The architecture of the ZEBRA computer

Version 1.0 **Kees Pronk**

PART I Introduction to the ZEBRA computer

History of the ZEBRA development

This article will give an overview of the hardware architecture and the software of the ZEBRA computer. This computer was designed in the fifties of the previous century by Willem L. van der Poel (1926 .. now) (Figure 1, 2).

Figure 1 Willem L. van der Poel (around 1980)

The story begins during the German occupation. As a student van der Poel had already worked on the construction principles of computers. After the war he studied at the Delft Polytechnic where he met prof. A. C. S. van Heel of the physics department. The first computer designed by Willem van der Poel was a relay computer called the ARCO (**A**utomatische **R**elais **C**alculator voor **O**ptische berekeningen). This machine did not calculate very fast and therefore it was nick-named TESTUDO (Latin for turtle). Most subsystems of this computer have been preserved and one part of it is available in our Study Collection in the basement of the EEMCS building in Delft. After his graduation, van der Poel started working for the Dutch PTT on the development of computers. The PTT was interested in performing capacity calculations for telephone systems. Van der Poel and his boss Leen Kosten started at the Mathematical Department of the Central Laboratory of the Dutch PTT (later the dr. Neher laboratory of the PTT) in The Hague with developing two experimental computers based on vacuum tubes and relays, the ZERO (**Z**eer **E**envoudig **Re**ken**O**rgaan; Very Simple Calculating Device) and the PTERA (**PT**T **E**lektronische **R**eken**A**utomaat). These machines have not been preserved; their parts were reused for the next development starting around 1955: the ZEBRA computer (**Z**eer **E**envoudige **B**inaire **R**eken**A**utomaat, Very Simple Binary Computer).

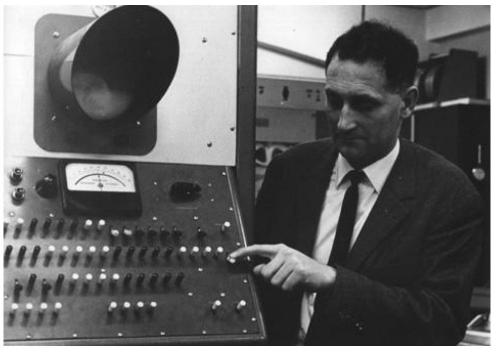


Figure 2 Willem L. van der Poel operating the controls of the ZEBRA (around 1960)

The ZEBRA was designed to be a simple computer containing a minimal number of vacuum tubes and a straightforward architecture. The machine contains some 600 tubes and around 500 transistors. It should be noted that vacuum tubes are sensitive to aging. Having less vacuum tubes simply means obtaining higher reliability. After the logical design had been readied van der Poel tried to sell the computer to Dutch industries to have them produced in series. This wasn't a success and he found the English firm Stantec (Standard Telephones and Cables Lt.) willing to set-up a production facility. Stantec produced some 55 machines and sold them throughout the world. Several Dutch universities and institutions have bought a ZEBRA. Stantec sold these machine for about *f* 160.000,- (one hundred sixty thousand Dutch guilders). This amount was equivalent to eight yearly salaries of a full professor in Delft in those years. Figure 3 shows the ZEBRA as it was installed around 1958 at the Julianalaan building in Delft. The set-up shows two computer cabinets, a third cabinet containing the power supply in the back and the table of the operator showing a printer, a small console and the paper tape punch. A complete ZEBRA using vacuum tubes is on display at the Study Collection in the basement of the EEMCS building. Later on Stantec produced a fully transistorized version of the ZEBRA which is also on show at the Study Collection.

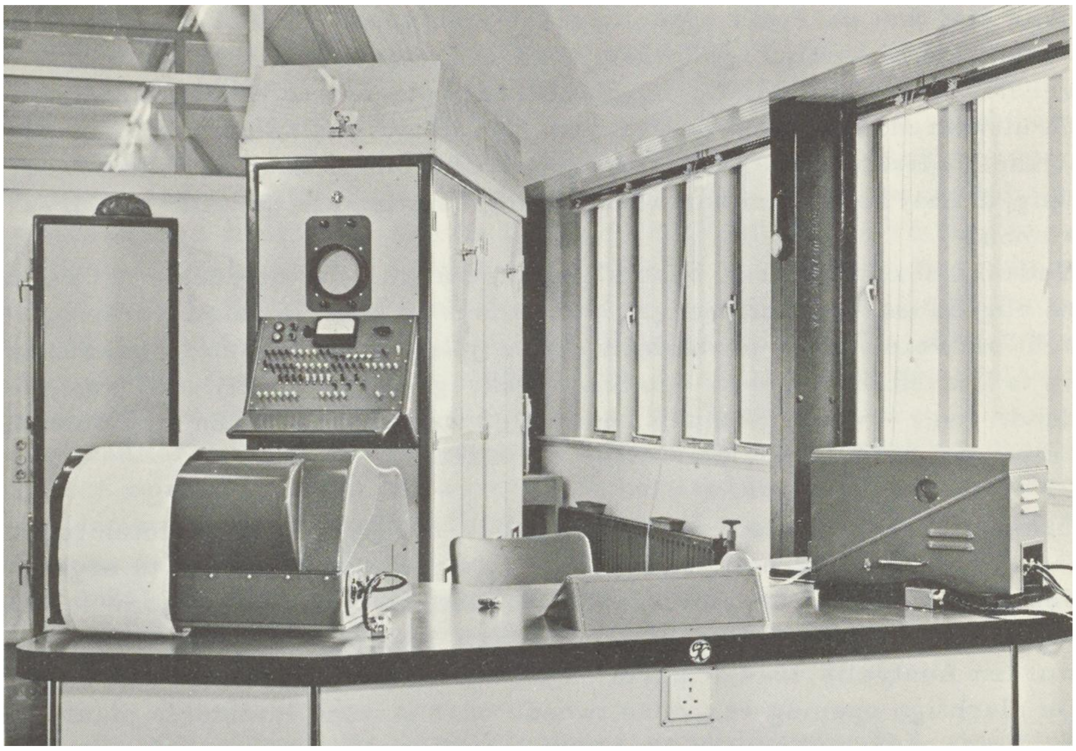


Figure 3 The ZEBRA set-up at the Mathematics Department in the Julianalaan building in Delft

PART II Hardware of the ZEBRA

Before embarking on a description of the ZEBRA architecture three provisos must be made.
[1] Although the ZEBRA has been advertised as a 'simple' machine, it turns out to be a very ingenious machine of which not all details can be discussed in this article. In fact there are three machines: the hardware, the Normal Code (software) machine and the Simple Code (software) machine. First we take a look at the hardware; more about Normal Code and Simple Code will follow. Much additional material to study the ZEBRA is available in the Study Collection (https://studieverzameling.ewi.tudelft.nl).
[2] During the development of the ZEBRA and in later years computer terminology has changed. In this article a more modern terminology will be used than in the original documents.
[3] It is impossible to discuss all the details of the ZEBRA. In a few sections an issue must be mentioned but the reader is not supposed to understand all the details. In that case the issue will be called 'magic'.

It should be noted that in the fifties the ZEBRA was the very first stored program computer scientist could have at their disposal. In the earlier years all calculations were done using (electro-) mechanical calculators. When using a calculator always intermediate results would have to be written down on paper. These values had to be re-entered in the calculator later on by hand. This used to be time consuming operations also introducing errors. Being able to use a stored program computer made it possible to speed-up computations considerably.

The hardware of the ZEBRA

Figure 4 gives a view on the hardware. The computer is housed in three cabinets; one of the cabinets being solely used for the power supply. The figure shows the two cabinets with the electronics. In the left side the magnetic drum may be seen.

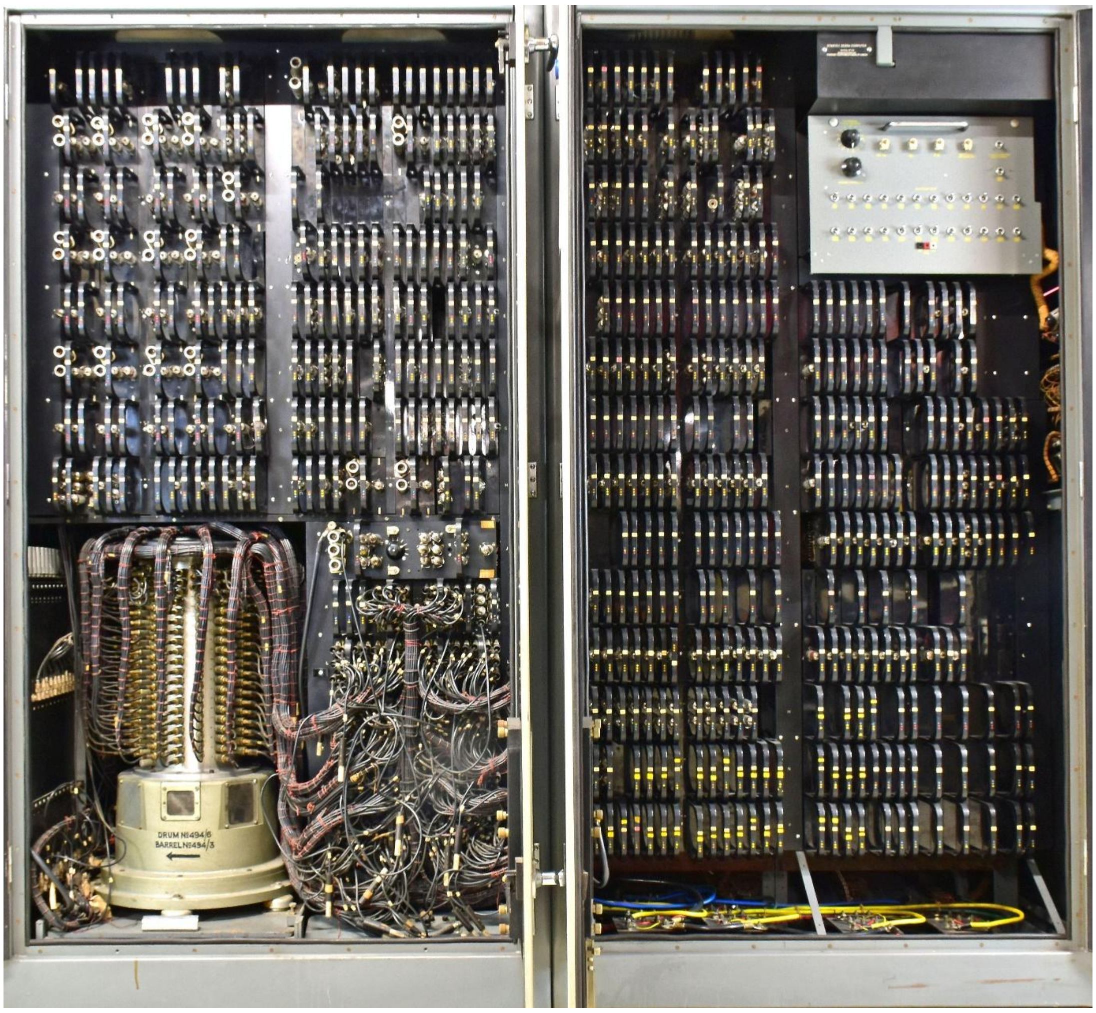
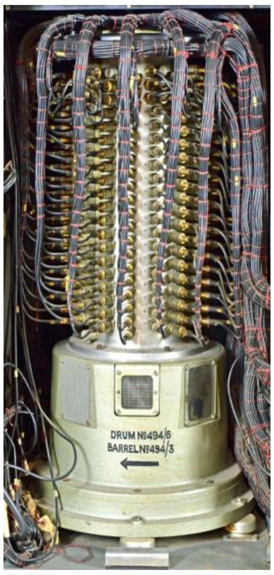


Figure 4 Two cabinets with electronics in the ZEBRA



The main memory of the ZEBRA is formed by a so-called magnetic drum (see Figure 5). The surface of the drum consists of magnetizable material; the drum rotates with 6000 rpm. Using read/write heads one may store or retrieve information on the drum. The drum contains 256 user tracks; each track has 32 positions to read or write a word, so the total storage is restricted to 8192 words , numbered 0 .. 8191. The minimum access time of information on the drum is 312 microseconds. The drum functions as a linear random access memory. Using a drum for main memory was very common in that period of time. Obviously, using a drum means one has to wait until the drum has rotated and the required word is under the read/write head which may take up to 10 milliseconds. Several functions have been built in the machine to remedy that shortcoming. Electrical pulses from a special part of the disk regulate all the timing inside the ZEBRA. On the surface of the enclosure of the drum (Figure 5) one may see the read/write heads.

Figure 5 The housing of the magnetic drum with the read/write heads

The construction of the machine

Plug-in electronic units are provided incorporating wire wrapped connections on the back side (Figure 6).

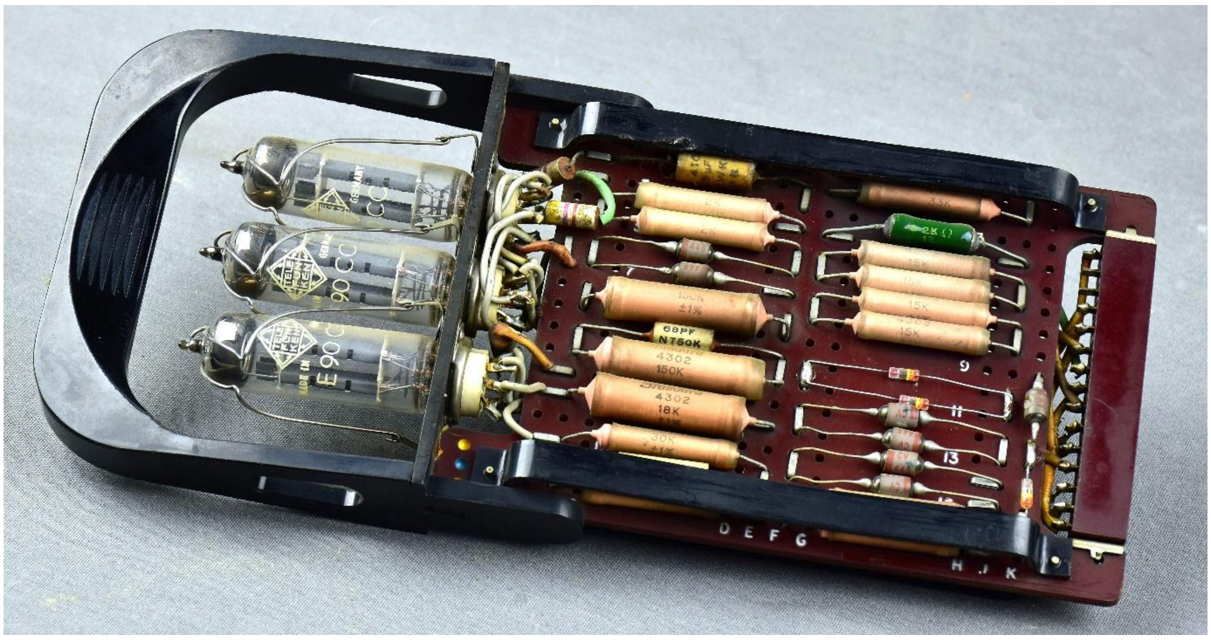


Figure 6 An example ZEBRA plug in unit with E90CC professional double triodes

Because of the tubes approximately 3 KVA is required by the machine. Test programs are provided to be used with marginal testing where variations of high voltages can be used to detect incipient component failures.

The I/O devices

The I/O devices connected to the ZEBRA are:

A paper tape reader able to read 200 characters per second. Five hole paper tape using a proprietary code was user for program text input.
A paper tape punch able to punch up to 50 characters per second.
A Creed teleprinter operating at 7 characters per second.
A small console for the operator contains a loudspeaker and telephone dialler intended to give numeric commands to the computer (Figure 7). The loudspeaker was connected to the innards of the computer and was used to monitor the progress, or the lack thereof, of the execution of the program.
The front panel of the ZEBRA contains an oscilloscope on which the value of some of the memory locations may be displayed, a number of switches to control the machine and an efficiency meter (Figure 8). This meter indicated the time the processor spent to the execution of the program; or, was idle, waiting for the drum.

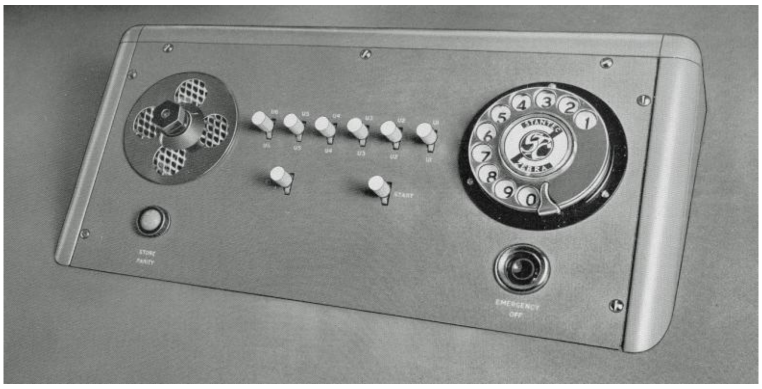


Figure 7 A small console with loudspeaker, switches and dialler

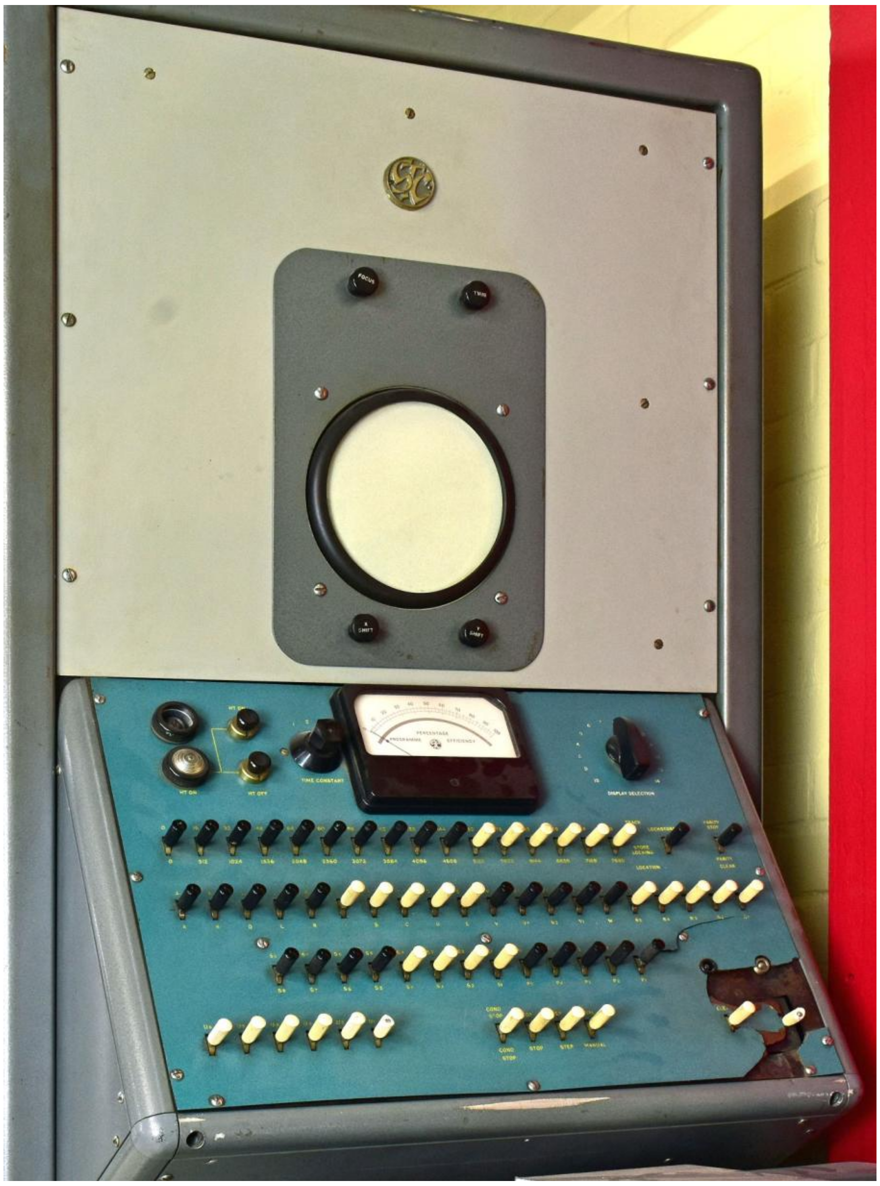


Figure 8 The main console of the ZEBRA containing CRT, switches and indicators.

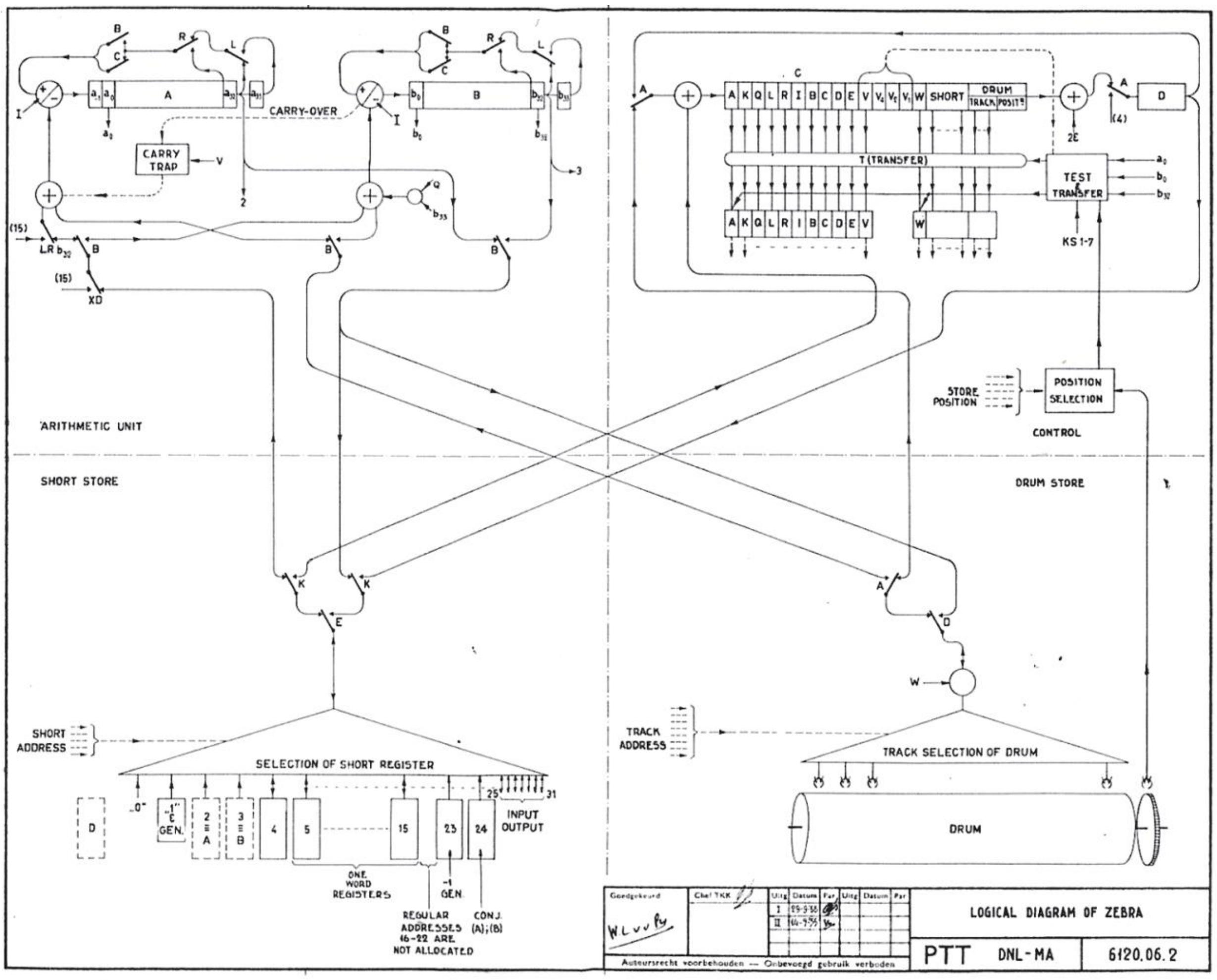


Figure 9 The hardware organisation of the ZEBRA

The registers of the ZEBRA

Figure 9 gives a schematic overview of the hardware. The left-upper corner shows the Arithmetic Unit containing the A- and B accumulators and some adders. Both accumulators may be put in series to provide for double precision integer arithmetic. Here the numeric calculations will be done. The right-upper corner shows the Control Unit (C, D and E-registers) handling the instruction sequencing. An instruction retrieved from the memory will be stored in the Control Unit, in which the C register holds the next instruction to be processed. The D register works like the program counter and the E register holds the instruction that is currently being executed. The A, B, C and D registers are constructed using shift registers and the E register contains flip-flops.

Because the details are not clearly visible in figure 9, figures 10 and 11 have been added to more clearly show the interesting parts of the Control Unit and the Fast Registers.

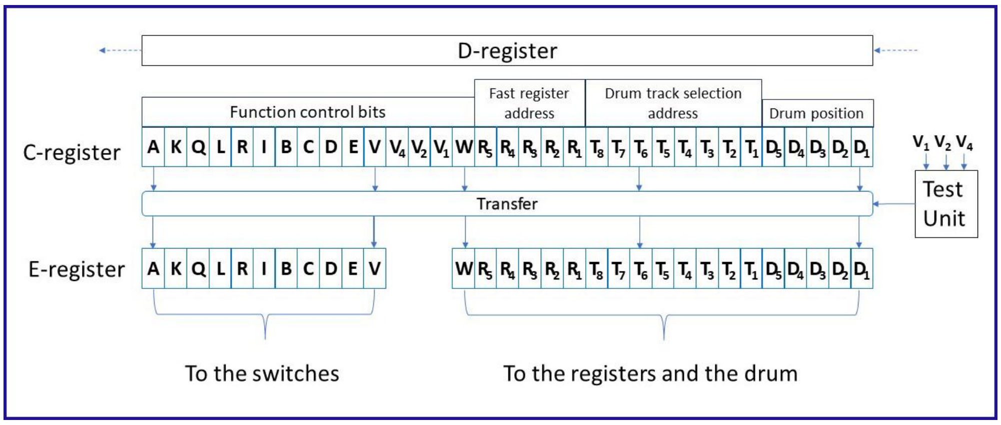

D-register
Function control bits
Fast register address
Drum track selection address
Drum position
C-register
A K Q L R I B C D E V V4 V2 V1 W R5 R4 R3 R2 R1 T8 T7 T6 T5 T4 T3 T2 T1 D5 D4 D3 D2 D1
V1 V2 V4
Test Unit
Transfer
E-register
A K Q L R I B C D E V
W R5 R4 R3 R2 R1 T8 T7 T6 T5 T4 T3 T2 T1 D5 D4 D3 D2 D1
To the switches
To the registers and the drum

Figure 10 Partial copy of Control Unit Diagram

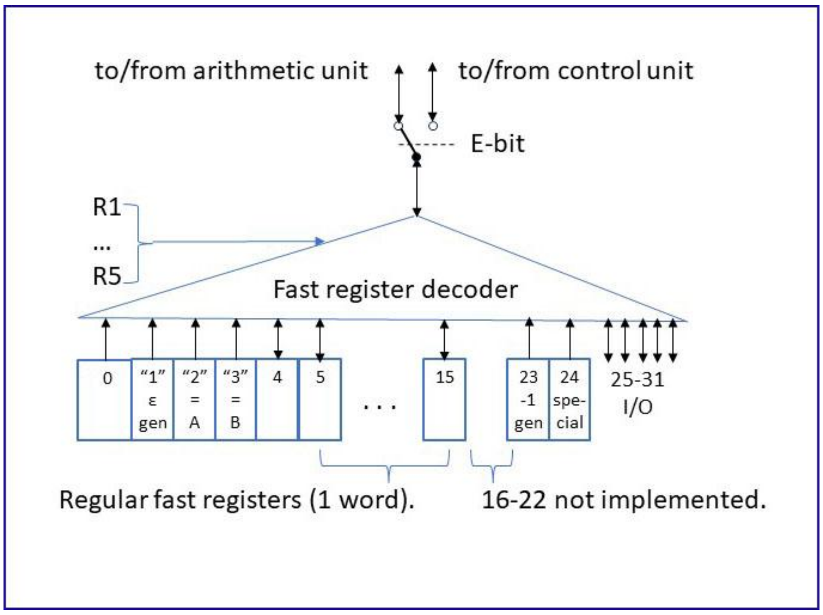


Figure 11 The Fast Registers in more detail

Words in the ZEBRA may contain either user data or instructions. Instructions use 33 bits words and data use 33 bit integers. No other data format is supported by the hardware. Because of the need to produce a reliable computer using a minimum of tubes the computer has been set-up as a binary serial machine. Data flows serially between the various components of the machine; least significant bit first.

Instructions are organized in several fields: the operation codes (function bits A .. V, V_4, V_2, V_1, W), the fast register addresses (R5 .. R1) and the main storage address divided up in track selection (T8 .. T1) and drum positions (D5 .. D1). Note that the ZEBRA may be used as a one address machine (main memory) and as a two address machine (main memory plus fast registers) and that the ZEBRA may also perform computations on the address part of instructions.

The fast registers in figure 11

The fast registers 0 and 1 contain constants; fast registers 2 and 3 are equivalent to the A- and B-accumulator; fast registers 4 .. 15 may be used for intermediate storage of data or instructions. Fast registers 16 .. 21 have not been implemented and registers 22 … 31 are reserved for I/O devices.

Reading from the drum may involve some latency; the processor is waiting for the drum. This is not acceptable for the accumulators and the fast registers. Each of these have their own track on the drum and by some mechanism the information is repeatedly written on that track such that upon reading the information is readily available. This is visible in figure 5 where it can be seen that on the top of the drum more read/write heads have been mounted than on the lower parts.

PART III Programming the ZEBRA

The three layer model

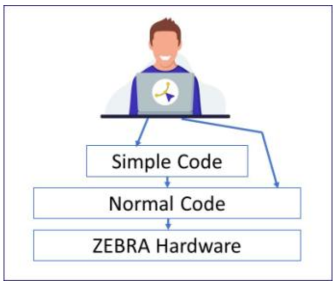


Figure 12 The three layer model

The ZEBRA follows the standard three layer format (figure 12). The user programs the Simple Code machine; this code is translated to Normal Code which is then executed by the hardware. With some additional effort the user may write programs using a mixture of Simple and Normal code. For programs requiring optimal efficiency, the user will probably use Normal Code directly.
There will be a short introduction to Normal Code here. The bulk of the explanation of Normal Code will take place in ANNEX B.

Introducing the Normal Code

The function bits of the ZEBRA (see Figure 11) are called **AKQLRIBCDEVV$_4$V$_2$V$_1$W** and determine which operations are going to take place in the current cycle of the machine. In those years users of the ZEBRA referred to the string **AKQLRIBCDEVV$_4$V$_2$V$_1$W** as “the Dutch Alphabet”. The computer effectuates them using the active function bits from right to left. Thanks to the parallel availability up to 12 bits may be combined in one instruction effectively speeding up the processing. This kind of ‘parallelism’ allows the ZEBRA to have a much greater throughput than other machines of that era could give.

For most of the function bits the connections between the memory, the arithmetic unit and the control unit will be made using the switches indicated in Figure 9 and the requested functions will be performed. As programming in Normal Code is rather difficult the discussion is postponed to ANNEX B, after Simple Code has been introduced.

Programming the ZEBRA in Simple Code

Simple Code was introduced to allow programmers to access the ZEBRA without having to learn the complex Normal Code. The memory model of Simple Code exists of two memory areas of 1200 words each; one area for instruction storage and one area for data storage. Both areas start at address 0. The Simple Code machine has one accumulator called *A*. In Simple Code there exists only one data type: the floating point data type. A floating point number is defined as $\pm$ *mantissa* $* 10^n$. The mantissa is a fraction: $0.1 < |mantissa| \leq 1$ and the exponent *n* has a value $-1024 < n < +1024$ (effectively $-999 < n < +999$) and the base is 10. The mantissa contains 33 bits allowing for nine digits of precision[1]. This clearly shows the technical/scientific application domain for which the ZEBRRA was designed. In addition to the accumulator A there are 7 registers named α, β, γ, δ, ε, θ and τ. These registers are used for (integer) counting and for managing subroutine returns. Subroutine jumps and goto’s use a set of 99 label locations of which a part is in use for the scientific subroutines and the remainder can be used by the programmer who is responsible for managing the labels used. Due to lack of space both registers and labels will not be discussed here.

The first example of Simple Code is a simple calculation. It is required to calculate the expression ab + cd, where a, b, c, d are stored in locations 0 .. 3 and store the result in location 4.

[1] The Stantec implementation uses an extra bit for parity checking on the drum.

Given the following set-up:

```
  Location | contents

      0    |  a
      1    |  b
      2    |  c
      3    |  d
      4    |

Program    | Action / Comment

      H    | a -> A
      V1   | ab -> A
      T4   | ab -> 4; 0 -> A
      H2   | c -> A
      V3   | cd -> A
      A4   | ab + cd -> A
      T4   | ab + cd -> 4
```

Someone understanding Dutch will recognize that H = Haal (get) and V = vermenigvuldig (multiply).

The following annotated example will give an impression about how to solve a simple mathematical problem. This example has been borrowed from the Syllabus document.

Calculate e^x using a series development as given below, but this is done in iterative form in the program.

$$e^x = \sum_{n=0}^{n=\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$$

In this program: (1) = 1, (2) = 10^{-7}, (A) = *x* and locations 0, 3 ,4, 5 and 6 needed but are not yet used. The notation (n) means take the contents of location n. The program is contained in figure 13.

```
------code------|--------comment ---

          U3        (A) -> 3; copy x to location 3
          H1        (1) -> A get the value of (1) into A
          U4        (A) -> 4 as zeroth partial sum; keep in A
          T5        (A) -> 5 as zeroth term; clear A
          U6        0 -> 6 as value for n
Q2        H6   ┐    (6) -> A
          A1   ├─add 1;    increment n
          U6   ┘    (A) -> 6; store in 6 again
          Z28  ┐    1/(A) -> A
          V3   ├─calculate the nth term = x/n . (n - 1)-th term
          V5   ┘
          U         store temporarily in 0; keep in A
          A4        add (A) to partial sum in 4
          U4        and store again in 3
          H         retrieve nth term again from 0
          Z18       take absolute value
          S2        subtract 2; (A) – (2) -> A
          E2        jump to Q2 when (A) > 0; i.e. |nth term| > 10^-7
          H4        get end result into A
```

Figure 13 Simple Code of the exponential power example

Program preparation

Programs are usually written off-line and punched on five hole paper tape paper tape. The tape may contain both instructions and data, but using several tapes (e.g. for libraries) was quite common. Figure 14 gives an example of such a program on a tape. The (Simple/Normal Code) tape is read into the computer by the Simple/Normal Input Program, a utility residing in high memory. The SIP/NIP checks the instructions on the paper tape and transforms them into binary numbers to be stored in the proper location. After loading the program and the data, processing could start and the computer would present the results on the Creed typewriter or the punch. The output on the typewriter was produced using one line of text, followed by one blank line; again followed by a line of text and blanks. The output could be held for a series of black stripes which became another idea for calling the machine the ZEBRA.
Debugging of programs was quite problematic; one could display a few memory locations on the oscilloscope and one could use a special instruction to have certain memory locations and their contents being printed. Once you found an error you had to re-punch your program or data tape.

The code has to be punched on paper tape in a propriety ZEBRA code. (see figure 1)

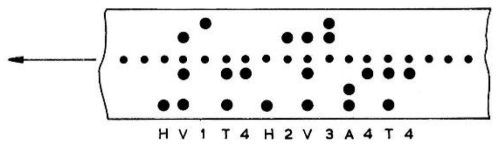


Figure 14 Punched tape for the example program

Speed of execution of Simple Code

The (floating point!) instructions **`Hn`** and **`Tn`** take 20 ms to execute; the **`An`** and **`Sn`** instructions take 40 ms and the **`Vn`** instruction takes 35 ms to execute.

Summing up

Using Simple Code the scientific programmer could solve his calculational problems after a couple of days of training. ANNEX A will give a shortened overview of the common instructions available in Simple Code.

Putting it all together

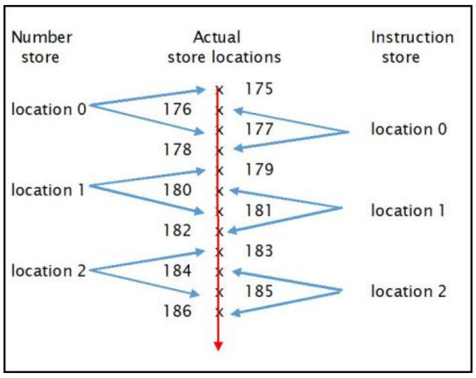


Figure 15 Interweaving Simple Code Programs instructions and data

The drum image contains a short input program (now: bootloader) on track 0 and some special locations on the following tracks. Simple Code may start on location 175 and continue to location 6147. Then follows the Simple Code Input Program and interpreter (now: meta assembler), the output programs, the Normal Input Program and the common (scientific) subroutines. Thanks to the compactness of the Normal Code used for all system programs, around 6000 out of 8192 locations may be used by the programmer. To protect the drum against unintentional overwriting it is possible to use switches on the ZEBRA panel to make tracks on the drum read-only.

In reality the instruction store and the data (number) store are not separated stores but instructions and data are interwoven into one stream. The instructions and data are put into alternate sequential locations in the store. Instructions require two locations for address and operand; data requires two locations for mantissa and exponent (see figure 15).

PART IV Further development and discussion

Emulators

The ZEBRA computer was a success; both commercially and intellectually. When the computers were finally taken out of service it became an intellectual challenge to extend their life by writing a so-called emulator; a program written in another language emulating the ZEBRA. Two emulators are known. The oldest one was written in Pascal and runs on a command line DOS system using a 4086. It was written by Don Hunter and later edited by Willem van der Poel. It may be obtained from the author of this paper. This emulator only works in a MSDOS-box using a 16-bit Pascal compiler. It has been measured that this emulator works two thousands time *faster* than the original ZEBRA! The second emulator was written by Jurjen N. E. Bos https://bitbucket.org/jneb/zebra/overview . The code has been written in Python3. There are two versions of this emulator: a command line version which is operating correctly and a graphical version also using the program tk. The latter one gives some minor problems under Linux because of the tk-part. On inspecting the code one may find many TODOs showing that writing a program that correctly emulates both hardware and software is a very difficult job. A particular nicety on this emulator is that it tries to emulate the timing of the original ZEBRA.

High Level Languages

A drawback of programming computers in assembly code for a particular machine is that programmers need both to learn a new language for every new computer they start working on and rewrite all their old code. In the late fifties machine-independent languages (high level languages) overcoming this drawback were developed. Of course it has been tried to use high level languages on the ZEBRA. As far as known compilers for the languages Algol 60 and Lisp were developed.

A short evaluation of the ZEBRA architecture.

Although the ZEBRA has been quite successful, it is interesting to evaluate its architecture with respect to the long term development of computers. The most eye catching feature of the ZEBRA, the use of multiple function bits in one instruction, has as far as known not survived in later generations of computers.

The use of serial bit streams for data transport and calculation was based upon the need for simplicity of the arithmetic unit therewith needing the least amount of unreliable vacuum tubes. Later generations of computers used reliable transistors instead of vacuum tubes allowing parallel data transport and parallel arithmetic units. Apart from the fully transistorized ZEBRA that was developed somewhat later, to the knowledge of the author, serial bit streams have only been used in a cheap version of the PDP-8 architecture, the PDP-8S (S of Serial) a rather slow implementation of the PDP-8 by DEC (Digital Equipment Corp).

In the ZEBRA the drum provides for all the clock signals needed for the processor. This results in a strong coupling of the drum timing with the processor timing. In later bus-driven systems the components of a computer operate more a-synchronously.

The way the ZEBRA addressed the memory and registers on the drum made it difficult to extend the memory over 8 kilowords. In the transistorized version of the ZEBRA Stantec realized (optional) blocks of ferrite core that could extend the memory size but this was realized through a simple mechanism of bank switching.

In short, although the ZEBRA architecture has had a lot of enthusiastic users and the computer has contributed much to the understanding of computer architectures, it had not many follow-ups.

Conclusion

Before we jump to the conclusions: the very interesting information about Normal Code is available in Annex B.
This article has provided a short and incomplete overview of the hardware and software of the ZEBRA computer as it was sold in the late fifties and early sixties of the previous century. Scientists using the ZEBRA formed a successful ZEBRA club to exchange ideas and programs. Compared to current architectures the ZEBRA is unique in providing multiple active bits to execute in one cycle of the machine and also in taking measures to remedy the waiting time inherent to the use of a rotating drum. The design of the ZEBRA was a great success. The endeavour of van der Poel has been recognized by the IEEE with the prestigious Computer Pioneer award in 1984. It is appropriate to mention here dr. G. van der Mey; a deaf blind person who was heavily involved in this kind of program development work and worked especially on the construction of an Algol 60 compiler.

The author wants to thank P.J. Trimp and O. Rompelman and for their helpful comments on earlier drafts of this article. S.M. Nijhuis (library TUD) is thanked for the retrieval of photographs of the Julianalaan set-up of the ZEBRA.

Literature

General ZEBRA documents:

The Logical Principles of Some Simple Computers.
Willem Louis van der Poel
Dissertation, 1 feb. 1956, University of Amsterdam

Microprogramming and Trickology
W.L. van der Poel
In: Hoffman W., Digital Information Processors, Vieweg + Teubner Verlag, Wiesbaden.

The Simple Code for Zebra
W.L. van der Poel
Het PTT-bedrijf; aug. 1959; Deel IX, no 2.

Less is more in the Fifties; Encounters between Logical Minimalism and Computer Design during the 1950s.
L. De Mol, M. Bullynck, E. G. Daylight.
IEEE Annals of the History of Computing, Institute of Electrical and Electronics Engineers, 2018

Introduction to Stantec Zebra (June 1959).
Standard Telephones and Cables Ltd., Information Processing Division, Newport, Mon

An Outline of the Functional Design of the Stantec Zebra; Stantec; Sept. 1958

Stantec Zebra flyer 1958,

Simple Code:

SYLLABUS eenvoudige code voor Zebra
Electronisch rekencentrum, Rijksuniversiteit Utrecht, March 1961 (in Dutch)

Simple Code Instruction List, Stantec, Sept. 1959 .. 1961

The STANTEC-ZEBRA simple code and its interpretation
R. J. Ord-Smith (STANTEC)
https://www.sciencedirect.com/science/article/pii/0066413860900380

Chengetai Deansed Kadenge
Understanding the Stantec-Zebra
Sept. 2016, College of Science, Swansea university, UK

Additional materials:

G. van der Mey
PTT Report 164 MA
Process for an Algol Translator Part 1: The translator
http://bitsavers.trailing-edge.com/pdf/stantec/Zebra_Algol-60_Part1_Jul62.pdf
Process for an Algol Translator Part 2: The interpreter
http://bitsavers.trailing-edge.com/pdf/stantec/Zebra_Algol-60_Part2_Jul62.pdf
Process for an Algol Translator Part 0: Introduction
Process for an Algol Translator Part 3: The tables
http://bitsavers.trailing-edge.com/pdf/stantec/Zebra_Algol-60_Part0_Part3_Jul62.pdf
Process for an Algol Translator Part 4: Flow diagrams

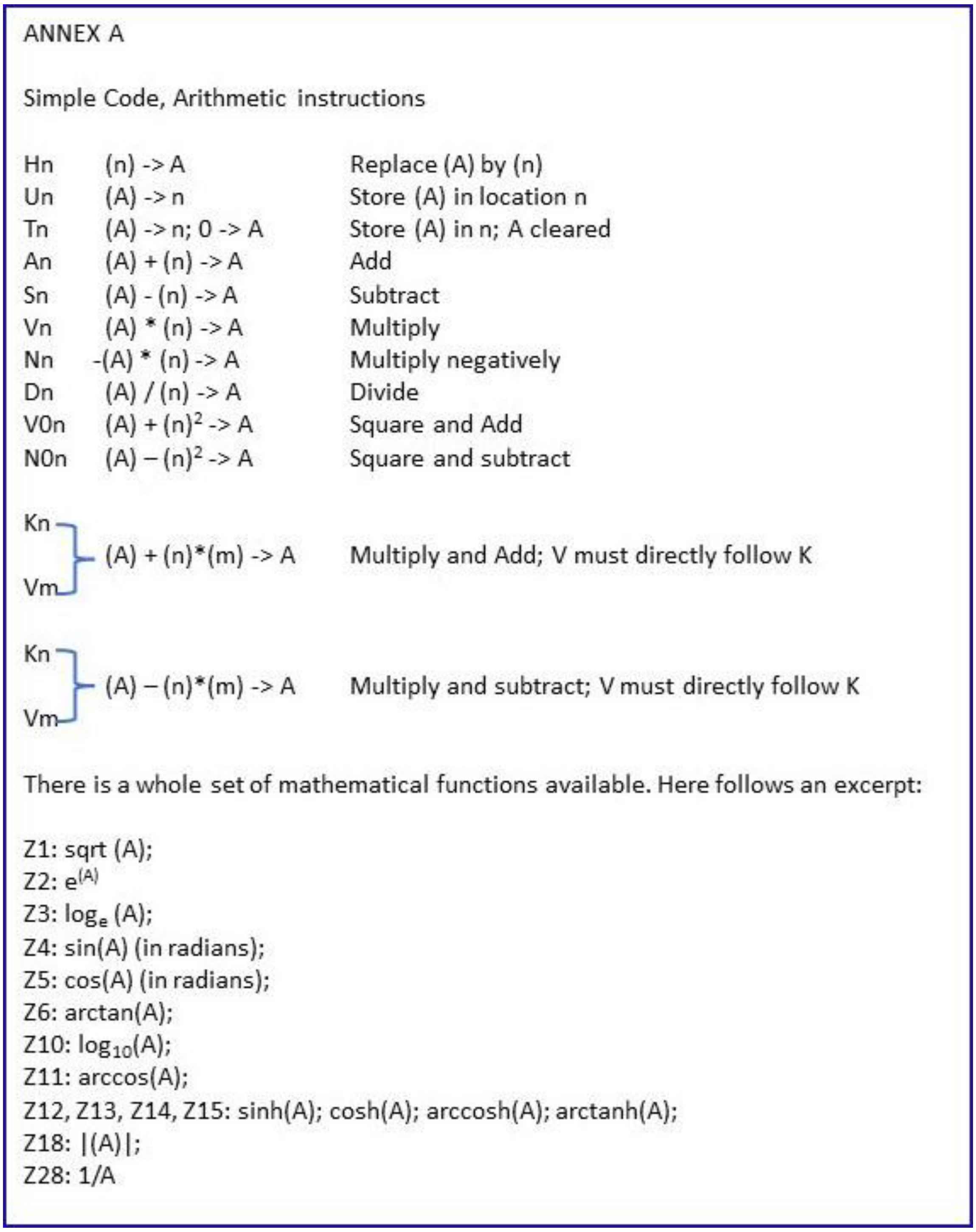
Erik Verhagen; Geheugentrommels; ADFO BOOKS (in Dutch)

HeerdeBeer.org—Bouw en gebruik van rekenapparaten bij de Mathematische Afdeling van het Centrale laboratorium der PTT (in Dutch)

https://computer.org/profiles/willem_vanderpoel

ANNEX A: Overview Simple Code Instructions

The complete Simple Code consists of more than 100 codes. Here only the codes for numerical processing will be given in figure 16. Please note that the only data type in Simple code is the floating point number.



ANNEX A

Simple Code, Arithmetic instructions

Hn	(n) -> A	Replace (A) by (n)
Un	(A) -> n	Store (A) in location n
Tn	(A) -> n; 0 -> A	Store (A) in n; A cleared
An	(A) + (n) -> A	Add
Sn	(A) - (n) -> A	Subtract
Vn	(A) * (n) -> A	Multiply
Nn	-(A) * (n) -> A	Multiply negatively
Dn	(A) / (n) -> A	Divide
V0n	(A) + $(n)^2$ -> A	Square and Add
N0n	(A) – $(n)^2$ -> A	Square and subtract
Kn Vm	(A) + (n)*(m) -> A	Multiply and Add; V must directly follow K
Kn Vm	(A) – (n)*(m) -> A	Multiply and subtract; V must directly follow K

There is a whole set of mathematical functions available. Here follows an excerpt:

Z1: sqrt (A);
Z2: $e^{(A)}$
Z3: $\log_e$ (A);
Z4: sin(A) (in radians);
Z5: cos(A) (in radians);
Z6: arctan(A);
Z10: $\log_{10}$(A);
Z11: arccos(A);
Z12, Z13, Z14, Z15: sinh(A); cosh(A); arccosh(A); arctanh(A);
Z18: |(A)|;
Z28: 1/A

Figure 16 Simple Code instructions for mathematics

ANNEX B: Programming the ZEBRA in Normal Code

In Normal Code the programmer may write commands to access all of the facilities of the hardware obtaining maximum performance. In current parlance Normal Code would be named Assembler Code. For proper understanding please visit figures 10, 11 and 12 again. Please note that the integer is the only data type in Normal Code. The function bits of the ZEBRA (see Figure 11) are known as **AKQLRIBCDEVV$_4$V$_2$V$_1$W** and determine which operations are going to take place in the current cycle of the machine. In a machine cycle the functions are executed *right* to *left*.

First, the meaning of the individual bits of the operation codes will be discussed. Then the way to compose the operation codes will be elucidated. This will be followed by several 'simple' programs.

The meaning of the operation bits explained

The A bit:
When A = 1:
(a) the main store and the arithmetic unit are connected.
(b) a transfer path is provided from the D control register to the C control register.
(c) a transfer path is provided from fast register 4 to the D control register.
(d) Semantics: A + (*n*) + (*m*) -> A ; The current contents of the A accumulator will be added to the contents of the drum address *n* and the fast register *m*, The result will stay in A. This action takes about 20 milliseconds.

When A = 0 = X: (the X must be written when A equals zero; meaning Ne**x**t instruction).
(a) the main store and the control unit are connected. This will read the next instruction from the drum.
(b) a transfer path is provided from the C control register to the D control register.

The K bit:
When K = 1: The fast registers and the control unit are connected.
When K = 0: The fast registers and the arithmetic unit are connected.

The D bit:
When D = 1: Writing into the drum will occur from either of the accumulators depending upon the B bit.
When D = 0: Reading from the drum into the control unit or the arithmetic unit occurs.

The B bit:
When B = 1: The B accumulator is selected.
When B = 0: The A accumulator is selected.

The E bit:
When E = 1: Writing from the drum into the fast registers or the arithmetic unit occurs.
When E = 0: Reading from a fast register or to the arithmetic unit will occur.

The C bit:
When C = 1: The accumulator selected by the B bit is cleared.

The I bit:
when I = 1: The number entering the arithmetic unit is subtracted from, instead of added to the accumulator selected by the B bit.

The L and R bits designate a one position left (right) shift of both accumulators (in series). L and R being both 1 has a special meaning with multiplication.

The W bit is automatically provided when needed. When the W bit = 0, the computer will wait on the drum information becoming available. When the W bit = 1, the computer will not wait for the drum; this means it will take its instructions from the Fast Registers.

The $\mathbf{V_1}$, $\mathbf{V_2}$ and $\mathbf{V_4}$ bits allow checking one of the A or B accumulators for <0, <=0, 0, >0, >=0 etc. When the test fails the next instruction is fetched from memory. When the test succeeds, the instruction is copied to the E-Register from where it will be effectuated using the active function bits from right to left.

Intermezzo on the A and K function bits

As shown before function bits enable a large number of instructions to be created. Figure 16 will show the use of combining the A and K bits.

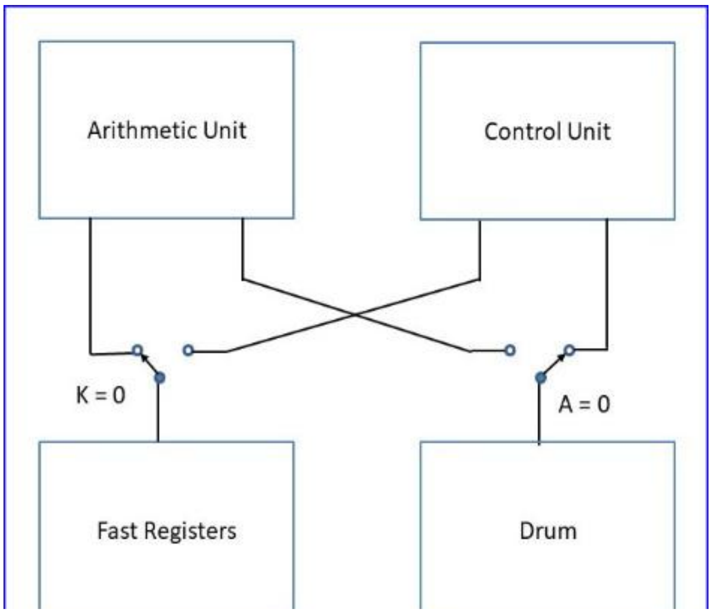


Figure 17   Using both the A and K bits

When A = 0 and K = 0 (as shown in the figure) a so-called *adding jump* is executed. The contents of the addressed fast register is added to the A accumulator and the next address is fetched from the drum to the control unit.
When A = 0 and K =1 a *double jump* is executed. A word from one of the fast registers is added to an instruction fetched from the drum (address arithmetic).
When A = 1 and K = 0 a *double addition* is executed. The contents of the addressed fast register is added to the contents of the drum location; both are added to the A-accumulator.
When A = 1 and K = 1 a *jumping addition* occurs. A word from a fast register may modify an instruction and a word from the drum is added to the A accumulator.
The reader should not confuse the A-bit with the A-accumulator or with addition. The A-bit connects the various parts of the computer. Addition is done according to the above text. The B-bit decides whether the A-accumulator or the B-accumulator will be used. The reader should also understand that there is no automatic sequencing of instructions in the ZEBRA. The X-bit (zero-A-bit) determines the next instruction to be executed. See Ex1 and Ex2 further on. **End of intermezzo.**

Composing function bits into one operation code

As said earlier many function bits may be combined into one instruction. Two examples:
- the instruction X327BCE5 means: Take your next instruction from the drum location 327. Write the contents of the B accumulator into fast register 5. Clear the B accumulator.
- the instruction AD4015BCE9 means: write the contents of the B accumulator into the main memory location 4015 and into fast register 9. Clear the B accumulator. Your next instruction will come from the D register.

In the following examples (n) will indicate 'the contents of location n'.

Some simple programming examples:

Ex. 1: (300) + (500) + (700) –> A Add the contents of memory addresses 300, 500, 700 in accumulator A.

```
Address     | Instruction    Comment

      100   | X101           Take
      101   | A300           Do
      102   | X103           Take
      103   | A500           Do
      104   | X105           Take
      105   | A700           Do
      106   | X107           Take
      107   | X200           Take and continue
```

Ex. 2: Combining Do and Take while using the fast registers and B: (5)+(7)+(12)+(14) -> B

```
      100   | X101BC5        (5) -> cleared B
      101   | X102B7         (5)+(7) -> B
      102   | X103B12        (5)+(7)+(12) -> B
      103   | X104B14        (5)+(7)+(12)+(14) -> B
```

Ex.3 : Showing the **I** bit (subtraction) and the (sometimes needed) point separator (5)+(6)-(7) -> A

```
      100   | X101C5         (5) -> cleared A
      101   | X102.6         (5)+(6) -> A
      102   | X103I7         (5)+(6)-(7) -> A
```

Some shorthand: `X next_address` may be abbreviated as `N`.
The previous example could be written as:

```
      100   | NC5
      101   | N6
      102   | NI7
```

In the same way `'N_address'` may be written as `NKK:`

```
      101   | ABC102   => NKKBC
      102   | X002.1
      103   | A104CE15 => NKKCE15
```

The **V**, $\mathbf{V_1}$, $\mathbf{V_2}$ and $\mathbf{V_4}$ bits allow testing the A or B accumulator for <0, <=0, 0, >0, >=0. Additionally these bits can be used for checking the positions of the various switches on both consoles. This is done in the Test Unit in figure 11. When the test fails the next instruction is fetched from memory. When the test succeeds, the current instruction is copied from the C to the E Register from where it will be effectuated using the active function bits from right to left.

Normally, the drum address precedes the fast register address. When the fast register address is written first, another address *p* can be written. This will cause an *inactive* drum address 8192 - 2*p* to be input. The W-bit will be automatically activated. Thus:
XK5   means functional bits *K* and *W* are present, drum address 0000, fast register 5
and thus
X5K7   means functional bits *K* and *W* are present, drum address 8178, fast register 5
which means: repeat the instruction in fast register 5 seven times.

Here some magic is involved. The intention is to move loop code from the slow drum to the fast registers where the loop will be executed. The basic idea of repeating an instruction stems from the fact that when a fast register is serving as the next instruction source, the drum address is blocked because the W-bit is present. Nevertheless the address field of the instruction is augmented by 2 every cycle. This does not influence the fast register address until the drum address is equal to 8192. Because the address has only 13 bits, it will overflow after getting added *p* times 2 to it. Because of the overflow the fast register bits (R5 .. R1) will be incremented. At this fast register address the return address (where to go after the loop) needs to be provided before the execution of the loop starts. Hence the notation *X5K7* for: repeat the instruction in fast register 5 seven times. End of magic.

Before embarking on explaining the programming example of integer multiplication in the ZEBRA it may be appropriate remembering how binary multiplication works. The following example shows how the multiplier is shifted and how the result is constructed. Something similar occurs is the program below by shifting both the A and the B register.

```
    101101   multiplicand
      1101   multiplier
---------- *
    101101   * 1
   000000    * 0, shifted one place
  101101     * 1, shifted two places
 101101      * 1, shifted three places
---------- +
1001001001   result of multiplication
```

In the following code the A and B accumulators are forming a double-length accumulator, the multiplicand is placed in register 15, the multiplier is placed in B, and A is cleared initially. At every cycle the least significant bit of B is tested and only if it is one, the contents of 15 is added to A. Then A and B are both shifted to the right, thereby dropping the right hand bit of the multiplier (in B) and shifting one bit from the product from A to B.

The following example gives a complete open subroutine for multiplication of A and B without pre-supposing any contents of the registers. The repetition of the multiplication runs as follows starting at address 100. Here fast register 5 will contain the *ALR* instruction and fast register 6 contains the return instruction.

100	| *NE5*	Pre-instructions activity only starts after the next instruction (magic again)
101	| *NKKCE15*	Store multiplicand -> 15, *ALR* -> A
102	| *ALR*	Constant. After action of *NE5* stores *ALR* -> 5.
103	| *NKE6LRC*	Place return instruction in 6, Clear *A* and add (15) conditionally, Shift right.
104	| *X5K15LR*	Repeat ALR fifteen times (*X*5*K*15*LR* itself is done 16 times).

```
105    | NLRI         After action of E5 destroys ALR but that does not matter
```

The performance of the ZEBRA for integer calculations in Normal Code is: addition and subtraction take 312 µs; multiplication takes 11 ms and division takes 35 ms. An average operational time of all operations is 2 ms.

Minimizing the number of slow drum accesses could be arranged by copying an instruction from the drum to the fast registers and subsequently modifying the contents of the fast registers in order to transform the current instruction I into the next instruction I' ,and I' into I'', and so forth. Until the drum was accessed a second time, the program was executing "under water," using van der Poel's terminology. The reduced number of (slow) accesses to the drum allowed the program to maintain a high execution speed. Using many functional bits in the same instruction was called by van der Poel: "microprogramming".

There do exist instructions for calling subroutines an returning from those. These will not be elaborated upon here. What is missing in the ZEBRA architecture are the notions such as stack pointers and activation records. When needed, one has to program them using Normal Code.

A software library containing integer multiply and divide, a package for extended precision arithmetic, the floating point format data type, floating point arithmetic operations and mathematical functions (sin, cos, etc.) are all available. As may be seen this programming in Normal Code is the work of specialists. Using Normal Code the programmer is able to get the optimum performance of the machine. Unfortunately, Normal Code is too difficult for the layman programmer and therefore Simple Code was developed.

-.-.-.-