TRS-80 With A Keccak Sponge Cake

Jean-Marie Chauvet

MassiveRand
jmc@massiverand.com
http://www.massiverand.com
62, ave. Pierre Grenier, 92100 Boulogne-Billancourt, France

Abstract. The subject of this paper, an improbable implementation of a recently standardized cryptographic hash function on a thirty-five-year-old microcomputer, may strike some as unusual and recreative at best. In the tedious discipline of the process, however, lessons were learned in implementation trade-offs for basic cryptographic primitives which may prove interesting in the current context of securing (small to nano) machine to machine communications. More importantly, that such insights might stem out of revisiting how earlier computing platforms relate to the code written on them to cast a distant light on modern connections of code to material, historical and contextual factors certainly illuminates the joys of retrocomputing.

Keywords: Cryptography, Random Bit Generator, Stream Cipher, SHA-3, Keccak, Z80, TRS-80, Retrocomputing, Crazy Ideas, Remix

1 Introduction

This paper briefly reports on an experimental implementation of Keccak, a family of *sponge functions*, for the TRS-80 Model 1, first released in August 1997. A sponge function is a generalization of the concept of cryptographic hash function with infinite output and can perform quasi all symmetric cryptographic functions, from hashing to pseudo-random number generation and to authenticated encryption [2]. Keccak is a finalist in this year's RFP for SHA-3 a successor to SHA-2.

In confronting this retrocomputing challenge, we drew solace from the Keccak designers claim that:

> "Keccak excels in hardware performance, with speed/area trade-offs [...] Keccak has overall good software performance. [...] On constrained platforms, Keccak has moderate code size and RAM consumption requirements."

and inspiration from the recent comprehensive work on implementing cryptographic primitives on microcontrollers [12].

2 The TRS-80 Model I

It was with minimal expectations that, in August 1977, Tandy Corporation teamed up with Radio Shack to release the TRS-80, one of the first personal computers available to consumer markets. And as it turned out, the TRS-80 surpassed even the most cautious sales estimates by tenfold within its first month on the market despite a hefty $600 price point; the burgeoning prospects of a new era in personal electronics and computing could no longer be denied. While the original machine shipped with BASIC Level I, based on Wang's free TinyBASIC, and 4KB of memory, a 12" monitor and a Radio Shack tape recorder as datacassette storage, later shipments in 1978 came with Level II BASIC, licenced from Microsoft, and 16KB of memory: this is the target platform of the present implementation. (The ability to expand memory up to 48KB, to access floppy drives, a second cassette port, a Centronics parallel printer and an optional RS232 port required the proprietary Expansion Interface, a bulky box that fit under the monitor.) The TRS-80 uses a Zilog Z80 CPU clocked at 1.77MHz.

At about the same time, in late 1977, the Senate Select Committee on Intelligence undertook a classified study concerning allegations that the National Security Agency (NSA) was improperly involved in the development of a data encryption standard (DES) [1]. The following allegations were investigated by the Senate Select Committee on Intelligence: that the NSA exerted pressure on officials in the National Science Foundation (NSF) to withhold grant funds for scholarly research in the field of public cryptology and computer security; that the NSA directed an employee, who was also a member of the Institute of Electrical and Electronic Engineers (IEEE), to write a letter to IEEE warning its members that certain actions related to an upcoming Information Theory Group Conference could be in violation of Government regulations affecting the publication and export of cryptographic information: that U.S. Government harassment brought about a chilling effect in universities doing research in cryptanalysis and even resulted in one university with- drawing already published material from its library shelves; that the NSA, under the guise of testing the mathematical formulae (algorithms) submitted to the National Bureau of Standards (NBS) for consideration as a Data Encryption Standard (DES), "tampered" with the final algorithm in order to weaken it and create a "trapdoor" which only the NSA could tap; that the NSA forced the company (IBM) whose algorithm was chosen, to compromise the DES's security by reducing the key size used in the encryption and decryption process; and that the DES failed to allow for future technological advancements which will permit successful brute force attacks within the next several years. The study report, dated April 1978, cleared the NSA of all suspicion but acknowledges its instrumental role in convincing IBM that a reduced key size was sufficient, assisting indirectly in the design of the S-box structure, and in certifying that the final DES algorithm was, to the best of their knowledge, free of any statistical or mathematical weaknesses.

2.1 The Z80 early microprocessor

All data in the TRS-80 and most in the Z80 is handled in 8-bit bytes. There are 14 general purpose registers in the CPU, designated A, B, C, D, E, H and L and the so-called primed counterparts, A' to L'. At any given time one set, primed or non primed, is active. The A register is used as a default operand in many of the instructions and is often referred to as the accumulator. Registers are paired, as BC, DE, HL and AF, in the few 16-bit arithmetic and pointer operations in the instruction set. Special-purpose registers are the 16-bit program counter, PC, the 16-bit stack pointer, SP, and two 16-bit index registers, IX and IY. The flag registers are F, F' respectively, 8 bits signalling sign, zero, half-carry, parity/overflow and carry after arithmetic and logic operations [13]. The Z80 instruction set, a superset of the older 8080 and 8008 instruction sets, affords more than 500 combinations altogether covering data movement, arithmetic, logical and compare, decision-making and jumps, stack operations, bit shifting and I/O operations [7]. Note that in contrast to modern microcontrollers such as e.g. the ATmega and ATxmega cores family from Atmel, there is no multiplication instruction–not necessarily an auspicious starting point for public-key cryptography.

We use the representation of large integers and elements of finite fields as byte arrays using radix 2^8 typical on such 8-bit architectures.

2.2 The modern toolchain

Like all early microcomputers, the TRS-80 is designed for users to key in their BASIC programs and store them on audio cassettes. Thus BASIC became the prominent instrument in the programming usage the budding home-computing community engaged in [5] at the time. Programming know-how was further enhanced and broadened by the development of an ecosystem of hobbyists exchanging audio tapes or program listings (to be carefully re-keyed in) in dedicated newsletters. The hobbyist subculture of the early eighties was in fact the harbinger of the modern PC software industry.

In our IDE age, the claim in the Level II BASIC Reference Manual that

> "LEVEL I users undoubtedly spent lots of time retyping long program lines, all because of a typo, or maybe just to make a minor change.[...] Level II's editing features eliminate much of this extra work. In fact, it's so easy to alter program lines, you'll probably be able to do much more experimenting with multi-statement lines, complex expressions, etc."

might seem a bit of an overstatement, considering the full chapter devoted to the description of the `EDIT` command (and its delicious dialect: *nSpacebar*, $n \leftarrow$, `SHIFT` $\uparrow$, `L`, `X`, `I`, `A`, `E`, `Q`, `H`, *n*`D`, *n*`C`, *n*`Sc`, *n*`Kc`) [10].

The TRS-80 soon also had an integrated editor-assembler for machines with at least 16K memory, called EDTASM, which complemented nicely the TBUG monitor, for these hobbyists interested in exploring assembly language and machine language. EDTSASM would be loaded from its distribution cassette, as a

machine language program, and once started would offer an editor for typing in assembly code. A set of commands then allowed the source to be assembled and written back in binary machine language to a fresh audio tape. The machine would be manually reset and the binary cassette just produced rewound and reloaded the usual way before running the program. The complete edit-build-run cycle indeed involved audio cassette manipulation and reinitialisation of the TRS-80.

Its transformative nature is a major aspect of retrocomputing. Here the audio cassette plays a central role as the link between the original machine and modern programming environments. In the current instance Knut Roll-Lund's excellent program, PlayCAS, is critical in setting up a modern toolchain as it transforms the PC into a TRS-80 tape deck, playing binary formatted (`.cas`) assembly or BASIC files produced by many emulators, as audio files transferred to the TRS-80 through the standard audio/mic jacks. With PlayCAS we are given back the `CLOAD`, `CSAVE`, and `SYSTEM` commands to transfer programs back and forth between the TRS-80 and the modern PC. Our toolchain can then be selected from a large choice of Z80 compilers, assemblers, and TRS-80 simulators. We used the *Small Device C Compiler*, SDCC, which targets the Zilog Z80 among many others and includes assembler, simulator, and debugger in a comprehensive suite of tools. We also used SDLTRS intensively, a Radio Shack TRS-80 Model I/III/4/4P emulator for Macintosh OSX, Windows, and Linux. We created a short Python script to convert from the Intel Hex Format (`.ihx`) to the TRS-80 cassette format (`.cas`) documented in [6].

3 The Keccak Sponge Cake

3.1 The cryptographic sponge construction

Keccak is a family of so-called sponge functions. The sponge function is a generalization of both the cryptographic hash function and the stream cipher with infinite output. It can perform quasi all symmetric cryptographic functions, from hashing to pseudo-random number generation to authenticated encryption [2]. Early october 2013, news was released that a subset of the Keccak family was selected for NIST's current proposal for SHA-3 [3].

Keccak functions rely on one of seven core operations named Keccak-f[b], with b=25, 50, 100, 200, 400, 800 or 1600 bits representing the size of an internal state. In this work, we only looked into b=200, 400, 800 and 1600 as these sizes are easily represented as arrays of complete bytes on the 8-bit Z80. For instance when b=200, the state is an array of 5-by-5 so-called *lanes* each of which is 1 byte long; when b=1600, each of these 25 lanes is 8-byte long.

The Keccak-f[b] function proceeds in a sequence of *rounds*, each one operating a complex series of permutations, rotations and exclusive or operations on individual bytes of the state. The number of rounds is $n = 12 + 2l$ where 2^l is the length of a lane measured in bits; for our experiments, n ranges from 18 for b=200 to 24 for b=1600.

In the context of cryptography, the sponge construction is a mode of operation, based on a fixed-length permutation and a padding rule, which builds a function mapping variable-length input to variable-length output. Its basic operation takes as an input a binary string of any length, in several rounds of an *absorbing* phase, and then produces a binary output string of requested length through additional rounds, in a *squeezing* phase. Applying the sponge construction with a random permutation results in a so-called random sponge. It turns out that a random sponge is as strong as a random oracle, except for the effects induced by the finite memory. Random sponges are thus well suited to replace random oracles for expressing security claims [4].

3.2 Randomness and the TRS

Entropy pool and randomness generation presented quite another interesting issue. Applications of Keccak use the OS-provided random number generator (RNG), reading random bytes from `/dev/urandom`. The BASIC Level II interpreter in ROM [9] indeed features a pseudo-RNG, through two calls: `RANDOM` to seed the PRNG, and `RND(N)` which returns a single-precision floating point value between 0 and 1 if N is 0; a random integer between 1 and N, included, when $N > 0$. After some disassembling of the ROM and further interacting with the TRS-80 discussion groups, it surfaced that this PRNG is a Linear Congruential Generator based on the following iteration: $x_n = 4253261x_{n-1} + 372837 \mod 2^{24}$. The seed (24-bit long) is stored at a specific lower RAM address where `RANDOM` overwrites its middle byte with the internal R register. There is no seed initialization at power-up or reset, so that failure to call `RANDOM` causes the same sequence of random numbers to be produced in successive calls to `RND`.

Interestingly enough, the same LCG was apparently already present in the older Level I, which was based, however, on a completely different BASIC, namely TinyBASIC. A 2^{24} modulus LCG is also present in the later QuickBasic, GW Basic (with constants 214013 and 2531011), and even Visual Basic 6 (with constants 1140671485 and 12820163) all released by Microsoft in the following decade.

4 Implementing Keccak

4.1 Assembly Language and BASIC Level II

Bit operations exclusive or and rotations are the elementary transformations performed during a Keccak round. These bit operations are directly supported by the Z80 instruction set (`RLCA`, `RLA`, and similar, and `XOR`, `CPL`), which makes implementation in assembly language straightforward. On the other hand BASIC Level II only offers bit AND and bit OR on one- or two-byte integers, making it more challenging. Furthermore, BASIC Level II integers are represented as signed 2-byte numbers in the range $[-32767, 32767]$ which adds new difficulties for high-range Keccak functions.

As a matter of reference point we implemented Keccak-f[200] both in BASIC Level II (see Appendix) and assembly language, while we only developed assembly code for b=400, 800 and 1600. In the BASIC implementation, we simply created a subroutine for the exclusive or operation based on the equality $XOR(a, b) = OR(AND(a, NOT(b)), AND(b, NOT(a)))$ and another one to arithmetically compute the left 1-bit rotation of a one byte integer. Keccak higher level permutations are then built on top of these (hardly optimal) primitives.

Some implementation trade-offs were different for the Keccak variants depending on the size of the internal state. The Z80 offers a few instructions on 16-bit long data which could be leveraged for the cases where $b > 200$ for instance. Similarly loops implementing bit rotations by more than 8 bits were either unfolded or replaced by whole byte manipulations in the higher b variants. Finally, stack size considerations drove us to pre-compute and store constants in the global data segment, more specifically permutation indices modulo 5 and numbers of bit rotations modulo the lane size. All numerical results were tested against the Python reference implementation and test vectors provided on the Keccak Web site.

4.2 Simple BASIC API

The question of the BASIC API is important in maintaining at least a light varnish of authenticity to the effort, as the language was the prominent instrument through which the budding home-computing community throve [5] at the time. We kept the API to a rough minimum, however, as memory didn't really allowed much sophistication. The sponge functions are machine language routines first read from cassette by the `SYSTEM` command. In Level II, the BASIC commands `POKE I,B` and `PEEK(I)` respectively sets the value of memory location I to B and returns the value at memory location I. The individual bytes of the state are "poked" at determined locations in memory, in the absorbing phase. Then the machine language routines are called using the `USR(N)` well-known idiom in BASIC Level II, once the entry point of the selected routine has been "poked" in reserved addresses 16526 and 16527, LSB first. Similarly, during the squeezing phase, results are "peeked" at the same determined memory locations.

4.3 Results

A preliminary remark: on the Z80, machines cycles are divided up into system clock cycles called T-states, so that performance is measured in T-states or as plain execution time on a vintage 16K Model I machine. Table 1 compares the Keccak-f[200] Level II BASIC and Z80 assembly implementations. The number of T-states execution for one call of the sponge function is reported here as traced by the SDLTRS emulator, running on 2.2 GHz AMD LD1250 monocore CPU. While the code size is comparable in both implementations, both using a few extra hundred bytes for data, execution time for the assembly implementation is 39x faster than the BASIC program. Table 2 shows performance of the assembly

Table 1. Keccal-f[200] in Level II BASIC and assembly

	BASIC	Assembly
Code Size (bytes)	2,140	2,112
Execution Time (s)	156	4
T-States Executed (10^6)	307	13.9

Table 2. Keccal-f[b] in assembly

	b=200	b=400	b=800	b=1600
Code Size (bytes)	2,112	3,280	4,920	8,742
Data Size (bytes)	70	228	898	1,930
Execution Time (s)	4	6	14	25
T-States Executed (10^6)	13.9	21.8	36	42.8

language implementations for varying internal state size. Execution time is measured on a 16K Model I machine, while the number of T-states is provided by tracing execution on the SDLTRS emulator. Different optimizations were used in the $b = 1600$ implementations to accelerate bit rotation on 8-byte long integers.

5 Conclusions

It is generally understood that the world of retrocomputing, an ecosystem of largely non-professional practices involving old computing technology, serves the goals of collection and preservation, particularly in regards to historic software [11]. On the other hand, retrocomputing may also challenge established notions of authenticity.

The implementation of an innovative cryptographic hash function, very recently selected by NIST for standardization – in a time of controversy and mistrust over the facts revealed by the Snowden documents – could be considered as a simple port to a specific hardware architecture, as are comparable efforts targeting current FPGA and microcontrollers, for instance. Since the announcement of Keccak as the winner of the competition to choose a new hash algorithm, NIST has been working hard to turn Keccak into a standard. (NIST can't just point to the academic paper and materials submitted by the Keccak team and call that a standard. NIST has to write the algorithm up in a standards-compliant format and include it in other NIST cryptographic standards documents.)

In perfect retrocomputing spirit, 35 years after the staff report of the Senate Select Committee on Intelligence, NIST today is widely criticized after changes it purported to make on Keccak turning it into a standard. As presented by NIST's John Kelsey at the Workshop on Cryptographic Hardware and Embedded Systems in August 2013:

> In the name of increased performance (running faster in software and hardware), the security levels of Keccak were drastically reduced. The four versions of the winning Keccak algorithm had security levels of 224 bits, 256 bits, 384 bits, and 512-bits. However NIST intends to standardize only two versions, a 128-bit and a 256-bit version.
>
> Some of the internals of the algorithm had been tweaked by NIST, some in cooperation with the team that submitted Keccak, to improve performance and allow for new types of applications.

Critics were prompt to emphasize that, essentially, NIST had changed Keccak to something very different from what won the 5-year competition. The Keccak team provided the community with a response to these issues last October, arguing that some of the latter remarks are incorrect and misleading. In pretty much the same wording of rebuttal found more than three decades ago in the 1978 report of the Senate Select Committee[1], the design team stated that:

> The current [NIST-reworked standard] proposal is a subset of the (unmodified) Keccak family. The proposal uses exactly the same cryptographic primitive as selected at the end of the SHA-3 competition. As explained in our post, one can generate test vectors for that proposal using our reference code submitted for round 3 (January 2011). This alone proves that there cannot be any internal tweaks.

This perfect timing in the background against which this retrocomputing project takes place indeed helps dramatize the *transformative* aspect of this development of Keccak for the TRS-80 in what differentiates this remix from a simple port.

In the first place, the assemblage remixes fragments from the past with with newer elements, joining components from different historical times in new ways. The use of audio transport from the modern PC, in this instance through the .WAV format introduced by IBM and Microsoft in 1991, to emulate audio cassette storage (and revive BASIC commands such as `CSAVE, CLOAD`) and exchange code and data, back and forth not only between different machines but literally between computing epochs in time, is a case in point.

Moreover developing for the environment, curiously constrained to the modern eye, of an 8-bit processor in a memory-limited microcomputer obsolete architecture may produce new design and optimization ideas which can then be reintegrated into living, on-going contemporary practices [8]. This reintegration may then hopefully contribute to a deeper understanding and a broader development perspective.

Finally the antiquated, but still supportive of modern needs, nature of the architecture may suggest new meaning to retrocomputing-based perfect forward secrecy. In the contemporary concern about tampering and traps in our computing devices which echoes over and again past controversies, should we assume

[1] Established by S. Res. 400, 94th Cong., 2nd Sess., the Select Committee counted 22 members including Senators Goldwater, Stevenson, Biden, Hart, Moynihan, Case and Inouye.

that NSA had the foresight to plant backdoors in a 1977 newly-released microcomputer in view of forward engineering vulnerabilities that would allow it to break not-yet envisioned encryption schemes on rare vintage machines still running thirty years later?

References

1. Birch Bayh and Barry Goldwater. Involvement of NSA in the Development of the Data Encryption Standard. Staff Report of the Senate Select Committee on Intelligence, April 1978. `http://www.intelligence.senate.gov/pdfs/95nsa.pdf`.
2. G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. The Keccak reference, January 2011. `http://keccak.noekeon.org/`.
3. G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. The Keccak SHA-3 submission, January 2011. `http://keccak.noekeon.org/`.
4. G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Permutation-based encryption, authentication and authenticated encryption. Directions in Authenticated Ciphers, July 2012.
5. David Brin. Why Johnny can't code. *Salon.com*, 2006.
6. Pierre Giraud and Alain Pinaud. *La pratique du TRS-80*, volume 2. Editions du P.S.I., 1980.
7. Lance A. Leventhal. *Z80 Assembly Language Programming*. Osborne and Associates, 1979.
8. Nick Montfort, Patsy Baudoin, John Bell, Ian Bogost, Jeremy Douglass, Mark C. Marino, Michael Mateas, Casey Reas, Mark Sample, and Noah Vawter. *10 PRINT CHR$(205.5+RND(1));: GOTO 10*. The MIT Press, 2012.
9. Edwin R. Paay. *Level II ROM Reference Manual*. MICRO-80 Products, 1980.
10. Radio Shack. *Level II BASIC Reference Manual*. Radio Shack, Fort Worth, Texas 76102, USA, 1979.
11. Yuri Takhteyev and Quinn DuPont. Retrocomputing as preservation and remix. *Library Hi Tech*, 31(2):355–370, 2013.
12. Erich Wenger, Thomas Unterluggauer, and Mario Werner. 8/16/32 shades of elliptic curve cryptography on embedded processors. In Goutam Paul and Serge Vaudeney, editors, *Progress in Cryptology - INDOCRYPT 2013*. Springer, 2013. in press.
13. Jr. William Barden. *TRS-80 Assembly-Language Programming*. Radio Shack, 1979.

A BASIC Level II Implementation of Keccak-f[200] (non-optimized)

```
20 DIMA%(25),B%(25),C%(5),D%(5),PB%(25),R%(25),RC%(24)
30 FORI%=1TO25:A%(I%)=0:NEXT
35 ' EXAMPLE INITIAL STATE. SET UP ABSORBING PHASE HERE.
40 A%(1)=128:A%(2)=2:A%(3)=3:A%(4)=64:A%(5)=255
50 GOSUB400:GOSUB150
60 FORN%=1TO18:GOSUB200:GOSUB500:GOSUB600
70 X%=RC%(N%):Y%=A%(1):GOSUB100:A%(1)=Z%
80 PRINT"ROUND";N%:GOSUB150
90 NEXTN%:END
99 ' XOR
100 Z%=X%AND(255-Y%):Z%=Z%OR(Y%AND(255-X%)):RETURN
119 ' RLA ON 8 BITS
120 IFX%>127THENZ%=1ELSEZ%=0
130 Y%=X%+X%:Z%=Z%+(Y%-256*INT(Y%/256))
140 RETURN
149 ' PRINT STATE
150 FORI%=1TO5
160 FORJ%=1TO5:PRINTA%(J%+5*(I%-1));:NEXTJ%:PRINT:NEXTI%:RETURN
199 ' THETA
200 FORI%=1TO5:X%=A%(I%):Y%=A%(I%+5):GOSUB100
210 X%=Z%:Y%=A%(I%+10):GOSUB100
215 X%=Z%:Y%=A%(I%+15):GOSUB100
220 X%=Z%:Y%=A%(I%+20):GOSUB100
230 C%(I%)=Z%:NEXT
240 X%=C%(2):GOSUB120:X%=Z%:Y%=C%(5):GOSUB100:D%(1)=Z%
250 X%=C%(3):GOSUB120:X%=Z%:Y%=C%(1):GOSUB100:D%(2)=Z%
260 X%=C%(4):GOSUB120:X%=Z%:Y%=C%(2):GOSUB100:D%(3)=Z%
270 X%=C%(5):GOSUB120:X%=Z%:Y%=C%(3):GOSUB100:D%(4)=Z%
280 X%=C%(1):GOSUB120:X%=Z%:Y%=C%(4):GOSUB100:D%(5)=Z%
290 FORI%=1TO5:FORJ%=1TO5
300 X%=A%(I%+5*(J%-1))
310 Y%=D%(I%):GOSUB100
320 A%(I%+5*(J%-1))=Z%:NEXTJ%,I%:RETURN
399 ' CONSTANTS
400 FORI%=1TO25:READPB%(I%):NEXT
410 FORI%=1TO25:READR%(I%):NEXT
415 FORI%=1TO24:READRC%(I%):NEXT
420 DATA 1, 11, 21, 6, 16, 17, 2, 12, 22, 7, 8, 18, 3, 13, 23, 24
425 DATA 9, 19, 4, 14, 15, 25, 10, 20, 5
430 DATA 0, 1, 6, 4, 3, 4, 4, 6, 7, 4, 3, 2, 3, 1, 7, 1, 5, 7, 5
435 DATA 0,2, 2, 5, 0, 6
440 DATA 1, 130, 138, 0, 139, 1, 129, 9, 138, 136, 9, 10, 139, 139
```

```
445 DATA 137, 3, 2, 128, 10, 10, 129, 128, 1, 8
450 RETURN
499 ' RHO PI
500 FORI%=1TO25
510 Z%=A%(I%):IFR%(I%)=0THEN530
520 FORJ%=1TOR%(I%):X%=Z%:GOSUB120:NEXTJ%
530 B%(PB%(I%))=Z%:NEXTI%:RETURN
599 ' KHI
600 FORJ%=1TO5
610 X%=B%(1+5*(J%-1)):Y%=B%(3+5*(J%-1))AND(255-B%(2+5*(J%-1)))
615 GOSUB100:A%(1+5*(J%-1))=Z%
620 X%=B%(2+5*(J%-1)):Y%=B%(4+5*(J%-1))AND(255-B%(3+5*(J%-1)))
625 GOSUB100:A%(2+5*(J%-1))=Z%
630 X%=B%(3+5*(J%-1)):Y%=B%(5+5*(J%-1))AND(255-B%(4+5*(J%-1)))
635 GOSUB100:A%(3+5*(J%-1))=Z%
640 X%=B%(4+5*(J%-1)):Y%=B%(1+5*(J%-1))AND(255-B%(5+5*(J%-1)))
645 GOSUB100:A%(4+5*(J%-1))=Z%
650 X%=B%(5+5*(J%-1)):Y%=B%(2+5*(J%-1))AND(255-B%(1+5*(J%-1)))
655 GOSUB100:A%(5+5*(J%-1))=Z%
660 NEXTJ%:RETURN
```

Note the simplistic implementation of `XOR` in the subroutine at line 100 for 1-byte long arguments, and the heavy-handed implementation of left rotation by 1 bit of a byte long argument in the subroutine at line 120. In assembly these operations are directly accomplished using Z80 instructions