# 22:10 A Tourist's Guide to Эльбрус

*by evm*

In the tradition of the many high quality tourist guides that have appeared in this fine publication, let's take a magical tour around Russia's modern computer architecture, the Elbrus 2000.[45] [46] [47]

## At A Glance

**Common Models**
Elbrus-1S+, Elbrus-4S, Elbrus-8S, Elbrus-8SV, Elbrus-16S

**Architecture**
Von Neumann
Very Long Instruction Word
Register Windowing (32-bit Base Registers)

**Registers**
`g0`–`g31`: Global Registers
`r0`–`r17`: General Purpose (Windowed)
`b0`–`b7`: Overlay Register within Window
Pred0-Pred31: Boolean Predicate Registers

**Address Space**
64-bit Virtual Addressing
Unknown Physical Memory Map

## Background & History

Elbrus is a Russian CPU architecture that has been around in some form for over 40 years. It started at Lebedev Institute of Precision Mechanics and Computer Engineering. It was the first superscalar, out-of-order execution processor developed in the Soviet Union (when the Elbrus 1 debuted in 1979). The architecture was extended to be a very long instruction word (VLIW) architecture with Elbrus 3 in 1990. Once fully integrated as a microprocessor architecture in 2001 (previous versions had used many discrete chips), the architecture became known as Elbrus 2000, or E2K for short. Elbrus is designed in Russia but currently manufactured by TSMC in Taiwan because of a lack of Russian production facilities capable of producing chips at advanced technology nodes.[48]

In the early '90s, the Lebedev Institute spun off a joint stock company called the Moscow Center of SPARC Technologies (now shortened to just MCST). MCST currently produces new Elbrus chips and Elbrus-based PCs, laptops, and servers. Elbrus-8S and 8SV are the current top-of-the-line processor models (eight core versions for servers and desktops), and a lower-cost 1S+ (single core) is available as well. Note the transliteration from Cyrillic where the model names appear as Эльбрус-8С, Эльбрус-8СВ, and Эльбрус-1С+, respectively. Anecdotally, the 8S CPUs are about three times slower than a comparable Intel CPU,[49] but the draw of Elbrus is that it's a fully domestically designed Russian processor. The Russian military has reportedly ordered thousands of ruggedized laptops based on the Elbrus-1S+,[50] although there is no indication that the order was ever delivered.

There is currently very little public documentation on Elbrus because MCST controls most documentation under nondisclosure agreements. This means we don't have full processor documentation like we normally would for a commercial CPU.

[45] Travis Goodspeed and Ryan Speers, "A Tourist's Phrasebook for Reversing Embedded ARM in the Dialect of the Cortex M Series," PoC||GTFO 11:6
[46] Ryan Speers and Travis Goodspeed, "A Tourist's Phrasebook for Reversing MSP430," PoC||GTFO 11:08
[47] Chris Hewitt, "A Tourist's Guide to Altera NIOS," PoC||GTFO 21:7
[48] Ian Cutress, "Russia's Elbrus 8CB Microarchitecture: 8-core VLIW on TSMC 28nm," AnandTech, June 1, 2020.
[49] Anton Shilov, "Russian-Made Elbrus CPUs Fail Trials, 'A Completely Unacceptable Platform'," Tom's Hardware, December 24, 2021.
[50] Inna Sidorkova, "Цены на военные ноутбуки достигли Эльбруса." July 9, 2018, RBC.

We used three sources of information for this article: (1) a Russian guide to Elbrus programming and optimization published by MCST,[51] (2) source code published by the OpenE2K group (a hobbyist group seemingly unrelated to MCST), and (3) leaked Linux kernel source code.

MCST is currently on the US sanctions list but thanks to the Reverend and friends we had access to an Elbrus-1S+ machine and used it to play around with some code examples. Our Elbrus was running a version of Linux made by MCST, but other Russian Linux distros are also available for Elbrus (e.g., Astra Linux). The Elbrus machine has a compiler called lcc, which is the MCST compiler based on gcc. It produces standard Linux ELF binary files. The options for disassembly at the moment are limited to ldis, which is part of lcc, and objdump, which is part of the binutils package put out by the OpenE2K group. ldis produces cleaner output, including resolution of symbol names, while objdump has a debug flag in the build that will prefix the output with the decoded instructions in hex. Anecdotally, ldis seems to miss some things (e.g., not disassemble all functions), although that could be due to operator error.

In order to explore the Elbrus instruction set we updated rix's Smashing C++ VPTRs from Phrack 56:8. That is a whole story for a another day, but you will find my code examples and the corresponding Elbrus disassembly attached to this PDF.[52]

## Basics of Instruction Set Decoding

The first thing we needed to figure out was how the instruction format works since the official documentation left this topic out entirely. Fortunately we found that the OpenE2K binutils release has a preprocessor flag `ENABLE_E2K_ENCODINGS`, which causes objdump to print out the instruction bytes and their groupings.[53] A version of objdump with this flag was what we used to produce the disassembly for most of this article.

In Elbrus documentation, the VLIW is called a "wide command" (широкой командой). A wide command contains multiple instructions, each of which is targeted at individual execution units in the CPU pipeline. The documentation variously uses the terms "commands" (команд), "instructions" (инструкций), and "operations" (операций) for the component instructions within the instruction word.

The OpenE2K objdump code refers to the way these component instructions are encoded as "syllables." A nice feature of Elbrus is that the instruction encoding is fairly simple when compared against modern DSP architectures we've experienced. Instruction counting is an exploitation task that can be pretty complicated on some architectures, but not Elbrus. It's fairly simple to determine the length of an instruction from the initial "HS" syllable (shown on page 49).

The HS syllable determines the presence of the other instruction syllables, which appear in a particular order. The order is: SS; ALU; CS0; ALES half syllables 2 and 5; CS1; ALES half syllables 0, 1, 3, and 4; AAS half syllables; a gap check; CDS; PLS; and finally LTS (literals). Literal syllables (i.e., immediate values) occur at the end of the syllables. The OpenE2K objdump code looks for all of the syllable presence flags above, reads them in order (minding the possible gap), and then compares the number of syllables read against the size field in HS. Any extra syllables are read as literals. For syllables that contain "half syllables" (i.e., 16-bit values), the order of the syllables is flipped as they appear sequentially in memory.

```
Byte order          | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
                    ---------------------------------
Half syllable order |   1   |   0   |   3   |   2   |
```

This makes more sense if you think about the bytes being read in as 4-byte little-endian values.

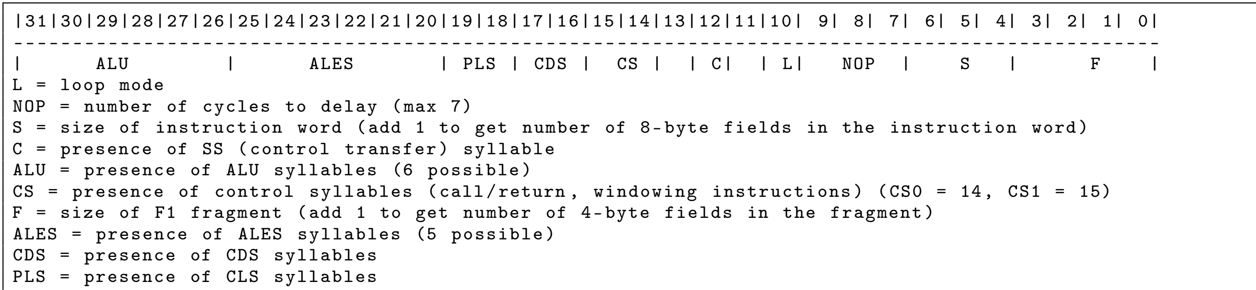
```
Word order          |       0       |       1       |
                    ---------------------------------
Byte order          | 3 | 2 | 1 | 0 | 7 | 6 | 5 | 4 |
                    ---------------------------------
Half syllable order |   0   |   1   |   2   |   3   |
```

## Register Set

Elbrus's basic registers consist of 18 general-purpose registers (`r0`–`r17`), 32 global registers (`g0`–`g31`), and a sliding set of windowed registers (`b0`–`b7`). More will be explained about the register windowing in the next section. Registers are prefixed with an access width, similar to x86.

For example, `sr0` is single (32-bit) `r0`, and `dr0` is double (64-bit) `r0`, which is also the default. When the registers are used with floating point values,

[51] `unzip pocorgtfo22.pdf elbrusprog.pdf`
Murad Neumann-zadeh and Sergei Korolev, "Руководство по эффективному программированию на платформе «Эльбрус»"
[52] `unzip pocorgtfo22.pdf vptrs.zip`
[53] `git clone https://git.mentality.rip/OpenE2K/binutils-gdb.git`

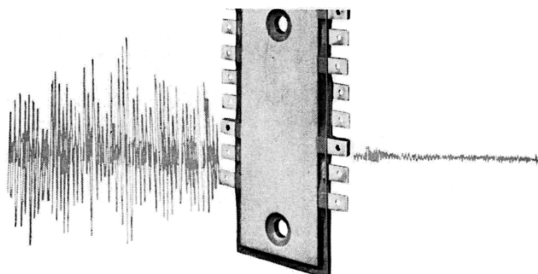


```
|31|30|29|28|27|26|25|24|23|22|21|20|19|18|17|16|15|14|13|12|11|10| 9| 8| 7| 6| 5| 4| 3| 2| 1| 0|
-----------------------------------------------------------------------------------------------
|      ALU        |       ALES       | PLS | CDS |  CS |  | C|  | L|   NOP  |    S   |      F    |
L = loop mode
NOP = number of cycles to delay (max 7)
S = size of instruction word (add 1 to get number of 8-byte fields in the instruction word)
C = presence of SS (control transfer) syllable
ALU = presence of ALU syllables (6 possible)
CS = presence of control syllables (call/return, windowing instructions) (CS0 = 14, CS1 = 15)
F = size of F1 fragment (add 1 to get number of 4-byte fields in the fragment)
ALES = presence of ALES syllables (5 possible)
CDS = presence of CDS syllables
PLS = presence of CLS syllables
```

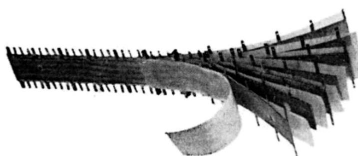
Encoding of the initial HS syllable, which determines the presence of other syllables, in an Elbrus-wide instruction word. It is unclear what the ALES, CDS, and PLS syllables are used for as we did not generate any of those instructions in the example code.

there is an `xr0`, which is an 80-bit version, presumably using the long double format from x86. According to the documentation, two double registers can be accessed as a quad register – for example `qr[i]` where `[i]` is even – but `gdb` on our box doesn't seem to be aware of this notation.

CAUTION: Elbrus has a word size of 32-bits for both registers and memory accesses, so the notion of single/double/quad on Elbrus is double what you might be used to on 64-bit x86, where the length of a word dates back to its early ancestors.

Laminated Bus Bars For Noise Reduction



Flat bus conductors laminated with Eldre's thin, rugged insulation will reduce electrical noises which cause havoc in high speed, solid state equipment. Lower the inductance and control the capacitance of your vital power distribution lines. Ground shields are interleaved with the voltage-carrying conductors so that effective shielding can be adequately provided. The terminations of each conductor, as shown, are for soldering but other types can be incorporated into the bus design. This compact and completely molded bus can replace a bulky harness and repetitive wiring.
Increase the reliability of your circuit with a bus system and obtain efficiency.

ELDRE COMPONENTS INC. • 1239 UNIVERSITY AVENUE • ROCHESTER, N.Y. 14607


## Basic Arithmetic, and Memory Operations in Elbrus

Here we show the various ALU register operations:

| Integer Arithmetic Instructions | |
|---|---|
| add | Addition |
| sub | Subtraction |
| rsub | Reverse subtraction |
| umul/smul | (Un)signed integer multiplaction |
| udiv/sdiv | (Un)signed integer division |
| umod/smod | (Un)signed modulo |
| sxt | Sign Extend |
| Bitwise Operations | |
| and/andn | Boolean and/nand |
| or/orn | Boolean or/nor |
| xor/xorn | Boolean xor/xnor |
| shl/shr | Shift left/right |
| scl/scr | Shift cyclic |
| sar | Shift right arithmetic (signed) |
| insf/getf | Set/get bitfield |
| Floating Point Arithmetic Instructions | |
| fadd | Floating point addition |
| fsub | Floating point subtraction |
| frsub | Floating point rev. subtraction |
| fmax/fmin | Floating point maximum/minimum |
| fmul/fscale | FP mult. / mult. by power of 2 |
| fdiv/frcp | Floating point division/reciprocal |
| fsqrt | Floating point square root |

A basic ALU instruction looks like this:

```
ALS0  1181d48d  addd,0 %dr1, _f16s,_lts0hi 0xfff0, %dr13
```

This translates to "add double precision, using channel 0, the 64-bit register %dr1 to the signed 16-bit value 0xfff0, and place the result in dr13." There are six ALU channels, so you can do up to six ALU instructions in one wide instruction. There is no simple register "move," so the compiler tends to use a zero-add as a "move" instruction. The full list of

ALU register operations is shown in the table above. Notice that this is a fairly small number of operations. Outside of the VLIW construct, the Elbrus instruction set feels pretty RISC-like.

Memory operations are also pretty simple. Operations are load and store with a variety of width specifiers. Addresses can be a register plus an immediate offset, or the sum of two registers. Here is an example of a basic load operation:

```
ALS0  678dc08c  ldd,0 %dr13, 0x0, %dr12
```

ldis renders this (a bit more clearly) as:

```
    ldd,0 [ %dr13 + 0x0 ], %dr12
```

This translates as "load double word (64-bits) from memory, using channel 0, from the address `dr13` + `0`, and store in register `dr12`." There are also array memory load/store operations (`ldaa`/`staa`) that work similarly. As far as we can tell from the documentation, the array mode doesn't add any special addressing. It's still the sum of two registers or a register plus a constant; the main advantage is that there's a built-in post-increment operation.

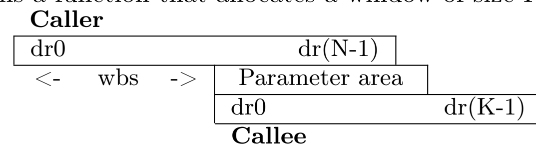
## Register Windowing

Probably the simplest way to understand register windowing is that it functions similarly to local variables within a stack frame in a memory stack. In processors without windowing (which is nearly all processor families with some notable exceptions, like SPARC and Itanium), we are used to code transferring registers around between function calls, meaning that some registers need to be saved on the memory stack or transferred to nonclobbered registers (those guaranteed by the application-binary interface to not get modified by the called function) prior to a function call.

A function of reasonable complexity will save registers it's not supposed to clobber to the memory stack so that they are available for calculations and then restore the previous values from the stack at the end of the function. Register windowing aims to reduce some of this register bucket-brigading overhead by making the register set function more like a memory stack. In the same way that a function allocates a stack frame for itself, a function allocates a window of registers.

On Elbrus, this is accomplished in a function prologue with the `setwd` instruction. After `setwd` executes, the "register" `r0` is really a reference to the first item in the register window. Now the function can use `r0` to `r<N>` without having to save any registers from the calling function. How about parameter passing in registers? Just like architectures with stack-passed parameters, we need a calling function and the called function to share an overlapping area.

This is done with a `wbs` parameter in the `call` instruction. `wbs` indicates the start of shared function parameters within the current window. After a call, `r0` in the called function now refers to the base of the shared parameter area. This is illustrated here, where a caller function has a window of size N and calls a function that allocates a window of size K:




Elbrus also offers a sliding or mobile base register (`b`), which a function can use within its own function window. The base register is just an overlay on the existing register window; it points to a given register within the window. Accesses to registers with the base register use an array notation—for example, `db[0]` means "access first double register (64-bits) at the base pointer." The instruction `setbn` is used to set this pointer.

The operand `rbs` (the offset to set b from the base of the window) is also specified in quadwords. In practice, it looks like lcc uses the base pointer to point to the parameter area, so `db[0]`, `db[1]`, `db[2]`, etc. are parameter 0, parameter 1, parameter 2, etc. for functions that are about to be called.

Since functions return values in `dr0`, this also means that `db[0]` holds the return value from the calling function's point of view.

## Calls and Branches in Elbrus

Calls and branches are somewhat unique in Elbrus, they occur in two phases instead of in a single instruction the way it works on most architectures. Elbrus uses the `disp` instruction to set up any kind of control transfer instruction. This sets the `ctptr1` register to the target address. The `call` instruction executes the control transfer. This allows the pipeline to get a little bit of advance warning for the call, allowing it to set up state for the target function, which can be undone or ignored if the call doesn't execute. The documentation refers to the `ipd` portion as specifying the "swap depth," but it is unclear what this means.

Return instructions happen similarly with a `return` instruction first that sets up the return and the `ct` instruction to execute control transfer. (This is also used for branches, as we'll discuss in the next section.) Notice that the function never seems to do anything with the return address. This is because Elbrus has a completely separate call chain stack, called the Procedure Chain Stack (PCS). Architecturally this is referenced via the Procedure Chain Stack Pointer (PCSP) register. The PCSP is not accessible from user mode; rather, it is set up by the kernel similarly to how user stack memory gets set up on a per-process basis.

### The Procedure Chain Stack (PCS)

The PCSP is pretty simple—it's a 128-bit register with 64-bit "lo" and "hi" parts. The "lo" part contains the base address, and the "hi" part contains an index to the current frame.[54] It is unclear at this point what the "rw" field is actually used for.

```
(gdb) info registers pcsp_lo
pcsp_lo     0x1800c2e00002b000 1729596524238909440
    base    0xc2e00002b000 214267328638976
    rw      0x3 3

(gdb) info registers pcsp_hi
pcsp_hi     0x200000000060 35184372088928
    ind     0x60 96
    size    0x2000 8192
```

The Linux kernel source code shows the format of the stack frames, in the form of the `e2k_mem_crstack` struct. Each frame is 32 bytes and consists of four saved 64-bit register values, the "lo" and "hi" parts for `cr0` and `cr1`, respectively. Again we are left without documentation on what exactly the `cr0` and `cr1` registers do, but they must be involved in control transfers. The Linux code shows that `cr0` "hi" contains the return address, and `cr1` contains a bunch of fields pertaining to the current procedure's register window.

Here is the definition of `e2k_mem_crstack` (PCSP frame structure) in the E2K Linux kernel (`arch/e2k/kernel/e2k_syswork.c`):

```
typedef struct e2k_mem_crstack {
  e2k_cr0_lo_t   cr0_lo;  //pf?
  e2k_cr0_hi_t   cr0_hi;  //return address
  e2k_cr1_lo_t   cr1_lo;  //mess of fields - includes
                          // interrupt enable flags
  e2k_cr1_hi_t   cr1_hi;  //more fields - includes register
                          // window and stuff
} e2k_mem_crs_t;
```

[54] Current frame address = base + index.

So the return `%ctpr3` instruction is essentially saying "pop the current frame off the PCS into `cr0` and `cr1` and stick the return address in `ctpr3`."

### Branching

A similar construction is used for basic branches. Rather than flags or conditions registers like in x86 or ARM, VLIW processors often have a full set of condition registers called "predicate" registers. These allow the compiler to set up a sequence where multiple comparisons can happen in advance of a branch, and then a branch can be based on multiple predicates, or a sequence of branches can occur using the different predicate registers.

Here's a common design pattern seen in Elbrus branches. The following is essentially implementing `if (condition) { function(); }` in C.

```
disp %ctpr1, 0x10d48
cmpedb,0 %dr0, 0x0, %pred0
ct %ctpr1 ? %pred0
```

First the `disp` instruction indicates to the pipeline the control transfer target, the function address. Then in the `cmpedb` instruction `dr0` is compared to `0` and the result placed in `%pred0` (true or false). Finally if `%pred0` is true then the ct instruction causes a control transfer, otherwise we fall through to the next instruction.

## Conclusion

Elbrus processors are pretty capable and make decent Linux machines. While the Elbrus CPU may be under powered compared with similar Intel or ARM server processors, given the Russian geopolitical situation, these guys are going to stick around for a while. Elbrus's VLIW architecture and register windowing will pose additional challenges for exploit writers. Fortunately, the Elbrus component instructions are very RISC-like, despite the wide command format.

In this article, we've explored the basics of the instruction set and the PCS using publicly available documentation. There's a lot more to learn, however. We'll need some full documentation to start plumbing the depths of things like virtual memory, interrupt and exception handling, and the boot process.