarrow \bar{7}$	$e+79)$	$\sigma \rightarrow \bar{5}$
$e+17)$	$\mathcal{N}e+81_{(-30)}$	$e+38)$		$e+59)$	$\bar{3} + \bar{6}$	$e+80)$	$e+27 \rightarrow \mathcal{C}$
$e+18)$	$\dots \rightarrow \mathcal{C}$	$e+39)$	$\bar{1}\bar{8} - \bar{1}\bar{8}$	$e+60)$	$\sigma \rightarrow \bar{3}$	$e+81)$	$a \rightarrow \mathcal{C}$
$e+19)$	$\mathcal{N}e+47_{(-30)}$	$e+40)$	$e+44 \rightarrow \mathcal{C}$	$e+61)$	$\bar{5} + \bar{6}$		
$e+20)$	$\mathcal{N}e+64_{(-30)}$	$e+41)$	$\bar{1}\bar{9} - \bar{1}\bar{8}$	$e+62)$	$\sigma \rightarrow \bar{5}$		

Table 4.6 The meshing routine as developed in section 10 of von Neumann's manuscript, starting at long tank location e . It still contains occurrences of the variables p , a , and e (highlighted).

4.7 Loading and calling the meshing routine

The code in Table 4.6 still contains the variables p , e and a , and the location of the routine in memory is given relative to the variable e . Before the routine can be executed, the variables must be removed, leaving code that can be converted to binary form and placed in memory. Von Neumann's treatment of this issue in the manuscript is rather terse, however.

Once the routine was loaded into memory, some substitutions would be needed to prepare it for use. As shown in Table 4.4, the short tanks $\bar{1}$ to $\bar{6}$ hold the parameters of the routine and they would have to be substituted with the actual parameters before calling the routine; similarly, the variable a in long tank $e+81$ would have to be substituted by the required return address. In addition to these seven words, however, von Neumann stated that the six words in locations $e+14, \dots, e+17, e+19$ and $e+20$ containing e , and the two words in locations $e+23$ and $e+24$ containing p , would require substitution.

Von Neumann named this group of substitutions S_{15} , and observed that they would not all necessarily have to be carried out at the same time. The substitutions for e would only have to be carried out once, when the routine's location in memory was fixed, but substituting the parameters in the short tanks and the return location

a would typically have to be done each time the routine was called. In the code, the nine long tank words appearing in S_{15} were to be replaced by ‘blanks’, giving rise to a sequence of words called G_{82} .

Von Neumann wrote that G_{82} could be placed in the long tanks, but replacing the nine long tank words dealt with by S_{15} by blanks does not result in variable-free code. There are ten other words in Table 4.6 containing variables, and the manuscript does not explain what should be done with them.

The eight orders in locations $e + 3$, $e + 26$, $e + 40$, $e + 42$, $e + 50$, $e + 63$, $e + 67$ and $e + 80$ contain the variable e . These orders are all jumps to locations within the routine. In the order $e + 11 \rightarrow C$ in location $e + 3$, for example, 11 represents the offset within the routine of the target location of the jump. As discussed in Section 6.3, Goldstine and von Neumann later described how the code of a routine would be automatically adjusted by a preparatory routine before the S_{15} substitutions were applied. Once the routine was placed in memory and the value of e known, the preparatory routine would convert offsets like 11 to the correct address, $e + 11$. It is likely that von Neumann had some similar mechanism in mind as he wrote the meshing manuscript, and hence that the variable e in these eight orders should be replaced by 0 when forming G_{82} .

The two orders in locations $e + 58$ and $e + 75$ contain the variable p , representing the length of the complexes being meshed, and would not have been dealt with by the preparatory routine. Like the orders in $e + 23$ and $e + 24$, they would need to be altered if the value of p changed, and it seems a simple oversight on von Neumann’s part that these two words were not included in the substitutions of S_{15} .

Loading G_{82} into memory amounts to choosing a value for the variable e . Table 4.7 shows the sequence G_{82} in the case where e is assigned the value 100. Aside from a couple of oversights, applying the S_{15} substitutions to this code would produce a complete and executable routine.

It would be possible to run the meshing routine as a stand-alone program, for testing purposes. This could be done by writing orders which carried out the S_{15} substitutions and then transferred control to e . In general, however, the meshing routine would form part of a larger program: in particular, von Neumann planned to use it as an integral component of the sorting routine.

Von Neumann described this more general case in section 11 of the manuscript, explaining that the orders effecting S_{15} could be part of a ‘main routine’ of a program that used the meshing routine, which would itself be a ‘sub routine’. A subroutine could be called many times in a main routine, but before the second and subsequent calls only those parts of S_{15} that differed would need to be repeated. If the meshing routine was used as a subroutine in a mergesort program, for example, p and e would remain constant throughout and the orders executed before invoking the meshing routine would have only to set up the short tanks $\bar{1}$ to $\bar{5}$, and perhaps the return address a in location 181.

100)	$\uparrow \bar{9} 2$	121)	$\mathcal{N}0$	142)	$44 \rightarrow \mathcal{C}$	163)	$27 \rightarrow \mathcal{C}$
101)	$\dots \uparrow \bar{7}$	122)	$\mathcal{N}-1$	143)	$\bar{8} - \bar{7}$	164)	$\bar{4} \rightarrow \bar{2}\bar{1}$
102)	$\dots \uparrow \bar{8}$	123)	$\mathcal{N}1_{(-30)}$	144)	$\bar{1}\bar{6} \text{ s } \bar{1}\bar{7}$	165)	$\bar{5} \rightarrow \bar{2}\bar{2}$
103)	$11 \rightarrow \mathcal{C}$	124)	0	145)	$\sigma \rightarrow \bar{1}\bar{5}$	166)	$\uparrow \bar{2}\bar{3}$
104)	$\bar{3} \rightarrow \bar{9}$	125)	0	146)	$\bar{1}\bar{5} \rightarrow \mathcal{C}$	167)	$73 \rightarrow \mathcal{C}$
105)	$\bar{4} \rightarrow \bar{1}\bar{0}$	126)	$27 \rightarrow \mathcal{C}$	147)	$\bar{3} \rightarrow \bar{2}\bar{1}$	168)	$\bar{2}\bar{1} \rightarrow \mathcal{C}$
106)	$\bar{9} \rightarrow \mathcal{C}$	127)	$\bar{9} - \bar{1}$	148)	$\bar{5} \rightarrow \bar{2}\bar{2}$	169)	
107)		128)	$\bar{1}\bar{3} \text{ s } \bar{1}\bar{1}$	149)	$\uparrow \bar{2}\bar{3}$	170)	
108)		129)	$\sigma \rightarrow \bar{1}\bar{2}_1$	150)	$56 \rightarrow \mathcal{C}$	171)	
109)		130)	$\bar{9} - \bar{1}$	151)	$\bar{2}\bar{1} \rightarrow \mathcal{C}$	172)	
110)		131)	$\bar{1}\bar{4} \text{ s } \bar{1}\bar{2}$	152)		173)	$\bar{1}\bar{0} + \bar{2}\bar{0}$
111)	$\uparrow \bar{9} 13$	132)	$\sigma \rightarrow \bar{1}\bar{3}_1$	153)		174)	$\sigma \rightarrow \bar{1}\bar{0}$
112)	$\mathcal{N}0$	133)	$\bar{1}\bar{0} - \bar{2}$	154)		175)	$\bar{p} + \bar{2}\bar{5} \rightarrow \bar{8}$
113)	$\mathcal{N}0$	134)	$\bar{1}\bar{3}_1 \text{ s } \bar{1}\bar{2}_1$	155)		176)	$\bar{4} + \bar{6}$
114)	0	135)	$\sigma \rightarrow \bar{1}\bar{5}$	156)	$\bar{9} + \bar{2}\bar{0}$	177)	$\sigma \rightarrow \bar{4}$
115)	0	136)	$\bar{1}\bar{5} \rightarrow \mathcal{C}$	157)	$\sigma \rightarrow \bar{9}$	178)	$\bar{5} + \bar{6}$
116)	0	137)		158)	$\bar{p} + \bar{2}\bar{5} \rightarrow \bar{7}$	179)	$\sigma \rightarrow \bar{5}$
117)	0	138)		159)	$\bar{3} + \bar{6}$	180)	$27 \rightarrow \mathcal{C}$
118)	$\dots \rightarrow \mathcal{C}$	139)	$\bar{1}\bar{8} - \bar{1}\bar{8}$	160)	$\sigma \rightarrow \bar{3}$	181)	0
119)	0	140)	$44 \rightarrow \mathcal{C}$	161)	$\bar{5} + \bar{6}$		
120)	0	141)	$\bar{1}\bar{9} - \bar{1}\bar{8}$	162)	$\sigma \rightarrow \bar{5}$		

Table 4.7 The sequence of words G_{82} formed by assigning a value (here 100) to e and thus fixing the position of the routine in memory. In principle, this is the variable-free code whose binary equivalent would be placed in the long tanks, though as well as the short tank variables $\bar{1}\bar{2}_1$ and $\bar{1}\bar{3}_1$ von Neumann appears to have overlooked the variable p in orders 158 and 175. Before running the code, a preparatory routine would adjust the targets of the jump instructions, and the substitutions S_{15} would initialize the nine blank words and the six short tanks holding the routine's parameters.

4.8 Process overview

Von Neumann's development of the meshing routine can best be understood as the series of transformations between different symbolic representations of the program summarized in Table 4.8. Conventional mathematical notation was used to define the problem and the criteria for an acceptable solution, and then a case analysis, also familiar from informal mathematics, described the overall structure of the program. This articulated a framework that described how the basic sequences of instructions would be linked using loops and conditional branches. The orders were first written in an extended version of EDVAC's code allowing the use of variables of various types. In a rather involved process, these were then reduced to the variable-free form ready for translation into binary code. This last stage was not described in the manuscript, but von Neumann (1945e) had earlier described a special-purpose key punch that would ease the process of transforming code symbols into paper tape perforations ready for loading into \mathcal{M} .

Step	Transformation	Resulting form of program
1 (sect. 1)	Give a mathematical formulation of the problem.	The mathematical definitions of sorting and meshing.
2 (sect. 3)	Identify the basic sequences and their interaction.	The case analysis.
3 (sect. 5)	Write extended code for each basic sequence.	68 coded instructions and short tanks $\bar{1}_1$ to 4_5 .
4.1 (sect. 6)	Replace short tank address variables with actual tank numbers.	The 68 instructions of Table 4.5 and the short tanks $\bar{1}$ to $\bar{p}+25$.
4.2 (sect. 9)	Replace long tank address variables with values dependent on e .	The 82 code words of Table 4.6 and the short tanks $\bar{1}$ to $\bar{p}+25$.
4.3 (sect. 11)	Replace the variables p , a and e with blanks.	The code of Table 4.7 (G_{82}) and the short tanks $\bar{1}$ to $\bar{p}+25$.
4.4 (sect. 11)	Adjust the offset location numbers in G_{82} and perform the S_{15} substitutions.	Initialized code, to be invoked by an instruction $e \rightarrow C$.

Table 4.8 A summary of the development of the meshing routine.

The first transformations were not rigorously defined. Von Neumann seems to have assumed that it was obvious that the procedure expressed by the case analysis would in fact mesh two sequences in accordance with the mathematical definition. Similarly, no argument is given to demonstrate that the code sequences produced to implement the various parts of the analysis do what they are supposed to do.

The transformation to variable-free code is rather different, however. It proceeds in a number of steps, at each of which a certain class of variables in the code is substituted by constant values. In step 4.1, the allocation of short tanks is decided upon and the short tank variables in the instructions are rewritten. In step 4.2, a new variable e is introduced to represent the address of the first instruction, and long tank variables are replaced by expressions of the form $e + i$. Step 4.3 removed the remaining variables to leave code that, once translated into binary form, could be loaded into EDVAC's memory. The transformations to this code summarized in step 4.4 would then be carried out by the machine itself.

The development process exemplified in the manuscript formed the basis for the more complex process defined in the *Planning and Coding* reports, as described in the following chapter.

Chapter 5

Planning and coding

In the open-source community, a project is said to fork when a group of developers takes the project's codebase and starts independent work on it, often against a background of technical disagreement or personal acrimony. Something like a fork took place in the EDVAC project in late 1945 when von Neumann persuaded his home institution, the Institute for Advanced Study (IAS) in Princeton, to support a project to build an electronic computer. He soon severed his ties with the Moore School and enticed Herman Goldstine to the IAS as his lieutenant on the new project. Eckert and Mauchly also left the Moore School to form a start-up company and begin work on commercial computer development. Meanwhile, the EDVAC project continued at the Moore School with a restructured project team.

Although the IAS machine is often characterized in terms of its difference from EDVAC, at the first meetings of the new project team in November von Neumann gave a thorough overview of current thinking on the EDVAC project, covering both hardware and coding and including a brief discussion of the sorting application. In many ways, EDVAC can be thought of as a first draft of the IAS machine.

Over the next few years, Goldstine and von Neumann produced a series of more formal project reports. First was a *Preliminary Discussion of the Logical Design of an Electronic Computing Instrument*, written in collaboration with Arthur Burks, and this was followed by a three-volume treatise on the *Planning and Coding of Problems for an Electronic Computing Instrument*, issued in 1947 and 1948. The earlier report documented the high-level design of the machine and its code, and the *Planning and Coding* reports went on to outline a methodology for program development and included a compendium of typical programming examples. These reports are often taken to mark the beginning of the systematic study of computer programming.

Von Neumann's 1945 treatment of the meshing problem informed the *Planning and Coding* reports in a number of ways. It exemplified the development process that the later reports would describe more formally, and meshing and sorting problems featured as one of the more complex examples of coding. Even the discussion of subroutines in the final report was prefigured in von Neumann's earlier comments about the reuse of the meshing routine.

5.1 The IAS machine and its code

Once the Electronic Computer Project (ECP) was given the go-ahead, a series of meetings was quickly arranged between von Neumann and key representatives of the project partners, the IAS, Princeton University, and RCA, the Radio Corporation of America. The minutes of the first meeting, on 12 November, noted that ENIAC and EDVAC would be studied ‘for the education of the group’ (IAS 1945, #1), and the meetings involved substantial presentations on this material by von Neumann. The minutes therefore provide a valuable record of how thinking about EDVAC had evolved since the summer.

As in the *First Draft*, von Neumann noted that ‘the storage tube and acoustic tank seem to be the obvious candidates for inclusion as memory devices’ (IAS 1945, #1, 6), and the discussions proceeded on the basis of a memory consisting of long and short tanks and the assumption that ‘completely random reading is not in fact possible, i.e. that the procedure is split up into switching and waiting for the right number to come up’ (IAS 1945, #2, 1).

In the second meeting, on 19 November, von Neumann outlined a code ‘simply to prove that it is possible to do the job’, describing the operations it contained as being ‘sufficient for complete programming (neglecting magnetic tapes)’ (IAS 1945, #2, 5). Despite its rather provisional nature, this code is of interest as it represents a transitional phase between the EDVAC code used for the meshing routine and the code for the IAS machine described in the *Preliminary Discussion* the following year.

Number and order words were still distinguished by the value of their first bit, but their format had changed significantly:

0	±:																		ξ																
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31		
1	Order code	x						y						X																					

All order words had three address fields: x and y were 6-digit short tank numbers, while X could hold either a short tank number or a (suitably disambiguated) location number for a long tank word, split between a 7-digit tank number and five digits identifying a minor cycle. To distinguish ‘the “real” numbers of the computer’ from those that merely denoted memory locations, von Neumann reverted to a metaphor that he had considered earlier, only to discard:

we have 12 digits for the location of any given word, 7 digits for the tank or ‘street’ number and 5 digits for the location of the word in the tank, i.e. the ‘house number’ proper. (IAS 1945, #2, 3)

The code, summarized in Table 5.1, is markedly simpler than the second EDVAC code. Substitution is described as a ‘partial’ operation and the transfer operation as ‘total’. Four orders provided for partial and total substitution or replacement in either direction between long and short tank words. The code also appears to be the first to include a conditional jump instruction, perhaps made possible by the three-address format of the order words:

Type	Short symbol	Description
1	$\mathcal{N}\xi$	The number ξ .
Arithmetic order		
2	$x \ \omega \ y \rightarrow X$	Carry out the operation $x \ \omega \ y$ and dispose of the result in X .
Transfer orders (total substitutions)		
3	$x \rightarrow X$	Replace the contents of X with those of x .
4	$X \rightarrow x$	Replace the contents of x with those of X .
Substitution orders		
5	$x \rightarrow X$ partial	Substitute the address fields of X with values from x .
6	$X \rightarrow x$ partial	Substitute the address fields of x with values from X .
Transfer of control orders (unconditional and conditional jumps)		
7	$\mathcal{C} \rightarrow X$	\mathcal{C} connects to long tank X .
8	$\mathcal{C} \rightarrow X$ if $x \geq y$	\mathcal{C} connects to long tank X if $N(x) \geq N(y)$.

Table 5.1 The code outlined by von Neumann at the ECP meeting on 19 November, 1945 (IAS 1945, #2). The discussion did not fully resolve the question of when and where to store the results of arithmetic operations. $N(x)$ denotes the number in tank x .

It is desirable, however, to be able to transfer the control on order. Denote this order by $\mathcal{C} \rightarrow X$. In addition you want the ability to exercise an option depending on the outcome of a particular computation, for example if the number in one short tank is greater than or equal to the number in another short tank (denoted by $x \geq y$). If we always code as $\mathcal{C} \rightarrow X$ if $x \geq y$ then we can get unconditional transfer by putting all 0's in for x and y , or we might have a special order for unconditional transfer. (IAS 1945, #2, 4-5)

By the beginning of 1946, however, plans for the new machine's memory had changed dramatically. RCA had begun development of a new type of storage tube, the Selectron, which would remove the variable waiting time associated with delay line storage, and allow any item of data to be retrieved in a determined time period. The architecture of the IAS machine, as described in the *Preliminary Discussion* of June, 1946, included a memory built from 40 Selectrons. Words consisted of 40 bits, with one bit stored at the corresponding position in each Selectron.

With a delay line memory, short tanks had been used as registers to make the operands and results of arithmetic operations easily available. With the Selectron storage, the rationale for this distinction evaporated, and the design of the arithmetic unit changed. It now contained two registers, an *accumulator* to hold the result of additions and subtractions, and a *register* which held the multiplier and part of the product in multiplications, and the divisor and remainder in divisions.

The code given in the *Preliminary Discussion* is summarized in Table 5.2. Orders reverted to the one-address format of the *First Draft* and, like the registers in the arithmetic unit of the first EDVAC design, the accumulator served as the go-between in any transfer of data from one location in memory to another. Only 20 bits were

Order	Short Symbol	Description
Arithmetic orders		
1, 2	$x -$	Clear accumulator A , and add (subtract) the number stored in $S(x)$ to (from) A .
3, 4	$x -M$	Clear A , and add (subtract) the absolute value of the number stored in $S(x)$ to (from) A .
5, 6	$x h-$	Add (subtract) the number stored in $S(x)$ to (from) A , without clearing.
7, 8	$x h-M$	Add (subtract) the absolute value of the number stored in $S(x)$ to (from) A , without clearing.
9	$x R$	Clear register R and add the number stored in $S(x)$ to it.
10	A	Clear A and move the number in R to A .
11	$x \times$	Clear A and multiply the number in $S(x)$ by the number in R , storing product in A and R .
12	$x \div$	Clear R and divide the number in A by the number in $S(x)$, leaving quotient in R and remainder in A .
20, 21	$L \quad R$	Multiply (divide) the number in A by 2 (i.e. shift A left (right)).
Transfer of control orders (unconditional and conditional jumps)		
13, 14	$x C \quad x C'$	Transfer control to the left (right) order in $S(x)$.
15, 16	$x Cc \quad x Cc'$	Transfer control to the left (right) order in $S(x)$ if the number in A is ≥ 0 .
Substitution orders (total and partial)		
17	$x S$	Move the number in A to $S(x)$.
18, 19	$x Sp \quad x Sp'$	Move the appropriate digits of A to the address field of the left (right) order in $S(x)$.

Table 5.2 The code of the IAS machine described by Burks et al. (1946) and used by Goldstine and von Neumann (1946). Most orders consist of a single number x followed by an identifying symbol. $S(x)$ is the Selectron location indexed by x ; two orders were stored in each location.

needed to code an instruction, divided between an 8 bit operation code and a 12 bit number specifying a storage location, and to save storage it was proposed to pack two orders in each word. Pairs of orders were provided where necessary to distinguish the two halves of a word.

Perhaps the most striking thing about this code, however, is how similar it is to the code von Neumann had outlined the previous November, despite the radical change in memory technology. A wide range of arithmetic operations was specified, presumably to make the coding of typical applications as convenient as possible, but apart from these the code only contains the partial and total substitutions and conditional and unconditional jumps of the November code.

5.2 Programming with diagrams

Von Neumann's 1945 manuscript contains no visualization of the meshing program. As he and Goldstine worked on example problems for the *Planning and Coding* reports, however, the limitations of a purely textual approach became apparent and, famously, the first report introduced the flow diagram notation. The rationale for this was clearly explained in a draft version of the report:¹

From our limited experience with the coding of numerical problems we have acquired a conviction that this programming is best accomplished with the help of some graphical representation of the problem. We have attempted, in a preliminary way, to standardize upon a graphical notation for a problem in the hope that this symbolism would be sufficiently explicit to make quite clear to a relatively unskilled operator the general outline of the procedure. We further hope that from such a *block-diagram* the operator will be able with ease to carry out a complete coding of a problem.

This unfamiliar terminology is suggestive, as ENIAC's first cadre of operators later recalled that they had learned to program the machine by examining its block diagrams. Those imposing blueprints hid the complex electronic design of ENIAC's units, representing them as networks of functionally defined components. Similarly, Goldstine and von Neumann's block diagrams presented a view of a routine that hid the details of coding while revealing its large-scale structure more clearly. The diagrams' ideography was similar to that of their electronic predecessors:

The 'route' travelled by the control is symbolized by lines with arrows on them indicating the direction of 'motion', thus \longrightarrow . In appropriate places along the route we introduce storage boxes for the retention of 'static' data, i.e., of number-words as distinguished from order-words, thus $\xrightarrow{\square}$. [...] Finally we introduce boxes en route to show the functions the control is to perform, i.e., to reveal the microscopic orders, thus $\rightarrow \square \rightarrow$. In this case the control executes the orders in the box before proceeding further in its itinerary.

A block diagram is a picture of a coded routine in memory: locations holding numbers are shown as *storage boxes* and the in-line *control boxes* correspond to locations containing orders, while the arrows show how control can move through memory as the program runs. Figure 5.1 shows the first example diagram in the draft report, a simple loop to calculate N values of the polynomial $x_n^2 + ax_n + b$ and tabulate the results in memory.

Von Neumann drew an analogy between inductive processes in mathematics and loops in programs and referred to the variable controlling a loop, n in this example, as the 'inductive variable'. The trickiest aspect of representing code graphically was to find a way of showing how the values of the inductive variable and the other stored numbers that depended on it changed as a routine executed.

The storage box in Figure 5.1 lists the memory locations used by the program and their contents. The variables a , b , N , and n represent the numerical data of the problem, and memory locations are identified by the address variables $0'$, $1'$, $2'$, \dots .

¹ Goldstine and von Neumann (1946). All unattributed quotations in the following sections are from the text of the draft report.

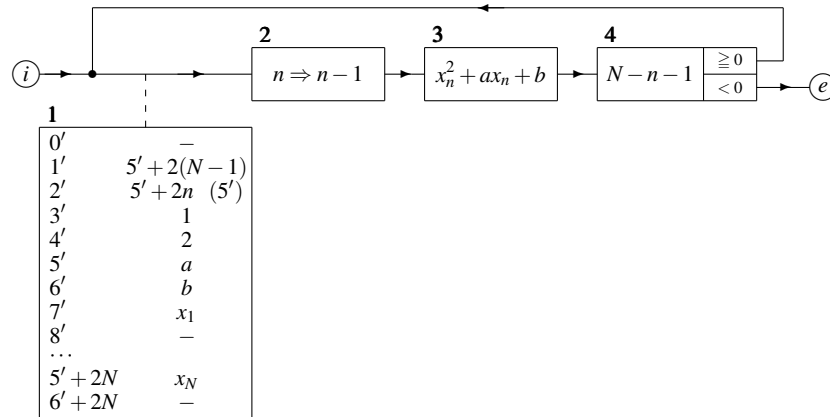


Fig. 5.1 Block diagram of a routine to tabulate values of $x_n^2 + ax_n + b$. After Goldstine and von Neumann (1946).

The contents of memory are defined by constant or variable expressions. Location $0'$ is working storage and locations $8', 10', \dots, 6' + 2N$ will hold the calculated values of the polynomial.

The storage box is attached to the flow line inside the loop body and documents the values stored at the moment control passes that point. Values depending on n will differ on each iteration. At the beginning of each iteration of the loop, location $2'$, for example, will hold the value $5' + 2n$. This expression represents an address: for $n = 1, 2, \dots, N$ it will take the values $7', 9', \dots, 5' + 2N$, the addresses of the locations holding the values of x_n . The ($5'$) in line $2'$ in the storage box is an initial value; this implies that the initial value of n is 0, something that is not otherwise stated.

Storage boxes are all alike, but control boxes differ according to the kind of operation they portray. The three control boxes in Figure 5.1 correspond to the major steps in the loop: incrementing the value of n , calculating the next value of the polynomial, and testing whether or not the induction, or iteration, is complete. It is striking that although the progress of the iteration is determined by the values of N and n , these numbers are not actually stored. Goldstine and von Neumann wrote that 'we find it more convenient to store $5' + 2N, 5' + 2n$ ', implying that the code will control the loop by manipulating location numbers rather than, as the control boxes suggest, storing and modifying the value of n .

A box containing a formula

merely indicates that a group of microscopic orders are to produce a given arithmetic result, thus $\boxed{n^2 + 3n + 1}$ is an abbreviation for a set of orders that will produce the polynomial $n^2 + 3n + 1$.

Microscopic orders 'explain how a particular expression is calculated', and were contrasted with macroscopic orders which 'govern the overall characteristics of the program', such as its looping and termination properties. These operation boxes thus

performed an essential function of any block diagram by blackboxing some of the low-level details of the program.

Conditional transfer orders were represented by ‘alternative boxes’ with different output arrows to be followed in the two cases where the expression in the box was ≥ 0 or < 0 . As with the operation boxes, the expression in an alternative box was treated as an abbreviation for the microscopic orders required to calculate it, and the sign of the result was used to choose between the two possible paths.

The third type of control box in Figure 5.1, called a ‘substitution box’, was

used for those orders that can alter numbers of other orders [and] symbolized as a rectangle with an instruction such as $n + 1 \Rightarrow n$ inside of it, i.e., $\boxed{n + 1 \Rightarrow n}$. This box is used to mean that any order containing a memory-location number which depends on n and any number-word which depends on n are to be altered by replacing n by $n + 1$.

A substitution box appearing in a loop typically represented a change in the value of the variable controlling the loop and like the other control boxes, the expression in the box denoted the microscopic orders required to bring this change about. These orders might be more complicated than simply incrementing the number in a given memory location: in this example, the induction variable n only appears within the expression $5' + 2n$, and the microscopic orders would have to change the contents of location $2'$ consistently with the value of n increasing by 1.

Finally, the block-diagram notation included ‘the symbols i and e to denote the initiating and ending program signals’. Symbols in circles were connectors used to link non-contiguous parts of a large diagram.

After presenting the block diagram, Goldstine and von Neumann went on to show how it could be used, as promised, to carry out the coding of the problem ‘with ease’. Table 5.3 shows sequences of orders corresponding to the three control boxes in the block diagram of Figure 5.1. As in the meshing routine manuscript, initial coding was carried out in an extended code and the code laid out in a similar format. Address variables 2.1 to 4.4 denote the locations of the orders are stored and appear in substitution and transfer orders. The symbol e represents the end of the routine and would typically be replaced by an unconditional jump to the first instruction of the next part of a complete program.

The coding of the substitution box is slightly confusing. For reasons unexplained in the text, the diagram shows the substitution $n \Rightarrow n - 1$ instead of the expected $n + 1 \Rightarrow n$. Order 2.1 loads the value of location $2'$ into the accumulator, but the stated result of this order is to load the value $5' + 2(n - 1)$ into the accumulator rather than the $5' + 2n$ that the storage box records as the value of $2'$; it appears that the variable n appearing in the storage box has been substituted by the expression $n - 1$. Order 2.2 adds 2 to the accumulator and the resulting value $5' + 2n$ is then stored in location $2'$, re-establishing consistency with the storage box. The partial substitutions in orders 2.4, 2.5, 2.6 and 2.8 update the orders whose address field depends on the value of n , namely 3.1, 3.2, 3.4 and 3.8. The code corresponding to the substitution box therefore carries out all the changes necessary to make the stored numbers consistent with the incremented value of the induction variable n .

As in the meshing routine manuscript, the next step was to assign the order and number words to specific locations in memory and replace the variables in the coded

Memory Position	Control Order	Result of Order	
		A	R, S, C
2.1	2'	$5' + 2(n-1)$	
2.2	4' h	$5' + 2n$	
2.3	2' S		$2' : 5' + 2n$
2.4	3.1 Sp		3.1 : $(5' + 2n)$ R
2.5	3.2 Sp		3.2 : $(5' + 2n) \times$
2.6	3.4 Sp		3.4 : $(5' + 2n)$ R
2.7	3' h	$6' + 2n$	
2.8	3.8 Sp		3.8 : $(6' + 2n)$ S
3.1	$5' + 2n$ R		R : x_n
3.2	$5' + 2n \times$	x_n^2	
3.3	0' S		0' : x_n^2
3.4	$5' + 2n$ R		R : x_n
3.5	$5' \times$	ax_n	
3.6	0' h	$x_n^2 + ax_n$	
3.7	6' h	$x_n^2 + ax_n + b$	
3.8	$6' + 2n$ S		$6' + 2n : x_n^2 + ax_n + b$
4.1	2' -	$-(5' + 2n)$	
4.2	1' h	$2N - 2 - 2n$	
4.3	2.1 Cc		$N - n - 1 \geq 0$
4.4	e		

Table 5.3 Sequences of orders corresponding to the substitution, operation and alternative boxes in Figure 5.1. The effect of an order on the contents of the accumulator is shown in column A and the changed contents of memory locations and the register in column R, S, C, along with the condition under which the conditional transfer takes place. After Goldstine and von Neumann (1946).

instructions of Table 5.3 with the actual addresses. However, the design of the IAS machine added a complication at this point:

Accordingly, we proceed to the last phase of our programming: the assignment of actual positions in the memory. Recalling that there are two orders per word, we make the assignment as follows: There are in all 20 orders, which will occupy positions 1 through 10 in the memory; the static storage will then be assigned positions 11 through $(17 + 2N)$ in the memory. We now systematically renumber all our preceding work and thus arrive at our final set of machine orders.

In addition, the induction variable n was assigned its initial value of 0. This allocation of orders and static storage locations to fixed positions in memory gives the version of the program shown in Table 5.4.

The details of this example demonstrate how the general approach to coding adopted by von Neumann for the meshing routine was carried over to the IAS project. A number of basic sequences were identified and coded in an extended code including address variables. Values were then substituted for these variables to give a version of the code that could easily be translated into binary form. With only minor modifications, this is the approach that was followed in the examples in the published version of the *Planning and Coding* reports.

Address	Orders	Address	Contents
1	13 : 15 h	11	
2	13 S : 5 Sp	12	$14 + 2N$
3	5 Sp' : 6 Sp'	13	16
4	14 h : 8 Sp'	14	1
5	16 R : 16 ×	15	2
6	11 S : 16 R	16	a
7	16 × : 11 h	17	b
8	17 h : 17 S	18	x_1
9	13 - : 12 h	...	
10	1 Cc : e	$16 + 2N$	x_N
		$17 + 2N$	

Table 5.4 The final version of the code for the polynomial program. Order and number words have been assigned to definite memory locations, and the only variable remaining is e . After Goldstine and von Neumann (1946).

The major innovation in the draft report is the introduction of the block diagram notation. The diagram of Figure 5.1 replaces the manuscript's case analysis as a way of summarizing the relationships between the basic sequences, defining which sequence followed which and when each sequence would be invoked. Goldstine and von Neumann clearly found the graphical notation clearer and more expressive than the conventions of informal mathematics, perhaps particularly when it came to defining iterative processes. Furthermore, by distinguishing operation, alternative and substitution boxes, the notation introduced a three-fold categorization of the types of basic sequence that allowed program structure to be modelled at a more abstract level than that of individual orders.

Another change from the manuscript is the treatment of the starting address of the program. In the manuscript, von Neumann originally left this as a variable e (confusingly now used for the exit point of a routine), but coded routines in the draft report are assigned a fixed starting address. We will see in the next chapter how this was reconciled with the need, also identified in the manuscript, to allow routines to easily appear at different places in memory.

5.3 The meshing routine

Two sections of the draft *Planning and Coding* report, in Goldstine's handwriting, develop code for meshing and sorting routines. The two operations are defined in almost the same words as the opening section of von Neumann's manuscript, but the report gives a block diagram for the meshing problem instead of von Neumann's case analysis. The code is also rather different; in part this is the inevitable effect of migrating the problem to a different machine, but there are two significant changes which led to modifications to the block diagram notation and its use.

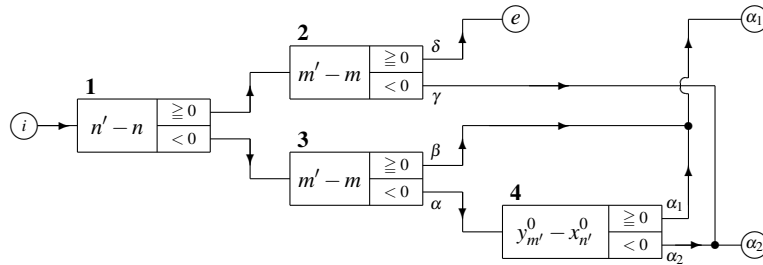
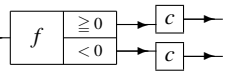


Fig. 5.2 The case analysis at the start of the meshing program depicted as a cascade of alternative boxes. Adapted from Goldstine and von Neumann (1946); the mathematical notation is that of the 1945 manuscript, and small Greek letters have been added to cross-reference the order sequences in the earlier program with the control paths in the block diagram.

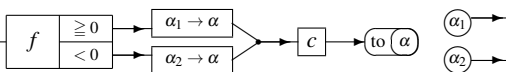
The first change had to do with the way the conditional branching at the start of the program was expressed. In the manuscript, von Neumann had used the technique described in section 3.3, substituting the starting location of an instruction sequence into an unconditional transfer order. In contrast, the later version of the program used conditional transfer orders. Four conditional branches were required, and the block diagram in the draft report depicts these as the cascade of alternative boxes shown in Figure 5.2.

Von Neumann seems to have noticed a redundancy here: boxes 2 and 3 carry out the same test and so the corresponding code will be repeated in the final program on both control paths leading from box 1. In the meshing program this is insignificant, but he discussed the general case in a letter to Goldstine (von Neumann 1946):

It can happen that a conditional transfer is wanted, but in such a manner that it should become effective only after certain other calculations have taken place. Denote these calculations by a block $x \rightarrow c \rightarrow xx$, the case I have in mind is then, that you wish to make the test, as to which of the two branches of the conditional transfer is to be used, before c (at x), but that you wish to carry out the bifurcation after c (at xx). One way to do this is to code c twice:

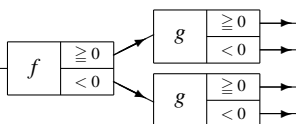
Scheme A: \rightarrow  , but this may be wasteful. Another method is to

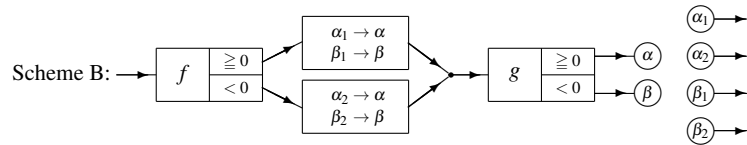
code c only once, to let an unconditional transfer $(to \alpha)$ follow c , and to substitute α from the conditional transfer before c :

Scheme B: \rightarrow 

[...]

If c is sufficiently involved, then Scheme B may be preferable to Scheme A. Even for a 4-way alternative: $f \geq$ or < 0 , $g \geq$ or < 0 (which we have e.g. in the meshing problem), one should be familiar with both approaches:

Scheme A: \rightarrow  , and



This new notation, the implicit transfer of control from a terminal containing a symbol (e.g. α) to one of a set of matching terminals containing that same symbol subscripted (e.g. α_1), became known as a *variable remote connection*. The boxes containing formulas such as $\alpha_1 \rightarrow \alpha$ represent partial substitutions modifying the addresses in the transfer orders; in the notation of the draft report these should be written $\alpha_1 \Rightarrow \alpha$, etc. Even von Neumann was confused by the symbolism, and he wrote under the first Scheme B diagram, ‘Am I using the proper kind of arrow?’.

Goldstine added a summary of von Neumann’s proposal to the draft report, along with an illustration of how to apply scheme B to boxes 1, 2 and 3 of Figure 5.2, but both the draft and the published *Planning and Coding* reports exclusively used the more flexible scheme A to express complex conditional branching.

As well as being a useful modelling notation, variable remote connections made the block diagram notation more *complete*, in the sense of being able to depict more of the capabilities of the code. Partial substitutions into arithmetic orders were adequately represented by substitution boxes, but partial substitutions into transfer orders will in general change the topology of the block diagram, in effect moving the pointy end of an arrow to another box. They therefore cannot be modelled by diagrams containing only fixed connections between boxes.

A second change to the block diagram notation arose indirectly from the nature of the IAS machine’s storage. The meshing routine moves complexes, contiguous groups of $p + 1$ words, from one place to another in memory. In a delay line memory, it was difficult to arrange for the complexes to be efficiently moved on a word-by-word basis, and the version of the EDVAC code used in the 1945 manuscript included instructions to move sequences of words in one operation.

With the Selectron storage this constraint no longer held, and the code for the IAS machine only provided for moving one word at a time. This meant that the meshing routine now had to include an inner loop to move the $p + 1$ words in a complex. In the sequence of examples in the draft report, this was the first program with a nested loop, or a ‘double induction’ as von Neumann called it, and this raised an issue with the use of storage boxes.

Block diagrams were originally drawn with a single storage box summarizing the memory locations used by the program, as the example in Figure 5.1 illustrates. A formula described the contents of each location, and their initial values could also be shown. With only one box, however, it was not possible to show the relationship between the control flow through the diagram and the changing values in memory. Von Neumann described an alternative scheme, with reference to the diagram in figure 5.1, as follows:

An alternative procedure would be this: Attach the storage box in its initial form, i.e. with a $5'$ opposite the memory location number $2'$, outside the n -induction loop, i.e. between \textcircled{i} and the first \rightarrow . Attach at the storage box’s present location, i.e. within the n -induction

loop, a small storage box which indicates only the change that takes place during the induction: $\boxed{2' : 5' + 2n}$. At present the simpler notation of the text will be used, however there are more complicated situations (e.g. multiple inductions) where only a notation of the above type is unambiguous. (Goldstine and von Neumann 1946)

Under this new scheme, a storage box at the beginning of a block diagram showed the initial values held in memory locations, and small boxes were attached to the flow lines of the diagram at points, particularly inside loops, where the stored values changed. Goldstine and von Neumann then redrew the existing block diagrams to include smaller distributed storage boxes that more clearly showed the changes in memory that took place as the process evolved. After redrawing the block diagram for the meshing routine, Goldstine commented:

It should be noted [...] how one shows the changes effected in the various positions of main storage [box] by means of small details from it inserted into the appropriate places in the various induction loops. (Goldstine and von Neumann 1946)

This notational change did not affect the control boxes in the block diagram, however, but only the way the changes they brought about were described. The coding of the meshing routine from the original block diagram, following the same process as for the simpler polynomial example, was unchanged by the update to the diagram.

5.4 The sorting routine

The draft *Planning and Coding* report contains the first full treatment we have of a sorting routine using the mergesort procedure, explained as follows:

we consider the problem of sorting a sequence of complexes $(x_l; u_l^1, \dots, u_l^p)$ ($l = 0, 1, \dots, L - 1$), i.e., of obtaining a monotone permutation of the given sequence. It will be shown below that this can always be achieved by repeated meshings. We start by meshing the 1-sequences $(x_{2r}; u_{2r})$ and $(x_{2r+1}; u_{2r+1})$ ($r = 0, 1, 2, \dots$) and obtain a number of 2-sequences—and possibly one 1-sequence. We then mesh these sequences, obtaining a set of four-sequences, etc. (Goldstine and von Neumann 1946)

The sorting routine was designed as a ‘double induction on two indices n and r , which specify the length and number of monotone sequences to be meshed’. n controls the outer loop: each time this loop is entered, the length of the subsequences being meshed is doubled, so as n takes on the values $0, 1, 2, \dots$, the sequences being meshed are of length $2^n = 1, 2, 4, \dots$. In the n^{th} iteration, all the subsequences of length $\leq 2^n$ have been sorted, and the routine meshes pairs of these into sorted subsequences of length $\leq 2^{n+1}$. r controls the inner loop, counting how many pairs of subsequences have been meshed. At each iteration of the inner loop, there are three possibilities: two subsequences of length 2^n can be meshed and the inner loop repeated; if there is only one subsequence of length 2^n remaining, it can be meshed with any remaining complexes; otherwise, any remaining complexes are simply added to the end of the sequence.

The block diagram for the sorting routine was revised twice as Goldstine and von Neumann took into account the modifications to the notation described in the previous section. The first diagram used the original form of the notation with a single large storage box and the first revision introduced small boxes distributed round the diagram to show the changes in stored values more clearly.

As von Neumann had proposed in his 1945 manuscript, the sort routine called the meshing routine as a subroutine, to ‘utilize our preceding work on meshing’ as the draft report put it, and represented it in the block diagram by the special box $\rightarrow * \boxed{M} \rightarrow$. In the first two versions of the diagram, the box representing the meshing routine was drawn twice and linked directly to the following code, in effect following scheme A from von Neumann’s letter. The final version of the diagram followed scheme B: the box was only drawn once and the different branches of the sort routine merged just before it. Variable remote connections specified where control would pass to once meshing was complete.

This change reflects some uncertainty about the semantics of the asterisked boxes, which Goldstine and von Neumann had originally described as being like footnotes. Boxes in block diagrams represented blocks of words in memory: if the subroutine box appeared twice, this would mean that its code would appear twice in the finished program, or that it would be an ‘open’ subroutine, to use terminology that emerged a few years later. In fact, the meshing routine was used as a ‘closed’ subroutine whose code was separate from that of the sort routine and called from different places in it, a situation more accurately reflected by the final version of the diagram.

A further notational issue was how to represent the initialization of the subroutine’s parameters when it was called. Goldstine and von Neumann dealt with this rather ingeniously by including the meshing routine’s initial storage box in the block diagram of the sort routine. This box defined the storage used by the subroutine and by means of suitable substitutions, the sort routine diagram could indicate how that storage was initialized before the meshing routine was called.

When they began to consider the sort routine, Goldstine and von Neumann had already fully coded the meshing routine and assigned it to storage locations 1–31. The block diagram and the code of the sort routine used these location numbers to cross-reference to the meshing routine. This was not a viable long-term strategy, however: as von Neumann had pointed out, subroutines would have to be ‘mobile’ to be useful, and a technique that required actual location numbers to be included in a block diagram that used a subroutine was clearly inadequate. The solution they proposed to the mobility problem is discussed in Section 6.3; in the draft report, the final stage of the coding of the sort routine was, perhaps tellingly, ‘left to the reader as an exercise’.

5.5 Planning and coding

As the draft report was transformed into the three volumes of the final *Planning and Coding* report, many changes were made and much new material added. The first chapter recast the block diagrams of the draft as *flow diagrams* and situated them within an explicit step-by-step process for program development that codified von Neumann's earlier practice outlined in Table 4.8.

Goldstine and von Neumann enumerated four major stages in the process, each containing a number of specific steps; these are summarized in Table 5.5. Despite the introduction of flow diagrams, the overall shape and purpose of this process is very similar to the earlier approach. In both, after a stage of mathematical analysis the program is written in an extended symbolic code containing variables of different kinds. These variables are then systematically removed, leaving variable-free code which can be mechanically transformed into its binary form.²

The report began with some general comments about coding. Emphasizing the fact that a running program could modify its own instructions, Goldstine and von Neumann pointed out that in general the coded sequence of instructions in memory would change as program execution progressed. It was therefore important to understand not only the static properties of the code expressed in the initial sequence of symbols, but also its dynamic, temporal behaviour:

in planning a coded sequence the thing that one should keep primarily in mind is not the original (initial) appearance of that sequence, but rather its functioning and continued changing while the process that it controls goes through its course. [...] the basic feature of the functioning of the code in conjunction with the evolution of the process it controls, is to be seen in the course which the control *C* takes through the coded sequence, paralleling the evolution of that process. We therefore propose to begin the planning of a coded sequence by laying out a schematic of the course of *C* through that sequence, i.e. through the required region of the selectron memory. This schematic is the *flow diagram of C*. (Goldstine and von Neumann 1947-8, vol. 1, 4)

Goldstine and von Neumann did not explain why they changed their terminology from block to flow diagram, but perhaps they felt that the strong associations of the earlier term with electronic design would be misleading. Using the metaphor of flow to describe computational processes depicted as directed graphs already had some currency. As well as Cunningham's diagrams of the punched card workflow surrounding ENIAC, mentioned earlier, BRL mathematicians Haskell Curry and Willa Wyatt had described a depiction of the topology of a complex ENIAC set-up as a 'flow chart' (Curry and Wyatt 1946).

Like block diagrams, flow diagrams showed the paths *C* can take through the code of a routine laid out in memory, but there was now a greater emphasis placed on the changes taking place as the process unfolded. The equivalence between types

² Von Neumann does not appear to have considered automating this final transformation. His May 1945 letter to Goldstine described a typewriter-like key-punch device with keys labelled with the symbols of the EDVAC code. When a key was pressed, the machine would make the corresponding perforations in some tape (von Neumann 1945e).

Step	Transformation	Resulting form of program
1	A mathematical stage of preparation	
1.1	Give a mathematical formulation of the problem.	Equations and conditions.
1.2	Replace the equations and conditions with 'arithmetical and explicit procedures'.	A step-by-step process, usually employing approximations and multiple inductions.
1.3	Estimate the numerical precision of the process defined in step 1.2.	
2	The <i>dynamic</i> or <i>macroscopic</i> stage of coding	
2.1	Allocate storage areas.	
2.2	Identify basic sequences of instructions needed and their connections.	Flow diagram with storage tables.
3	The <i>static</i> or <i>microscopic</i> stage of coding	
3.1	Write instructions for each operation and alternative box and variable remote connection in the flow diagram.	The <i>preliminary enumeration</i> of the code: sequences containing storage positions and variables.
4	Assign all storage positions and all orders their final numbers	
4.1	Assign numbers to storage positions.	
4.2	Linearize operation sequences, adding unconditional jumps where necessary.	
4.3	Assign orders to words, pairing where possible. Assign final numbers and L/R marking to orders.	
4.4	Replace the address variable in each order with its final value.	The <i>final enumeration</i> of the code, containing variables <i>i</i> and <i>e</i> for entry and exit addresses.
4.3	Replace <i>i</i> and <i>e</i> with their actual values.	Variable-free binary code.

Table 5.5 The methodology of the *Planning and Coding* reports. The descriptions of the four main stages are taken verbatim from Goldstine and von Neumann, (1947-8, vol. 1).

of boxes and orders was maintained. *Operation boxes* denoted simple sequences of arithmetical operations and transfers of numbers, and:

The alternative boxes which we introduced correspond to the conditional transfer orders $x\text{Cc}$, $x\text{Cc}'$. I.e., the intention that they express will be effected in the actual code by such an order. (Goldstine and von Neumann 1947-8, vol. 1, 7)

The most significant innovation in the flow diagram notation was to do with the way substitution was handled. To begin with, an explicit distinction between *storage* and *variables* was introduced. The memory locations used by a program to store numbers were classified as *fixed* or *variable* storage. Both terms were relative to the problem being coded: the Selectrons provided no read-only storage. Storage was distinguished from variables, which were introduced as follows:

A mathematical-logical procedure of any but the lowest degree of complexity cannot fail to require *variables* for its description. (Goldstine and von Neumann 1947-8, vol. 1, 10)

Variables were also classified into two kinds. Some were basically parameters, given values at the beginning of the problem and never changed. Borrowing a term from formal logic, Goldstine and von Neumann referred to these as *free* variables. *Bound* variables, in contrast, were those which only existed within the problem, assumed a sequence of different values as a computation progressed, and couldn't meaningfully be given values from outside a routine. A typical example of a bound variable is the 'induction variable' that counted the iterations of a loop.

Storage locations, then, represented the physical locations updated by a running program. Variables, on the other hand, were symbols used only in the mathematical description of the problem. It began to seem like something of a category mistake to think of a program instruction changing the value of a variable. In the flow diagram notation, this change in perspective was reflected in a technical change in the status of substitution boxes. They no longer corresponded to the code that carried out the substitutions, but only to a change in the way the computation was described.

A substitution box never requires that any specific calculation be made, it indicates only what the value of certain bound variables will be from then on. Thus if it contains ' $f \rightarrow i$ ', then it expresses that the value of the bound variable i will be f in the immediately following constancy interval, as well as in all subsequent constancy intervals, which can be reached from there without crossing another substitution box with a ' $g \rightarrow i$ ' in it. The expression f is to be formed with all bound variables in it having those values which they possessed in the constancy interval immediately preceding the box, at the stage in the course of C immediately preceding the one at which the box was actually reached. (This is particularly significant for i itself, if it occurs in f .) (Goldstine and von Neumann 1947-8, vol. 1, 17)

To illustrate the effect of this change, consider the basic case of a loop which is traversed N times under the control of an induction variable n . Figure 5.3 shows how such a loop would be represented in the two notations. As in many of Goldstine and von Neumann's examples, the value stored is not n , but some more general expression $f(n)$ depending on it, perhaps the address of the n^{th} element of an array. Following the notational conventions for the two types of diagram, the location at which this value is stored is denoted by the symbols $1'$ and $A.1$, respectively.

The crux to understanding a loop is the transformation carried out by the boxes labelled 1 in Figure 5.3. These boxes stand for the instructions that compute and store $f(n+1)$ in place of the existing value $f(n)$. Box 1 in the block diagram increments the induction variable and all values that depend on it: after this code has completed, therefore, the new value stored at $1'$ can only be consistently described as $f(n)$, misleadingly suggesting that no change in storage has been effected by box 1. In the flow diagram, however, box 1 is an operation box which does not affect the value of the variable n , and the following storage box clearly indicates the updated value of f .

However, this increased expressivity comes at a cost. Following the loop after box 1 in the flow diagram, we re-encounter the storage box which states that the value stored in $A.1$ is $f(n)$: this is inconsistent with the new value $f(n+1)$ that has just been calculated. To make the diagram consistent, We need to state explicitly that the value of the bound variable n has increased. This is recorded in the in-line substitution box containing the expression $n+1 \rightarrow n$. Unlike the substitution box

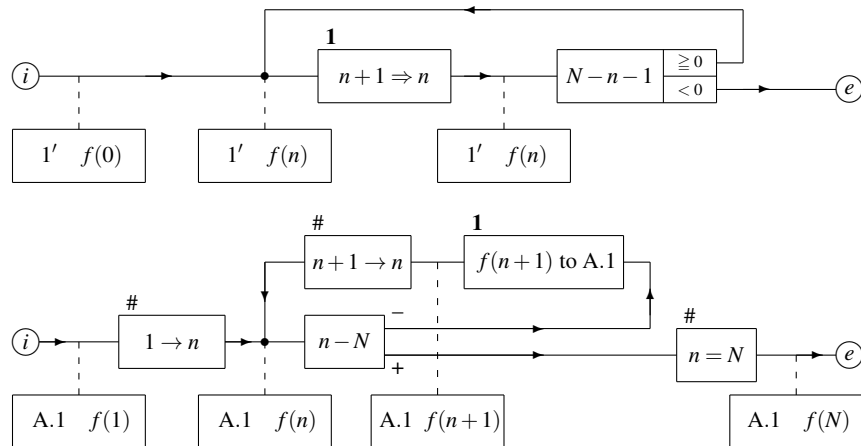


Fig. 5.3 The control of a simple loop expressed as a block diagram (top) and flow diagram (bottom). The block diagram is adapted from Figure 5.1, and loops with the same structure as the flow diagram appear in (Goldstine and von Neumann 1947-8, vol. 1). Control passes through each loop N times, and the inductive variable is n . An expression dependent on n , represented by the function $f(n)$, is computed and stored, not n itself.

in the block diagram, this does not correspond to any program instructions, and is distinguished from other control boxes by being marked with $\#$. As Goldstine and von Neumann (1947-8, vol. 1, 11) expressed it, ‘we prefer to treat these variable value changes merely as changes in notation which do not entail any actual change in the relevant storage items’.

To use a term introduced by von Neumann (1947), a substitution box *reconciles* the storage boxes on either side of it. If the result of applying the substitution to the expression in the storage box following the substitution box is the expression in the preceding storage box, the diagram is consistent. This technique also applies nicely to the initialization at the beginning of the loop: the substitution $1 \rightarrow n$ reconciles the first two storage boxes in the flow diagram, whereas the initial value $n = 0$ in the block diagram is only stated implicitly.

A flurry of correspondence between Goldstine and von Neumann in February and March 1947 resulted in the introduction of one more piece of notation, the *assertion box*, to the flow diagrams. The issue was how to record the overall effect of a loop after it had terminated. When the loop in the flow diagram of Figure 5.3 terminates, the induction variable n has the value N and storage location A.1 holds the value $f(N)$, as recorded in the final storage box. The function of the assertion box containing the expression $n = N$ is to reconcile this with the preceding storage box, just before the alternative box, which showed that A.1 stored the value $f(n)$.

Like a substitution box, an assertion box describes properties of variables rather than storage and is marked with $\#$ to indicate that it:

never requires that any specific calculations be made, it indicates only that certain relations are automatically fulfilled whenever C gets to the region which it occupies (Goldstine and von Neumann 1947-8, vol. 1, 17)

Flow diagrams can therefore be read on two levels. If the references to storage locations are treated simply as labels for values, a diagram can be read as a purely mathematical description of a computational process. If, on the other hand, the boxes marked # are ignored, the diagram functions, like the earlier block diagrams, as a picture of a program in memory that serves as a guide when coding the routine.

Historian Nathan Ensmenger (2016) has described flow diagrams as they were used in the computing industry as boundary objects, mediating between different groups of participants in the business of software production. In Goldstine and von Neumann's method, they exist rather on the boundary between planning and coding, or between mathematics and computing, mediating between the computer-independent work carried out in stage 1 and the machine-specific coding of stage 3. Furthermore, this boundary is inscribed in the very heart of the notation itself, in the distinction drawn between storage and variables and the use made of it.

5.6 The final versions of the sorting routines

Versions of the meshing and sorting routines appeared in the second of the *Planning and Coding* reports, in 1948, where they were described as having a 'combinatorial' rather than an 'analytical' character and as providing a test for the 'efficiency of the non-arithmetical parts of the machine' (Goldstine and von Neumann 1947-8, vol. 2, 49).

There was a subtle change to the wording of the mathematical definitions of the two operations. In the manuscript and the draft report, the operation of meshing two sequences had been defined as that of sorting the concatenation of the sequences. In the final report, however, meshing was described as the operation of finding a monotone permutation of the sum, or concatenation, of two sequences and sorting, as before, as that of finding a monotone permutation of a sequence. This change in wording made no material difference, but did at least remove the misleading suggestion that the meshing routine would be defined in terms of the sorting routine. On the contrary, the meshing problem was described as the simpler of the two, and one 'whose solution can conveniently be used' in solving the sorting problem.

Meshing was described as an 'inductive process' and was defined in detail by means of an elaborated and rather more formal version of the case analysis that von Neumann had presented in his 1945 manuscript. However the different cases were reordered to bring out the equivalence between the order in which the decisions were made and the structure illustrated in Figure 5.2. The flow diagram for the meshing routine preserved the structure of the block diagram from the draft report, and added a lot of detail in storage boxes to fully document the course of the meshing process. The code was presented as being derived from the flow diagram in essentially the same way as in the simple polynomial example described in Section 5.2.

Similarly, after a relatively informal description of the idea underlying the merge-sort algorithm, the sorting routine was formalized as an inductive process. The flow diagram, however, had a rather different structure from the earlier block diagram,

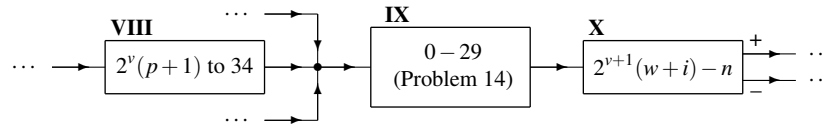


Fig. 5.4 Calling the meshing routine from the sorting routine. After (Goldstine and von Neumann, 1947-8, vol. 2, 61).

reflecting a continuing uncertainty on Goldstine and von Neumann's part about how to model the use of the meshing routine as a subroutine.

Goldstine and von Neumann wrote that

the subsidiary sequence [of the meshing routine] will occur in the interior of the main sequence [of the sorting routine] (Goldstine and von Neumann 1947-8, vol. 2, 59)

This wording seems to suggest that they planned to use the the meshing routine as an open subroutine and form a single sequence of instructions to carry out mergesort. In the code the produced, however, the meshing routine formed a closed subroutine occupying storage locations 0–41. The sorting routine occupied words 42–86, and substituted parameters values in the storage area used by the meshing routine before transferring control to location 0 to call the routine.

The meshing routine was only called once by the sorting routine, and as in the earlier diagrams it was represented in the flow diagram of the sorting routine as a single box. The relevant section of the flow diagram is shown in Figure 5.4. The meshing routine is represented by the box labelled IX: the asterisked form of the box is not used, and the number range 0–29 refers specifically to the order words in the subroutine. This notation therefore respects the basic semantics of the diagrams where boxes represented blocks of memory. The number words, or storage area, used by the meshing routine, in locations 30–41, were not shown on the diagram of the sorting routine, but were assumed to be available. A note on the diagram stated that 'Numbers 0–41 refer to 0–41 in 11.3', i.e. the preceding section of the report that had described the meshing routine.

The mergesort procedure called the meshing routine on three conceptually separate occasions. As in the final version of the diagram in the draft report, the flow diagram was structured to show the subroutine only once, with the three control paths merging just before the subroutine box. Rather than using variable remote connections, however, the diagram was significantly restructured to use a cascade of alternative boxes after control returned from the meshing routine to split the control flow back into three separate parts. The first of these alternative boxes is shown in Figure 5.4.

As far as the representation of subroutines is concerned, then, the flow diagram notation presented in the final *Planning and Coding* reports seems to take a step back from the draft report. There is no notation that is specifically adapted to denote or describe properties of a typical subroutine call and return process. The box representing the order words of the meshing subroutine seems to be nothing more than an abbreviation for a subdiagram: while this might well introduce the possibil-

ity of hierarchically structuring diagrams as a way of reducing complexity, it does not reflect the semantics of subroutines. In fact, by strongly suggesting that what is in fact a closed subroutine is implemented as an open subroutine, it seems rather to obfuscate those semantics. The issue of Goldstine and von Neumann's treatment of subroutines is considered in more detail in the following chapter.

After presenting the coded version of the mergesort routine, derived from the flow diagram in the standard way, Goldstine and von Neumann concluded by discussing operations more generally and comparing the performance of their programs with that of IBM punched card equipment.

There is no evidence that Goldstine or von Neumann continued work on meshing and sorting problems after the publication of the *Planning and Coding* reports. It is outside the scope of this book to trace the legacy of the reports and the code they contained, but an illustration of their fate is provided by the Los Alamos MANIAC computer, which became operational in 1952. MANIAC's design was based closely on that of the IAS machine, and its programmers' manual incorporated an adapted version of the *Planning and Coding* reports and included a version of the mergesort routine.

The problem had evolved significantly. It was simplified to sort sequences of numbers rather than complexes, and the length of the sequence to be sorted was assumed to be a power of 2. It was no longer coded as an internal sort, but was designed to process data held on the magnetic drum forming MANIAC's 'outer' memory. These assumptions simplified the program's structure: there was no subroutine to carry out meshing and it was described by a single flow diagram that contained 'three induction loops' corresponding to the subprocesses of meshing, sorting, and transferring data between the drum and the internal memory (Jackson and Metropolis 1954).

Chapter 6

Subroutines

The use of subroutines was an ever-present theme in von Neumann's programming work. In 1945 he specifically commented on the use of the meshing routine as a subroutine in a sorting program, the topic was extensively discussed in the 1946 *Planning and Coding* draft, and the final published volume of the report presented a scheme for partially automating the process of combining routines into a complete program.

The idea of organizing computations hierarchically so that previously obtained results and routines that had already been coded could be reused in new contexts was familiar and widespread. Manual computation relied on printed volumes of tabulated functions, and mathematicians used well-known interpolation routines to compute values that lay between those provided in a table. These volumes allowed the prior work of the computers who had drawn up the tables to be easily reused, and their use had the curious side-effect of distributing computation widely across space and time.

The designers of the first automatic computers adopted a variety of strategies for reuse: for example, the Harvard Mark I had special-purpose interpolation units which read tabular data from tapes and computed the intermediate values required by a program; in contrast, ENIAC had hardware units on which tabulated functions could be set up and programmers were expected to design their own interpolation routines to be called into action when required.

The challenge facing EDVAC's designers, then, was not that of coming up with a brand-new scheme to enable reuse but rather of adapting a suite of familiar and widely used practices to the demands of the new machine. As discussed earlier, von Neumann assumed that tabular data would be stored in memory and had used the example of table look-up to motivate the use of substitution in EDVAC's code. Adapting the idea of reusable instruction sequences to the new style of programming would, however, throw up an unexpected challenge.

6.1 Master and subsidiary routines

Von Neumann structured the meshing routine so that the fixed parts of the routine were separated from those that would vary from one occasion of use to another, clearly distinguishing the code of the meshing procedure itself from the substitutions used to initialize it for a particular meshing operation.

G_{82} [...] is a fixed routine. With a suitable choice of S_{15} it will, therefore, cause any desired meshing process to take place. Thus G_{82} can be stored permanently outside the machine, and it may be fed into the machine as a 'sub routine', as a part of the instructions of any more extensive problem, which contains one or more meshing operations. Then S_{15} must be part of the 'main routine' of that problem, it may be effected there in several parts if desired. If, in particular, the problem contains several meshing operations, only those parts of S_{15} need be repeated, in which those operations differ. (von Neumann 1945i)

While this may be one of the earliest recorded uses of the term 'subroutine', the strategy of organizing a computation hierarchically so that a main or master routine would delegate common operations to subsidiary routines was by no means new, as Goldstine and von Neumann well knew.

The desirability of establishing fixed routines for such processes which are common to many problems (e.g. interpolation routines, the square-rooting or the cube-rooting routines in machines without built-in square-rooters or cube-rooters, etc.), as distinguished from routines unique to a specific problem (e.g. the statement of the specific equation of the problem), has of course been known for a long time. Various existing machines contain varying features designed to achieve such ends: Made-up plug-boards which can be stored and inserted as units when called for, various 'master' and 'subsidiary' control tapes which can be combined with a certain freedom, etc. (Goldstine and von Neumann 1946)

The use of 'subsidiary routines' was discussed at a meeting in September, 1945, between the EDVAC team and representatives of BRL's Ballistic Computations Committee. After the meeting, Haskell Curry wrote to Goldstine with a suggestion:

In regard to our conference in Princeton a week ago, Dr. Dederick has pointed out that his point in regard to subsidiary routines was rather squelched in the discussion. [...] when [EDVAC's] input mechanism is designed, it will be highly desirable to have a library of subsidiary routines recorded on tapes [...] which can be called in by the master routine of the particular problem. The situation is analogous to the problem and routine tapes on the BTL machine. There the master tape is called the problem tape. It is necessarily different for each problem fed to the machine but there is a provision whereby the problem tape calls in certain routine tapes [...] I think Dr. Dederick's point is that it would be desirable to have some such feature as this on the EDVAC. (Curry 1945)

Goldstine replied with a very palpable squelch of Curry's own observation:

The analogy you state between the EDVAC and the BTL machine is, I think, not completely valid because the latter has practically no registers for storing program information, whereas the EDVAC will contain a large number of units capable of remembering program instructions. In fact, all the programming information for a given program will be handled inside the EDVAC and the magnetic tapes will be used only as a means for bringing those instructions into the machine before the actual program is started. (Goldstine 1945b)

He then went on to outline how 'routine tapes' would be adapted to EDVAC:

In the chain of events in the main program sequence, there will be inserted at the proper place the instructions to transfer the control [to] a given tank wherein is contained the instructions for carrying out the required sub-routine, together with the instructions which will send the control back to the main program upon completion of the sub-routine. Evidently one would collect in his library tapes for handling standard types of problems such as interpolations or integrations.

When the operator is preparing a tape for a new problem, he will punch the tape with all the instructions except the sub-routine instructions which are already contained in the library. He will merely insert into his programming instructions the number of the tank or tanks in which the sub-routines are to appear, and then, by means of a reperforator he would attach to his tape the sub-routine from his library tape. (Goldstine 1945b)

At the point of use, it was not hard to adapt the familiar technique of organizing computations hierarchically to EDVAC's demands: the sequences of instructions making up the main and subsidiary routines would be stored at different locations within memory instead of being placed on separate physical devices, and the code needed to initialize parameters and transfer control between main and subsidiary routines was quite straightforward.

The new machine design raised a tricky logistical problem, however, as there was no guarantee that a subroutine would appear at the same place in memory each time it was used. In fact, exactly the opposite was true, and von Neumann (1945i) described this 'mobility within the long tanks' as an 'absolute necessity' in any practical scheme for using subroutines. The problem arises because the code of a subroutine makes explicit reference to memory locations within the subroutine, such as local storage locations and the targets of jump instructions. These references can only be coded once the actual position of the subroutine in memory is known.

This requirement for mobility, or relocatability, meant that a library subroutine could not be used like a routine tape on the Bell machine, which could simply be copied and mounted on the appropriate tape reader. Because of its unified memory space, the code of EDVAC's library subroutines would, paradoxically, have to be different each time they were used as part of a larger program.

The necessary changes were not arbitrary, of course, but depended solely on the address of the first memory location occupied by the subroutine. Von Neumann had introduced the variable e to stand for this address in the meshing routine; as table 4.6 shows, 14 of the routine's instructions depended on e , and before it was called those instructions would have to be changed to be consistent with the actual value of e . In the draft *Planning and Coding* report Goldstine and von Neumann proposed a way of automating this task, but not before devoting quite a lot of attention to the more general question of how to develop programs using subroutines.

6.2 General program sequences

The treatment of subroutines in the draft and the final versions of the *Planning and Coding* reports is strikingly different. The second half of the draft report was devoted to the topic of 'general program sequences'. As von Neumann explained:

Such general routines will be stored in a library of tapes, and the equipment for preparing tapes will be arranged so that they can be easily recopied on a new program tape. The coding of a specific problem will then involve only the statement of the unique features of the problem along with a list of the ‘fixed’ subsidiary routines used, and an explanation of the modifications which have to be applied to them. (Goldstine and von Neumann 1946)

The same point of view is present in the published reports, but the presentation of example programs in the draft report foregrounds the identification and use of subroutines in programming much more than in the final texts.

After the introductory example of polynomial tabulation, described in chapter 5, Goldstine and von Neumann coded a routine for Lagrangean interpolation using a formula that contained two very similar subexpressions to form repeated products. They began the development by coding a general-purpose subroutine which would be called twice to evaluate the two different products:

We first, for simplicity describe a routine for forming the product

$$\prod_{n \neq m} (y - a_n).$$

(Goldstine and von Neumann 1946)

Rather than coding a specific routine for, say, quadratic interpolation, Goldstine and von Neumann designed the main routine to perform interpolation of any order N . The completed subroutine consisted of $N + 8$ number words and 28 instructions placed in 14 order words, a total of $N + 22$ words starting at memory location r . As well as N and r , the variables y , m , and a_i ($i = 0, \dots, N - 1$) from the mathematical definition of the product would have to be assigned values before the subroutine was called.

In the code, y , m and a_i only appeared in expressions denoting the contents of the number words, the subroutine’s local storage. Each time the subroutine was called, the initial values of these variables would be substituted into the appropriate places by a routine such as S_{15} . The situation was different with N and r , however: these variables appeared in the coded instructions, and would be fixed for any program that used the subroutine. They would have to be substituted as the subroutine was copied into memory as part of a complete program.¹

The initial coding of the interpolation routine using this subroutine did not take into account the fixed-point nature of the machine. All the numerical quantities used in a computation had to fall in the range $-1 \leq x < 1$, and considerable care had to be taken to ensure that intermediate values stayed in range and that a suitable level of numerical precision was maintained. Rather than simply rewriting the main routine to carry out a correctly scaled calculation, however, Goldstine and von Neumann coded two example subroutines to correctly evaluate formulas of the forms $c + b/a$ and $a + b$. The first of these subroutines was then used in the interpolation routine:

¹ A subroutine in which the order of interpolation N was also a run-time parameter would have required a different amount of numerical storage each time it was called, introducing complications that Goldstine and von Neumann apparently chose not to entertain.

the original naive calculation was replaced by new instructions that called the first of the new subroutines, placed in 21 words assumed to begin at location u .

This relatively simple example therefore illustrated two ways in which candidate subroutines could be identified in the course of developing a program. At the start of the process, a useful problem-specific routine was identified on the basis of the mathematical definition of the problem and coded in advance of the main routine. Once the initial coding had been completed, the benefits of developing some more generally useful subroutines were identified, and once they were coded the main program was 'refactored' to make use of one of the new routines.

The second half of the draft report considered the issue of 'general program sequences' in more detail, promising 'a very far-going systematisation of "fixed" subsidiary routines, and very flexible methods to modify and adjust such routines to the specific problems in which they are used'. Subroutines for integration were developed, and then combined with the earlier interpolation routine in a complete program for a hydrodynamical application. Finally, Goldstine and von Neumann turned to the practical issues of how these subroutines could be used.

The most significant issue had to do with the changes required to the code of a library subroutine after its location in memory was fixed. Also, some subroutines have parameters that have a fixed value throughout a given program, like the variable N in the interpolation routine. These values can conveniently be set at the time the subroutine is loaded into memory while other parameters will have to be initialized when the subroutine is called, as will its return address.

It would of course be possible to make these changes manually when assembling a complete program tape. However, Goldstine and von Neumann recognized that this would negate much of the benefit of having a subroutine library and that it would simplify operational procedures if the code on a subroutine tape could simply be copied onto an input tape. They therefore designed what they at first called a 'substitution sequence', itself intended as a library routine, to automate the process of modifying the subroutine's code.

6.3 Preparatory routines

The previous section described two subroutines that were coded in the course of developing an interpolation routine. In the text of the draft *Planning and Coding* report, the starting addresses of these subroutines were represented by the variables r and u . Variables did not occur in the binary version of the code, however, and so before these subroutines could be written on paper tape or magnetic wire and stored in the library, r and u had to be replaced by definite values. Assigning the value 0 to r , say, would mean that the product subroutine was coded as if it was to be placed in locations from 0 to $N + 28$, and the internal references in the code would be restricted to this range.

When a complete program tape was assembled and loaded, the subroutine would usually be placed in a memory at a different location, 237 say. To make its code

consistent with this location, each internal memory reference would have to be incremented by, in this case, 237. The substitution sequence was intended to automate this task: provided with the necessary data, such as the length and position of a program's subroutines, it would adjust the code to make it consistent with a particular occasion of use. As well as dealing with issues of relocatability, the substitution sequence could also initialize parameters that would be constant throughout an application, such as the order of interpolation N assumed in a particular problem.

In the final *Planning and Coding* report, the substitution sequence was retitled the 'preparatory routine', and described in two stages: the first routine adjusted the code of a single subroutine, and the second routine applied the first to all the subroutines in a program. The functionality of the preparatory routine was essentially the same as that of the substitution sequence in the draft routine, however.

The preparatory routine only automated part of the process of assembling a program, however, and at the end of the report Goldstine and von Neumann described how it would fit into the complete workflow. This process depended on details of the machine's input and output facilities that had not yet been considered, and before describing the assembly process, they outlined their assumptions in a rather programmatic way.

Coded routines, both library subroutines and main routines, would be stored on magnetic wire and fed into the machine under manual control. Operators would specify the starting location in memory to which the code on the tape should be transferred, and the number of words to be read from the wire. In a footnote, Goldstine and von Neumann suggested that this would also be the basic capability of the machine's eventual input order.

In principle, operators would be able to enter individual words into memory by putting the code on a very short tape. Recognizing the inefficiency of this, Goldstine and von Neumann also suggested that operators would be able to manually enter small numbers of words. The locations to be updated would be specified manually, and the coded words would then be "fed" directly into the machine by typing them with an appropriate "typewriter" (Goldstine and von Neumann, 1947-8, vol. 3, 20).

Given these input capabilities, the process of assembling a complete program would have involved the following steps.

1. The main routine and the library subroutines were all stored on their own pieces of wire. The starting locations of the routines would be calculated manually and set by the operators as each routine was fed into the machine.
2. The preparatory routine was also stored on its own wire and would be fed into the machine at the 'end of memory', where it would not interfere with the program routines.
3. The data required by the preparatory routine, such as the starting address of each routine and the value and location of certain parameters, would be typed into memory directly.
4. The 'machine is set going, with the control set manually at the beginning of the preparatory routine' (Goldstine and von Neumann, 1947-8, vol. 3, 21). When this routine finished, the code of all the library subroutines would have been adjusted to be consistent with their actual positions in memory.

5. Some further changes still needed to be made manually. Among these were the adjustments required to enable when one subroutine to call another, which could not be carried out until the final positions of all the subroutines were known. Goldstine and von Neumann didn't specify what these adjustments were, but in any call of a subroutine by another, the starting location of the subroutine being called, its parameters, and the address to which it should return control would all have needed manual adjustment.
6. Finally, the main routine could be set going.

This procedure would have involved a lot of work for the machine's operators, who would have to plan the format of the input tape in detail and calculate the starting addresses of the subroutines as well as entering a substantial amount of data by hand. In the draft report, Goldstine and von Neumann commented that the process 'involves filling out only a few dozen sets of data and is thus trivial in character' but they later considered the trade-off between manual and automatic procedures in slightly more detail, concluding that:

References to another subroutine [...] are likely to be rare and irregularly distributed. They are therefore less well suited for automatic treatment, by a special preparatory routine, than to ad hoc, manual treatment, by direct typing into the machine. (Goldstine and von Neumann, 1947-8, vol. 3, 23)

One point that Goldstine and von Neumann appear not to have considered is that all this work would be required every time a program was loaded. Citing an example where a simple manual set-up error on the differential analyzer had led to errors in the computation of a firing table, Curry commented on the proposal to use a typewriter to insert individual words into memory as follows:

Such an arrangement is very desirable for trouble-shooting and computations of a tentative sort, but for final computations of major importance it would seem preferable to proceed entirely from a program or programs recorded in permanent form, not subject to erasure, such that the computation can be repeated automatically [...] on the basis of the record. (Curry 1950, 4)

Goldstine and von Neumann's approach to the automation of program assembly can seem to be, in Martin Campbell-Kelly's words, a 'surprisingly pedestrian piece of work'. Contrasting it with the techniques developed for the EDSAC, Campbell-Kelly suggested that:

The absence of input-output instructions [in the code of the *Planning and Coding* reports] constrained the degree to which they could automate the programming process – indeed, perhaps even to *think* about automation. (Campbell-Kelly 2011, 26)

It is unlikely that this can be the whole story. Although they had not coded any input or output routines, Goldstine and von Neumann had a clear idea of what the relevant orders would do and the input order they described is essentially the same as the one provided in the EDSAC code.

The EDSAC counterpart of the preparatory routine was the 'Initial Orders', two versions of which were developed by David Wheeler in 1948-9. Whereas Goldstine and von Neumann had assumed that code would be loaded into memory manually,

the only way that code could be read into EDSAC's memory was under the control of a running program. The Initial Orders were this program, and they were set up on uniselectors and loaded manually before a program could be read from tape.

Crucially, programs were punched onto EDSAC's tapes not in binary form but in the equivalent of von Neumann's short symbols. An instruction to add the contents of memory location 52 to the accumulator, for example, would be punched as the three symbols A52 in a standard teletype encoding. The Initial Orders had the job of translating this into binary form before loading the encoded instructions in memory, a process that involved a decimal to binary conversion, among other things.


In 1949, Wheeler devised a second version of the Initial Orders that would deal with the assembly of a complete program out of a collection of subroutines. A single input tape would be prepared, containing the main routine and reperfored copies of the subroutine tapes. The code used on this tape augmented the short symbols of EDSAC's code in two ways. A number of variables were defined to indicate to the Initial Orders where, for example, an address in a subroutine needed to be modified to take into account the subroutine's actual location in memory. In addition to this, several control codes were defined that acted solely as directives controlling the behaviour of the Initial Orders and were not translated into binary EDSAC code.

The final version of the Initial Orders was therefore a sophisticated routine that generated binary code from an input tape language that was an extension of the symbolic representation of EDSAC's code. Once the input tape had been punched, the process of loading and running the program was completely automatic.

The fundamental difference between the two approaches is that Goldstine and von Neumann assumed that binary code would be stored externally and loaded into memory, but the EDSAC team assumed that symbolic code would be stored, and the binary code generated and loaded when required. EDSAC's binary code was not stored, but was generated by the Initial Orders and existed only as transient pulses in EDSAC's delay lines as a program was running. When the load process was extended to incorporate the use of subroutines, both groups simply extended their initial approach. Goldstine and von Neumann assumed that all routines would be given in binary form, which would therefore require *in situ* modification, whereas Wheeler generalized the existing linguistic capabilities of the Initial Orders.

6.4 Diagramming subroutines

As illustrated in Chapter 5 in connection with the meshing and sorting routines, the representation of subroutines in block or flow diagrams turned out to be surprisingly problematic. In the draft *Planning and Coding* report, Goldstine and von Neumann had introduced a special kind of control box to help simplify complicated diagrams:

If either such a group [of orders] is highly complicated or has already been programmed and is in the computer library, we indicate the box as  and then give a more detailed statement of the box outside the main diagram as if it were a footnote. (Goldstine and von Neumann 1946)

The block diagram of the interpolation routine described above used this notation twice to indicate the points at which the subroutine to calculate products was called. No attempt was made to illustrate the process of subroutine call and return: the asterisked boxes were simply placed inline in the interpolation diagram at the points where a product was to be calculated.

This new notation did introduce some hierarchical structure to the block diagram notation. It would be possible to simplify a complex diagram by using an asterisked box to represent a separately drawn ‘subdiagram’. As the quotation above indicates, however, this device was not introduced specifically to represent subroutines, and in fact it failed in several ways to accurately model the semantics of subroutines.

Firstly, the notation took no account of storage. An asterisked box represents only the order words of a subroutine; when its local storage was not simply ignored, the subroutine’s storage box was included in the diagram of the calling routine. This had the advantage of making it possible to show how the main routine would initialize the subroutine’s parameters before calling it, but also introduced scope for confusion and ambiguity.

Secondly, each box in a diagram is supposed to correspond to a particular block of memory locations. Multiple instances of a subroutine box on a diagram therefore suggest that there will be two copies of the subroutine code in the final program. In other words, such diagrams suggest that the subroutine is an ‘open’ subroutine where the subroutine code is physically copied into the main routine, rather than the ‘closed’ subroutines that Goldstine and von Neumann exclusively used in their coding, where the subroutine code appears only once and is separated from the main routine, with control being transferred to and from it when necessary.

A closed subroutine could be represented by simply moving its block diagram out of the diagram of the main routine, but this raised the new problem of how to represent the transfer of control when a subroutine is called. Multiple calls could be represented by simply drawing an arrow leading to the input arrow of the subroutine. It is trickier, however, to show how the subroutine may return control to the different places in the main routine from where it was called. A solution to this problem was found by using the variable remote connections described in Section 5.3. At the point where a subroutine was called, the main routine assigned a specific value – α_1 , say – to a label α and then transferred control to the subroutine box. The subroutine box itself transferred control to a connector α , from where control would transfer to the connector labelled α_1 , as shown in Figure 6.1. This technique was used in the version of the sort routine presented in the draft *Planning and Coding* report, which called the meshing subroutine at two different places.

The variant shown in the lower half of Figure 6.1 was used in a real application in the flow diagram for the Monte Carlo program run on ENIAC in early 1948 (Haigh et al. 2016). This program included a subroutine to calculate pseudo-random numbers which was called at two separate points in the main routine. Connectors ρ linked the points where the subroutine was called to the input of the subroutine box. The random number routine was a single series of instructions and the asterisked form of the operation box was not used. A descendent of this subroutine was illustrated using exactly the same notational conventions in the programming manual for

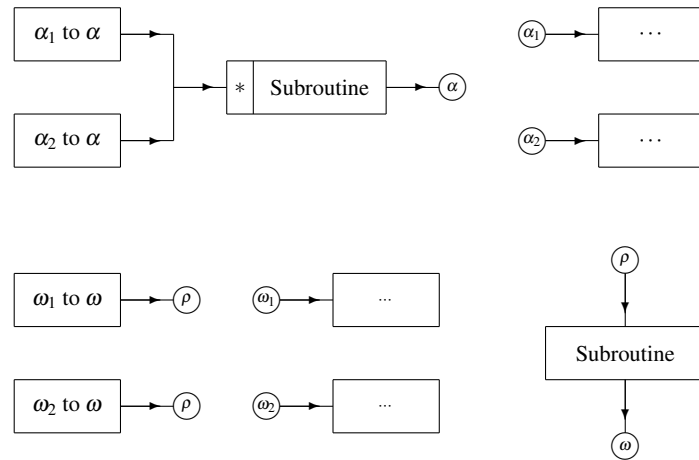


Fig. 6.1 Two ways of representing subroutine call and return using remote connectors. The upper diagram is adapted from (Goldstine and von Neumann 1946) and the lower diagram from the ENIAC Monte Carlo flow diagram of December, 1947 (Haigh et al. 2016).

the MANIAC, the clone of the IAS machine that was built at Los Alamos (Jackson and Metropolis 1954, 240-5).

By late 1947, then, when the Monte Carlo flow diagram was drawn, a practical and semantically plausible technique for graphically depicting the use of closed subroutines had been developed and was being used in practice. It is somewhat surprising, then, that this technique was not adopted for the flow diagram notation or the example routines described in the published version of the *Planning and Coding* reports. The description of the flow diagram notation in the first report does not mention the asterisked boxes of the block diagrams, and provides no other way of introducing hierarchical structure to simplify complex diagrams. Despite promising to ‘proceed from typical parts, which occur usually as parts of larger problems, to increasingly completely formulated self-contained problem’ (Goldstine and von Neumann, 1947-8, vol. 1, 195), there are only a few occasions in the three reports where one coded routine is used as a subroutine in another example.

As illustrated in Section 5.6, on these occasions the subroutine was represented by an inline box that referred to the numbers of the memory locations assigned to the subroutine when it was being coded. These numbers could appear in the flow diagram of the calling routine to show the initialization of the subroutine’s parameters. This seems a poor notational choice: as the location numbers would usually be changed when the subroutine was allocated its final position in a complete routine, the references to these numbers in the main routine would become inconsistent. Goldstine and von Neumann did not discuss, and may not even have noticed, this issue.

The end result is that the flow diagram notation presented in the *Planning and Coding* reports provided no adequate way to represent closed subroutines, despite the fact that Goldstine and von Neumann recognized the use of such routines as a

crucial part of a practical coding technique. This uncertainty is reflected in the text of the reports: in the final *Planning and Coding* report, they wrote that a subroutine is a coded sequence ‘which is formed with the purpose of possible substitution into other routines’ (Goldstine and von Neumann, 1947-8, vol. 3, 2). The flow diagrams they drew certainly suggest such a substitution, but the relationship between the corresponding coded sequences is one of composition rather than substitution.

6.5 Second-class citizens

In two ways, then, the published *Planning and Coding* reports take a step back from fully integrating subroutines into the business of programming. As described above, in the draft report the identification of subroutines occurred as a natural part of the planning part of the process, and the addition of variable remote connections to the block diagram notation provided a way of modelling the transfers of control involved in subroutine call and return. And yet, neither of these topics are discussed in the final version.

The omission of subroutines from the exhaustively presented methodology of Table 5.5 is even more striking than the notational issue. The examples in the draft report demonstrated a fluent and confident use of subroutines: based on the mathematical description of the interpolation routine, a problem-specific subroutine was identified; once the main routine had been coded some general subroutines for fixed point arithmetic were developed, and the interpolation code was then refactored to make use of one of these subroutines. The published method, on the other hand, gives the impression that program development is a linear and deterministic process proceeding smoothly from the mathematical description of a problem to its coded equivalent. There is no trace of the iterative and craftsmanlike engagement with the practice and materials of coding that is vividly documented in the draft report.

The collision between the methodology and the demands of practical software production are visible in the ENIAC Monte Carlo application, development of which got under way shortly after the publication of the first *Planning and Coding* report (Haigh et al. 2016). The documentation of this project clearly illustrates the long series of changes, corrections, enhancements, refactorings and other interventions that transformed von Neumann’s initial flow diagram into code that actually ran. In the course of this process, the interaction of mathematicians and computer operators, among others, created a pidgin version of the complex flow diagram notation that emphasized the flow of control through the program at the expense of keeping track of the changing values of the data stored.

But with the exception of the pseudo-random number routine mentioned above, there were no subroutines in the Monte Carlo program. While there is no doubt that Goldstine and von Neumann attached great importance to the provision of a library of useful and easily reusable subroutines, the reader of the reports is left with the impression that the use of the library exhausts the applicability of subroutines in

programming.² Subroutines are not presented on a par with conditional branches and loops as elementary structures that might be used in program design. They are instead rather self-contained, and their use is exceptional. A hypothetical example of a routine that made use of nine subroutines was described as representing ‘a very high level of complication’, and calls from one subroutine to another were thought ‘likely to be rare’ (Goldstine and von Neumann, 1947-8, vol. 3, 22-3).

In restricting the use of subroutines almost exclusively to the use of a library, Goldstine and von Neumann replicated a division of labour familiar from manual computation in the new arena of software development. Historically, mathematical tables had been produced by groups ranging from the artisanal calculators employed by the UK’s Nautical Almanac Office to the stratified and disciplined workforces employed by Gaspard de Prony after the French Revolution and the Mathematical Tables Project in Depression-era America (Grier 2005), and were used by scientists and mathematicians hoping to relieve themselves of the tedium of calculation.

A very similar division of labour is visible in some early computer installations. For example, Douglas Hartree described how EDSAC’s subroutine library ‘relieves the user of the machine of the greater part of the work of programming calculations in detail’ (Wilkes et al. 1951, xiii). Those users were, of course, the elite scientists of Cambridge University: a glossy booklet produced to celebrate the 75th anniversary of the University’s Computing Laboratory noted that the machine ‘served a large number of Cambridge scientists including some who went on to win Nobel prizes’ (Ahmed 2013, 35). The developers of EDSAC’s subroutine library were not equally well-known or rewarded.

This view of subroutines had a long posterity. It was only in the mid-1950s, and in a context far removed from the traditional scientific uses of the computer, that Allen Newell and Herbert Simon began to explore the practical consequences of the idea of organizing complex programs as structures of interacting subroutines (Priestley 2017). In 1967, computer scientist Christopher Strachey commented that procedures in Algol 60 were ‘second class citizens—they always have to appear in person and can never be represented by a variable or expression’ (Strachey 2000, 32).³ Perhaps this discriminatory treatment had its roots in Goldstine and von Neumann’s erasure of subroutines from the *Planning and Coding* reports 20 years earlier.

² The same perspective appears in the very influential description of the EDSAC programming system in the book by Wilkes et al. (1951).

³ Julian Rohrhuber (2018) has written on the status of functions in a wide range of computational formalisms, and I am grateful to him for drawing my attention to Strachey’s comment. On the face of it an odd description to use of a technical concept in an obscure programming language, the phrase ‘second-class citizens’ had considerable currency in the UK media at the time and was particularly associated with the treatment of immigrant communities such as the ‘Kenyan Asians’ (Winder 2005).

Chapter 7

Contexts and conclusions

There can be no question about the historical significance of von Neumann's work on programming described in this book. And yet in many ways his achievement represents the translation of existing practices into a new environment rather than the invention of a completely new way of doing things. If his work was, in the words of Arthur Burks, 'revolutionary', it was a curiously conservative kind of revolution, more 1688 than 1917.

In part this can be explained by his background. After giving up active work in logic at the start of the 1930s, von Neumann increasingly came to identify himself as an applied mathematician. He did not actively engage in the logicians' effort to define 'effective calculability'; in contrast, his interest in computation was largely instrumental, as reflected in his extensive range of scientific consultancy positions, a tendency amplified by the demands of his war work. His approach to the design and use of the EDVAC code was strongly shaped by the detailed knowledge and experience he had gained of the existing range of automatic computing machinery.

This chapter summarizes and places in a wider context some of the important historiographical and technical themes that have arisen in earlier chapters.

7.1 The genesis of the general purpose computer

Reflecting on his first sorting program in May 1945, von Neumann commented:

I think, however, that it is legitimate to conclude already on the basis of the now available evidence, that the EDVAC is very nearly an 'all purpose' machine (von Neumann 1945e)

Slippage between 'all purpose' and Turing's description of certain machines as 'universal' has lent support to an oft-told story about the invention of the computer. In George Dyson's avowedly mythopoeic account, the 'stored-program computer' was 'conceived by Alan Turing and delivered by John von Neumann', a tale in which the Institute for Advanced Study features as an unlikely surrogate mother (Dyson 2012, ix).

Von Neumann used the term frequently in 1945. In January, for example, he described ENIAC as ‘an absolutely pioneer venture, the first complete, automatic, all-purpose digital electronic computer’ (von Neumann 1945g), in the *First Draft* he mentioned the desire to make EDVAC ‘as nearly as possible *all purpose*’ (von Neumann 1945a, 3), and in November he characterized the proposed IAS machine in almost the same terms as ENIAC, as ‘a fully automatic, digital, all-purpose electronic computing machine’ (von Neumann 1945h). For von Neumann, the class of all-purpose machines was not limited to those like EDVAC and the IAS machine that held numbers and orders in a single large memory, but could include machines with very different designs, such as ENIAC.

Instead, von Neumann used the term to draw a boundary between automatically sequenced machines and those, often analogue, which could only deal with specific problems or forms of equations. In October, IAS director Frank Aydelotte wrote to von Neumann seeking reassurance that the Institute’s machine would be faster than the new MIT differential analyzer that had just been announced in the newspapers; von Neumann replied that:

The electronic device which we are planning would of course surpass it in speed, precision and flexibility (all-purpose character) [...] I would like to repeat that those problems which offer the most interesting and important uses for future computing machines, and in particular for the device which we plan, cannot be done at all on any differential analyzer with practicable characteristics. (von Neumann 1945b).

In von Neumann’s view, these problems were almost exclusively mathematical and scientific: from the beginning of his association with the EDVAC project he had described it as a scientific instrument, as he did later with the IAS machine. In particular, he emphasized the machines’ ability to make the numerical solution of non-linear partial differential equations feasible for the first time (Priestley 2018).

This all-purpose character had a very specific technical source. As Grace Hopper and Howard Aiken put it, ‘numerical methods [have reduced] the processes of mathematical analysis to a sequence of the five fundamental operations of arithmetic: addition, subtraction, multiplication, division and reference to previously computed results’ (Harvard 1946, 10). As many people from Babbage onwards had realized, a machine that could automatically carry out specified sequences of those operations could in principle compute anything. After describing the features that would give the Bell Labs machine the ability to do just that, for example, Samuel Williams noted that ‘all of these functions being orderly controlled by a tape, will provide the flexibility required for a universal system’ (Williams 1944, 1).

Von Neumann’s description of EDVAC as ‘all purpose’, then, does not describe it as a new kind of machine but rather places it in a particular lineage. Following the team’s realization in April 1945 that computing and sorting could be, and indeed for practical reasons had to be, combined in a single machine, the discovery that EDVAC could be applied to sorting was interesting and reassuring but did not lead von Neumann or anyone else to suppose that a markedly new type of machine had been invented. A computer did not have to be able to sort to be called ‘all-purpose’, and the ability to sort did not promote EDVAC to a new category of machine.

The idea of a ‘lineage’ evokes philosopher of technology Gilbert Simondon’s (2012) observation that technical objects acquire their individuality not from their use but from their ‘genesis’, the course of technological evolution leading up to them. He identified ‘concretization’ as a crucial process in this evolution whereby the components of a technical object acquire multiple functions in an overlapping causal field, such as the cooling fins on the cylinder of an engine which combine the functions of structural support and heat dispersion.

The specificity of the modern computer is founded on an act of concretization in Simondon’s sense. As the *First Draft* made clear, EDVAC’s memory combined capabilities that had been kept distinct in earlier automatically sequenced machines. By being able to represent both numbers and orders \mathcal{M} bound together functions which had been performed by separate devices in earlier machines. This initiated a more specific line of technological enquiry that was very prominent in the early years of computer development, namely the search for a storage medium capable of economically combining the linear access characteristic of instructions read from tape with the random access needed to stored numbers.

The unification of memory also opened up new possibilities for programming, exemplified by von Neumann’s use of substitution in EDVAC’s code. But as with the developments in hardware, these represented not so much a new beginning as an evolution of existing practice.

7.2 The history of programming

Historians of computing sometimes give the impression that the development of programming was dependent on the development of the so-called ‘stored-program computer’, implying that programming could only happen—indeed, could only be thought—after the creation of the modern computer. For example, Martin Campbell-Kelly (2011) has described programming as being ‘invented’ in the period between 1947 and 1951, presenting the story as one of transition from the ‘theory’ stated in the *Planning and Coding* reports to the ‘practice’ made possible once machines like EDSAC were available for use.

Campbell-Kelly’s argument hinges on two observations. The first is that in 1947 plans for the IAS machine’s input and output facilities were still highly fluid and Goldstine and von Neumann therefore chose to describe only those parts of their example computations that took place in the computer’s central units. Secondly, the task of describing computations in such a way that they could be carried out by machines rather than human computers turned out to be surprisingly difficult. Without the experiential feedback gained from running complete programs on a functioning machine, the *Planning and Coding* reports could only offer a partial account of the task of programming.

As valid as these points are, this account assumes that the novelty of the EDVAC proposals makes it reasonable to ignore the substantial practical experience that had been gained before 1949 of running computations on automatically sequenced

machines. Large calculations had been performed on Mark I and ENIAC, and the designers and operators of both machines were well aware of the importance and the difficulty of introducing data into the machine and having results produced in a way that would be useful to users. In the course of this experience, the difficulty of getting the instructions right had been noticed. Reflecting on his experience in setting up a problem on ENIAC in 1946, Douglas Hartree (1949, 92-3) talked about the surprising difficulty of programming and the need to take the ‘machine’s-eye view’ when planning a computation, and no doubt the Mark I programmers had had similar experiences.

So while the text of the *Planning and Coding* reports might indeed be described as theoretical and incomplete, they should be read in the broader context of ongoing experimental and production coding work on automatically sequenced machines, including machines which have attracted little attention from historians, such as the Harvard Mark II and IBM’s Selective Sequence Electronic Calculator (SSEC). Only if the work carried out on these earlier machines is considered to be something less than ‘programming’ does it make sense to talk about programming being invented between the *Planning and Coding* reports and EDSAC.

The use of the term ‘programming’ in computing originated with the ENIAC project (Grier 1996). In a natural extension of its everyday use in connection with such things as concert programmes, it referred to the activity of determining which sequences of operations would be carried out. The agent doing the programming could vary: the ‘programming circuits’ of ENIAC’s units controlled the operation of their arithmetic circuits, while the machine’s multiplier was described as ‘automatically programming’ the accumulators to carry out the sequences of additions and transfers involved in performing a multiplication. By 1945, usage had expanded and the human activity of preparing instructions for a machine was also described as programming; gradually, the instructions rather than the operations became the machine’s ‘program’.

Of greater significance than these linguistic observations, however, is the fact that the problem to which programming, in its various senses, is the solution—controlling the behaviour of automatically sequenced machines—is wider than and predates the use of machines of EDVAC’s type. The history of programming, in other words, is not coextensive with the history of programming so-called ‘stored-program’ machines. While ‘programming’ acquired its modern sense in connection with those machines, the activity it denotes has a much wider currency. Even if their own terminology differed, Charles Babbage and Ada Lovelace can reasonably be described as thinking about the programming of the Analytical Engine, Leslie Comrie as having programmed the National Accounting Machines, and Richard Bloch and Grace Hopper as writing programs for Mark I. Von Neumann himself offers some warrant for this kind of mild anachronism: when writing to Aydelotte, he introduced the word ‘program’ as a synonym of the older ENIAC term ‘set up’ (von Neumann 1945b).

In this perspective, von Neumann’s work on EDVAC’s logical control represents a development *within* the history of programming, and the *Planning and Coding* reports can indeed be described as ‘a kind of feasibility study [that] enabled other

computer groups to press ahead without any nagging doubts concerning the technical soundness of the stored-program concept' (Campbell-Kelly 2011, 27). The new style of programming should not be identified with the new architecture, however. While von Neumann's innovations were developed in the context of, and no doubt inspired by, the plans for EDVAC's memory, they are independent of any particular hardware configuration, both in theory and in practice. The first programs written in the new style to be executed actually ran on the architecturally obsolescent ENIAC in early 1948, as discussed below.

7.3 Planning

As signalled in the very title of the *Planning and Coding* reports, von Neumann split the task of program preparation into two major phases, the first concerned with understanding the computation that was to be carried out and the second with expressing that computation in way suitable for machine interpretation. These final sections offer some concluding reflections on von Neumann's ideas about planning and coding, and their legacy.

The methodology articulated in the reports can be understood as an extension of conventional mathematical practice (Priestley 2018). ENIAC and Mark I project reports described three-stage processes echoing the division of labour set up by de Prony at the end of the eighteenth century. Mathematical analysis of the problem was followed by the preparation of a detailed computing plan which was passed to a computer, human or mechanical, for execution. The process outlined in Table 5.5 elaborates this simple process with details of the syntactic work necessary to produce machine-readable code, but its basic structure remains unchanged.

Von Neumann's aim seems to have been to establish a predictable linear process for the development of code. As the previous chapter described, some of the more intuitive and iterative examples of coding found in the draft *Planning and Coding* report were removed in the published version in favour of a presentational style that emphasized the quasi-deductive nature of the work. At the same time, the flow diagram notation was extended with substitution and assertion boxes so that the step-by-step progress of the computation could be expressed in mathematical terms. The goal was completeness and predictability, an attitude well expressed by Wilkes et al. (1951, 1) who wrote that in a program for a machine, 'every contingency must be foreseen'.

There is a tension, however, between the prescriptive nature of this process and the realities of software construction. A trivial example can be found in the definition of the meshing operation, which was originally defined in terms of sorting. Von Neumann found it convenient to reverse this dependency in the code, however, and in the final *Planning and Coding* report the mathematical definition of meshing was adjusted so as to remove its apparent dependency on sorting. A more substantial example can be found in the use of flow diagrams in the development of the ENIAC Monte Carlo application (Haigh et al. 2016).

In the example flow diagrams in the *Planning and Coding* reports, a strikingly large proportion of the space taken up by the diagrams on the page is occupied by storage boxes. As much importance is given to the storage and substitution boxes that document the effect of the program as to the control boxes that describe the control flow and operations. When von Neumann came to draw up the initial flow diagram for the Monte Carlo problem, he followed the official notation closely, but there is a striking difference in the use of storage boxes. In the examples in the reports, every change of value of a variable or storage location is documented. In the Monte Carlo diagram, by contrast, storage boxes are much less prominent and in many cases simply repeat an assignment of variable to storage location that is clearly stated in a preceding operation box. The imbalance is even more striking in the final diagram, produced in December 1947. Of the 90 or so boxes on the diagram, only around a dozen are storage or substitution boxes, the remainder being either operation or alternative boxes. This gives a dramatic illustration of how, in non-pedagogic uses of the notation, the procedural modelling of control flow and operations was quickly privileged over the declarative description of the effect of those operations.

Nevertheless, the published form of the notation and methodology was widely circulated and used for pedagogic purposes, as the examples given by Jackson and Metropolis (1954) illustrate, and it was only in the mid-1950s that an influential counter-proposal was first articulated, when several of the participants at the famous 1956 Dartmouth conference on AI took issue with very idea of planning as a suitable model for programming. As Nathaniel Rochester put it, ‘one ordinarily provides the machine with a set of rules to cover each contingency which may arise [but] if the machine had just a little intuition or could make reasonable guesses, the solution of the problem could be quite direct’ (see Priestley 2017).

7.4 Coding

The EDVAC approach quickly achieved dominance. Following the Moore School lectures in the summer of 1946, the overwhelming majority of new computer projects adopted an EDVAC-type design and its associated style of coding, the most notable exception being the plans for the ACE produced by Alan Turing at the UK’s National Physical Laboratory.

In a striking example of the autonomy of programming, however, programs were running in an EDVAC-style code some time before any machines based on the new architecture had been completed. ENIAC was largely out of action in 1947 as it was moved out of the Moore School, but almost as soon as it was recommissioned at BRL it was set up to run programs written in a code that included conditional branching and, within the limits of its hardware capabilities, address modification, nicely validating Turing’s observation that ‘computing processes [...] can all be done with one digital computer, suitably programmed’ (Turing 1950, 441-2). This

code was first used for the Monte Carlo application which ran in March and April, 1948 (Haigh et al. 2016).

This conversion had an unanticipated consequence. In 1946 ENIAC was the only operational electronic computer, and as well as running a number of conventional mathematical applications it served as a test-bed and a stimulus for exploring and thinking about different approaches to computation. The mathematician Derrick Lehmer was, like Curry, a member of BRL's Ballistic Computations Committee, and over a holiday weekend used ENIAC to explore a problem in number theory. His set-up, as reconstructed by historians Maarten Bullynck and Liesbeth de Mol (2010), made essential use of ENIAC's capabilities for parallel computation. As a result of his detailed engagement with the ENIAC in 1946, Curry himself came to a novel understanding of the structure of its programs. He later developed his insights into a general theory of program composition, which he described as 'sufficiently different' from Goldstine and von Neumann's views on subroutines that 'it is not immediately obvious just how the two investigations should be put together' (Curry 1950).¹

But the conversion put an end to all that. Its imposition limited the machine's capabilities and from that point on ENIAC ran only one set-up, the implementation of the conversion code. Large areas of the machine were mothballed, such as the master programmer which had essentially no role in the post-conversion economy. Although the conversion significantly decreased ENIAC's rate of computation, the ease of programming that it offered must have made it seem an attractive trade-off. In early Cold War America, the priority was to put ENIAC to work.

Substitution is the *leitmotiv* of von Neumann's work on programming. It appears in many different contexts and with many different meanings including, in roughly chronological order, the following.

1. The fundamental operation that \mathcal{C} carries out when storing a word in \mathcal{M} is called substitution (see Section 3.4).
2. Von Neumann's top-down approach to coding is to write a routine in an extended code that includes variables, and then systematically to replace those variables by actual values. He does not explicitly describe this process as substitution, but the replacement of variables by constant terms in this way is the classic logical sense of the term (see Section 4.5).
3. Both the block diagram and flow diagram notations included 'substitution boxes' to model some of the changes taking place as a routine executes (see Sections 5.2 and 5.5).
4. The preparatory routine that adjusts the code of a subroutine to be consistent with its actual position in memory was originally called the 'substitution sequence' (see Section 6.3).
5. The relationship between a main routine and a subroutine is described, rather problematically, as one of substitution (see Section 6.4).

¹ De Mol et al. (2015) describe Curry's proposals in detail, using them as a platform from which to launch a critique of Goldstine and von Neumann's work in the *Planning and Coding* reports.

Once he had conceived of an operation that would replace only part of an order word, it is not surprising that von Neumann, with his background in logic, would have seen in it an analogy with substitution. It is less predictable that he would take substitution as the fundamental notion to describe *all* transfers of information to or within \mathcal{M} . In logic, substitution is a metasyntactic operation providing a means of transforming one formula into another, however, and such a transformation takes place in EDVAC's memory at every step of a computation. In a symbolic model of computation, substitution provided a unified way of modelling change.

Code—which for Goldstine and von Neumann always included numbers as well as orders—was not a fixed entity, but just the starting point of a dynamic process which would modify the contents of memory in various and unpredictable ways, even constructing new orders for the machine to execute. It was part of a continuum of symbolic representations of a computational process stretching from traditional notations of mathematics, through coded symbols ready to be typed directly into memory, to the binary forms of those symbols held in memory and evolving as computation progressed. Goldstine and von Neumann's process described notations and techniques suitable for recording and verifying the effects of the decisions taken by programmers and they seem to have found substitution in its broader senses a natural way to describe many of the transformations involved.

In seeking to define what he thought of as a new type of logical activity, von Neumann reached for the tools and techniques familiar to him from his experience as an active logician in the 1920s. Strikingly, at the end of that decade Curry (1929) had published an analysis of substitution which noted that the 'complexity of this process is manifest'. Following earlier work of Moses Schönfinkel, he proposed to eliminate it in favour of a simpler combinatory mechanism. Like von Neumann, then, Curry reached for the familiar when faced with the challenge of theorizing programming, and he accurately described his work of the late 1940s as a 'program composition technique'.

A third migrant from logic to computation brought with him different baggage. Turing encountered logic in the 1930s in the context of recursive function theory which emphasized the bottom-up composition of a handful of primitive functions to define more complex operations. This influence of this way of thinking is visible both in his famous 1936 paper and also in his thoughts on programming, where the role of subroutines as the building blocks of programs is quite different from the view described in Chapter 6.

The general purpose computer was a technological development, a response to specific and localizable practical demands such as those that led to the unification of sorting and computing described in Chapter 2. The road from logic to practical computation led across an unfamiliar terrain, and von Neumann was one of several making the crossing, each breaking their own trail and hopeful of the opportunities to be found in the land beyond. Even if it is now somewhat out of fashion, the work he was involved with in the mid-1940s formed and enriched many aspects of the field of computing, laying a foundation for much of what was to come.

Appendix A

Von Neumann's second EDVAC code

Abstract This appendix reproduces the text of a manuscript preserved in box 20 of the Herman H. Goldstine papers at the American Philosophical Society in which von Neumann defines a code for an EDVAC architecture using short delay lines. Von Neumann mentioned this topic in a letter to Haskell Curry on 20 August, 1945, and a version of the code is described in Eckert and Mauchly's EDVAC progress report, dated 30 September, 1945. This text can therefore be dated with some confidence to the late summer of 1945. A small number of obvious slips in the manuscript have been silently corrected.

1. For $x, y = 0, 1, 2, \dots, \xi = 0, 1, \dots, 2^5 - 1$, x determines y, ξ uniquely by virtue of the relation

$$x = 2^5 y + \xi.$$

Define accordingly

$$y = Qx, \xi = Rx.$$

2. Time is measured by a variable $t = 0, 1, 2, \dots$. Each t stands for the time interval $t, t + 1$. The length of this unit interval is in conventional units $2^5 \tau$, where τ is a pulse-time: $\tau \sim 10^{-6}$ sec.

3. *Words.*

A *word* is a sequence of 2^5 *pulses*. Each pulse occupies a pulse-time (τ), hence a word occupies a time unit of $2 \cdot (2^5 \tau)$. The memory consists of

$$C = 2^5 A + B + 1$$

words, which are enumerated as follows:

- 1) $2^5 A$ words Wx , $x = 0, 1, \dots, 2^5 A - 1$.
(x is to be taken mod $2^5 A$.)
- 2) B words $W\bar{z}$, $z = 0, 1, \dots, B - 1$.
(z is to be taken mod B .)

3) 1 word σ .

Actually $A \sim 2^8$, $B \sim 2^5$ to 2^6 . We assume that

$$A \leq 2^8, B \leq 2^6.$$

4. *Gates.*

The device contains

$$D = A + B + 1$$

gates, which are enumerated as follows:

- 1) A gates Gy , $y = 0, 1, \dots, A - 1$.
(y is to be taken mod A .)
- 2) B gates $G\bar{z}$, $z = 0, 1, \dots, B - 1$.
(z is to be taken mod B .)
- 3) 1 gate σ .

We must now discuss the following matters: *Access, Control, Operation, Substitution.*

5. *Access.*

At each time t each gate has *access* to exactly one word. Specifically:

- 1) At each time t gate Gy has access to word $x = 2^5y + Rt$. (I.e. $Qx = y$, $Rx = Rt$.)
- 2) Gate $G\bar{z}$ has always access to word $W\bar{z}$.
- 3) Gate σ has always access to word σ .

6. This is a list of all possible words, or rather of the written symbols which denote them:

- | | | |
|-----------------------------|-------------------------------------|--------------------------------|
| 0) 0 | 3) $\bar{z}' \rightarrow x' r$ | 7) $\bar{z}' \omega \bar{z}''$ |
| 1) $x' \rightarrow C$ | 4) $x' \rightarrow \bar{z}' r$ | 8) $\rightarrow \bar{z}' r$ |
| 2) $\bar{z}' \rightarrow C$ | 5) $\bar{z}' \rightarrow \bar{z}''$ | 9) $\bar{z}' \rightarrow r$ |
| | 6) $\sigma \rightarrow \bar{z}''$ | 10) $\mathcal{N}\xi$ |

Here x' is a 14-binary-digit integer; \bar{z}' , \bar{z}'' are 6-binary-digit integers; r is a 5 digit binary integer, $r = 0$ being omissible; ω is one of the symbols enumerated in 8.; ξ is a 30-binary-digit fraction with sign, $-1 \leq \xi < 1$.

7. *Control.*

Assume that at time t the control organ C is connected to $\left\{ \begin{array}{l} Gy, \text{ i.e. to } Wx \text{ with } \\ G\bar{z}, \text{ i.e. to } W\bar{z} \text{ ----} \end{array} \right\}$
 $\left\{ \begin{array}{l} x = 2^5y + Rt, \text{ whence } y = Qx, \quad Rt = Rx \\ \text{-----} \end{array} \right\}$. Then the following events will take place:

$\left\{ \begin{array}{l} Wx \\ W\bar{z} \end{array} \right\}$ is the word:	Events:
0) 0	\mathcal{C} disconnects, and at time $t + 1$ connects to $\left\{ \begin{array}{l} GQ(x+1), \text{ i.e. to } W(x+1) \\ G\bar{z}+1, \text{ i.e. to } W\bar{z}+1 \end{array} \right\}$.
1) $x' \rightarrow \mathcal{C}$	\mathcal{C} disconnects. Waiting until the first $t' > t$ with $Rt' = Rx'$. Then \mathcal{C} connects to GQx' , i.e. to Wx' .
2) $\bar{z}' \rightarrow \mathcal{C}$	\mathcal{C} disconnects, and at time $t + 1$ connects to $G\bar{z}'$, i.e. to $W\bar{z}'$.
3) $\bar{z}' \rightarrow x' \mid r$	\mathcal{C} disconnects. Waiting until the first $t' > t$ with $Rt' = Rx'$. At each time $t' + s$, $s = 0, 1, \dots, r$, for the duration of that time unit, $GQ(x' + s)$ connects to $G\bar{z}' + s$, i.e. $W(x' + s)$ to $W\bar{z}' + s$. At that time $W\bar{z}' + s$ is substituted into $W(x' + s)$. After the $s = 0, 1, \dots, r$ have been exhausted, $\left\{ \begin{array}{l} \text{waiting until the first } t'' > t' + r \\ \text{time } t'' = t' + r + 1 \text{ follows.} \dots \end{array} \right\}$ $\left\{ \begin{array}{l} \text{with } Rt'' = R(t + 1). \text{ (This is } t'' + 2^5 + 1 \text{ or} \\ \text{-----} \\ \text{ } t'' + 2^6 + 1.) \end{array} \right\}$ Then \mathcal{C} connects to $\left\{ \begin{array}{l} GQ(x+1) \\ G\bar{z}+1 \end{array} \right\}$, i.e. to $\left\{ \begin{array}{l} W(x+1) \\ W\bar{z}+1 \end{array} \right\}$.
4) $x' \rightarrow \bar{z}' \mid r$	Same as 3), except that now $W(x' + s)$ is substituted for $W\bar{z}' + s$.
5) $\bar{z}' \rightarrow \bar{z}''$	\mathcal{C} disconnects. $G\bar{z}'$ connects with $G\bar{z}''$, i.e. $W\bar{z}'$ with $W\bar{z}''$. $W\bar{z}'$ is substituted into $W\bar{z}''$. Then, at time $t + 1$, \mathcal{C} is reconnected as in 0).
6) $\sigma \rightarrow \bar{z}''$	Same as 5), except that $G\bar{z}'$ and $W\bar{z}'$ are replaced by σ and σ .
7) $\bar{z}' \omega \bar{z}''$	\mathcal{C} disconnects. $G\bar{z}'$, $G\bar{z}''$, i.e. $W\bar{z}'$, $W\bar{z}''$, connect with the inputs of the arithmetical organ \mathcal{A} , the output of \mathcal{A} is σ . Between these three \mathcal{A} performs the operation ω as described in 8. Then $G\bar{z}'$, $G\bar{z}''$ disconnects. Waiting until the first $t' \geq$ the completion of these operations $\left\{ \begin{array}{l} \text{for which } Rt' = R(t + 1) \\ \text{-----} \end{array} \right\}$. Then \mathcal{C} connects to $\left\{ \begin{array}{l} GQ(x+1) \\ G\bar{z}+1 \end{array} \right\}$, i.e. to $\left\{ \begin{array}{l} W(x+1) \\ W\bar{x}=1 \end{array} \right\}$.

- 8) $\rightarrow \bar{z}' \mid r$ \mathcal{C} disconnects. At each time $t + 1 + s$, $s = 0, 1, \dots, r$, for the duration of that time unit, $\left\{ \frac{GQ(x+1+s)}{Gz+1+s} \right\}$ connects to $G\bar{z}' + s$, i.e. $\left\{ \frac{W(x+1+s)}{Wz+1+s} \right\}$ to $W\bar{z}' + s$. At that time $\left\{ \frac{W(x+1+s)}{Wz+1+s} \right\}$ is substituted into $W\bar{z}' + s$. After the $s = 0, 1, \dots, r$ have been exhausted, time $t + 2 + r$ follows. Then \mathcal{C} connects to $\left\{ \frac{GQ(x+2+r)}{Gz+2+r} \right\}$, i.e. to $\left\{ \frac{W(x+2+r)}{Wx+2+r} \right\}$.
- 9) $\bar{z}' \rightarrow \mid r$ Same as 8), except that now $W\bar{z}' + s$ is substituted into $\left\{ \frac{W(x+1+s)}{Wz+1+s} \right\}$.
- 10) $\mathcal{N}\xi$ Same as 0).
-

8. Operation.

When ω is called in by 7) in 7., $W\bar{z}'$, $W\bar{z}''$ must be of the form 10) in 7.: $\mathcal{N}\xi'$, $\mathcal{N}\xi''$. σ is automatically of the form 10) in 7.: $\mathcal{N}\xi^*$. Now ω operates as follows:

Symbol	Replaces the ξ^* in the $\mathcal{N}\xi^*$ of σ by:
C1) +	$\xi' + \xi''$
C2) -	$\xi' - \xi''$
C3) \times	$\xi' \times \xi''$
C4) \div	ξ' / ξ''
C5) $\sqrt{\quad}$	$\sqrt{\xi'}$
C6) db	The decimal equivalent of ξ' .
C7) bd	The binary equivalent of the decimal interpretation of ξ' .
H1) + _h	$\xi' + \xi'' + \xi^*$
H2) - _h	$\xi' - \xi'' + \xi^*$
H3) \times _h	$\xi' \xi'' + \xi^*$
H4) s	$\begin{cases} \xi' & \text{for } \xi^* \geq 0 \\ \xi'' & \text{for } \xi^* < 0 \end{cases}$

9. *Substitution.*

When a word w' is substituted into a word w'' , the following will take place: (The categories 0) – 10) are those enumerated in 6. and 7.)

- A) w' is a word 0) – 9) (i.e. anything other than a word 10): $\mathcal{N}\xi$):
 w'' is replaced by w' in its entirety.
- B) w' is a word 10): $\mathcal{N}\xi$):
- Ba) w'' is a word 0) or 10):
 w'' is replaced by w' in its entirety.
- Bb) w'' is a word 1):
 x' is replaced by digits 17–30 of ξ .
- Bc) w'' is a word 2):
 z' is replaced by digits 11–16 of ξ .
- Bd) w'' is a word 3) or 4):
 x' is replaced by digits 17–30 of ξ ,
 z' is replaced by digits 11–16 of ξ ,
 r is replaced by digits 6–10 of ξ .
- Be) w'' is a word 8) or 9):
 z' is replaced by digits 11–16 of ξ .
- Bf) w'' is a word 5), 6) or 7):
 No replacement is made.

Due to Bb) – Be) the 14-, 6-, 5- binary digit integers x' , z' , r appear in the replacing ξ as fractions $2^{-30}x'$, $2^{-16}z'$, $2^{-10}r$. In order to abbreviate, define

$$\overline{/}x' = 2^{-30}x', \quad \overline{/}z' = 2^{-16}z', \quad \overline{/}r = 2^{-10}r.$$

Appendix B

Von Neumann's meshing routine manuscript

- (1) A $p+1$ -complex: $X^{(p)} = (x^0; x^1, \dots, x^p)$ consists of the *main number*: x^0 , and the *satellites*: x^1, \dots, x^p . Throughout what follows $p = 1, 2, \dots$ will be fixed. A complex $X^{(p)}$ *precedes* a complex $Y^{(p)}$: $X^{(p)} \leq Y^{(p)}$, if their main numbers are in this order: $x^0 \leq y^0$.

An n -sequence of complexes: $\{X_0^{(p)}, \dots, X_{n-1}^{(p)}\}$.

If $0', \dots, (n-1)'$ is a permutation of $0, \dots, (n-1)$, then the sequence $\{X_{0'}^{(p)}, \dots, X_{(n-1)'}^{(p)}\}$ is a *permutation* of the sequence $\{X_0^{(p)}, \dots, X_{n-1}^{(p)}\}$. A sequence $\{X_0^{(p)}, \dots, X_{n-1}^{(p)}\}$ is *monotone* if its elements appear in their order of precedence: $X_0^{(p)} \leq X_1^{(p)} \leq \dots \leq X_{n-1}^{(p)}$, i.e. $x_0^0 \leq x_1^0 \leq \dots \leq x_{n-1}^0$.

Every sequence $\{X_0^{(p)}, \dots, X_{n-1}^{(p)}\}$ possesses a monotone permutation: $\{X_{0'}^{(p)}, \dots, X_{(n-1)'}^{(p)}\}$ (at least one). Obtaining this monotone permutation is the operation of *sorting* the original sequence.

Given two (separately) monotone sequences $\{X_0^{(p)}, \dots, X_{n-1}^{(p)}\}$ and $\{Y_0^{(p)}, \dots, Y_{m-1}^{(p)}\}$, sorting the composite sequence $\{X_0^{(p)}, \dots, X_{n-1}^{(p)}, Y_0^{(p)}, \dots, Y_{m-1}^{(p)}\}$ is the operation of *meshing*.

- (2) We wish to formulate code instructions for sorting and for meshing, and to see how much control-capacity they tie up and how much time they require. It is convenient to consider meshing first and sorting afterwards.
- (3) Consider the operation of meshing the two (separately) monotone sequences $\{X_0^{(p)}, \dots, X_{n-1}^{(p)}\}$ and $\{Y_0^{(p)}, \dots, Y_{m-1}^{(p)}\}$.

A natural procedure to achieve this is the following one:

Denote the meshed sequence by $\{Z_0^{(p)}, \dots, Z_{n+m-1}^{(p)}\}$. Assume that the l first elements $Z_0^{(p)}, \dots, Z_{l-1}^{(p)}$ have already been formed, $l = 0, 1, \dots, n+m$. Assume that they consist of the n' (m') first elements of the X - (Y -) sequence:

$X_0^{(p)}, \dots, X_{n'-1}^{(p)}$ and $Y_0^{(p)}, \dots, Y_{m'-1}^{(p)}$, with $n' = 0, 1, \dots, n$ and $m' = 0, 1, \dots, m$ and $n' + m' = l$.

Then the procedure is as follows:

- (α) $n' < n, m' < m$:
Determine whether $x_{n'}^0 \leq$ or $> y_{m'}^0$.
 - (α_1) $x_{n'}^0 \leq y_{m'}^0$: $Z_l^{(p)} = X_{n'}^{(p)}$,
replace l, m', n' by $l + 1, m', n' + 1$.
 - (α_2) $x_{n'}^0 > y_{m'}^0$: $Z_l^{(p)} = Y_{m'}^{(p)}$,
replace l, m', n' by $l + 1, m' + 1, n'$.
- (β) $n' < n, m' = m$:
Same as (α_1).
- (γ) $n' = n, m' < m$:
Same as (α_2).
- (δ) $n' = n, m' = m$:
The process is completed.

(4) In carrying out this process, the following observations apply:

- (a) The process consists of steps which are enumerated by the index $l = 0, 1, \dots, n + m$. It begins with $l = 0$, ends with $l = n + m$, and l increases by 1 at every step — hence there are $n + m + 1$ steps.
- (b) Each step is characterised not only by its l , but also by its n', m' . Since $l = n' + m'$, it is preferable to characterise it by n', m' alone, and to obtain l from the above formula. Thus the process begins with $(n', m') = (0, 0)$, ends with $(n', m') = (n, m)$, and at every step either n' or m' increases by 1 while the other remains constant.
- (c) At the beginning of every step it is necessary to sense which of the 4 cases (α) – (δ) of (3) holds. (δ) terminates the procedure. (β), (γ) are related to (α): Indeed (β), (γ) correspond to (α_1), (α_2). Hence in the cases (β), (γ) one may replace $x_{n'}^0, y_{m'}^0$, (when they are being inspected) by 0, 0 or 0, -1 and then proceed as in (α).
- (d) At the end of (α) (i.e. (α_1) or (α_2), by (c) equally for (β) or (γ)) the complex $X_{n'}^{(p)}$ ($Y_{m'}^{(p)}$) must be placed in the position of the complex $Z_l^{(p)}$. This amounts to transferring the elements of $X_{n'}^{(p)}$ ($Y_{m'}^{(p)}$), i.e. since x_n^0 (y_m^0) is already available, it amounts to transferring $x_{n'}^1, \dots, x_{n'}^p$ ($y_{m'}^1, \dots, y_{m'}^p$). This is an unbroken sequence of p elements, followed by $x_{n'+1}^0$ ($y_{m'+1}^0$). At the next step, $x_{n'}^0$ ($y_{m'}^0$) will have to be replaced (for the next inspection) by $x_{n'+1}^0$ ($y_{m'+1}^0$), hence it is simplest to transfer at this point a sequence of $p + 1$ elements, i.e. $x_{n'}^1, \dots, x_{n'}^p, x_{n'+1}^0$ ($y_{m'}^1, \dots, y_{m'}^p, y_{m'+1}^0$).
- (e) The arrangement made at the end of (d) implies, that for $l \neq 0$ the quantities to be inspected for the step l , i.e. $x_{n'}^0, y_{m'}^0$, are already available at the beginning

of the step. In the interest of homogeneity it is therefore desirable to have the same situation at the beginning of the step $l = 0$, i.e. $(n', m') = (0, 0)$. Hence the step $l = 0$ must be preceded by a preparatory step, say step —, which makes x_0^0, y_0^0 available.

(5) The remarks of (4) define the procedure more closely. Specifically:

- (f) At the beginning of a step, say step (n', m') , the following quantities must be available, i.e. placed into short tanks: $n', m', x_{n'}^0, y_{m'}^0$. Denote the short tanks containing these quantities by $\overline{1}_1, \overline{2}_1, \overline{3}_1, \overline{4}_1$. Now the first operation must turn about determining which of the cases $(\alpha) - (\delta)$ holds. This consists in determining which of $n' - n, m' - m$ are ≥ 0 or < 0 . Hence n, m , too must be available, say in the short tanks $\overline{5}_1, \overline{6}_1$. According to which of the 4 cases holds, \mathcal{C} must be sent to the place where its instructions begin, say the (long tank) words $1_\alpha, 1_\beta, 1_\gamma, 1_\delta$. Their numbers must be available, i.e. in short tanks, say in the short tanks $\overline{7}_1, \overline{8}_1, \overline{9}_1, \overline{10}_1$. Finally, the order which will send \mathcal{C} to $1_\alpha - 1_\delta$ must be in a short tank, say in the short tank $\overline{11}_1$.
- (g) We now formulate a set of instructions to effect this 4-way decision between $(\alpha) - (\delta)$. We state again the contents of the short tanks already assigned:

$$\begin{array}{cccc} \overline{1}_1) \mathcal{N}n'_{(-30)} & \overline{2}_1) \mathcal{N}m'_{(-30)} & \overline{3}_1) \mathcal{N}x_{n'}^0 & \overline{4}_1) \mathcal{N}y_{m'}^0 \\ \overline{5}_1) \mathcal{N}n_{(-30)} & \overline{6}_1) \mathcal{N}m_{(-30)} & \overline{7}_1) \mathcal{N}1_{\alpha(-30)} & \overline{8}_1) \mathcal{N}1_{\beta(-30)} \\ \overline{9}_1) \mathcal{N}1_{\gamma(-30)} & \overline{10}_1) \mathcal{N}1_{\delta(-30)} & \overline{11}_1) \dots \rightarrow \mathcal{C} & \end{array}$$

Now let the instructions occupy the (long tank) words $1_1, 2_1, \dots$:

$$\begin{array}{l|l} 1_1) \overline{1}_1 - \overline{5}_1 & \sigma) \mathcal{N}n' - n_{(-30)} \\ 2_1) \overline{9}_1 \text{ s } \overline{7}_1 & \sigma) \mathcal{N}^{1_\gamma}_{1_\alpha(-30)} \quad \text{for } n' \overline{=} n \\ 3_1) \sigma \rightarrow \overline{12}_1 & \overline{12}_1) \mathcal{N}^{1_\gamma}_{1_\alpha(-30)} \quad \text{for } n' \overline{=} n \\ 4_1) \overline{1}_1 - \overline{5}_1 & \sigma) \mathcal{N}n' - n_{(-30)} \\ 5_1) \overline{10}_1 \text{ s } \overline{8}_1 & \sigma) \mathcal{N}^{1_\delta}_{1_\beta(-30)} \quad \text{for } n' \overline{=} n \\ 6_1) \sigma \rightarrow \overline{13}_1 & \overline{13}_1) \mathcal{N}^{1_\delta}_{1_\beta(-30)} \quad \text{for } n' \overline{=} n \\ 7_1) \overline{2}_1 - \overline{6}_1 & \sigma) \mathcal{N}m' - m_{(-30)} \\ 8_1) \overline{13}_1 \text{ s } \overline{12}_1 & \sigma) \mathcal{N} \dots \overline{13}_1) \dots \quad \text{for } m' \overline{=} m \\ & \dots \overline{12}_1) \dots \\ & \text{i.e.} \\ & \sigma) \mathcal{N}^{1_\delta}_{1_\gamma} \mathcal{N}^{1_\beta}_{1_\alpha(-30)} \quad \text{for } \begin{array}{l} m' = m, n' = n \quad m' = m, n' < n \\ m' < m, n' = n \quad m' < m, n' < n \end{array} \\ & \text{i.e. for } \begin{array}{l} (\delta) (\beta) \\ (\gamma) (\alpha) \end{array}, \text{ respectively.} \\ 9_1) \sigma \rightarrow \overline{11}_1 & \overline{11}_1) 1_\alpha, 1_\beta, 1_\gamma, 1_\delta \rightarrow \mathcal{C} \text{ for } (\alpha), (\beta), (\gamma), (\delta), \text{ respectively.} \\ 10_1) \overline{11}_1 \rightarrow \mathcal{C} & \end{array}$$

Now

$$\overline{11}_1) 1_\alpha, 1_\beta, 1_\gamma, 1_\delta \rightarrow \mathcal{C} \quad \text{for } (\alpha), (\beta), (\gamma), (\delta), \text{ respectively.}$$

Thus at the end of this phase \mathcal{C} is at $1_\alpha, 1_\beta, 1_\gamma, 1_\delta$, according to which case $(\alpha), (\beta), (\gamma), (\delta)$ holds.

- (h) We now pass to the case (α) . This has 2 subcases (α_1) and (α_2) , according to whether $x_{n'}^0 \geq$ or $< y_{m'}^0$. According to which of the 2 subcases holds, \mathcal{C} must be sent to the place where its instructions begin, say the (long tank) words $1_{\alpha_1}, 1_{\alpha_2}$. Their numbers must be available, say in the short tanks $\overline{1_2}, \overline{2_2}$.
- (i) We now formulate a set of instructions to effect this 2-way decision between $(\alpha_1), (\alpha_2)$. We state again the contents of the short tanks additionally assigned:

$$\overline{1_2} \mathcal{N} 1_{\alpha_1(-30)} \quad \overline{2_2} \mathcal{N} 1_{\alpha_2(-30)}$$

Now the instructions follow:

$$\begin{array}{l|l} 1_\alpha) \overline{4_1} - \overline{3_1} & \sigma) \mathcal{N} y_{m'}^0 - x_{n'}^0 \\ 2_\alpha) \overline{1_2} \text{ s } \overline{2_2} & \sigma) \mathcal{N} 1_{\alpha_1(-30)} \quad \text{for } x_{n'}^0 \leq y_{m'}^0 \\ & \text{i.e. for } \begin{matrix} (\alpha_1) \\ (\alpha_2) \end{matrix}, \text{ respectively.} \\ 3_\alpha) \sigma \rightarrow \overline{11_1} & \overline{11_1}) 1_{\alpha_1}, 1_{\alpha_2} \rightarrow \mathcal{C} \text{ for } (\alpha_1), (\alpha_2), \text{ respectively.} \\ 4_\alpha) \overline{11_1} \rightarrow \mathcal{C} & \end{array}$$

Now

$$\overline{11_1}) 1_{\alpha_1}, 1_{\alpha_2} \rightarrow \mathcal{C} \quad \text{for } (\alpha_1), (\alpha_2), \text{ respectively.}$$

Thus at the end of this phase \mathcal{C} is at $1_{\alpha_1}, 1_{\alpha_2}$, according to which case $(\alpha_1), (\alpha_2)$ holds.

- (j) Before turning to $(\alpha_1), (\alpha_2)$, let us dispose of the cases $(\beta), (\gamma), (\delta)$. According to (c), the cases $(\beta), (\gamma)$ can be handled as follows:
Additional short tanks assigned:

$$\overline{3_2} \mathcal{N} 0 \quad \overline{4_2} \mathcal{N} -1$$

The instructions for (β) :

$$\begin{array}{l|l} 1_\beta) \overline{3_2} - \overline{3_2} & \sigma) \mathcal{N} 0 \\ 2_\beta) 2_\alpha \rightarrow \mathcal{C} & \end{array}$$

and from here on like (α) with 0,0 for $x_{n'}^0, y_{m'}^0$.

The instructions for (γ) :

$$\begin{array}{l|l} 1_\gamma) \overline{4_2} - \overline{3_2} & \sigma) \mathcal{N} -1 \\ 2_\gamma) 2_\alpha \rightarrow \mathcal{C} & \end{array}$$

and from here on like (α) with 0, -1 for $x_{n'}^0, y_{m'}^0$.

(For both cases cf. (c).)

Assuming that after the conclusion of the procedure \mathcal{C} is to be sent to the (long tank) word a , the instructions for (δ) are as follows:

$$1_\delta) a \rightarrow \mathcal{C}$$

- (k) We now pass to (α_1) , (α_2) . It is necessary to state at this point, where the complexes $X_0^{(p)}, \dots, X_{n-1}^{(p)}$ and $Y_0^{(p)}, \dots, Y_{m-1}^{(p)}$ are stored, and where the complexes $Z_0^{(p)}, \dots, Z_{n+m-1}^{(p)}$ are to be placed. Let the X -complexes form a sequence which begins at the (long tank) word b , also the Y -complexes a sequence beginning at c , and the Z -complexes a sequence beginning at d . Since every complex consists of $p+1$ numbers, therefore $X_{n'}^{(p)}$ begins at $b+n'(p+1)$, $Y_{m'}^{(p)}$ begins at $c+m'(p+1)$, $Z_l^{(p)}$ begins at $d+l(p+1)$. Hence $x_{n'}^u$ is at $b+n'(p+1)+u$, $y_{m'}^u$ is at $c+m'(p+1)+u$, z_l^u is at $d+l(p+1)+u$. To conclude: The X complexes occupy the interval from b to $b+n(p+1)-1$, the Y complexes occupy the interval from c to $c+m(p+1)-1$, the Z complexes occupy the interval from d to $d+(n+m)(p+1)-1$. At the beginning of the (α_1) or (α_2) phase the following further quantities must be available, i.e. placed into short tanks: $b+n'(p+1)$, $c+m'(p+1)$, $d+l(p+1)$. It is also convenient to have $p+1$. Denote the short tanks containing these quantities by $\overline{1}_3, \overline{2}_3, \overline{3}_3, \overline{4}_3$. Hence these are the short tanks additionally assigned:

$$\begin{aligned} \overline{1}_3 & \mathcal{N}b+n'(p+1)_{(-30)} \\ \overline{2}_3 & \mathcal{N}c+m'(p+1)_{(-30)} \\ \overline{3}_3 & \mathcal{N}d+l(p+1)_{(-30)} \\ \overline{4}_3 & \mathcal{N}p+1_{(-30)} \end{aligned}$$

Finally, the transfer of the complex $X_{n'}^{(p)}$ (in (α_1)) or $Y_{m'}^{(p)}$ (in (α_2)) to the place of the complex $Z_l^{(p)}$ must be channeled through the short tanks. According to (d), the numbers $x_{n'}^1, \dots, x_{n'}^p, x_{n'+1}^0$ or the numbers $y_{m'}^1, \dots, y_{m'}^p, y_{m'+1}^0$ must be brought in (from X or Y) and the numbers $x_{n'}^0, x_{n'}^1, \dots, x_{n'}^p$ or $y_{m'}^0, y_{m'}^1, \dots, y_{m'}^p$ must be taken out (to Z). Consequently the numbers $x_{n'}^0, x_{n'}^1, \dots, x_{n'}^p, x_{n'+1}^0$ or the numbers $y_{m'}^0, y_{m'}^1, \dots, y_{m'}^p, y_{m'+1}^0$ must be routed through the short tanks. It is clearly best to have them in the form of an unbroken sequence. The length of this sequence is $p+2$. Denote the short tanks which are designated to hold this sequence by $\overline{1}_4, \overline{2}_4, \dots, \overline{(p+1)}_4, \overline{(p+2)}_4$.

We add: The primary function of (α_1) [(α_2)] is to move $x_{n'}^0, x_{n'}^1, \dots, x_{n'}^p$ [$y_{m'}^0, y_{m'}^1, \dots, y_{m'}^p$] into the (long tank) words $d+l(p+1), d+l(p+1)+1, \dots, d+l(p+1)+p$. However, there is also a secondary function: It must prepare the conditions for step $l+1$. This means that it must replace the numbers $n', x_{n'}^0, b+n'(p+1), d+l(p+1)$ [$m', y_{m'}^0, c+m'(p+1), d+l(p+1)$] in the short tanks $\overline{1}_1, \overline{3}_1, \overline{1}_3, \overline{3}_3$ [$\overline{2}_1, \overline{4}_1, \overline{2}_3, \overline{3}_3$] by the numbers $n'+1, x_{n'+1}^0, b+(n'+1)(p+1), d+(l+1)(p+1)$ [$m'+1, y_{m'+1}^0, c+(m'+1)(p+1), d+(l+1)(p+1)$].

To conclude: There are also two orders, affecting the transfers of X [Y] into the short tanks, and of Z out of the short tanks, and these orders are best placed into short tanks, say $\overline{1}_5, \overline{2}_5$. They must be followed by an order returning \mathcal{C} to the (α_1) or (α_2) sequence in long tanks. Hence this third order must be in $\overline{3}_5$, and it must depend on (α_1) or (α_2) . I.e. it must be transferred into $\overline{3}_5$ from the (α_1) or (α_2) sequence.

- (l) We now formulate 2 sets of instructions to carry out the tasks of (α_1) and (α_2) , as formulated in (k).

Additional short tanks assigned:

$$\overline{1_5}) \dots \rightarrow \overline{1_4} | p+2 \quad \overline{2_5}) \overline{1_4} \rightarrow \dots | p+1 \quad \overline{3_5}) \dots \quad \overline{4_5}) \mathcal{N}1_{(-30)}$$

The instructions for (α_1) :

$1_{\alpha_1})$	$\overline{1_3} \rightarrow \overline{1_5}$	$\overline{1_5}) b + n'(p+1) \rightarrow \overline{1_4} p+2$
$2_{\alpha_1})$	$\overline{3_3} \rightarrow \overline{2_5}$	$\overline{2_5}) \overline{1_4} \rightarrow d+l(p+1) p+1$
$3_{\alpha_1})$	$\overline{3_5}$	$\overline{3_5}) 6_{\alpha_1} \rightarrow \mathcal{C}$
$4_{\alpha_1})$	$6_{\alpha_1} \rightarrow \mathcal{C}$	
$5_{\alpha_1})$	$\overline{1_5} \rightarrow \mathcal{C}$	
<hr/>		
	$\overline{1_5}) b + n'(p+1) \rightarrow \overline{1_4} p+2$	$b + n'(p+1)) \mathcal{N}x_{n'}^0$ to $\overline{1_4}) \mathcal{N}x_{n'}^0$ $b + n'(p+1) + 1) \mathcal{N}x_{n'}^1$ to $\overline{2_4}) \mathcal{N}x_{n'}^1$ \dots \dots $b + n'(p+1) + p) \mathcal{N}x_{n'}^p$ to $\overline{(p+1)_4}) \mathcal{N}x_{n'}^p$ $b + (n'+1)(p+1)) \mathcal{N}x_{n'+1}^0$ to $\overline{(p+2)_4}) \mathcal{N}x_{n'+1}^0$
	$\overline{2_5}) \overline{1_4} \rightarrow d+l(p+1) p+1$	$\overline{1_4}) \mathcal{N}x_{n'}^0$ to $d+l(p+1)) \mathcal{N}x_{n'}^0$ $\overline{2_4}) \mathcal{N}x_{n'}^1$ to $d+l(p+1)+1) \mathcal{N}x_{n'}^1$ \dots \dots $\overline{(p+1)_4}) \mathcal{N}x_{n'}^p$ to $d+l(p+1)+p) \mathcal{N}x_{n'}^p$
	$\overline{3_5}) 6_{\alpha_1} \rightarrow \mathcal{C}$	
$6_{\alpha_1})$	$\overline{1_1} + \overline{4_5}$	$(\sigma) \mathcal{N}n' + 1_{(-30)}$
$7_{\alpha_1})$	$\sigma \rightarrow \overline{1_1}$	$\overline{1_1}) \mathcal{N}n' + 1_{(-30)}$
$8_{\alpha_1})$	$\overline{(p+2)_4} \rightarrow \overline{3_1}$	$\overline{3_1}) \mathcal{N}x_{n'+1}^0$
$9_{\alpha_1})$	$\overline{1_3} + \overline{4_3}$	$(\sigma) \mathcal{N}b + (n'+1)(p+1)_{(-30)}$
$10_{\alpha_1})$	$\sigma \rightarrow \overline{1_3}$	$\overline{1_3}) \mathcal{N}b + (n'+1)(p+1)_{(-30)}$
$11_{\alpha_1})$	$\overline{3_3} + \overline{4_3}$	$(\sigma) \mathcal{N}d + (l+1)(p+1)_{(-30)}$
$12_{\alpha_1})$	$\sigma \rightarrow \overline{3_3}$	$\overline{3_3}) \mathcal{N}d + (l+1)(p+1)_{(-30)}$
$13_{\alpha_1})$	$1_1 \rightarrow \mathcal{C}$	To begin step $l+1$ according to (g).

The instructions for (α_2) :

$1_{\alpha_2})$	$\overline{2_3} \rightarrow \overline{1_5}$	$\overline{1_5}) c + m'(p+1) \rightarrow \overline{1_4} p+2$
$2_{\alpha_2})$	$\overline{3_3} \rightarrow \overline{2_5}$	$\overline{2_5}) \overline{1_4} \rightarrow d+l(p+1) p+1$
$3_{\alpha_2})$	$\overline{3_5}$	$\overline{3_5}) 6_{\alpha_2} \rightarrow \mathcal{C}$
$4_{\alpha_2})$	$6_{\alpha_2} \rightarrow \mathcal{C}$	
$5_{\alpha_2})$	$\overline{1_5} \rightarrow \mathcal{C}$	
<hr/>		
	$\overline{1_5}) c + m'(p+1) \rightarrow \overline{1_4} p+2$	

	$c + m'(p + 1)) \mathcal{N}y_{m'}^0$ to $\overline{1_4}) \mathcal{N}y_{m'}^0$
	$c + m'(p + 1) + 1) \mathcal{N}y_{m'}^1$ to $\overline{2_4}) \mathcal{N}y_{m'}^1$
	...
	$c + m'(p + 1) + p) \mathcal{N}y_{m'}^p$ to $\overline{(p+1)_4}) \mathcal{N}y_{m'}^p$
	$c + (m' + 1)(p + 1)) \mathcal{N}y_{m'+1}^0$ to $\overline{(p+2)_4}) \mathcal{N}y_{m'+1}^0$
$\overline{2_5}) \overline{1_4} \rightarrow d + l(p + 1) p + 1$	$\overline{1_4}) \mathcal{N}y_{m'}^0$ to $d + l(p + 1)) \mathcal{N}y_{m'}^0$
	$\overline{2_4}) \mathcal{N}y_{m'}^1$ to $d + l(p + 1) + 1) \mathcal{N}y_{m'}^1$
	...
	$\overline{(p+1)_4}) \mathcal{N}y_{m'}^p$ to $d + l(p + 1) + p) \mathcal{N}y_{m'}^p$
$\overline{3_5}) 6\alpha_2 \rightarrow \mathcal{C}$	
$6\alpha_2) \overline{2_1} + \overline{4_5}$	$\sigma) \mathcal{N}m' + 1_{(-30)}$
$7\alpha_2) \sigma \rightarrow \overline{2_1}$	$\overline{2_1}) \mathcal{N}m' + 1_{(-30)}$
$8\alpha_2) \overline{(p+2)_4} \rightarrow \overline{4_1}$	$\overline{4_1}) \mathcal{N}y_{m'+1}^0$
$9\alpha_2) \overline{2_3} + \overline{4_3}$	$\sigma) \mathcal{N}c + (m' + 1)(p + 1)_{(-30)}$
$10\alpha_2) \sigma \rightarrow \overline{2_3}$	$\overline{2_3}) \mathcal{N}c + (m' + 1)(p + 1)_{(-30)}$
$11\alpha_2) \overline{3_3} + \overline{4_3}$	$\sigma) \mathcal{N}d + (l + 1)(p + 1)_{(-30)}$
$12\alpha_2) \sigma \rightarrow \overline{3_3}$	$\overline{3_3}) \mathcal{N}d + (l + 1)(p + 1)_{(-30)}$
$13\alpha_2) 1_1 \rightarrow \mathcal{C}$	To begin step $l + 1$ according to (g).

- (6) Let us restate, which short tanks are occupied at the beginning of step l , and how. This is the list:

$\overline{1_1}) \mathcal{N}n'_{(-30)}$	$\overline{2_1}) \mathcal{N}m'_{(-30)}$	$\overline{3_1}) \mathcal{N}x_{n'}^0$	$\overline{4_1}) \mathcal{N}y_{m'}^0$
$\overline{5_1}) \mathcal{N}n_{(-30)}$	$\overline{6_1}) \mathcal{N}m_{(-30)}$	$\overline{7_1}) \mathcal{N}1\alpha_{(-30)}$	$\overline{8_1}) \mathcal{N}1\beta_{(-30)}$
$\overline{9_1}) \mathcal{N}1\gamma_{(-30)}$	$\overline{10_1}) \mathcal{N}1\delta_{(-30)}$	$\overline{11_1}) \dots \rightarrow \mathcal{C}$	
$\overline{1_2}) \mathcal{N}1\alpha_1_{(-30)}$	$\overline{2_2}) \mathcal{N}1\alpha_2_{(-30)}$		
$\overline{3_2}) \mathcal{N}0$	$\overline{4_2}) \mathcal{N}-1$		
$\overline{1_3}) \mathcal{N}b + n'(p + 1)_{(-30)}$	$\overline{2_3}) \mathcal{N}c + m'(p + 1)_{(-30)}$	$\overline{3_3}) \mathcal{N}d + l(p + 1)_{(-30)}$	$\overline{4_3}) \mathcal{N}p + 1_{(-30)}$
$\overline{1_4}) \dots$	$\overline{2_4}) \dots$	\dots	$\overline{(p+1)_4}) \dots$ $\overline{(p+2)_4}) \dots$
$\overline{1_5}) \dots \rightarrow \overline{1_4} p + 2$	$\overline{2_5}) \overline{1_4} \rightarrow \dots p + 1$	$\overline{3_5}) \dots$	$\overline{4_5}) \mathcal{N}1_{(-30)}$

The first thing to note is, that this requires $11 + 4 + 4 + (p + 2) + 4 = p + 25$ short tanks. Hence, if the total number of short tanks is 32 [64], this gives the upper limit 7 [39] for p .

The second observation is, that these short tanks have the following contents when the sequence of steps $l = 0, 1, \dots, n + m$ begins, i.e. at the beginning of the step $l = 0$. This is the list:

$\overline{1}_1) \mathcal{N}0$	$\overline{2}_1) \mathcal{N}0$	$\overline{3}_1) \mathcal{N}x_0^0$	$\overline{4}_1) \mathcal{N}y_0^0$
$\overline{5}_1) \mathcal{N}n_{(-30)}$	$\overline{6}_1) \mathcal{N}m_{(-30)}$	$\overline{7}_1) \mathcal{N}1_{\alpha(-30)}$	$\overline{8}_1) \mathcal{N}1_{\beta(-30)}$
$\overline{9}_1) \mathcal{N}1_{\gamma(-30)}$	$\overline{10}_1) \mathcal{N}1_{\delta(-30)}$	$\overline{11}_1) \dots \rightarrow \mathcal{C}$	
$\overline{1}_2) \mathcal{N}1_{\alpha_1(-30)}$	$\overline{2}_2) \mathcal{N}1_{\alpha_2(-30)}$		
$\overline{3}_2) \mathcal{N}0$	$\overline{4}_2) \mathcal{N}-1$		
$\overline{1}_3) \mathcal{N}b_{(-30)}$	$\overline{2}_3) \mathcal{N}c_{(-30)}$	$\overline{3}_3) \mathcal{N}d_{(-30)}$	$\overline{4}_3) \mathcal{N}p+1_{(-30)}$
$\overline{1}_4) \dots$	$\overline{2}_4) \dots$	\dots	$\overline{(p+1)}_4) \dots \quad \overline{(p+2)}_4) \dots$
$\overline{1}_5) \dots \rightarrow \overline{1}_4 \mid p+2$	$\overline{2}_5) \overline{1}_4 \rightarrow \dots \mid p+1$	$\overline{3}_5) \dots$	$\overline{4}_5) \mathcal{N}1_{(-30)}$

Of these $p + 25$ short tanks the following must form unbroken sequences: $\overline{1}_5, \overline{2}_5, \overline{3}_5$ because of their rôle in (I) (between 5_{α_1} and 6_{α_1} , and between 5_{α_2} and 6_{α_2}); $\overline{1}_4, \overline{2}_4, \dots, \overline{(p+1)}_4, \overline{(p+2)}_4$ because of their rôle in (I) (at $\overline{1}_5$ and $\overline{2}_5$, in the two intervals mentioned above).

These are $3 + (p + 2) = p + 5$ short tanks. Of these $p + 3$, namely $\overline{3}_5$ and $\overline{1}_4, \overline{2}_4, \dots, \overline{(p+1)}_4, \overline{(p+2)}_4$ require no preliminary substitution; 2, namely $\overline{1}_5, \overline{2}_5$ have a fixed content.

The remaining $(p + 25) - (p + 5) = 20$ short tanks can be classified as follows: 12, namely $\overline{1}_1, \overline{2}_1, \overline{7}_1, \overline{8}_1, \overline{9}_1, \overline{10}_1, \overline{11}_1, \overline{1}_2, \overline{2}_2, \overline{3}_2, \overline{4}_2, \overline{4}_5$ have a fixed content; 6, namely $\overline{5}_1, \overline{6}_1, \overline{1}_3, \overline{2}_3, \overline{3}_3, \overline{4}_3$ have to be substituted from the general data of the problem (they contain $n, m, b, c, d, p + 1$); 2, namely $\overline{3}_1, \overline{4}_1$ have to be substituted from the sequences X, Y (they contain x_0^0, y_0^0). It is desirable that all short tanks with a fixed constant form an unbroken sequence, so that they can be substituted by one order. I.e. the 14 given here and the 2 given above must form an unbroken sequence. These 2 last are $\overline{1}_5, \overline{2}_5$, as noted still earlier, they must be followed by $\overline{3}_5$. This gives an unbroken sequence of $12 + 2 + 1 = 15$ short tanks.

Finally, it is desirable to have the 6 short tanks with $n, m, b, c, d, p + 1$ at the beginning, and the 2 with x_0^0, y_0^0 immediately afterwards. Also, to have the sequence of indefinite length $(p + 2)$ at the end.

This gives the following final assignment of the $p + 25$ short tanks used:

$\overline{1}) \dots \overline{5}_1) \mathcal{N}n_{(-30)}$	$\overline{9}) \dots \overline{1}_1) \mathcal{N}0$
$\overline{2}) \dots \overline{6}_1) \mathcal{N}m_{(-30)}$	$\overline{10}) \dots \overline{2}_1) \mathcal{N}0$
$\overline{3}) \dots \overline{1}_3) \mathcal{N}b_{(-30)}$	$\overline{11}) \dots \overline{7}_1) \mathcal{N}1_{\alpha(-30)}$
$\overline{4}) \dots \overline{2}_3) \mathcal{N}c_{(-30)}$	$\overline{12}) \dots \overline{8}_1) \mathcal{N}1_{\beta(-30)}$
$\overline{5}) \dots \overline{3}_3) \mathcal{N}d_{(-30)}$	$\overline{13}) \dots \overline{9}_1) \mathcal{N}1_{\gamma(-30)}$
$\overline{6}) \dots \overline{4}_3) \mathcal{N}p+1_{(-30)}$	$\overline{14}) \dots \overline{10}_1) \mathcal{N}1_{\delta(-30)}$
$\overline{7}) \dots \overline{3}_1) \mathcal{N}x_0^0$	$\overline{15}) \dots \overline{11}_1) \dots \rightarrow \mathcal{C}$

$\bar{8}) \cdots \bar{4}_1) \mathcal{N}y_0^0$	$\bar{16}) \cdots \bar{1}_2) \mathcal{N}1_{\alpha_1(-30)}$
	$\bar{17}) \cdots \bar{2}_2) \mathcal{N}1_{\alpha_2(-30)}$
	$\bar{18}) \cdots \bar{3}_2) \mathcal{N}0$
	$\bar{19}) \cdots \bar{4}_2) \mathcal{N}-1$
	$\bar{20}) \cdots \bar{4}_5) \mathcal{N}1_{(-30)}$
	$\bar{21}) \cdots \bar{1}_5) \cdots \rightarrow \bar{24} \mid p+2$
	$\bar{22}) \cdots \bar{2}_5) \bar{24} \rightarrow \cdots \mid p+1$
	<hr/> $\bar{23}) \cdots \bar{3}_5) \cdots$
	$\bar{24}) \cdots \bar{1}_4) \cdots$
	$\bar{25}) \cdots \bar{2}_4) \cdots$
	\cdots
	$\overline{p+24}) \cdots \overline{p+14}) \cdots$
	$\overline{p+25}) \cdots \overline{p+24}) \cdots$

(7) We now come to the step — mentioned in (e). We foresaw there that — would have to substitute x_0^0, y_0^0 into the proper short tanks (as we saw in (6), into $\bar{7}, \bar{8}$). We see now, however, that — has to take care of the substitution into all short tanks. More precisely: No substitutions into $\bar{23}$ and $\bar{24}, \bar{25}, \dots, \overline{p+24}, \overline{p+25}$ are needed. And $\bar{1}, \dots, \bar{6}$ should be substituted when the problem is set up as such. Hence the short tanks left for the step — are $\bar{7}, \bar{8}$ and $\bar{10}, \dots, \bar{22}$.

We will substitute $\bar{7}, \bar{8}$ first and $\bar{10}, \dots, \bar{22}$ afterwards. During the first operation the short tanks $\bar{1}, \dots, \bar{6}$ are already occupied as indicated above (by $n, m, b, c, d, p+1$), while $\bar{9}, \dots$ are still available. Hence we can use $\bar{9}, \dots$ while carrying out the first step, the substitution of $\bar{7}, \bar{8}$, and substitute $\bar{9}, \dots$, or more precisely $\bar{9}, \dots, \bar{22}$, in the final form only subsequently, as a second step.

Actually it is desirable to place during the first step, the substitution of $\bar{7}, \bar{8}$, some orders into short tanks. In accordance with what was said above, we use for this $\bar{9}, \dots$.

We will now formulate the instructions which carry out all these substitutions. Let these instructions occupy the (long tank) words $1_0, 2_0, \dots$:

$1_0) \quad \updownarrow \bar{9} \mid 2$	
$2_0) \quad \cdots \updownarrow \bar{7}$	$\bar{9}) \cdots \updownarrow \bar{7}$
$3_0) \quad \cdots \updownarrow \bar{8}$	$\bar{10}) \cdots \updownarrow \bar{8}$
$4_0) \quad 8_0 \rightarrow \mathcal{C}$	$\bar{11}) 8_0 \rightarrow \mathcal{C}$
$5_0) \quad \bar{3} \rightarrow \bar{9}$	$\bar{9}) b \updownarrow \bar{7}$
$6_0) \quad \bar{4} \rightarrow \bar{10}$	$\bar{10}) c \updownarrow \bar{8}$
$7_0) \quad \bar{9} \rightarrow \mathcal{C}$	
$\bar{9}) \quad b \updownarrow \bar{7}$	$\bar{7}) \mathcal{N}x_0^0$
$\bar{10}) \quad c \updownarrow \bar{8}$	$\bar{8}) \mathcal{N}y_0^0$
$\bar{11}) \quad 8_0 \rightarrow \mathcal{C}$	

8 ₀) $\overline{9} \mid 13$	
9 ₀) $\mathcal{N}0$	$\overline{9}$) $\mathcal{N}0$
10 ₀) $\mathcal{N}0$	$\overline{10}$) $\mathcal{N}0$
11 ₀) $\mathcal{N}1_{\alpha(-30)}$	$\overline{11}$) $\mathcal{N}1_{\alpha(-30)}$
12 ₀) $\mathcal{N}1_{\beta(-30)}$	$\overline{12}$) $\mathcal{N}1_{\beta(-30)}$
13 ₀) $\mathcal{N}1_{\gamma(-30)}$	$\overline{13}$) $\mathcal{N}1_{\gamma(-30)}$
14 ₀) $\mathcal{N}1_{\delta(-30)}$	$\overline{14}$) $\mathcal{N}1_{\delta(-30)}$
15 ₀) $\dots \rightarrow \mathcal{C}$	$\overline{15}$) $\dots \rightarrow \mathcal{C}$
16 ₀) $\mathcal{N}1_{\alpha_1(-30)}$	$\overline{16}$) $\mathcal{N}1_{\alpha_1(-30)}$
17 ₀) $\mathcal{N}1_{\alpha_2(-30)}$	$\overline{17}$) $\mathcal{N}1_{\alpha_2(-30)}$
18 ₀) $\mathcal{N}0$	$\overline{18}$) $\mathcal{N}0$
19 ₀) $\mathcal{N}-1$	$\overline{19}$) $\mathcal{N}-1$
20 ₀) $\dots \rightarrow \overline{24} \mid p+2$	$\overline{20}$) $\dots \rightarrow \overline{24} \mid p+2$
21 ₀) $\overline{24} \rightarrow \dots \mid p+1$	$\overline{21}$) $\overline{24} \rightarrow \dots \mid p+1$
22 ₀) $\mathcal{N}1_{(-30)}$	$\overline{22}$) $\mathcal{N}1_{(-30)}$
23 ₀) $1_1 \rightarrow \mathcal{C}$	To begin step 0 according to (g).

(8) We now have a complete list of instructions:

First, in short tanks $\overline{1}, \dots, \overline{6}$, as described at the end of (6).

Second, in long tanks the words $1_0, \dots, 23_0$ (cf. (7)); $1_1, \dots, 10_1$ (cf. (g)); $1_\alpha, \dots, 4_\alpha$ (cf. (i)); $1_\beta, 2_\beta$ (cf. (j)); $1_\gamma, 2_\gamma$ (cf. (j)); 1_δ (cf. (j)); $1_{\alpha_1}, \dots, 13_{\alpha_1}$ (cf. (l)); $1_{\alpha_2}, \dots, 13_{\alpha_2}$ (cf. (l)).

Let us consider the second category of instructions, i.e. the words in long tanks, more closely. The first thing to note is, that this requires $23 + 10 + 4 + 2 + 2 + 1 + 13 + 13 = 68$ (long tank) words. The second observation is, that according to (j) $1_\beta, 2_\beta$ as well as $1_\gamma, 2_\gamma$ are always followed by $1_\alpha, \dots, 4_\alpha$, and according to (i) $1_\alpha, \dots, 4_\alpha$ are always followed by $1_{\alpha_1}, \dots, 13_{\alpha_1}$ or $1_{\alpha_2}, \dots, 13_{\alpha_2}$. Hence it is reasonable to make the final assignment of numbers to these (long tank) words in such a way that these precedences are maintained.

Actually it is best to delay the final assignment of numbers, for reasons which will appear in (9). We make, however, a secondary assignment of numbers as follows:

$1', \dots, 23'$	to	$1_0, \dots, 23_0$
$24', \dots, 33'$	to	$1_1, \dots, 10_1$
$34', 35'$	to	$1_\beta, 2_\beta$
$36', 37'$	to	$1_\gamma, 2_\gamma$
$38', \dots, 41'$	to	$1_\alpha, \dots, 4_\alpha$
$42', \dots, 54'$	to	$1_{\alpha_1}, \dots, 13_{\alpha_1}$
$55', \dots, 67'$	to	$1_{\alpha_2}, \dots, 13_{\alpha_2}$
$68'$	to	1_δ

- (9) The assignment of numbers at the end of (8), makes $1_\alpha, 1_\beta, 1_\gamma, 1_\delta, 1_{\alpha_1}, 1_{\alpha_2}$ equal to $38', 34', 36', 68', 42', 55'$. These numbers occur in $11_0, 12_0, 13_0, 14_0, 16_0, 17_0$, i.e. in $11', 12', 13', 14', 16', 17'$. Hence the content of these 6 (long tank) words depends explicitly on the final assignment of (long tank word) numbers to $1', \dots, 68'$.

If this final assignment were made now, in a rigid form, then the (long tank) words $11', \dots, 14', 16', 17'$ could be formulated accordingly, and the instructions would be completed. It is, however, preferable to have these instructions in such a form that they can begin anywhere, i.e. that their first (long tank) word can be chosen freely.

Let this first (long tank) word be e , i.e. e is $1'$. Hence $e, \dots, e+67$ should correspond to $1', \dots, 68'$. However, it is worth while to deviate from this simple sequential correspondence for the following reasons:

- (A) In $1_0, \dots, 23_0$ (i.e. $1', \dots, 23'$) the passage of \mathcal{C} from $\overline{11}$ to 8_0 involves a delay of about one long tank, if 8_0 follows immediately upon 7_0 : Indeed 7_0 is followed by $\overline{9}, \overline{10}, \overline{11}$. Hence it is better to intercalate 3 words between 7_0 and 8_0 to time correctly for $\overline{9}, \overline{10}, \overline{11}$, plus, say, 1 word for the long tank switching in $\overline{11}$. I.e., there should be 4 (empty) words between 7_0 and 8_0 , i.e. $7'$ and $8'$.
- (B) In $1_{\alpha_1}, \dots, 13_{\alpha_1}$ (i.e. $42', \dots, 54'$) there exists the same situation as in (A) between 5_{α_1} and 6_{α_1} , where $\overline{15}, \overline{25}, \overline{35}$ (i.e. $\overline{21}, \overline{22}, \overline{23}$) are intercalated. In order to avoid a delay of about one long tank, it is again necessary to intercalate $3+1=4$ (empty) words between 5_{α_1} and 6_{α_1} , i.e. $46'$ and $47'$.
- (C) In $1_{\alpha_2}, \dots, 13_{\alpha_2}$ (i.e. $55', \dots, 67'$) between 5_{α_2} and 6_{α_2} the situation is exactly the same as in (B). Hence it is again advisable to intercalate 4 (empty) words between 5_{α_2} and 6_{α_2} , i.e. $59'$ and $60'$.
- (D) 10_1 (i.e. $33'$) sends \mathcal{C} to $\overline{11}_1$ (i.e. $\overline{15}$), and this in turn sends \mathcal{C} to 1_α or 1_β or 1_γ or 1_δ (i.e. $38'$ or $34'$ or $36'$ or $68'$). In order to avoid a delay of about one long tank, it is necessary to intercalate 1 word after 10_1 to time correctly for $\overline{11}_1$, plus, say, 1 word for the long tank switching in $\overline{11}_1$. I.e. there should be 2 (empty) words after 10_1 , i.e. $33'$.

Taking these matters into account, the following final assignment of numbers obtains:

$e, \dots, e+6$	to	$1', \dots, 7'$	to	$1_0, \dots, 7_0$
$e+7, \dots, e+10$	to		empty (cf. (A))	
$e+11, \dots, e+26$	to	$8', \dots, 23'$	to	$8_0, \dots, 23_0$
$e+27, \dots, e+36$	to	$24', \dots, 33'$	to	$1_1, \dots, 10_1$
$e+37, e+38$	to		empty (cf. (D))	
$e+39, e+40$	to	$34', 35'$	to	$1_\beta, 2_\beta$
$e+41, e+42$	to	$36', 37'$	to	$1_\gamma, 2_\gamma$
$e+43, \dots, e+46$	to	$38', \dots, 41'$	to	$1_\alpha, \dots, 4_\alpha$
$e+47, \dots, e+51$	to	$42', \dots, 46'$	to	$1_{\alpha_1}, \dots, 5_{\alpha_1}$
$e+52, \dots, e+55$	to		empty (cf. (B))	
$e+56, \dots, e+63$	to	$47', \dots, 54'$	to	$6_{\alpha_1}, \dots, 13_{\alpha_1}$
$e+64, \dots, e+68$	to	$55', \dots, 59'$	to	$1_{\alpha_2}, \dots, 5_{\alpha_2}$

$$\begin{array}{lll}
e+69, \dots, e+72 & \text{to} & \text{empty (cf. (C))} \\
e+73, \dots, e+80 & \text{to} & 60', \dots, 67' \text{ to } 6_{\alpha_2}, \dots, 13_{\alpha_2} \\
e+81 & \text{to} & 68' \text{ to } 1_{\delta}
\end{array}$$

Hence the (long tank) words $11', 12', 13', 14', 16', 17'$ become

$$e+14, e+15, e+16, e+17, e+19, e+20,$$

and they contain the numbers $38', 34', 36', 68', 42', 55'$, and these become

$$e+43, e+39, e+41, e+81, e+47, e+64.$$

We rewrite these (long tank) words:

$$\begin{array}{l}
e+14) \mathcal{N}e+43_{(-30)} \\
e+15) \mathcal{N}e+39_{(-30)} \\
e+16) \mathcal{N}e+41_{(-30)} \\
e+17) \mathcal{N}e+81_{(-30)} \\
e+19) \mathcal{N}e+47_{(-30)} \\
e+20) \mathcal{N}e+64_{(-30)}
\end{array}$$

- (10) Disregarding for the time being the 6 substitutions required to produce the 6 (long tank) words enumerated at the end of (9), the total system of instructions, at the present stage, is this:

- (I) The 82 (long tank) words $e, \dots, e+81$ of (9).
- (II) The 6 short tanks $\bar{1}, \dots, \bar{6}$ of (6).

The quantities which actually determine the problem, as a function of the X, Y , are these:

$$(*) \ n, m, b, c, d, p, a, e. \text{ (For } a \text{ cf. } (1_{\delta}) \text{ in (j), for } e \text{ cf. (9).)}$$

Of these the 6 first, n, m, b, c, d, p , are given in (II), but p occurs again in (I). The others a, e , occur in (I) only. So we must discuss how the occurrences of

$$(**) \ p, a, e$$

in (I) are to be taken care of.

p occurs in $20_0, 21_0$ (cf. (7)), i.e. $e+23, e+24$ (cf. (9)). a occurs in 1_{δ} (cf. (j)), i.e. $e+81$ (cf. (9)). The occurrences of e have been summarized at the end of (9).

We rewrite the (long tank) words which contain these additional substitutions:

$$\begin{array}{l}
e+23) \dots \rightarrow \bar{24} \mid p+2 \\
e+24) \bar{24} \rightarrow \dots \mid p+1
\end{array}$$

and

$$e+81) a \rightarrow \mathcal{C}$$

- (11) The complete system of instructions, as derived in what preceded, can also be formulated as follows:

The 82 (long tank) words of (I) in (10), namely $e, \dots, e+81$, contain only fixed symbols, except for certain occurrences of the 3 variables of (**) in (10), namely

p, a, e , in the 9 words enumerated at the end of (9) and the end of (10). Assume, that $e, \dots, e + 81$ are stated, with blanks \dots in place of these 3 variables in the 9 words in question. Call this group of 82 words G_{82} .

Then, after G_{82} has been placed in the long tanks, in an unbroken sequence beginning at e , the following further steps are necessary:

First, 6 substitutions into short tanks, according to (II) in (10), and 9 substitutions into long tanks, according to the end of (9) and the end of (10). We restate these $6 + 9 = 15$ substitutions:

$$\begin{array}{lll} \bar{1}) \mathcal{N}n_{(-30)} & e + 14) \mathcal{N}e + 43_{(-30)} & e + 23) \dots \rightarrow \overline{24} | p + 2 \\ \bar{2}) \mathcal{N}m_{(-30)} & e + 15) \mathcal{N}e + 39_{(-30)} & e + 24) \overline{24} \rightarrow \dots | p + 1 \\ \bar{3}) \mathcal{N}b_{(-30)} & e + 16) \mathcal{N}e + 41_{(-30)} & e + 81) a \rightarrow \mathcal{C} \\ \bar{4}) \mathcal{N}c_{(-30)} & e + 17) \mathcal{N}e + 81_{(-30)} & \\ \bar{5}) \mathcal{N}d_{(-30)} & e + 19) \mathcal{N}e + 47_{(-30)} & \\ \bar{6}) \mathcal{N}p + 1_{(-30)} & e + 20) \mathcal{N}e + 64_{(-30)} & \end{array}$$

Denote this group by S_{15} .

After the substitutions S_{15} have been carried out, \mathcal{C} can be sent at any time to e . This will cause the meshing to take place as desired, and after its completion send \mathcal{C} to a .

The following final remark should be added: G_{82} , as defined above, contains only fixed symbols, i.e. it is a fixed routine. With a suitable choice of S_{15} it will, therefore, cause any desired meshing process to take place. Thus G_{82} can be stored permanently outside the machine, and it may be fed into the machine as a 'sub routine', as a part of the instructions of any more extensive problem, which contains one or more meshing operations. Then S_{15} must be part of the 'main routine' of that problem, it may be effected there in several parts if desired. If, in particular, the problem contains several meshing operations, only those parts of S_{15} need be repeated, in which those operations differ. And since G_{82} contains no explicit reference to its own position, i.e. to e , therefore G_{82} can be placed anywhere in the long tanks, it is only necessary that the 'main routine' take care of the proper e (by means of its S_{15}). This 'mobility' within the long tanks is, of course, an absolute necessity for 'sub routines' which are suited for use in a flexible general logical scheme of 'main routines' and (possibly multiple and interchangeable) 'sub routines'.

- (12) To conclude, we must estimate the duration of a meshing process according to the instructions which we derived.

We will not count the time in effecting S_{15} , hence we begin when \mathcal{C} reaches e . We follow the list of words $e, \dots, e + 81$ given in (9):

The process begins with the step — of (7), i.e. $1_0, \dots, 23_0$, i.e. $e, \dots, e + 26$. Apart from 26 words = $\frac{26}{32}$ ms = .81 ms, there are the following delays: $\bar{9}, \bar{10}$ each averages $\frac{1}{2}$ tank = .5 ms; $\bar{11}$ is 1 word = $\frac{1}{32}$ ms = .03 ms. The total is .81 + .5 + .5 + .03 = 1.84 ms.

Now consider a step $l = 0, 1, \dots, n + m$. Its make up is as follows: It begins with $1_1, \dots, 10_1$ of (g), i.e. $e + 27, \dots, e + 38$. These are 12 words = $\frac{12}{32}$ ms = .38 ms,

and no other delays. From here on the process splits, according to which of the 4 cases (α) , (β) , (γ) , (δ) obtains.

Consider (α) first. It begins with $1_\alpha, \dots, 4_\alpha$ of (i), i.e. $e + 43, \dots, e + 46$. Apart from 4 words $= \frac{4}{32}$ ms $= .13$ ms, there are the following delays: At the beginning of this sequence 7 words (from $e + 36$ to $e + 43$) $= \frac{7}{32}$ ms $= .22$ ms; from the time of $\overline{11}$ (which follows upon $e + 46$ and hence is $e + 47$) until the beginning of (α_1) or of (α_2) ($e + 47$ or $e + 64$), i.e. nothing or 17 words, averaging $\frac{1}{2}$ 17 words $= \frac{1}{2} \frac{17}{32}$ ms $= .27$ ms. This totals $.13 + .22 + .27 = .62$ ms. Next there is (α_1) [(α_2)], consisting of $1_{\alpha_1}, \dots, 13_{\alpha_1}$ [$1_{\alpha_2}, \dots, 13_{\alpha_2}$] of (l), i.e. $e + 47, \dots, e + 63$ [$e + 64, \dots, e + 80$]. Apart from 17 words $= \frac{17}{32}$ ms $= .53$ ms, there are the following delays: $\overline{15}$ averages $p + 1$ words and $\frac{1}{2}$ tank; $\overline{25}$ averages $p + 2$ words and $\frac{1}{2}$ tank; after 13_{α_1} [13_{α_2}] (i.e. $e + 63$ [$e + 80$]) there is a delay until 1_1 ($e + 27$), since this delay is to be taken modulo entire tank, i.e. modulo 32 words, it amounts to 28 [11] words, i.e. an average of $\frac{1}{2}(28 + 11) = 18\frac{1}{2}$ words. This totals $(p + 2) + (p + 1) + 18\frac{1}{2} = 2p + 21\frac{1}{2}$ words and $\frac{1}{2} + \frac{1}{2} = 1$ tank, i.e. $\frac{2p+21\frac{1}{2}}{3} + 1$ ms $= \frac{p}{16} + .67$ ms. The grand total for (α) is therefore $.62 + (\frac{p}{16} + .67)$ ms $= \frac{p}{16} + 1.29$ ms.

Consider next (β) , (γ) . These differ from (α) only inasmuch as they replace 1_α by $1_\beta, 2_\beta$ ($2_\gamma, 3_\gamma$) of (j). In either case, there is, with actual operation and delays, a direct sequence from 10_1 to 2_α , i.e. from $e + 36$ to $e + 44$. Hence there duration is the same as α .

Consider finally (δ) . This involves the delay from 1_{10} to 1_δ of (j), i.e. from $e + 36$ to $e + 47$, and the word 1_δ , i.e. $e + 47$, itself. This amounts to 12 words $= \frac{12}{32}$ ms $= .38$ ms.

Now of the $n + m + 1$ steps $l = 0, 1, \dots, n + m$ all but the last one, $n + m$, are (α) or (β) or (γ) ; $n + m$ is (δ) . Hence there are $n + m$ lasting $.38 + (\frac{p}{16} + 1.29)$ ms $= \frac{p}{16} + 1.67$ ms and 1 lasting $.38 + .38$ ms $= .76$ ms. The total duration of the entire meshing process is therefore this: $1.84 + (n + m)(\frac{p}{16} + 1.67) + .76$ ms $= 2.60 + (n + m)(\frac{p}{16} + 1.67)$ ms. For $p = 1$ this is $2.60 + (n + m)1.78$ ms, for $p = 7$ it is $2.60 + (n + m)2.11$ ms, for $p = 39$ it is $2.60 + (n + m)4.11$ ms. (Concerning these p values consider the first part of (6).)

References

- AWB-IUPUI** Arthur W. Burks papers, Institute for American Thought, Indiana University–Purdue University Indianapolis.
- ETE-UP** ENIAC Trial Exhibits, Master Collection, 1964–1973, University of Pennsylvania Archives and Records Center, Philadelphia.
- HHG-APS** Herman Heine Goldstine Papers, Mss. Ms. Coll. 19, American Philosophical Society, Philadelphia.
- HML-UV** Sperry-Rand Corporation, UNIVAC Division Records, Accession 182.5.I, Hagley Museum and Library, Wilmington, Delaware.
- JVN-LOC** John von Neumann and Klara Dan Von Neumann Papers, Library of Congress, Washington DC.
- JWM-UP** John W. Mauchly papers, Ms. Coll. 925, University of Pennsylvania, Philadelphia.
- Agar, J. (1998). Digital patina: Texts, spirit and the first computer. *History and Technology*, 15(1-2):121–135.
- Ahmed, H. (2013). *Cambridge Computing: The First 75 Years*. Third Millenium, London.
- Barany, M. J. (2018). The officer’s three names: the formal, familiar, and bureaucratic in the transnational history of scientific fellowships. Preprint available at <http://mbarany.com/publications.html> (downloaded 4 March 2018).
- Bullynck, M. and de Mol, L. (2010). Setting-up early computer programs: D. H. Lehmer’s ENIAC computation. *Archive for Mathematical Logic*, 49(2):123–146.
- Burks, A. W. (1945). Notes on meetings with Dr. von Neumann, March-April 1945. AWB-IUPUI.
- Burks, A. W. (1989). Manuscript page headed ‘From H. H. Goldstine Sept ’68’. Comments dated February 10, 1989. AWB-IUPUI.
- Burks, A. W. (1998). Draft material for Chapter 6 of an unpublished book. AWB-IUPUI.

- Burks, A. W., Goldstine, H. H., and von Neumann, J. (1946). Preliminary Discussion of the Logical Design of an Electronic Computing Instrument. The Institute for Advanced Study, 28 June 1946.
- Campbell-Kelly, M. (2011). From theory to practice: the invention of programming, 1947-51. In Jones, C. B. and Lloyd, J. L., editors, *Dependable and Historic Computing: Essays Dedicated to Brian Randell on the Occasion of his 75th Birthday*. Springer.
- Campbell-Kelly, M. and Williams, M. R., editors (1985). *The Moore School Lectures: Theory and Techniques for Design of Electronic Digital Computers*, volume 9 of *Charles Babbage Institute Reprint Series for the History of Computing*. The MIT Press.
- Cunningham, L. E. (1945). Proposed set-up for the punch-card equipment of the ENIAC. 4 April 1945. HHG-APS, box 23.
- Curry, H. B. (1929). An analysis of logical substitution. *American Journal of Mathematics*, 51(3):363–384.
- Curry, H. B. (1945). Letter to Herman H. Goldstine, September 26, 1945. HHG-APS, box 21.
- Curry, H. B. (1950). A program composition technique as applied to inverse interpolation. Naval Ordnance Laboratory Memorandum 10337. March 29, 1950.
- Curry, H. B. and Wyatt, W. A. (1946). A study of inverse interpolation of the Eniac. Ballistic Research Laboratories Report No. 615. Aberdeen Proving Ground, Maryland. 19 August 1946.
- de Mol, L., Carlé, M., and Bullynck, M. (2015). Haskell before Haskell: an alternative lesson in practical logics of the ENIAC. *Journal of Logic and Computation*, 25(4):1011–1046.
- Dyson, G. (2012). *Turing's Cathedral: The Origins of the Digital Universe*. Allen Lane.
- Eames, C. and Eames, R. (1973). *A Computer Perspective*. Harvard University Press.
- Eckert, J. P. and Mauchly, J. W. (1945). Automatic High Speed Computing: A Progress Report on the EDVAC. September 30, 1945.
- Eckert, J. P., Mauchly, J. W., and Warren, S. R. (1945). PY Summary Report No. 1. Moore School of Electrical Engineering, University of Pennsylvania, March 31, 1945.
- Ensmenger, N. (2016). The multiple meanings of a flowchart. *Information & Culture*, 51(3):321–351.
- Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.
- Goldstine, H. H. (1944). Report of a conference on computing devices at the Ballistic Research Laboratory on 26 January 1944. ETE-UP.
- Goldstine, H. H. (1945a). Letter to Calvin Mooers, November 15, 1945. HHG-APS, box 21.
- Goldstine, H. H. (1945b). Letter to Haskell B. Curry, October 3, 1945. HHG-APS, box 21.

- Goldstine, H. H. (1945c). Letter to John von Neumann, April 13, 1945. HHG-APS, box 9.
- Goldstine, H. H. and von Neumann, J. (1946). Draft version of Goldstine and von Neumann (1947-8). JVN-LOC, box 33, folder 7.
- Goldstine, H. H. and von Neumann, J. (1947-8). Planning and coding problems for an Electronic Computing Instrument. Vol. 1, 1 April, 1947; vol. 2, 15 April 1948; vol. 3, 16 August 1948. The Institute for Advanced Study.
- Grier, D. A. (1996). The ENIAC, the verb ‘to program’ and the emergence of digital computers. *IEEE Annals of the History of Computing*, 18(1):51–55.
- Grier, D. A. (2005). *When Computers Were Human*. Princeton University Press.
- Haigh, T., Priestley, M., and Rope, C. (2014). Reconsidering the stored program concept. *IEEE Annals of the History of Computing*, 36(1):2–30.
- Haigh, T., Priestley, M., and Rope, C. (2016). *ENIAC in Action: Making and Re-making the Modern Computer*. MIT Press.
- Hartree, D. R. (1949). *Calculating Instruments and Machines*. The University of Illinois Press.
- Harvard (1946). *A Manual of Operation for the Automatic Sequence Controlled Calculator*. The Annals of the Computation Laboratory of Harvard University, Volume I, Harvard University Press.
- IAS (1945). Minutes of E. C. meetings #1 – #4. November 12, 19, 21 and 26, 1945. Institute for Advanced Study. HHG-APS, box 27.
- IBM (1936). *IBM Electric Punched Card Accounting Machines Principles of Operation: Collator Type 077*. International Business Machines Corporation, New York, NY.
- IBM (1945). *Machine Methods of Accounting*. International Business Machines Corporation, New York, NY.
- Jackson, J. B. and Metropolis, N. (1954). The MANIAC. Report LA-1725, Los Alamos, New Mexico, December 15, 1951. Revised July 16, 1954.
- Jimmy (n.d.). Letter to Herman Goldstine, undated. HHG-APS, box 51.
- Knuth, D. E. (1970). Von Neumann’s first computer program. *Computing Surveys*, 2(4):247–260.
- Knuth, D. E. (1973). *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley.
- Kullback, S. (1942). The British GC & CS. August 1, 1942. Box 1134, entry 9032 (Historic Cryptographic Collection), RG 457, US National Archives and Records Administration, College Park, MD.
- Mahoney, M. S. (2005). The histories of computing(s). *Interdisciplinary Science Reviews*, 30(2):119–135.
- Mauchly, J. W. (1944a). Notes from a conversation with Mr. Madow of the Census Bureau. October 9, 1944. JWM-UP box 10, folder 12.
- Mauchly, J. W. (1944b). Notes on a conversation with Lieutenant-Colonel S. Kullback, lecturer in statistics, George Washington University, Wednesday, October 11, 1944. JWM-UP box 10, folder 12.
- Mauchly, J. W. (1945a). Notes on possible meteorological use of high speed sorting and computing devices. April 14, 1945. JWM-UP box 10, folder 12.

- Mauchly, J. W. (1945b). Notes on some problems requiring high speed sorting and/or computing. April 14 1945. JWM-UP box 10, folder 12.
- Mauchly, J. W. (1945c). Notes on sorting problems occurring in cryptanalysis. April 14, 1945. JWM-UP box 10, folder 12.
- Mauchly, J. W. (1946a). Automatic sorting. Unpublished typescript, April, 1946. HML-UV, box 261.
- Mauchly, J. W. (1946b). Sorting and collating. In Campbell-Kelly and Williams (1985), pages 271–287.
- McClung Fleming, E. (1974). Artifact study. *Winterthur Portfolio*, 9:153–173.
- Montfort, N., Baudoin, P., Bell, J., Bogost, I., Douglass, J., Marino, M. C., Mateas, M., Reas, C., Sample, M., and Vawter, N. (2013). *10 PRINT CHR(205.5+RND(1)); : GOTO 10*. MIT Press.
- Moore School (1943). *The ENIAC: a Report Covering Work until December 31, 1943*. University of Pennsylvania, Moore School of Electrical Engineering.
- Moore School (1944). *The ENIAC: Progress Report Covering Work from January 1 to June 30, 1944*. University of Pennsylvania, Moore School of Electrical Engineering.
- Nofre, D., Priestley, M., and Alberts, G. (2014). When technology became language: The origins of the linguistic conception of computer programming, 1950–1960. *Technology and Culture*, 55(1):40–75.
- Norberg, A. L. (2005). *Computers and Commerce: A Study of Technology and Management at Eckert-Mauchly Computer Company, Engineering Research Associates, and Remington Rand, 1946-1957*. MIT Press.
- Priestley, M. (2017). AI and the origins of the functional programming language style. *Minds and Machines*, 27(3):449–472.
- Priestley, M. (2018). The mathematical origins of modern computing. In Hansson, S. O., editor, *Technology and Mathematics: Philosophical and Historical Investigations*. Springer. (forthcoming).
- Rohrhuber, J. (2018). Sans-papiers as first-class citizens. Unpublished chapter.
- Simondon, G. (2012). *Du mode d'existence des objets techniques*. Aubier.
- Stibitz, G. R. (1944). Report to Applied Mathematics Panel on Conference at Aberdeen on January 26. HHG-APS, box 20.
- Strachey, C. (2000). Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13:11–49.
- Truesdell, L. E. (1965). *The Development of Punch Card Tabulation in the Bureau of the Census 1890–1940*. U.S. Department of Commerce, Bureau of the Census.
- Turing, A. M. (1936). On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society, Second Series*, 42:230–265.
- Turing, A. M. (1950). Computing machinery and intelligence. *Mind*, 59:433–60.
- von Neumann, J. (1944). Letter to J. R. Oppenheimer, August 1, 1944. Los Alamos National Laboratory Archives, LA-UR-12-24686.
- von Neumann, J. (1945a). *First Draft of a Report on the EDVAC*. Moore School of Electrical Engineering, University of Pennsylvania. June 30, 1945.

- von Neumann, J. (1945b). Letter to Frank Aydelotte. 1 November 1945. JVN-LOC box 12, folder 1.
- von Neumann, J. (1945c). Letter to Haskell B. Curry, August 20, 1945. ETE-UP.
- von Neumann, J. (1945d). Letter to Herman Goldstine, February 12, 1945. HHG-APS, box 9.
- von Neumann, J. (1945e). Letter to Herman Goldstine, May 8, 1945. HHG-APS, box 21.
- von Neumann, J. (1945f). Letter to Howard Aiken *et al.*, January 12, 1945. JVN-LOC, box 7, folder 14.
- von Neumann, J. (1945g). Memorandum on mechanical computing devices. 30 January 1945. HHG-APS box 21.
- von Neumann, J. (1945h). Memorandum on the program of the high-speed computer project. 8 November 1945. HHG-APS box 21.
- von Neumann, J. (1945i). Unpublished manuscript describing the development of a meshing routine. HHG-APS box 21. Reproduced in Appendix B of this book.
- von Neumann, J. (1946). Letter to Herman Goldstine, September 16, 1946. HHG-APS, box 21.
- von Neumann, J. (1947). Letter to Herman Goldstine, March 2, 1947. HHG-APS, box 20.
- Wilkes, M. V., Wheeler, D. J., and Gill, S. (1951). *The Preparation of Programs for an Electronic Digital Computer*. Addison-Wesley Press, Inc.
- Williams, S. B. (1944). Calculating system. Bell Telephone Laboratories, March 29, 1944. HHG-APS, box 20.
- Winder, R. (2005). *Bloody Foreigners: The Story of Immigration to Britain*. Abacus.

Index

A

Aberdeen Proving Ground *see* Ballistic Research Laboratory
Aiken, Howard 11, 82
American Philosophical Society 2, 89
Atanasoff, John 5
Aydelotte, Frank 82, 84

B

Babbage, Charles 84
Ballistic Research Laboratory 1, 4, 10, 19–20, 62, 86
 computations committee 10, 70, 87
Bell Telephone Laboratories 19
 relay computer 19–23, 25, 33, 71, 82
 Relay Interpolator 19, 35
Bletchley Park 10
Bloch, Richard 84
Block diagram *see* Diagram, block
Box *see* Diagram, box
BRL *see* Ballistic Research Laboratory
Burks, Arthur 6, 12, 15–16, 49, 81

C

Campbell-Kelly, Martin 75, 83
Comrie 84
Conditional branching 17, 40, 47, 56, 58–59, 80, 86
 jump instruction 50–52, 55, 63
 on collators 9
 on EDVAC 25–27, 30–32
Cunningham, Leland 10, 11, 19, 62
Curry, Haskell 28, 62, 70, 75, 87–89
Cycle

major 22–23
minor 5, 15, 22–30, 32, 40, 50

D

De Prony, Gaspard 80, 85
Delay line 22–23, 27–28, 40, 51, 59, 76, 89
 long tanks 27–32, 41–48, 50, 51, 97–99, 103–107
 short tanks 27–32, 36, 41–42, 44–48, 50, 51, 97–104, 106, 107
Diagram 10, 20, 39
 block 53–55, 57–62, 64–66, 76–79, 87
 box 39, 58, 59
 alternative 55–58, 63, 65, 67, 86
 assertion 65, 85
 control 53–55, 60, 65, 76, 86
 operation 54, 55, 63, 64, 77, 86
 storage 53–55, 59–61, 64–66, 77, 86
 subroutine 61, 67, 77
 substitution 55–57, 59, 64–65, 85–87
 flow *v.* 53, 62–68, 76–79, 85–87
Differential analyzer 1, 75, 82

E

Eckert, Presper 1, 4, 5, 9, 12–16, 27, 28, 30, 33, 34, 49, 89
ECP *see* Institute for Advanced Study, Electronic Computer Project
EDSAC 75–76, 80, 83, 84
EDVAC vi, 2, 4, 5, 7, 9–15, 17, 19–37, 40, 41, 47–51, 59, 62, 69–71, 81–86, 88, 89
 diagram of memory 23, 28
 First Draft of a Report on the 1, 2, 5, 6, 12–14, 17, 19, 21–25, 27–30, 32, 33, 36, 50, 51, 82, 83

word layout 26, 30, 50
 ENIAC 1–2, 4, 10, 20, 33–35, 37, 39–40, 50,
 53, 62, 69, 82, 84–87
 Monte Carlo application 77–79, 85–86
 Ensmenger, Nathan 66
 Equation 63, 82
 differential 1, 4, 10, 33–35, 82

F

Flow diagram *see* Diagram, flow
 Flowchart 10, 62

G

Goldstine, Herman v, 1–2, 4–6, 11–12,
 15–17, 20, 27, 36, 46, 49, 53–68, 70–80,
 83, 87, 88

H

Hartree, Douglas 10, 80, 84
 Harvard
 Mark I 20, 23, 33, 35, 37, 39–40, 69, 84,
 85
 Mark II 84
 Hollerith, Herman 10
 Hopper, Grace 82, 84

I

IAS *see* Institute for Advanced Study
 IBM 23
 punched card machines 7–13, 21, 36, 68
 SSEC 84
 Institute for Advanced Study 49, 50, 81, 82
 Electronic Computer Project 6, 15–17, 33,
 34, 49–52, 56, 59, 68, 78, 82, 83
Preliminary Discussion 34, 49–52

K

Knuth, Donald 3, 15, 17, 44
 Kondopria, Akrevoe 6
 Kullback, Solomon 10–12

L

Long tank *see* Delay line, long tanks
 Los Alamos 1, 4, 10, 21
 MANIAC 68, 78
 Lovelace, Ada 84

M

Madow, William 10
 Mahoney, Michael 3
 Mauchly, John 1, 5, 9–17, 27, 28, 30, 33, 34,
 49, 89
 Mergesort v, 2, 15–17, 46, 60, 66–68
 Mooers, Calvin 5–6
 Moore School of Electrical Engineering 1,
 4, 5, 49
 1946 summer school 5, 16, 86

N

Newell, Allen 80

O

Oppenheimer, Robert 21

P

*Planning and Coding of Problems for an
 Electronic Computing Instrument* 2,
 17, 48, 49, 53, 56, 57, 59, 60, 62, 66–69,
 71, 73–80, 83–87
 Punched cards *see* IBM

R

Rochester, Nathaniel 86

S

Short tank *see* Delay line, short tanks
 Simon, Herbert 80
 Simondon, Gilbert 83
 Stibitz, George 19–21
 Subroutine vi, 30, 34–36, 42, 46, 49, 61,
 67–80, 87, 88, 107
 Substitution 25–26, 29–34, 36, 42, 48, 55,
 56, 61, 63, 67, 69, 79, 83, 87–88, 90,
 102–103, 106
 box *see* Diagram, box, substitution
 orders 24, 29, 50–52, 55, 58–59, 87,
 91–93
 S_{15} 45–48, 70, 72, 107
 sequence 73–74, 87

T

Turing, Alan v, 3, 32, 81, 86, 88

U

US Army Ordnance Department *see*
Ballistic Research Laboratory
US Army Signal Corps 10–11
US Census Bureau 7, 10, 11, 13

V

Variable remote connections 59, 61, 63, 67,
77–79
Von Neumann architecture v, 21–23
Von Neumann, John v, vi, 1–6, 11–17, 19,
21–62, 64–89
EDVAC codes 24, 29, 51, 89–93

meshing routine manuscript 2–7, 9, 12,
16, 17, 19, 27, 31, 32, 35–38, 40–42,
44–48, 53, 55, 57–59, 61, 66, 95–108
meshing routines 2, 13, 17, 57–60, 66–68
sorting routines 2, 13, 60–61, 66–68

W

Wheeler, David 75–76
Wiener, Norbert 11
Wilkes, Maurice 80
Wilks, Samuel 11
Williams, Samuel 20–21, 23, 82
Wyatt, Willa 62