

# GIMLI, Lord of the Glittering TRS-80

Jean-Marie Chauvet

MassiveRand

[jmc@massiverand.com](mailto:jmc@massiverand.com)

<http://www.massiverand.com>

62, ave. Pierre Grenier, 92100 Boulogne-Billancourt, France

**Abstract.** Bernstein et al. have proposed a new permutation, GIMLI [1], which aims to provide simple and performant implementations on a wide variety of platforms ranging from the AVR ATmega and ARM-Cortex to the Intel Haswell. In a festive spirit of retrocomputing, this brief paper reports such a simple and (somewhat) performant implementation on the almost Third-Age-of-Middle-earth-dating TRS-80.

**Keywords:** Cryptography, Permutation, Stream Cipher, Sponge functions, Gimli, Z80, TRS-80, Retrocomputing, Crazy Ideas, Remix

## 1 Introduction

In their recent paper introducing GIMLI [1], a 384-bit permutation, Bernstein et al. explain: “What distinguishes Gimli from other permutations is its *cross-platform* performance.” While the platforms referred to there are most certainly contemporary architectures, and indeed the paper presents reference implementations for FPGA, ASIC, 8-bit microcontroller, 32-bit high-end embedded microcontroller, 32-bit smartphone CPU and 64-bit server CPU platforms, the choice of a name of a once famous character of the Third Age of Middle-earth is an invitation to investigate extending the notion of cross-platform to more antique architectures. Short of having a distributed network of Silmarils at hand, we settle for a TRS-80 Model 1 [4] – from 1977, complete with 16K memory and BASIC Level II – and embark on porting the new permutation to the 40-year old Z80 platform in festive retrocomputing spirit [3,2].

## 2 Z80: “Microprocessor for fourth generation computers”

All Z80 registers are implemented using static RAM. The registers include two sets of six general-purpose registers, named B, C, D, and E, that may be used individually as 8-bit registers or in pairs as 16-bit registers. There are also two sets of accumulator and flag registers, respectively A and F, and six special-purpose registers: Interrupt Vector I, Memory Refresh R, Index registers IX and IY, Stack Pointer SP and Program Counter PC. At any given time only one set of registers is active.

All instructions are executed as series of basic operations. Each of these operations can take from three to six clock periods to complete (or they can be lengthened to synchronize the CPU to the speed of external devices). The clock periods are referred to as T (time) cycles and the operations are referred to as M (machine) cycles. We look at the performance of the implementation in terms of number of T-cycles. In the TRS-80 the processor is clocked at 1.77MHz.

The Z80 CPU can execute 158 different instruction types falling into the usual major groups: load and exchange, block transfer and search, arithmetic and logical, rotate and shift, bit manipulation, jump, call and return, I/O, basic CPU control, an ample supply for our purposes. The arithmetic and logical instructions operate on data stored in the accumulator and other general-purpose CPU registers or external memory locations. The results of the operations are placed in the accumulator and the appropriate flags are set according to the result of the operation. The Z80 provides a variety of addressing modes: immediate (8b), immediate extended (16b), modified page zero addressing, relative (in jump instructions), extended (in jump instructions), indexed (an 8b offset from an index register), register indirect (using HL as a 16b pointer).

On a TRS-80 Model 1, although the Z80 can address 64k memory, not all of it is available for users' programs. The Level II ROM, memory mapped I/O, keyboard map, video RAM, fill in the first 0x42E8 bytes so Level II BASIC programs start at 0x42E9 and may expand up to 0x7FFF on a 16K system, the target machine for our port of GIMLI. (Introduced in 1977 the Model 1 Level 1 had only 4K RAM; Level II machines, introduced later, usually came with 16K and could be upgraded to 32K and 48K through the purchase of the "Expansion Interface".)

### 3 Implementation

Our starting point is the SP-box in assembly and the description of its implementation on a modern 8-bit microcontroller in the GIMLI paper [1], and specifically Listing 5.2 therein, showing application the SP-box to three 32-bit registers  $x$ ,  $y$ ,  $z$  using just two temporary registers  $u$  and  $v$ .

Fig. 1. SP-box assembly instructions from Listing 5.2

Algorithm 1	Algorithm 2	Algorithm 3	Algorithm 4
Rotate	Compute x	Compute y	Compute z
1: $x \leftarrow x \lll 24$	1: $v \leftarrow z \ll 1$	1: $v \leftarrow y$	1: $u \leftarrow u \wedge v$
2: $y \leftarrow y \lll 9$	2: $x \leftarrow y \wedge z$	2: $y \leftarrow u \vee z$	2: $u \leftarrow u \ll 3$
3: $u \leftarrow x$	3: $x \leftarrow x \ll 2$	3: $y \leftarrow y \ll 1$	3: $v \leftarrow v \oplus u$
	4: $x \leftarrow x \oplus v$	4: $y \leftarrow y \oplus v$	4: $z \leftarrow z \oplus v$
	5: $x \leftarrow x \oplus u$	5: $y \leftarrow y \oplus u$	

The Z80 provides 7 8-bit registers, two of which (HL) are preferentially used jointly as a 16-bit memory pointer, leaving then 5 registers to perform the actual computation – a long shot from the 32 AVR registers. This obviously does not allow the full 384-bit GIMLI state to stay in the registers but furthermore it does not even accommodate a half-state as in the optimized construction for the AVR architecture in [1] – in fact, a single column of 3 32-bit integers does not even fit in the available registers.

Some drastic simplifications are required in order to minimize the number of loads and stores we are forced to use in the main loop. First, we keep the state in a block of 48 bytes, and work column by column as in the reference C implementation. Each 32-bit word in a column however is processed byte by byte, from low byte 0 to high byte 3. The rotations in Algorithm 1 are done in-place on the 48-byte state, at the cost of one load and store for each byte. In contrast, the compute steps in Algorithms 2, 3, and 4 fully use all registers to operate on each byte of a column word in an unrolled loop of 4 substeps, one for each byte 0 to 3 respectively of words  $x$ ,  $y$ ,  $z$ ,  $u$ , and  $v$ .

**Listing 1.1.** Rotate left 24 bits

```

030B 1A          [ 7] 505      ld a,(de)
030C D5          [11] 506     push de
030D E1          [10] 507     pop hl
030E 23          [ 6] 508     inc hl
030F 0E 03      [ 7] 509     ld c,#0x03
0311 06 00      [ 7] 510     ld b,#0x00
0313 ED B0      [21] 511     ldir
0315 12          [ 7] 512     ld (de),a;

```

In Listing 1.1 we implement left rotation by 24 bits, 3 bytes, by permuting in-place the column word  $B[3..0]$  as  $c = B[0]$ ;  $B[0] = B[1]$ ;  $B[1]=B[2]$ ;  $B[2]=B[3]$ ;  $B[3]=c$ ;, register pairs HL and DE are used as pointers to the consecutive bytes of the word. The left rotation by 9 bits is implemented as a left rotation by 1 byte (see Listing 1.2, where IY points to the word), followed by a left rotation of one byte also in-place.

**Listing 1.2.** Rotate left 1 bit

```

0353 FD 7E 03    [19] 548     ld a,3 (iy)
0356 CB 17       [ 8] 549     rl a
0358 FD CB 00 16 [23] 550     rl 0 (iy)
035C FD CB 01 16 [23] 551     rl 1 (iy)
0360 FD CB 02 16 [23] 552     rl 2 (iy)
0364 FD CB 03 16 [23] 553     rl 3 (iy)

```

Note that all words in the state are stored low-byte first in memory. The first two columns in the listings display the address and its value for the assembly code in the last two columns; the third one shows the number of T-cycles between brackets for each instruction and the fourth column is simply a source file line number.

The only catch in the compute steps is in the correct implementation of the left shifts, respectively by 1 and 2 in Algorithm 2, by 1 in Algorithm 3 and by 3 in Algorithm 4. Since we operate byte by byte in an unrolled loop, we have to keep the carry result of every single shift at each substep for the next one. We allocate a register, if available, to store this carry result after each 1-bit shift. As only 2 registers remain free, the total 3 1-bit shifts in computing  $x$  and  $z$  requires one additional read-write operation in memory for the extra shift.

Listing 1.3. Compute y

```

0453 16 00      [ 7] 716      ld d,#0x00
0455 21r04r00  [10] 717      ld hl,#_y+0
0458 4E         [ 7] 718      ld c,(hl)
0459 21r10r00  [10] 719      ld hl,#_v+0
045C 71         [ 7] 720      ld (hl),c
045D 21r0Cr00  [10] 721      ld hl,#_u+0
0460 46         [ 7] 722      ld b,(hl)
0461 21r08r00  [10] 723      ld hl,#_z+0
0464 7E         [ 7] 724      ld a,(hl)
0465 B0         [ 4] 725      or b
0466 CB 27      [ 8] 726      sla a          ; shift left through carry
0468 CB 12      [ 8] 727      rl d          ; store carry in bit 0 of d
046A A8         [ 4] 728      xor b
046B A9         [ 4] 729      xor c
046C 21r04r00  [10] 730      ld hl,#_y+0
046F 77         [ 7] 731      ld (hl),a
0470 21r05r00  [10] 732      ld hl,#_y+1
0473 4E         [ 7] 733      ld c,(hl)
0474 21r11r00  [10] 734      ld hl,#_v+1
0477 71         [ 7] 735      ld (hl),c
0478 21r0Dr00  [10] 736      ld hl,#_u+1
047B 46         [ 7] 737      ld b,(hl)
047C 21r09r00  [10] 738      ld hl,#_z+1
047F 7E         [ 7] 739      ld a,(hl)
0480 B0         [ 4] 740      or b
0481 CB 1A      [ 8] 741      rr d          ; set carry from bit 0 of d
0483 CB 17      [ 8] 742      rl a          ; shift left through carry
0485 CB 12      [ 8] 743      rl d          ; store carry in bit 0 of d
0487 A8         [ 4] 744      xor b
0488 A9         [ 4] 745      xor c
0489 21r05r00  [10] 746      ld hl,#_y+1
048C 77         [ 7] 747      ld (hl),a
048D 21r06r00  [10] 748      ld hl,#_y+2
0490 4E         [ 7] 749      ld c,(hl)
0491 21r12r00  [10] 750      ld hl,#_v+2
0494 71         [ 7] 751      ld (hl),c
0495 21r0Er00  [10] 752      ld hl,#_u+2
0498 46         [ 7] 753      ld b,(hl)
0499 21r0Ar00  [10] 754      ld hl,#_z+2
049C 7E         [ 7] 755      ld a,(hl)
049D B0         [ 4] 756      or b
049E CB 1A      [ 8] 757      rr d          ; set carry from bit 0 of d
04A0 CB 17      [ 8] 758      rl a          ; shift left through carry
04A2 CB 12      [ 8] 759      rl d          ; store carry in bit 0 of d
04A4 A8         [ 4] 760      xor b
04A5 A9         [ 4] 761      xor c
04A6 21r06r00  [10] 762      ld hl,#_y+2
04A9 77         [ 7] 763      ld (hl),a
04AA 21r07r00  [10] 764      ld hl,#_y+3
04AD 4E         [ 7] 765      ld c,(hl)
04AE 21r13r00  [10] 766      ld hl,#_v+3
04B1 71         [ 7] 767      ld (hl),c
04B2 21r0Fr00  [10] 768      ld hl,#_u+3
04B5 46         [ 7] 769      ld b,(hl)
04B6 21r0Br00  [10] 770      ld hl,#_z+3
04B9 7E         [ 7] 771      ld a,(hl)
04BA B0         [ 4] 772      or b
04BB CB 1A      [ 8] 773      rr d          ; set carry from bit 0 of d
04BD CB 17      [ 8] 774      rl a          ; shift left through carry
04BF CB 12      [ 8] 775      rl d          ; store carry in bit 0 of d
04C1 A8         [ 4] 776      xor b
04C2 A9         [ 4] 777      xor c
04C3 21r07r00  [10] 778      ld hl,#_y+3
04C6 77         [ 7] 779      ld (hl),a

```

We present the computation of word  $y$  in Listing 1.3, for illustrative purposes,  $x$  and  $z$  unrolled loops are similar, if a bit longer. These 64 instructions are executed in 483 t-cycles on the TRS-80.

The resulting SP-box implementation is summarized in Table 2.

We also compare this implementation with TRS-80-hosted implementations of the Keccak permutation and the scalar multiplication on Curve25519 developed earlier in [3,2].

Table 3 shows that, in spite of the limited number of registers available on the platform, this implementation plays well along the compactness and high-throughput features of Gimli described in [1], when compared to other implementations on the TRS-80.

**Fig. 2.** T-cycles and instructions counts

	<b>T-cycles Instructions</b>	
Rotate	397	44
Compute x	692	89
Compute y	483	64
Compute z	659	79
Total	2,231	276

**Fig. 3.** Compared performance on the TRS-80

	<b>Keccak (b=1600)</b>	<b>Scalar Mul. c25519</b>	<b>Gimli (this paper)</b>
Code Size	8,742	7,880	1875
Data Size	1,930	733	86
T-Cycles	$42.8 \cdot 10^6$	$2,677 \cdot 10^6$	$\approx 74,000$

## 4 Conclusions

High security without sacrificing performance is also possible on the TRS-80 with Gimli, even though the antique machine predates the new permutation by 40 years – it was introduced on August 3, 1977. Simple hand-crafting of the inner SP-box assembly code produces an implementation which is about 30 % faster in T-cycles count than the reference code naively compiled e.g. with SDCC [5].

The screen shot 4 shows the result of running Gimli on a Level II, 16K machine, starting from an all-zero initial state.

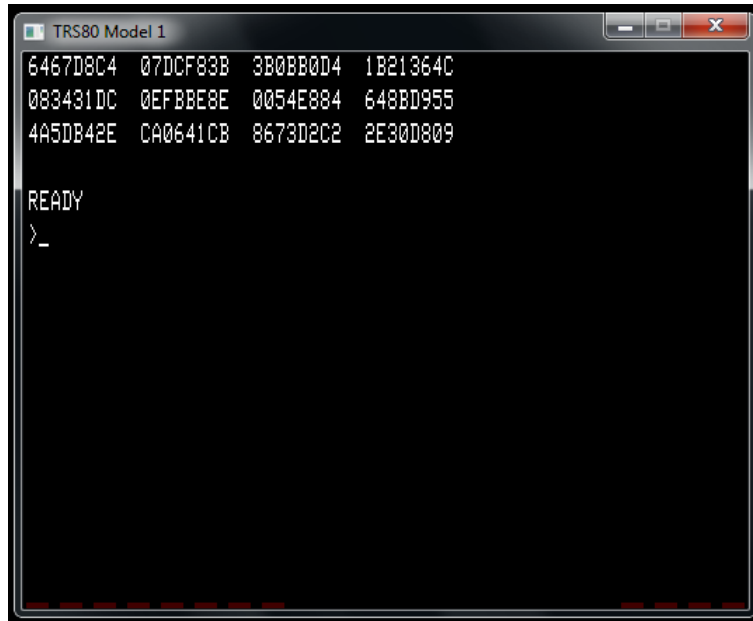
Even on this older system, Gimli, as advocated in the original paper, can easily be used to build high-security block ciphers, tweakable block ciphers, message-authentication code, hash functions etc. using BASIC Level II (within the memory limitations!). Indeed the PEEK and POKE BASIC instructions may be used to read and write the state byte by byte. Then, once the address of the main machine code entry point for Gimli is stored (using, e.g., POKE), little-endian, in addresses 16526 and 16527, the permutation is called by issuing the BASIC instruction `USR(0)`, with the argument 0 ignored.

We place all software implementations described in this paper into the public domain, they are available at <https://github.com/CRTandKDU/TRS80> .

## References

1. Daniel J. Bernstein, Stefan Kölbl, Stefan Lucks, Pedro Maat Costa Massolino, Florian Mendel, Kashif Nawaz, Tobias Schneider, Peter Schwabe, François-Xavier Standaert, Yosuke Todo, and Benoît Viguier. Gimli: a cross-platform permutation. Cryptology ePrint Archive, Report 2017/630, 2017. <http://eprint.iacr.org/2017/630>.

**Fig. 4.** Gimli running on a TRS-80 Model 1 Level II. Screen shot from `sdltrs` emulator on Windows showing the 48 consecutive 32-bit words of the final state.



```
TRS80 Model 1
6467D8C4 07DCF83B 3B0BB0D4 1B21364C
083431DC 0EFBBE8E 0054E884 648BD955
4A5DB42E CA0641CB 8673D2C2 2E30D809

READY
>_
```

2. Jean-Marie Chauvet. TRS-80 With A Grain Of Salt. Cryptology ePrint Archive, Report 2013/546, 2013. <http://eprint.iacr.org/2013/546>.
3. Jean-Marie Chauvet. TRS-80 With A Keccak Sponge Cake. Cryptology ePrint Archive, Report 2013/736, 2013. <http://eprint.iacr.org/2013/736>.
4. David Lien. *TRS-80 Micro Computer System*. Radio Shack, 1978.
5. SDCC Team. SDCC - Small Device C Compiler. <http://sdcc.sourceforge.net/index.php>.

Fig. 5. Teaching an old TRS to run new (crypto)tricks!

