Personal Recollections of Programming
DEUCE in the Late 1950s

MICHAEL WETHERFIELD

8 Bafford Lane, Charlton Kings, Cheltenham GL53 8DL, UK
*Corresponding author: robyw@machination.eclipse.co.uk

The author describes how he came to be employed, in 1957, as a programmer at Nelson Research Laboratories (Stafford), then the hub of the English Electric Company’s software activities, at a time when the throughput of the English Electric DEUCE computer had just been potentially improved by doubling the amount of data that could be punched on each Hollerith input card, necessitating a corresponding increase in the efficiency of the decimal-to-binary conversions used by card-reading subroutines. The DEUCE delay-line store, instruction code and input–output system are described in enough detail to enable readers to understand the difficulties, and how they were resolved.

Keywords: English electric DEUCE computer; experience of programming DEUCE; DEUCE instruction code; programming challenge in response to DEUCE enhancement

Received 03 February 2009; revised 20 September 2009
Handling editor: Martin Campbell-Kelly

1. INTRODUCTION

Bliss was it in that dawn to be alive

Wordsworth

This paper describes some of my early (1957–60) experiences as a programmer of the English Electric Company’s DEUCE computer, which had been operational since 1955 and whose prototype was the ‘Pilot Automatic Computing Engine (ACE)’ designed and developed at the National Physical Laboratory (NPL) at Teddington—the connection came about through an arrangement for English Electric engineers to assist NPL in the final stages of construction of the Pilot ACE, itself first demonstrated to the public in 1950. The ACE design arose from a 1945 official paper entitled ‘Proposed Electronic Calculator’ (usually referred to as the ‘ACE Report’—reprinted in [1]), written by the visionary A.M. Turing—already celebrated for his ‘Computable Numbers’ paper of 1936—when he joined the NPL Mathematics Division. The project to develop ACE was approved by the NPL Executive Committee in March 1946. In 1947, Turing himself effectively left the project, disillusioned by the lack of progress in constructing a working model, eventually resigning his NPL post on 28 May 1948 [2]. He died in 1954.

This paper aims partly to describe how DEUCE programming was ‘done’ at English Electric’s Stafford Laboratory (NRL) in 1957–60 and, in that context, discusses a programming problem (not mentioned in [1]) vitally affecting DEUCE’s throughput; this problem arose when, in 1957, DEUCE’s Hollerith punched-card input and output equipment was upgraded so that twice as much information could be punched into, and read from, each card. Although DEUCE software existed for converting decimal numbers read from cards to binary in ‘real time’, immutable timing constraints imposed by the card reader meant that there was no straightforward method of adapting these subroutines to work when twice as many numbers needed to be converted. How that problem was solved is the real subject of this paper. I have included what I hope is enough description of the DEUCE instruction code to enable readers to understand both the problem and its solution—if there seems to be too much detail, it is present to give a better picture of the challenges faced by DEUCE programmers.

2. HOW I BECAME A PROGRAMMER

In 1956, at the age of 23, I left Cambridge with a First-class Mathematics degree and very little idea of what sort of career I wanted, except that more advanced mathematics, and teaching, were ‘out’. Electronics and computers were a closed book to me; in my ignorance I tended to think of computers simply as superior calculating machines which were fed, and regurgitated, ‘media’—paper tape in the case of EDSAC (a lot of which...
seemed to find its way into the gutters around the building in Cambridge which housed it.

With paternal advice and assistance, a list of reputable potential employers was prepared, as well as a letter with which to introduce myself to them. After a round of interviews, I accepted an offer from Boots the Chemists—largely because I appreciated the friendly attitude of their interviewers—and took up employment in their Organization and Methods (O & M) department in Station Street, Nottingham, in the autumn of 1956, with a salary of £625 per annum.

Boots gave me a grounding in their business and also in the characteristics of various office machines which it was the responsibility of the O & M department to recommend for use within the company. It became apparent that they were preparing to ‘computerize’, and had identified me as a likely ‘computer man’. Boots had not only already decided on the first task of the computer, specifically, stock control; in order to provide a really accurate, machines—their chosen system was the ‘ELECOM 125’ (made in the USA by the Underwood Typewriter Company) which comprised two independent processors, one of them ‘conventional’ and the other an auxiliary dedicated to performing autonomous ‘merging sorts’ on four attached magnetic tape decks.

None of the staff at Boots at that time appeared to have had any direct computer experience. I and another O & M colleague, recruited at about the same time, were given some literature about the ELECOM 125, including a description of the instruction code; from this we learnt that the system’s immediate storage medium was a magnetic drum, faster access to a few ‘registers’ being obtained by providing multiple write heads round certain drum tracks so that storage locations on those tracks were effectively ‘replicated’, i.e. multiple copies of data would be written simultaneously, and when required could be read from the ‘copy’ nearest to the reading head. Rudimentary though this manual was, it really helped to stimulate my own burgeoning interest in programming—encouraged partly by the purchase of B. V. Bowden’s Symposium ‘Faster Than Thought’.

This first venture into the world of computers was destined to end in tears. Boots at that time did not have the experience to undertake the necessary systems analysis to implement their task, so help was inevitably sought from the computer’s manufacturer. After studying the task in Nottingham, the Underwood experts gave their opinion to the effect that the ELECOM 125 system would require more than 24 h to perform the Boots daily job! Whether or not multiple ELECOMs could have coped with it was a question left unanswered, because at about this time Underwood decided to withdraw from the computer market, and stopped making them altogether; Boots perforce cancelled their order—they eventually bought an EMIDEC computer.

By this time, I really wanted to get into computer programming, the sooner the better. There seemed little likelihood of Boots getting a computer quickly enough to satisfy this urge, so I decided to look for a programming job with an established computer manufacturer. My original series of interviews had included a positive one with the English Electric Company at their Atomic Power Division in Whetstone, near Leicester, and accordingly I wrote to them, explaining my situation; an interview was arranged for me at Nelson Research Laboratories (NRL), located in pleasant countryside about 2 miles East of Stafford. This was the hub of English Electric’s programming activities, with two English Electric DEUCE computers in operation. I was interviewed at NRL by Allan Gilmour, and was offered a job (at £800 p.a.) which I speedily accepted.

Boots released me amicably, though not without pointing out that they had just engaged John Boothroyd, a senior engineer from English Electric, to be their chief computer engineer. I remember thinking ‘He’s in for a disappointment’; and so it proved.

I started working for English Electric at NRL at the beginning of August 1957. My first job title was ‘Mathematician’; in practice this meant ‘DEUCE programmer’. I remained with English Electric, and its successors (ICL, and ultimately Fujitsu), in various departments and locations—not all the time as a programmer—until retiring in December 1991.

3. NRL AND THE DEUCE PROGRAMMING ENVIRONMENT

In 1957, English Electric was a big company, employing over 30000 people. The chairman was Lord Nelson and the managing director was his son, Sir George Nelson. The company had large factories dotted around the country; there was one in Stafford, separate from NRL, which produced electric motors, including those used to power pithead gear and propel submarines and other things. DEUCE computers were manufactured in Kidsgrove, about 20 miles North of Stafford. The company also produced warplanes for the RAF (Canberra and Lightning) at Warton, Lancashire, diesel locomotives (the highly-regarded ‘Deltic’) at Rugby, equipment for atomic power plants, guided weapons systems, valves, semiconductors, etc., as well as TVs, washing machines and refrigerators. It gave one quite a feeling of security to be a cog in this huge machine! English Electric also owned the name ‘Marconi’, which in those days still carried prestige.

NRL housed, besides the two DEUCEs mentioned and their maintenance engineers (and a number of development engineers, working on new designs), all the people who served the DEUCE computers—managers and secretarial staff, programmers, data preparation staff (the ‘punch room’), and also the ancillary Hollerith equipment for printing and copying the punched cards which were DEUCE’s primary input–output medium.

NRL was at the heart of English Electric’s involvement with DEUCEs in the field. At that time, there were up to 30 DEUCEs in operation, some of them at other English Electric sites, e.g. the
London computer bureau at Marconi House in the Strand, with whom NRL had close relations, and also other locations where they did vital work connected with the English Electric products developed and manufactured there; Stafford Works had its own DEUCE, independent of those at NRL, but sometimes sought programming assistance from NRL—one of my earliest tasks was to write a program enabling them easily to select from their catalogue the best ‘mine winder’ motors for pithead equipment, depending on the ‘duties’ they were required to perform. There were also DEUCEs at government scientific establishments such as the NPL (Teddington) and RAE Farnborough, and at various Universities, one or two abroad, e.g. at the New South Wales University of Technology in Sydney, virtually all of them engaged on ‘scientific’ work; though the NRL DEUCEs were used to schedule work for a textile mill in Leek, and I believe that the Agriculture Ministry DEUCEs at Guildford did largely non-scientific work. From NRL the DEUCE Users’ Group was run—this held periodic meetings, usually in London, at which papers were read and users could air their views and have them discussed; NRL published a monthly typed and duplicated ‘magazine’ called ‘DEUCE News’, for distribution to all users; and the extensive program and subroutine library was maintained at NRL, and thence distributed, on punched cards. Working at NRL, one came to meet quite a few users and sometimes to visit their sites, and relations were generally cordial. In addition, NRL ran a Bureau operation for various customers, e.g. the above-mentioned Leek textile mill.

Cliff Robinson was the driving force in charge of the DEUCE programming activity at NRL. Allan Gilmour, an expert in the solution of railway problems (who I believe also wrote the very successful election-result-forecasting program used when ITV televised the NRL DEUCEs ‘live’ as part of its coverage of the 1959 General Election), was Cliff’s second-in-command. The programming department was quite small when I joined; those with whom I worked most closely at that time included John Denison, Neville Hawkins, Peter Holland, Jim Lucking, David Ozanne, Roger Smith (who edited ‘DEUCE News’ and controlled the subroutine library) and also John Boothroyd, who as mentioned earlier had been recruited by Boots from English Electric, but subsequently re-joined; he eventually became head of the programmer-training department, with David Ozanne as his second-in-command.

There was a remarkably friendly atmosphere at NRL, partly because of the comparatively small number of people, but also, I think, due to exemplary management. Everybody knew everybody else. In our spare time, many of us were members of Stafford Players, a local amateur dramatic group. At Christmas, the two DEUCEs could be made to play duets; musical tones were generated by programs which switched the card output equipment on and off at the required frequency—loudspeakers being substituted for the card punch.

DEUCE was a valve machine—there were, as far as I know, no transistors in it. Its directly addressable storage was located in mercury delay lines (as specified by Turing in the ACE report).

It consisted of a grey cabinet, measuring approximately 10 feet long, 6 feet wide and 8 feet high, containing all the valve circuitry and the magnetic drum (backing store). The cabinet had doors on both sides, and also at one end giving access to the (warm) interior. At the other end was a console desk and various switches, indicator lights and oscilloscopes capable of showing the current contents of all the delay lines—essential for program testing; the card reader and card punch were positioned either side of the console. At NRL, the cabinets and consoles for both DEUCEs were very visible, positioned along the centre of one large glass-sided room. Near each DEUCE was a separate mushroom-shaped container holding all its delay lines (and maintaining the correct operating conditions for them), and other grey cabinets against the walls held power-supply units.

DEUCE itself had no line printer. The DEUCE card reader could read 200 cards per minute; the card punch could output 100 cards per minute. A key member of the staff was Wally Somerfield, the expert who knew how to ‘plug up’ the Hollerith tabulator in order to print, in any desired format, the alphanumeric data punched on cards either by DEUCE itself, or by the data preparation staff. All the Hollerith card machines, in particular, the DEUCE reader and punch, but also ancillary equipment including card copiers (which could serially number cards) and desk hand punches, were extremely reliable. Punched cards then seemed an ideal medium, as long as the cards in one’s pack were numbered—otherwise dropping a pack of cards could be disastrous. Some DEUCE users, Liverpool University for example, used paper tape input and output, but I believe that in this respect they were ‘self-supporting’.

The NRL DEUCEs had no programmed operating system or dedicated operators, and a programmer wishing to test a program or run a job would simply put his name in a book with the amount of time he required, prepare his card pack (as described below) and wait for his turn, usually then acting as his own operator. Each person having finished his time on the machine, which might be either of the two DEUCEs, would summon the next in the queue. Most work was done in normal hours, and the machines were turned off overnight; but sometimes they were ‘lent’ to parties of programmers from other English Electric sites requiring extra capacity who might use the machines into the small hours.

Initial training, in my own case, consisted of being given a programming manual written by George Davis and Vic Price of the London Bureau, and attending a short induction course given by Peter Holland for new joiners (the course must have included instruction in how to use the DEUCE subroutine library), during which I really came to appreciate the possibilities of DEUCE’s

---

2The cards measured \(7\frac{1}{2} \times 3\frac{1}{2} \text{ in.}\), and were 0.007 in. thick; the punched holes were rectangular.

3Sadly, this excellent manual (soon to be withdrawn) was then obsolescent due to its omission of enhancements recently made to the specification of DEUCE, including the ability to read and punch 64 card columns, and the ‘Automatic Instruction Modification’ feature applicable to instructions executed from certain locations.
instruction code: the particular examples Peter used to illustrate these included the ‘standard’ method of reading decimal numbers from punched cards and converting them to binary.

For the next month or two, I was left effectively on my own—albeit surrounded by colleagues always ready to answer any question or help with any problem—to attempt my first computer program (in England we had recently accepted the advantage, in this context, of spelling the word the American way). I wanted to calculate \( \pi \) to more than 100 decimal places, using one or more of the well-known arc cotangent identities—in the event, Machin’s. This involved devising some subroutines for performing arithmetic on 512-bit numbers, and a routine, based on these, to sum Gregory’s series for the inverse tangent, together with a framework for the whole which would culminate in the output of the result, in decimal format, to punched cards.

It all worked out very well, producing at least 145 correct decimal figures\(^4\)—moreover, thanks to my very careful checking both of the coding and of the cards onto which I had punched the program, it worked the first time I attempted to run it, thus disproving a current fallacy. Thereafter, I always expected my programs to work ‘first time’, and was rarely disappointed.

This was a revelatory moment in my career—indeed, in my life. Although my school and university experience had shown that I could appreciate the work of the mathematicians of the past, and more importantly could understand it well enough to be able to answer correctly almost any examination question I was faced with, I had had to come to terms with the fact that I could never consider myself a ‘real’ mathematician—that is, I did not feel equipped to create fresh mathematical ideas. But now here was this new, very satisfying creative activity, at which I seemed to be proficient—and I was getting paid to do it! It seemed then, and still seems to me, that the programming of electronic computers was one of the great new crafts invented in the twentieth century. Because of the quart-in-a-pint-pot ingenuity of its instruction code, DEUCE remains for me, and I am sure for many others, a ‘favourite’ computer. The late Jim Lucking, a much-missed former colleague, was so enamoured of DEUCE that later, when he was managing a section of programmers writing test programs in machine code for English Electric’s ‘System 4’ computers, he insisted (as he told me at the time) that they should all learn the DEUCE instruction code—what their reaction was, I never discovered. It was sometimes said of DEUCE: ‘If you can program DEUCE, you can program anything’—no doubt Jim had this (probably true) remark in mind.

DEUCE was often described as a ‘difficult’ computer to program, but I would dispute this. Mastery of any computer requires on the programmer’s part a thorough knowledge of the capabilities of the machine; a programmer’s skill lies in the ability to exploit these capabilities to the full. Virtually all of the external details of the instruction code which unlocked DEUCE’s capabilities are presented in the one-page Appendix at the end of this paper; that this is possible does not support the notion of ‘difficulty’—‘not easy to learn’ might be a fair description.

After initial training, my own very rewarding personal involvement with DEUCE was comparatively short—not much more than 2 years. For part of this time, I was employed in writing programs to assist different departments at English Electric’s Stafford Works, and also helping some crystallographers from Oxford University who were investigating computer possibilities; latterly, I spent most of my time developing software for the DEUCE program library, including some of the work described in Sections 8 and 9. I was than moved on to work on the development of English Electric’s ‘third-generation’ computer (KDF9). At about this time (1960), many of the programmers and engineers at NRL, including myself, were relocated to English Electric’s Kidsgrove site.

4. DEUCE SOFTWARE AND CARD PACKS

There was a comprehensive DEUCE program library, on cards, at NRL. The breadth of one’s experience of what it contained depended very much on the programming tasks one was set to do.

DEUCE’s ancestor having been the NPL Pilot ACE of 1950, a great deal of the DEUCE software was originally written for the Pilot ACE and punched on cards at the NPL, and had to be adapted to work on DEUCE because the delay line numbering for the two machines was somewhat different. Cliff Robinson, in a letter to me a few years ago, described an occasion during the time of software handover when, with the aid of his 3- or 4-year-old son, he had to spend a weekend changing punched binary 10s (1010) to 12s (1100) (presumably in instructions for filling the matching delay lines). Cliff’s task was to fill in the ‘2’ holes, using ‘chads’ from a card punch; his son was allowed to punch the ‘4’ holes! This use of chads to ‘stop up’ card holes was common practice among DEUCE programmers during program testing; the filled-in holes could usually survive transits through a card reader without the ‘stoppers’ falling out, but it was still advisable to copy the doctored cards sooner rather than later.

DEUCE subroutines existed for reading-in and punching-out decimal numbers; for performing floating-point operations (DEUCE’s arithmetic instructions were fixed-point only); and for standard mathematical operations, e.g. finding roots, trigonometric functions, etc. The library of subroutines was held on cards that could be copied and included in normal programs; each subroutine was available in versions to be stored in, and executed from, different delay lines, to suit the programmer’s convenience. The description of each subroutine would specify in which ‘short’ delay line the user’s ‘exit’ instruction, or ‘link’, was to be placed prior to entry. Subroutines which called other subroutines were described as a ‘second’ (or higher) ‘order’.

At the ‘automatic programming’ level, there was a very powerful General Interpreting Program (GIP), originating from the NPL, which could be used to program operations on matrices.

\(^4\)Recently re-ran this program successfully on my PC, using David Green’s DEUCE simulator, written in Australia.
and determinants; the user of GIP would call up very efficient linear algebra subroutines, called ‘bricks’. Many DEUCEs were sold simply on the merits of GIP [3, 4]—it was certainly much used at English Electric’s Warton site for aeronautical calculations. I personally acquired virtually no experience of GIP, not having been required to deal with problems involving matrices or determinants. However, the last DEUCE routine I actually wrote was a ‘brick’ for reading in cards punched with eight floating-point decimal numbers, each occupying eight card columns (one column for exponent, seven columns for ‘mantissa’) — this was to enable GIP users to exploit DEUCE’s enhanced ability (introduced in 1957) to read from 64 card columns instead of only 32.

There was also available a simple three-address autocode, called ‘Alphacode’, originated at NRL, which provided a very convenient method of performing input, output and arithmetic (including trigonometric) operations on fixed- and floating-point numbers. Alphacode programs, written on standard coding sheets, would be punched in a standard alphanumeric format, one instruction (or numerical constant) per card; these cards would be read, converted to a standard binary format and the result output onto cards by an assembler program actually called the ‘Alphacode Compiler’. These binary cards could then be read (followed, if required, by alphabetically punched variable data cards) and executed by the ‘Alphacode Interpreter’. Whatever Alphacode lacked in sophistication or efficiency it more than made up for in convenience; in this respect, it was the equivalent of a programmable pocket calculator. I used Alphacode myself on more than one occasion, and later wrote (in DEUCE machine code) the Mark 2 version of the Alphacode Compiler.

ACE and DEUCE software are comprehensively described in [1]—many of DEUCE’s library programs are described in more detail in [5, 6]. [7] describes a project by Fraser Duncan and David Huxtable (who later, at Kids grove, jointly developed an ALGOL Compiler for KDF9) to turn Alphacode into a ‘proper’ high-level language, i.e. to write a program called the ‘Alphacode Translator’ that would convert the output of the Alphacode assembler directly into DEUCE machine code, and thus achieve a much greater efficiency.

A normal card pack for running any DEUCE program would consist of a standard ‘initial card’, followed by cards holding the program, in binary form (i.e. as 32-bit ‘words’—a hole in a card represented a 1-bit, absence of a hole a zero), and finally binary or decimal data cards to be read in by the program. DEUCE punched cards had 80 columns, but information was (at first) only held in a subset of 32 columns; each of the 12 rows on a card could, therefore, hold the binary contents of one DEUCE word. Cards would be read in one row at a time. DEUCE program storage was allocated in units of 32 words, held, during execution, in the first eight (of 12) 32-word mercury ‘delay lines’. When reading in a program, three cards (a ‘triplet’) were required to hold the contents of one of these delay lines. By convention, the first (top) four rows of the leading card of each triplet would contain instructions causing the words in the remaining 32 rows of the triplet to be read into the appropriate delay line; in the last of these triplets, the instruction in the fourth row would also indicate, as its successor, the first instruction to be executed of the now-loaded program.

Parts of large programs requiring more storage than the available eight delay lines could be loaded to the DEUCE magnetic drum, whose function was solely to provide individual programs with additional data and instruction storage; this could hold two hundred and fifty-six 32-word units (‘tracks’) — a total of 8192 words. In a program pack, ‘drum’ triplets would follow the ‘delay line’ triplets described in the preceding paragraph; the contents of the first row of each drum triplet would simply identify the destination track(s), and a special subroutine included in the already-entered program would be used to load the contents of the triplet to those tracks, through a dedicated delay line (when the 32 available card columns were expanded to 64, each drum triplet was able to hold the contents of two tracks).

The last card in a program pack would normally have a hole punched in a certain column of its 12th row; the card reader could detect this, and would stop itself accordingly.

Programmers would write their programs on standard ‘coding sheets’ (an example is shown in Fig. 2), and could either punch their own programs, in binary form, onto cards, using Hollerith desk punches, or have them transferred to cards in the punch room. A turn-round time was involved with this option, and many programmers preferred to punch their own program cards; but large amounts of data were usually left to the punch room.

5. DEUCE DELAY LINE STORAGE

A ‘delay line’ was essentially a tube of mercury. Using electronic wizardry, a sequence of 0s and 1s, e.g. one or more computer binary ‘words’, could be ‘input’ at one end of a delay line, being converted to a string of binary pulses within the mercury, which travelled, in the form of a sound wave—whence the term ‘acoustic delay line’ sometimes used—to its other end and, using more wizardry, could there be ‘output’, i.e. converted back into the same sequence of 0s and 1s. ‘Storage’ for any length of time was effected by continual re-input of the ‘refreshed’ output. The pulse rate for all delay lines (i.e. the rate at which binary digits were converted to pulses on input, and back again on output) was 1 million bits per second, i.e. it took 32 μs to write or read a 32-bit word, but, as should become clear, this was not in practice the same as ‘access time’.

The amount of information that could be held in a delay line depended mainly on its physical length. In DEUCE, delay lines came in four sizes: one-word, of which there were four, called TS13, TS14, TS15 and ‘TS16’—‘TS’ standing for ‘temporary store’; two-word (three 64-bit delay lines, called DS19, DS20 and DS21); four-word (two 128-bit delay lines, called QS17 and QS18) and 32-word (1024 bits, called ‘long delay lines’; there
were 12 of these, DL1 to DL12). The numbers 1–21 uniquely designating these delay lines were used in 5-bit fields in DEUCE instruction words to address them, providing a total of 402 words of directly accessible storage.

‘Data’ words could be held in any delay lines. A field in every ‘instruction’ word was provided for specifying the long delay line from which its ‘successor’ instruction was to be ‘extracted’; however, because this field was only 3 bits long, the DL referred to had to be in the range 1–8, so instructions were normally held in these—though the DEUCE instruction code also provided one very powerful form of instruction whose function was to extract its successor from any delay line (specified in one of the above-mentioned 5-bit fields).

To ‘synchronize’ storage both within the set of delay lines, and between the delay lines and the instruction-sequencing and arithmetic units, DEUCE included a ‘clock’ which in effect ‘ticked’ every 32\(\mu\)s; this 32\(\mu\)s interval between ‘ticks’ was called a ‘minor cycle’ (m.c.). Input to, and output from, delay lines in effect started at the beginning of a minor cycle and was completed at the end of a minor cycle—in this way, the division of sequences of data pulses into 32-bit words was effected. A ‘major cycle’ comprised any 32 consecutive minor cycles (1024\(\mu\)s, i.e. approximately 1 ms). Minor cycles were numbered ‘modulo 32’, i.e. from 0 to 31. Within each 32-word delay line, each individual word was accessible just once in every major cycle, whereas the 1-word delay lines were accessible in any minor cycle, and individual words in 2- and 4-word delay lines could be accessed 16 times and 8 times, respectively, in every major cycle. From the programmer’s point of view each of the 32 locations in a long delay line was numbered according to the minor cycle in which it could be read or written, so that consecutive locations in, say, delay line 11 were referred to as DL11 m.c. 0, DL11 m.c. 1, and so on, up to DL11 m.c. 31—in shortened form, 110, 111, ..., 1131. Naturally, the numbering of minor cycles was synchronized between delay lines; it was possible using a single instruction to transfer one or more consecutive words from one long delay line to another, but they had to be in correspondingly numbered minor cycles.

Synchronization of magnetic drum tracks was along the same lines—when the first transfer (necessarily, of the contents of DL11, the ‘dedicated’ delay line) to the drum took place in a program, hardware ensured that any subsequent transfer to or from the drum preserved the same minor cycle numbering.

In each of the two-word delay lines DS19, DS20 and DS21, one word corresponded to an even minor cycle—conventionally identified in hand-written code, for some reason,\(^5\) by the suffix ‘2’ rather than ‘0’, e.g. 192—and the other to an odd minor cycle (identified by suffix ‘3’, e.g. 213). Direct transfers from one half to the other were ruled out; but transfers between ‘even’ halves and any of the even minor cycles in any of the multiple-word delay lines were permitted, and similarly for the ‘odd’ halves. In each of the four-word delay lines (QS17, QS18), the four words were conventionally identified with suffixes 0, 1, 2 and 3 (e.g. 170, 181), and access was more restricted: transfers to or from words with suffix \(n\) could only take place in minor cycles congruent to \(n\), modulo 4—for example, transfers between any long delay line and 173 (or 183) could only occur in minor cycles 3, 7, 11, 15, 19, 23, 27 or 31. Multiple-word transfers between delay lines could start in any minor cycle, subject to the foregoing rules—thus, for example, one ‘double-length’ instruction, starting in a minor cycle congruent to 0, modulo 4, could be used to transfer the contents of 170 and 171 to 192 and 193, respectively.

In general, transfers of single words between different minor cycle locations in long delay lines could only be effected through the short delay lines, using two or more instructions—for transfers between ‘even’ and ‘odd’, only through the one-word delay lines TS13, TS14, TS15 and TS16, which could be accessed in any even or odd minor cycle.

As far as the DEUCE instruction code was concerned, although delay line numbering was absolute, all explicit references to minor cycles were relative; as a rule, to the minor cycle in which the current instruction was ‘extracted’ from storage. The minor cycle referred to as ‘0’ in each long delay line, in effect the programmer’s point of reference, was redefined whenever a new program card pack was loaded, at which time the entire delay line store (and, in effect, the magnetic drum store) was ‘cleared’. As mentioned earlier, long delay lines were filled from triplets of cards; the delay line word into which the contents of the fifth row of the first card of the first triplet were inserted was designated ‘minor cycle 0’, thus defining the numbering of all minor cycles (but only for the duration of the program in the computer)—it goes without saying that subsequent triplets were loaded into matching minor cycles in the other long delay lines.

There was, however, one important constraint involved, due to the instruction code permitting 64-bit shifts to take place in DS21, as well as ‘carries’ and ‘borrows’ between the two halves of DS21: the necessary continuity of these two halves, defining their relative significance, and consequent asymmetry, was built into the hardware, and only worked in one way. By convention, the ‘even’ half of DS21 (212) was considered to be its less significant half and, therefore, it had to be ensured that the designated minor cycle 0 was ‘in synch’ with this half of DS21. Identifying it was a relatively straightforward trial-and-error task performed by instructions stored on the standard ‘initial card’ preceding the first triplet of any card pack, and executed in the course of reading that card.

The individual bits (pulses) of any DEUCE word were numbered from P1 (least significant) to P32, and they were emitted from any delay line in that sequence. This made it possible to add two operands as soon as binary digits started to emerge.

---

\(^5\)Probably the reason was that ‘0’ might easily be confused with ‘O’ (for ‘Odd’).
from the delay lines containing them (one of the operands necessarily being the single- or double-length ‘accumulator’ delay line, TS13 or DS21), and immediately to feed the resulting sum back into that ‘accumulator’, replacing its original contents. Subtraction was performed in the same way, a copy of the subtrahend effectively being negated as it emerged (least significant bit first) from its delay line, prior to addition. DEUCE used the ‘2’s complement’ convention for negative numbers, meaning that to negate a number it was only necessary to ‘invert’ each bit emerging after the first ‘1’ emitted. It was thus possible to generate and store the results of addition, subtraction, 1-bit shifting and ‘bit-wise’ logical operations, within the time required to extract the operands—32 or (double-length) 64 μs—giving DEUCE its very high potential speed of operation.

Each DEUCE instruction word could be extracted from its location during the last minor cycle of operation of its predecessor, but a further minor cycle was then always required for decoding the instruction before it could start its ‘operation’, which would last for one or more minor cycles. The fastest possible execution rate was therefore one instruction every two minor cycles; but this rate could only realistically be achieved during operations on the 1-word delay lines, which, as previously stated, were accessible in any minor cycle.

It remains a curiosity that in any DEUCE binary word punched on a card, P1 was on the left and P32 on the right. The Hollerith desk punch would, like a typewriter, traverse a card from left to right, its 12 keys enabling holes to be punched in any row; so, because no fields in a DEUCE instruction word were longer than 5 bits, programmers using this useful and reliable device to punch their instruction cards would literally have to know the binary representations of all the numbers up to 31 ‘backwards’ (sometimes referred to as ‘Chinese binary’)! On the oscilloscopes on the DEUCE console, 32-bit binary numbers and instructions would show as rows of bright and dim dots with the least significant binary digits on the left, corresponding to the way they appeared on a card.

6. THE DEUCE INSTRUCTION CODE (SEE ALSO APPENDIX)

In what follows examples of DEUCE instructions appear in bold type.

6.1. Functional addresses

DEUCE instruction words had a standard field layout, shown in detail in the Appendix. Every instruction incorporated a ‘Source’ (S) and a ‘Destination’ (D) field, each 5 bits long. Except when certain D values were used, an instruction effectively specified a transfer of information from one ‘Source’ delay line storage location (or sometimes a combination of two) to a ‘Destination’ location, replacing the previous contents of the latter. There was no ‘function code’ field in the instruction; instead, some of the most commonly required functions (e.g. add, subtract, multiply, divide, shift, logical AND, logical NEQ, tests for zero and negative numbers), as well as some numerical constants, were invoked by using special Source and Destination values outside the range 1–21, referred to as ‘functional addresses’. Thus, for instance, the instruction written ‘14-13’ (i.e. Source = 14, Destination = 13) simply meant ‘transfer the contents of TS14 to TS13’, but the instruction ‘23-25’ caused the contents of TS14 shifted down one binary place to be added to the contents of TS13, leaving the sum in the latter (and TS14 unchanged). Thus, TS13 could act as a 1-word accumulator, addition and subtraction being performed using Destinations 25 and 26, respectively. DS21 could also act as a 2-word accumulator; so, to subtract the contents of the pair of words in minor cycles 4 and 5 of DL9—the less significant word being in the even minor cycle 4—from the 64-bit quantity in DS21, one executed ‘9,5-23,d’.

The minor cycles and length—in this case ‘,d’, standing for ‘double’—of the operation were specified in other fields of the instruction. The instructions 13–25, 21–22,d and 24–14 could be used to shift up the contents of TS13, DS21 and TS14, respectively, by one place (in the first two cases, by adding the contents of TS13 or DS21 to itself; in the third case, the use of Source 24 caused the hardware in effect to delay the pulses by 1 μs as they emerged from TS14—Destinations other than 14 could also be used with Source 24). If these operations were made ‘long’, multiple shifts could be performed in a single instruction; though a shift of more than 16 places in DS21 would require more than one instruction. To take another example, the instruction ‘27-26,l’ (here ‘,l’ indicates ‘long’) caused the numerical constant ‘1’ to be subtracted from TS13 as many times as was specified by the duration of the instruction. The operands of bit-wise logical operations had to be in TS14 and TS15, the operation being specified by the Source value (25 or 26)—for example, the instruction ‘25-13’ caused the result of ‘ANDing’ the contents of TS14 and TS15 to be left in TS13, while ‘26-15’ would compare the (original) contents of TS14 and TS15 bit by bit, leaving in TS15 a word having 0’s in every position where the bits were the same and 1’s where they differed. All the ‘functional address’ values are listed in the Appendix.

Instructions with D = 24 were used to ‘trigger’ special functions, defined by the value of S; any delay lines used by these functions were implicit in their operation, and not specified by S or D. Some examples are given below. Instructions with D = 30 and 31 were used to instigate 32-word transfers between magnetic drum tracks (specified by S) and DL11.

6.2. Instruction sequencing

Unlike most modern computers, DEUCE did not execute instructions in storage sequence; instead, each instruction included fields specifying the location (DL and minor cycle) from which its successor was to be ‘extracted’. The delay line specification (‘NIS’, standing for Next Instruction Source),
in a 3-bit field, was absolute (NIS = 0 referring to DL8). However, the minor cycle in which an instruction started its own operation, and that in which its successor was extracted, were both specified relative to the minor cycle in which its own extraction took place, plus 2 (allowing for the two minor cycles in which extraction and decoding take place). In the former case, the relative difference in minor cycles was the number in the instruction’s 5-bit ‘W’ (for ‘Wait’) field, in the latter case the number in the 5-bit ‘T’ (for ‘Timing’) field. An obvious example of when a particular ‘W’ value was necessary was when an instruction required to access one or more consecutive words in a long delay line, so its operation had to start in the minor cycle corresponding to the first of these words: in this case, the ‘T’ field also specified the last word of the sequence to be accessed. For example, if an instruction located in, and extracted from, minor cycle 15 of a delay line was required to perform a ‘long’ transfer of consecutive words from minor cycles 21, 22 and 23 of delay line 9 to QS17 (necessarily, because of the ‘modulo 4’ rule, to 171, 172 and 173), the instruction’s Source and Destination fields would contain 9 and 17, respectively, and its ‘W’ and ‘T’ fields would contain 4 (=21 − (15 + 2)) and 6 (=23 − (15 + 2)), respectively. In this case, the ‘successor’ instruction also had to be located in minor cycle 23 of the NIS delay line; however, if only two consecutive words had to be transferred, the instruction could be specified as ‘double-length’, in which case ‘T’ would have no effect on the length of the transfer, so, the successor instruction—still specified by ‘T’—could be located in any convenient minor cycle. There were also instructions (Multiply and Divide) required to start operating in an odd minor cycle; so, even though the operands in these cases were implicit, the ‘W’ value was still significant.

A 2-bit field in each instruction word, referred to as the ‘Characteristic’ (C), specified whether a ‘transfer’ operation was to go on for 1 minor cycle (C=0: ‘single-length’), 2 minor cycle (C=2: ‘double-length’) or (C=1: ‘long’) a specific number (up to 32) of minor cycles—in the last case, as mentioned above, the minor cycle of the successor instruction was constrained to coincide with the final minor cycle of this ‘long’ operation and, therefore, the actual ‘length of operation’ of such an instruction was 1 greater than the difference (modulo 32) between the relative ‘start’ and ‘successor’ minor cycles specified in the instruction.

6.3. Instructions with Destinations 27 or 28

Instructions with Destinations 27 or 28 were used for testing whether the Source word (or words) were, respectively, negative (i.e. have P32 = 1) or non-zero. If they did not have the property tested for, the next instruction would be taken from the ‘successor’ DL and minor cycle specified. However, if any of the words tested were negative (D = 27) or non-zero (D = 28), the next instruction would be taken from the minor cycle in the ‘successor’ DL which immediately follows that specified. In other words, the alternative successor instructions for these ‘discrimination’ instructions had to be located in adjacent minor cycles of the NIS delay line, the first location being the ‘normal’ successor location.

6.4. Multiplication and division

DEUCE had autonomous multiplication and division facilities, both of which used DS21 and TS16 (TS16 no doubt originally chosen, rather than TS13, TS14 or TS15, so that instructions using the latter could be executed in parallel while autonomous multiplication or division were in progress).

Multiplication was triggered using the instruction ‘0-24’, which had to start in an odd minor cycle after first clearing 212 and placing the multiplier and multiplicand in TS16 and 213, respectively. After 65 minor cycles, in the course of which DS21 was automatically shifted up 32 binary places, the 64-bit product of the multiplier and multiplicand, both regarded as positive integers, would have replaced the contents of DS21. The actual mechanism was quite straightforward: as the multiplicand, in the more significant half of DS21, was shifted up, every time a ‘1’ bit ‘fell off its top end’ the multiplier (TS16), effectively extended with 32 zeroes at its more significant end, would be added to DS21—since the latter was being continuously shifted, the partial product would be correctly scaled. By the end of the operation, the multiplicand would have completely disappeared from 213 and the double-length product now occupied both halves of DS21. If either multiplier or multiplicand were actually negative, a correctly signed product was obtainable by subsequently subtracting the multiplicand or the multiplier, respectively (or if necessary both of them), from the more significant half of the product in 213. The value of this subtrahend could actually be computed in TS13 while the multiplication proceeded. Figure 1 shows draft DEUCE code for a simulation of the ‘0-24’ instruction—much slower than the real thing, of course, and necessarily lacking the facility of executing instructions in parallel.

Division was triggered using the instruction ‘1-24’, and operated on signed numbers. As with multiplication, the operation had to start operating in an odd minor cycle after first clearing the lower half of DS21 (213), and placing the dividend in 213 and the divisor in TS16; after 66 minor cycles the signed quotient—effectively, of the dividend shifted up 31 binary places, by the specified divisor—would be available in 212 and a signed remainder in 213.

6.5. Single shots

Normally, in every instruction word, P32 (the ‘Go’ digit) was a 1, ensuring the instruction would be executed as soon as it had been decoded. However, if this bit was 0 in a particular

---

Note: Due to the nature of the text, it is not possible to accurately reproduce the diagram mentioned (Figure 1). The text describes the mechanism of multiplication and division in DEUCE, detailing how instructions were handled and how the actual operations were executed within the machine's architecture.
instruction, that instruction would not execute until it had received an electronic ‘Single shot’. Typically, these were generated by the card reader whenever a new card row was ready for reading, or by the card punch when it was ready to punch a word sent to destination 29, thus, very effectively solving the problem of synchronizing a program with the input–output mechanisms. The effective delay would be an exact number of major cycles, ensuring that execution of the delayed instruction started in the correct minor cycle. Single shots could also be generated from the console, for program-testing purposes—i.e. with the ‘Go’ digit ignored, instructions could be executed one at a time at ‘human’ speed. This last-resort mechanism would normally require the program tester to halt the program at the start of a sequence of instructions to be examined, by ‘stopping up’ the card hole corresponding to the Go digit of that instruction, and then switching the machine to Single-shot mode.

6.6. Source 0 and Destination 0, and Automatic Instruction Modification

Transfers to Destination 0 caused the Source word to be extracted, decoded and executed, e.g. an instruction ‘13-0’ (with identical values in its W and T fields) would normally cause the ‘object’ instruction in TS13 to be executed next, rather than the instruction in the location specified by its NIS (P2-P4) field—the T field would still indicate the ‘extraction’ minor cycle, relative to which the object instruction’s own W and T values had to be specified. This facility was virtually essential for enabling the modification of instructions which operated on ‘arrays’—indeed, such instructions, when ‘sourced’ from QS17 or QS18, could be automatically pre-modified in situ, the instruction field(s) to be incremented or decremented being determined by P1-P4 in the appropriate 17-0 or 18-0 instruction; this powerful feature, introduced in 1957, was called ‘Automatic Instruction Modification’ (AIM), and it very effectively compensated for DEUCE’s lack of conventional index registers.

The ‘Destination 0’ facility was also useful for exit from subroutines when the ‘link’ instruction had been planted in, say, TS13 prior to entry. By convention, location 128 would always contain a ‘13-0’ instruction, with zero in the W and T fields, causing an instruction in TS13 to be extracted, decoded and to start its operation in minor cycles 30, 31 and 0, respectively—first-order subroutine links were always executed as if extracted from m.c.30, second-order subroutine links from m.c.31, and so on.

Consider also the instruction ‘0-0X’ which was executed if a zero (all 0s) was extracted for decoding (the ‘X’ signifying the absence of the Go digit, i.e. the instruction waited for a Single shot). ‘Source 0’, if the card reader was switched on, designated the next row from the card currently being read. This enabled instructions read directly from cards to be executed, and provided the means for ‘bootstrapping’ when

---

**Note: TS14 and DS20 involved for purposes of simulation only**

- **? - 16** Multiplier (X) transferred to TS16
- **? - 213** Multiplicand (Y) transferred to more significant half of DS21
- **30 - 212** Initialise less significant half of DS21 to zero
- **--------** (above instructions normal before call of 0 - 24)
- **4 - 24** Switch TCB off (i.e. DS21 acts as true double-length register)
- **30 - 203** Ensures contents of DS20 (X or 0 “extended” to 64-bits) “unsigned”
- **31 - 14** “All 1s” (loop below continues until every 1 has been shifted out)

↑

- **213 - 27** Test whether P32 bit (current top bit of multiplicand) is 1 or 0

  - **(0) ✓ ✓** (1)
  - **30 - 202 16 - 202** Addend is X if P32 of 213 is 1, otherwise 0

  - **21 - 22,d** Add DS21 to itself, i.e. shift up partial product and residue of Y
  - **20 - 22,d** (Shifted) partial product updated by adding contents of DS20
  - **24 - 14** Shift up TS14 (originally contained 32 ‘1’ bits)
  - **14 - 28** Are there any 1s left in TS14?

  - **nz ✓ ✓** (No 1s left => all 32 bits of Y have been tested)

  - **z** (Multiplication over - unsigned product now in DS21)

---

**FIGURE 1.** Code to simulate DEUCE unsigned multiplication (0-24 instruction). This is what DEUCE code might look like at the ‘drafting’ stage, i.e. before any store locations have been assigned to hold it. For an example of the ‘finished product’ for a different program, see Fig. 1.
6.7. Transfer control ‘A’

The designers of DEUCE provided a facility called Transfer Control ‘A’ (TCA). When TCA was ‘on’ the contents of DL10 could be extracted via TS16 after a delay of 1 minor cycle—e.g. in minor cycle 4, the contents of 100 was available from Source 16, and TS16 itself would be ‘refreshed’ with the contents of 101. An instruction of the form 16-10,l operating for a whole major cycle would cause the complete contents of DL10 to be ‘shunted’ round, so that 100 would now contain the word previously in 101, and so on—the previous contents of 100 would be in 1031. If the instruction had been 16-9,l DL10 would be unaltered, but DL9 would now contain a similarly ‘shunted’ copy of DL10. TCA was switched on by the instruction 3-24, and turned off by the first subsequent instruction with D = 16.

One suggested use for this instruction was in a program loop dedicated to performing the same operation on consecutive locations in a long delay line (e.g. dispatching them to the card punch one word at a time). Rather than somehow modifying the ‘W’ field in the instruction which performed the actual operation, if the operands were held in DL10 (in reverse order) and, with TCA ‘on’, every time the loop was executed an instruction of the form 16-10,l of a suitable duration was used to ‘shunt’ the operands as described above, then the ‘accessing’ instruction itself should not have to be altered (an example, perhaps, of the mountain being brought to Mohammed).

However, with the advent of AIM (see above), the problems involved in modifying ‘W’ fields virtually disappeared. A perhaps more worthwhile application of TCA (in conjunction with AIM) lay in using it to ‘stagger’ the relative positions of sequences of corresponding words in two different delay lines, thus, making it possible, in a program loop, to operate in quick succession on both members of a corresponding pair without incurring major time penalties.

Just as with 0-24 and 1-24, TS16 was no doubt chosen for use with this particular facility (in preference to TS13, TS14 or TS15) to permit single-length arithmetic and/or logical operations to be performed while TCA was ‘on’.

7. OPTIMUM CODING

Figure 1 illustrates a fragment of DEUCE coding (a simulation of the action of the 0-24 multiplication instruction) as it might have appeared at the ‘drafting stage, i.e. before instructions had been assigned to delay line locations.

As described in the preceding section, when an instruction (call it X) was ‘extracted’ in minor cycle m (either from one of the DLs 1-8, or from some other location specified by a ‘Destination 0’ instruction) for decoding and execution, its successor would be in minor cycle (m + 2 + T) of the delay line specified by X’s ‘NIS’ field, T being the contents of X’s ‘T’ field. The operation performed by X would start in minor cycle (m + 2 + W), where W was the contents of X’s ‘W’ field; its duration (1, 2, or more, minor cycles) was determined by the contents of X’s ‘C’ field—if ‘long’, it was (T − W + 1) or (T − W + 33) minor cycles. Whatever the duration, the instruction’s effective execution time was therefore (T + 2) minor cycles, or (T + 34) if the operation had not been completed by the end of the minor cycle containing the successor instruction. To reduce execution time, particularly in time-critical situations (e.g. in code executed between reading rows from a punched card, or within loops of instructions likely to be repeated many times), there was an incentive for programmers to try and reduce the values of both
T's and W's to a minimum. Success depended on the careful allocation of instructions (and data) to locations within the available long delay lines. This process, known as 'optimum coding', is now a lost, and largely unmentioned, art—somewhat akin to dishwasher-loading or, more realistically, school time-tableting; but it was practised assiduously by all conscientious DEUCE programmers. Fraser Duncan, referring to the tendency of programmers to 'optimize' even in situations where it was not justified, reportedly described it as 'spending mega seconds to save microseconds'—but, personally I believe this (widespread) tendency existed because somehow, magically, DEUCE encouraged conscientious programmers always to 'do their best'! Another manifestation of this tendency, which certainly applied to me, was that if a section of program was found to occupy slightly more than a multiple of 32 words, no effort would be spared in trying to reduce this number so that the code did not spill over into an extra delay line.

Figure 2 reproduces part of the first DEUCE program I ever wrote, in 1957—it is the complete coding of the 512-bit arithmetic (add, subtract and multiply) subroutines mentioned in my reference to this program in Section 3; in contrast to the 'draft' shown in Fig. 1, this code is ready for punching onto cards, i.e. all the instructions shown have been allocated to exactly 32 (!) minor cycles in a delay line, and all instruction fields filled in. Readers are not expected to follow the code, but those who do attempt it will observe that the 512-bit numbers are each organized as 16 'segments' of 32 bits, occupying the 16 even-numbered minor cycles in a delay line; that, for the sake of faster access, in the addition and subtraction routines TCA is used to 'stagger' the segments of one of the operands by four minor cycles relative to the corresponding segments of the other; and that the final loop executes (16 times) a short sequence of instructions (previously loaded into DLs 3-5 by the program itself), each of which is identical in every respect to the corresponding instruction executed the previous time round, but happens to be extracted from a location displaced by two minor cycles from that used the time before, and is thus enabled to operate on the 'next' segment—exploiting the fact that the operands of a DEUCE instruction were always addressed relative to the minor cycle in which the instruction was extracted.

8. THE 64-COLUMN CARD CHALLENGE

Initially, DEUCE could only read 32 bits from each row of each card; these were read from columns 21–52, and output was punched in the same card columns. However, Hollerith cards had 80 columns, so in 1957, with the aim of enhancing throughput, DEUCE card readers and punches were being modified so that two 32-bit words could be read from, or written to, each row, columns 17–48 containing the first word and columns 49–80 the second. When reading from a 64-column card, where previously one 'Source 0' instruction per row (needing a Single shot to synchronize with the card reader) was required, one now had to execute a second 'Source 0' instruction (not needing a Single shot) within a defined time interval after the first, if one wished to read a second 32-bit word from the same row. A similar arrangement was required for the card punch.

Any DEUCE program for reading data from cards was subject to timing constraints, the chief one being that the programmer had to ensure that not more than 15 major cycles (ms) elapsed between successive executions of the 'Source 0' instructions used to read consecutive rows. Between reading the 12th row of one card and the 1st ('Y') row of its successor, at most 80 ms were available. An 8-digit decimal number punched on a card would occupy eight adjacent card columns, with the most significant digit in the leftmost column, a digit n represented by a single hole punched in row n, the rows from the third down to the 12th being numbered 0–9. Negative numbers would be indicated by a hole punched in the second ('X') row of either the first or last (I forget which) of these columns. Software would be required to convert this decimal 'pattern' to a 32-bit positive or negative binary number. Library subroutines for reading decimal numbers from cards and converting them to binary were also obliged, as a matter of honour, to ensure that all conversions to binary were completed as soon after reading the 12th row of each card as possible—thus enabling any user of the subroutine (assumed to be reading a quantity of numbers, i.e. several cards all punched using the same number format) to have as much as possible of the 80 ms inter-card interval available for 'housekeeping' and not to have to stop and restart the card reader after every card. Such stopping and restarting was likely to slow down the rate of card input from 200 cards per minute to about one card per second, and excessive use was harmful to the reader mechanism.

When 64-column reading was introduced, the timings, and constraints, were not altered, so throughput was potentially doubled. This applied both when the cards carried binary instructions, so two drum tracks could be filled from one triplet of cards—this presented no problems, because no data conversion was involved—and also for decimal data. In the latter case, whereas existing subroutines were written to read-from-one-card-and-convert-to-binary (e.g.) three 9-digit, or four 8-digit decimal numbers, new subroutines would be expected to read twice as many numbers, of the same kind, from one card. The ‘challenge’ which now arose was to provide efficient versions of the required new subroutines. Although the existing subroutines were able to perform inter-row decimal-to-binary conversions (which only involved adding powers of 10, as described below) for three or four 8- or 9-digit numbers within the 15 ms inter-row intervals, application of the methods then in use to twice the amount of data would almost certainly violate this constraint; moreover, performing the necessary conversions ‘between cards’ would probably not be satisfactory either, because such conversions would involve a time-consuming polynomial summation for each number.
The accepted way for a DEUCE subroutine to read an 8-digit decimal number from a card and convert it to binary are summarized in the following rules (see [8], p. 253, for an example of the Pilot ACE code used for this—fortunately ACE’s TS, and functional, addresses do not match the DEUCE equivalents quoted below):

(i) At some time before reading row 0 (the third row) of the card, assign a 32-bit word (the ‘Value’) for accumulating the binary value of the number, and initialize it to 0.

(ii) When read into a DEUCE word, holes punched in any of rows 0–9 (within the eight columns containing the decimal number) appear as 1s in an 8-bit field (the ‘Columns’ field) within that word—the ‘0’ bits show where holes are absent. Accordingly, in another DEUCE word (usually TS15) assign an 8-bit field (the ‘Contributions’ field) whose individual bits correspond to those in the ‘Columns’ field and, some time before reading row 0, also initialize this ‘Contributions’ field to eight binary 1s. Each bit position in the ‘Columns’ field matches the position of a digit in the decimal number, so the power of 10 matching the decimal significance of that digit is understood to be associated with the corresponding bit position in the ‘Contributions’ field—if the latter occupies, for example, bits P25 to P32, 107 is associated with P25, 106 with P26, and so on, down to 100 (i.e. 1) associated with P32.

(iii) After reading each of the rows 0–9, align the ‘Columns’ field (moved, or read directly, into TS14) with the ‘Contributions’ field in TS15, and use a 26-15 instruction to change from 1s to 0s those bits in the latter which match holes punched in the row, appearing as 1s in the ‘Columns’ field in TS14 (a check, using a 25-28 instruction, may then also be performed to show up the illegal presence of more than one hole in a column, by detecting when a 0 in TS15 has been changed back to a 1). Finally, (except after reading row nine) scan the updated ‘Contributions’ field, and for any of the 8 bits which are still 1s, add the associated power of 10 to the ‘Value’ word. Thus, (for example) a column punched in row 6 would cause the appropriate power of 10 to be added after reading rows 0, 1, 2, 3, 4 and 5—i.e. just the required six times.

After reading row 9, there should be eight binary zeroes in the ‘Contributions’ field, unless a card had no hole punched. All being well, the ‘Value’ word now contains the binary equivalent of the decimal number on the card.

The time-consuming element of the above process lay in scanning the ‘Contributions’ field and incrementing the ‘Value’. This could be expected to take something like 3 ms for an 8-digit number; thus, it was not too difficult to write subroutines capable of reading four 8-digit numbers from each card without violating the 15 ms constraint, but reading and converting eight numbers would really present problems.

9. RESOLVING THE 64-COLUMN CARD CHALLENGE

The row-by-row scanning mechanism described in the last section, whereby the successive bits in the ‘Contributions’ field are all examined, and cause the ‘Value’ to be incremented according to whether the bit is a 1 or a 0, is similar to what happens when DEUCE multiplication (simulated in Fig. 1) is started by the instruction 0-24, whereupon the multiplicand (213) is automatically shifted up and has its top bit examined every two minor cycles; if this bit is non-zero, the contents of 212 (also shifted up) are incremented by the contents of TS16. How convenient it would be if one could store one or more ‘Contributions’ fields in 213, initiate multiplication, and continuously vary the contents of TS16 while this was going on so that the multiplier value became at the right moment the appropriate power of 10, to be added into 212!

At some time in 1958, it became apparent to us that something like this was actually feasible, without any change to existing hardware. It was realized that, in theory at any rate, switching TCA on while, or before, multiplication was in progress would cause TS16 (the multiplier) continuously to ‘assume’ the values actually stored in successive locations in DL10. It remained to see whether practical results could be obtained and, if successful, to determine the exact ‘rules of the game’.

It did not take long to establish that the technique was feasible. The first results were two new library subroutines, one of which, written by myself, read and converted eight signed 8-digit decimal numbers from one card, and the other, written by Jim Lucking, six signed 9-digit numbers. In the first case, the DEUCE multiplier was invoked four times between reading successive rows of a card, in the latter case three times—i.e. two numbers could be processed during each multiplication operation, one in each of the two major cycles that the operation lasted, the ‘partial product’ being extracted twice. Before the start of the second major cycle, after the first ‘product extraction’, the contents of DS21 had to be completely ‘reset’. In both major cycles exactly the same ‘pseudo-multipliers’, stored in 8 or 9 consecutive odd minor cycles of DL10, were used (it also turned out that even-numbered minor cycles in DL10 adjacent to these had to contain zeroes).

Jim Lucking and John Boothroyd had collaborated earlier to solve a problem arising from the fact that, in the patterns read from cards, the bits corresponding to the most significant decimal digits were in the least significant (i.e. using the new method, the last scanned) binary positions of the ‘Columns’ fields. This meant that the largest decimal powers were added into the ‘partial product’ last, by which time the latter would already have been shifted up 7 or 8 binary places—but it was essential, because of timing constraints, that the extracted value should not require any further scaling. Exploiting the fact that ‘partial products’ occupy 64, rather than 32, bits, the solution was:
(a) to store each ‘pseudo-multiplier’—a power of 10, to be added to $2^{12}$—in the appropriate location in $D_{10}$ already scaled up in such a way that, by the time the ‘partial product’ was extracted from $D_{21}$, the contribution of the leftmost (least significant) bit of this power of 10 would have become aligned with the least significant bit of $2^{15}$;

(b) this scaling up had to be ‘cyclic’, i.e. if it caused part of the binary representation of the power of 10 to overflow the top end of the word in $D_{10}$ to which it was assigned, the ‘overflowed’ bits were inserted at the bottom end of the same word, so that when the ‘partial product’ was extracted their contribution would be in $2^{12}$, correctly scaled relative to the ‘unoverflowed’ contribution in $2^{13}$, the latter having its least significant bit in the P1 position;

(c) the ‘partial product’ would be extracted by a ‘21-25, d’ instruction, which added both halves of $D_{21}$ together into $S_{13}$ (the ‘Value’ accumulator, pre-loaded with the partial sum for the decimal number obtained from previous rows of the card), thus ‘reuniting’ each pair of contributions. Because DEUCE multipliers are treated as unsigned, this technique works even when the scaled ‘pseudo-multiplier’ has ‘1’ in its P32 position.

Other DEUCE installations, becoming aware of the possibilities, began to make contributions. Whereas at NRL we tended to rely on John Boothroyd to tell us the exact details of how DEUCE multiplication and division worked, other installations apparently worked from DEUCE’s (reportedly excellent) logical circuit diagrams. Mike Kelly, at English Electric’s Whetstone site, learnt how to exploit the Divide instruction; John O’Brien, a programmer from the Luton site, included the conversion of (old) pence to pounds, shillings and pence, etc. In this report John, who had been one of the English Electric engineers who helped to build the Pilot ACE at NPL, included the statement: ‘It is believed that the use of TCA during multiplication was first suggested by B. Munday (then at NPL), so far as is known, the detailed coding was first done at Nelson Research Laboratories’. I never met Brian Munday—one of the NPL programmers of the ACE GIP [4, p. 164]—and can affirm that the idea occurred to me quite independently. In Section 10, I speculate briefly on the possibility that Turing might even have had the evidence suggests that this is unlikely.

The moral of all this, if there is one, is that when programming a computer at machine-code level, no properly-defined feature of any instruction should be neglected if improvements in effectiveness can be achieved by exploiting it, even if this usage appears ‘unorthodox’. In software organizations, there has sometimes been a regrettable tendency to regard ‘Unorthodox’ as synonymous with ‘Unmanageable’.

1 DID TURING ANTICIPATE THE SOLUTION?

I was taught Mathematics at school, until 1950, by Patrick Mermagen (afterwards Headmaster of Ipswich School). Readers familiar with the biography of A.M. Turing by Andrew

---

Hodges [2] may recall that Mermagen had been a respected school contemporary of Turing [2, p. 52]. He wrote to Turing in 1947 [2, pp. 362, 559] to seek his opinion on the place of mathematics and science in the post-war world; in his reply Turing mentioned that he himself would be looking for a number of mathematicians to write instructions for his computer—he had already mentioned this requirement in the ACE Report [9, p. 392], though he appears never to have actually described them as ‘computer programmers’. At the same time, Mermagen invited Turing to come and talk to his mathematical pupils about computers, but Turing would not commit himself at that time; I am rather ashamed to admit to having been unfamiliar with his name until much later.

Did Turing, who bequeathed so many features to DEUCE, via ACE [8, p. 228], actually originate the idea of enabling the multiplier to be varied during multiplication? Although it would have been useful at any stage (even before the advent of cards holding 64 columns of decimal data), this seems unlikely. The two essential requirements enabling this procedure to work are autonomous unsigned multiplication, and a TCA-type feature enabling the contents of a long delay-line to be ‘fed’ through the same TS which is normally used to hold the unshifted multiplication operand. Once Turing (being obsessed with ‘speed of operation’, as J.H. Wilkinson confirmed [10, p. 94]) had abandoned the idea of executing instructions in storage sequence, and introduced the ‘NIS’ field, it was perhaps inevitable that he should have proposed a TCA-like feature [11, p. 504]—though this seems not to have involved a TS at all, but would have operated directly on the contents of any specified long delay-line. Turing also specified an autonomous multiplication feature [10, pp. 493–494], and described how a ‘sign correcting subtrahend’ could be computed while multiplication was still in progress [11, pp. 512–513]. However, the fact that autonomous multiplication was not fitted to the Pilot ACE until 1951 [8, p. 227] suggests that, at NPL, it was not considered essential for converting from decimal to binary.

I have quoted in Section 9 John Boothroyd’s statement in DEUCE News 36 that Brian Munday at NPL had originated the technique. Indirect evidence of early awareness of its potential is provided by two NPL documents—Mike Woodger’s specification of the abortive 1947 ‘Test Assembly’ [8, p. 226; 12], and J.H. Wilkinson’s 1948 progress report on the Pilot ACE [2, p. 560; 13]. Both documents include descriptions of a TCA equivalent and an autonomous multiplication facility making use of the same TS—the ‘two essential requirements’ mentioned above. However, whereas Woodger’s document [12, p. 5] states explicitly that the TCA equivalent (then called ‘TC32’ in both [12] and [13]) must be ‘off’ while multiplication is in progress, Wilkinson’s [13, p. 36] does not reiterate this stricture. Since both documents aim to describe in detail the logical design of what was to become the Pilot ACE (as then envisaged), a difference of this kind may be regarded as significant. We can be thankful that Wilkinson saw no reason to rule out the simultaneous operation of TCA and multiplication in the Pilot ACE, and that this state of affairs was preserved in DEUCE.

ACKNOWLEDGEMENTS

My thanks are firstly due to the Computer Conservation Society for inviting me to give a talk on DEUCE to their Manchester branch in October 2007, which revived many old memories. Much of the material from that (unpublished) talk has found its way into this paper. I am grateful to the referees for many useful suggestions, in particular for drawing my attention to Copeland’s Symposium (Copeland, B.J. (ed.) (2005) Alan Turing’s Automatic Computing Engine. OUP, Oxford). I would also like to express my gratitude to the unfailingly helpful staff of the Radcliffe Science Library in Oxford.

REFERENCES

### APPENDIX: DEUCE INSTRUCTION WORD LAYOUT: SOURCES AND DESTINATIONS

Current instruction assumed to have been extracted in minor cycle M.

<table>
<thead>
<tr>
<th>P</th>
<th>(P1 least significant bit, P32 most significant bit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Only used, with P2-4, by 17-0 or 18-0, for ‘Automatic Instruction Modification’ (adds to object instruction in QS17 or 18, before execution, combinations of ± P5, P10, P17, P18, P22)</td>
</tr>
<tr>
<td>2-4</td>
<td>NIS (Next Instruction Source)</td>
</tr>
<tr>
<td>5-9</td>
<td>S (Transfer Source)</td>
</tr>
<tr>
<td>10-14</td>
<td>D (Transfer Destination)</td>
</tr>
<tr>
<td>15-16</td>
<td>C (Characteristic)</td>
</tr>
<tr>
<td>17-21</td>
<td>W (Wait) (relative to M+2)</td>
</tr>
<tr>
<td>22-25</td>
<td>J (‘Joe’)</td>
</tr>
<tr>
<td>26-30</td>
<td>T (Timing) (relative to M+2)</td>
</tr>
</tbody>
</table>

#### S (Transfer Source)

- **P** (P1 least significant bit, P32 most significant bit)
- **S** (Transfer Source) (D≠24,30,31) Delay line (1-21) or ‘Functional’.
- **D** (Transfer Destination) Delay line (1-21) or ‘Functional’.
- **C** (Characteristic) Transfer length (m.c.): C = 0 (1); C = 2 (2);
  - C = 1 => ‘long’ (T-W+1) or T-W+33 if W>T.
- **W** (Wait) (relative to M+2) Transfer/operation starts W+2 m.c. after m.c. M.
- **J** (‘Joe’) (May be used to effect ‘carry’ into T field.)
- **T** (Timing) (relative to M+2) Next instruction to be extracted from DL ‘NIS’ T+2 m.c. after m.c. M (T+34 if this would precede the last m.c. of the operation).
- **‘Go’** If 0, instruction waits for ‘Single-shot’

#### Functional S’s (incl. constants)

<table>
<thead>
<tr>
<th>S/D</th>
<th>Functional D’s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Next word (row) from card reader</td>
<td>0 Execute (S) next</td>
</tr>
<tr>
<td>DS21 (signed) shifted down 1 bit</td>
<td>22 (S) added to DS21</td>
</tr>
<tr>
<td>TS14 (signed) shifted down 1 bit</td>
<td>23 (S) subtracted from DS21</td>
</tr>
<tr>
<td>TS14 shifted up 1 bit</td>
<td>24 ‘Triggers’ (see below)</td>
</tr>
<tr>
<td>TS14 AND TS15</td>
<td>25 (S) added to TS13</td>
</tr>
<tr>
<td>TS14 NEQ TS15 (OR - AND)</td>
<td>26 (S) subtracted from TS13</td>
</tr>
<tr>
<td>*P1</td>
<td>27 If (S) &lt; 0 take next instruction from</td>
</tr>
<tr>
<td>*P17</td>
<td>28 If (S) NZ take ‘NIS’, m.c. M+T+3</td>
</tr>
<tr>
<td>*P32</td>
<td>29 Output (S) to card punch (or console)</td>
</tr>
<tr>
<td>Zero (‘all 0s’)</td>
<td>30 **Read from/Write to Drum using head (S)</td>
</tr>
<tr>
<td>–1 (‘all 1s’)</td>
<td>31 **Shift Drum Read/Write heads to position (S)</td>
</tr>
</tbody>
</table>

* ‘P1’ indicates the Source value has P1 = 1, remaining bits zero; and so on

**Read (into DL11) if C even, Write (contents of DL11) if C odd

<table>
<thead>
<tr>
<th>S</th>
<th>Actions if D = 24 (‘Destination Triggers’), dependent on 5-bit value in S field</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Start multiplier (65 m.c.; must start in odd m.c.) TS16 x 21 to DS21 (unsigned)</td>
</tr>
<tr>
<td>1</td>
<td>Start divider (66 m.c.; must start in odd m.c.) 21 ÷ TS16 to 21 (signed)</td>
</tr>
<tr>
<td>2</td>
<td>Test if 12th line of card reached (if so, next instruction taken from m.c. M+T+3)</td>
</tr>
<tr>
<td>3</td>
<td>TCA (causes TS16 to contain copy of 10m−1 in m.c. m; switched off by later D=16)</td>
</tr>
<tr>
<td>4</td>
<td>Causes DS21 to act as a 64-bit register (more significant half in 21)</td>
</tr>
<tr>
<td>5</td>
<td>Causes 21 and 21 to act as independent 32-bit registers</td>
</tr>
<tr>
<td>6/7</td>
<td>Switch audible alarm off/on</td>
</tr>
<tr>
<td>8</td>
<td>Clear ‘output staticisor’ (and switch 32-column fields on 64-column punch)</td>
</tr>
<tr>
<td>9</td>
<td>Stop card reader and/or card punch</td>
</tr>
<tr>
<td>10</td>
<td>Start card punch</td>
</tr>
<tr>
<td>12</td>
<td>Start card reader (11 unused, to avoid ‘Source 11’ interlocks during Drum transfers)</td>
</tr>
</tbody>
</table>