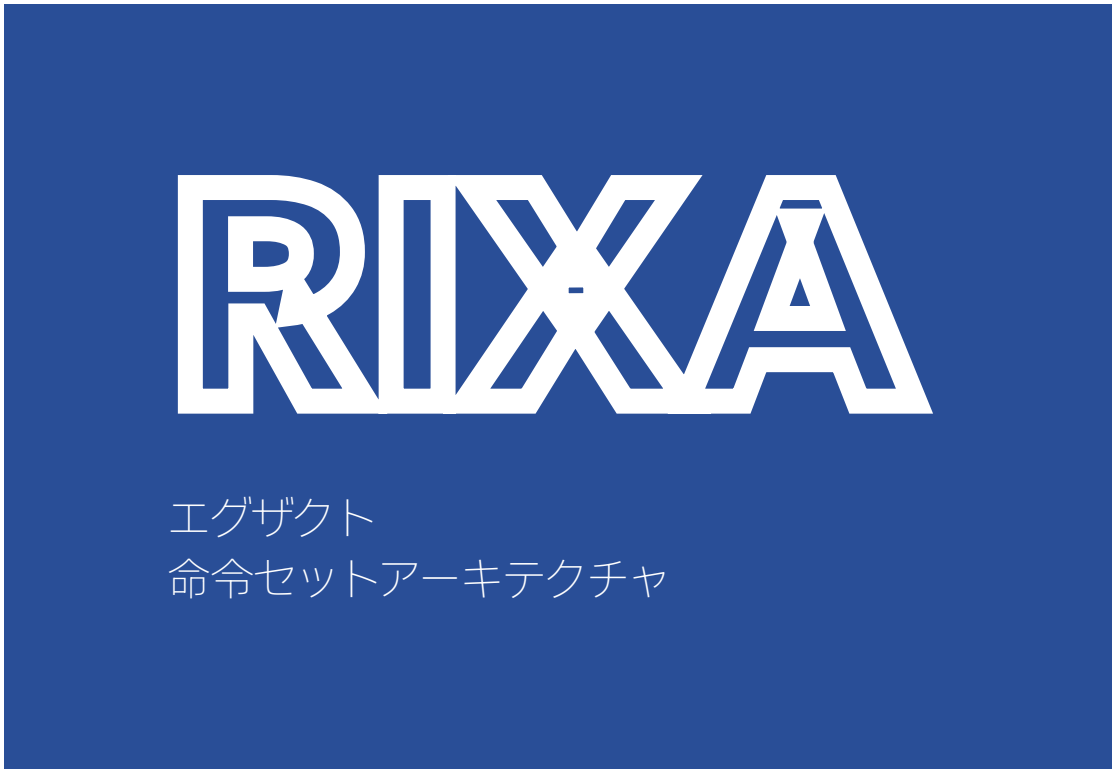
Inventions No. 5
sci-sci.org/rixa
DOI: 10.5281/zenodo.17284766

SciSci Research
サイサイ・リサーチ

Future Computing
コンフュー



RIXA (リクサ)

RISC Instruction Set Architecture for Exact Computing

James Douglas Boyd

SciSci Research, Inc. + Future Computing

SciSci Research, Inc.
Boulder, Colorado, United States
www.sci–sci.org

Citation Format:

Boyd, J.D. (2025). RIXA (リクサ): RISC Instruction Set Architecture for Exact Computing. SciSci Inventions, 1(5). DOI: 10.5281/zenodo.17284766

Contents

1 The RIXA Instruction Set Architecture

The Hensel Instruction Set Architecture (ISA) is a Reduced Instruction Set Computer (RISC) ISA. We'll call it the RISC ISA for EXact Arithmetic, or RIXA. In accordance with RISC philosophy, RIXA is designed for the Hensel to execute many simple instructions, preferring instruction quantity to algorithmic complexity. Thus, complexity is outsourced to the compiler, which issues lists of simple instructions for the Hensel to execute. We'll write the assembly language RIXA commands as ARIXA and machine code instructions as MRIXA.

Under RIXA, FC-3-2025 operands are of a fixed length, with the length depending on the CPU. In the case of, for instance, the Virtual Hensel I, whose processor register cluster has 2^5 registers, and whose addresses can match against $\chi^\mathfrak{d}$-IDs of length 5, the FC-3-2025 encodings will be of length 20 (since there are four blocks, each loaded to its own register.) In general, for a Hensel CPU of cluster depth $\mathcal{L} - 2 = 2^k$, the RIXA length for FC-3-2025 encodings will be $4 \times k$.

As discussed in the report on 2-adic arithmetic units (2AALUs), each arithmetic operation terminates with the computation of an FC-2-2025 code, which, as discussed in the Virtual Hensel and load-store reports, guides the load-store unit (LSU) in loading the output to the address-matching processor register. One need not issue a command for storing an operand in a particular register; this is done automatically. One the other hand, after loads are executed, their FC-2-2025 instructions are placed in a queue, and retrieval of a loaded operand requires specification of the queue position.

The following is but an introductory tutorial of use of elementary ARIXA commands and MRIXA instructions, as shown via examples.

1.1 PFC-3-2025 and FC-3-2025 Encoding Formats

Recall that an FC-3-2025 encoding consists of four blocks: N_{FC}, R_{FC}, L_{FC}, and T_{FC}. The last, T_{FC}, gives the 2-adic expansion entries to the right of the "decimal". L_{FC} gives non-repeating the 2-adic expansion entries to the left. R_{FC} gives repeating 2-adic expansion entries to the left. The N_{FC} block encodes the length of the repeating sequence in R_{FC}. The first three, taken together, give a pre-FC-3-2025 encoding (i.e., PFC-3-2025 format encoding). The Hensel CPU performs arithmetic on PFC-3-2025-encoded operands before converting them to FC-3-2025 format for register loading. Conversion to FC-3-2025 also involves some efficient coding tricks: N_{FC}, R_{FC}, and L_{FC} are given within their entries in reverse order. Thus, in the case of `LOAD` commands, the Hensel uses the FC-3-2025 encoding, and for arithmetic operations, the Hensel uses the PFC-3-2025 encoding. One always begins with PFC-3-2025, however: FC-3-2025 conversion is performed automatically before loads, as discussed in the 2AALU report.

1.2 `LOAD`: ARIXA and MRIXA

Suppose we want to compute $2 - \frac{1}{3}$ to obtain $\frac{5}{3}$. With ARIXA, loads are programmed as follows

```
LOAD   -1/3
```

One does not specify the register, as there is a unique collection of registers to which, under χ-$\mathfrak{A}$ matching, the $\chi^\mathfrak{d}$-IDs corresponding to the N_{FC}, R_{FC}, L_{FC}, and T_{FC} blocks of FC $\left(-\frac{1}{3}\right)$ are loaded, as determined by FC-2-2025 instructions. The `LOAD` command automatically triggers the loaders in the LSU circuit to load the four FC-3-2025 blocks to their respective registers in the processor cluster. In this case, the PFC-3-2025 encoding of $-\frac{1}{3}$ is:

```
00001 00001 00000
```

The MRIXA opcode for LOAD is 10000. So, the overall MRIXA instruction for loading $-\frac{1}{3}$ is

10000 00001 00001 00000

Prior to loading, the PFC-3-2025 encoding will then be converted to the following FC-3-2025 form for $-\frac{1}{3}$:

00010 10000 10000 00000

1.3 ADD, ADDC, and REG

Now, let's add 2. One need only give the opcode and the two operands, giving first the operand already loaded to the processor cluster:

ADDC REG1 2

(ADDC is a "cluster add", or an addition operation using an operand loaded to the processor cluster.) The MRIXA opcode for ADD is 01000. Thus, the MRIXA command for PFC $\left(-\frac{1}{3}\right) +$ PFC(2) is

01000
00001 00001 00000
00000 00010 00000

However, this looks ugly. What RIXA does instead is add 2 to an operand already loaded. (In this case, $-\frac{1}{3}$ is already loaded). The MRIXA opcode for ADDC with REG1 is 01100. Thus, the MIXRA instruction is:

01100 00000 00010 00000

The 2AALU will compute the following result (still in PFC-3-2025 form),

00001 00011 00000

automatically convert it to FC-3-2025 format,

01000 10000 11000 00000

and also automatically compute the FC-2-2025 instruction for loading

00000 00000 01000 00000

Intuitively, in the 2-adic case, adding 2 should require flipping only one bit.

1.4 RELOAD: ARIXA and MRIXA

Next, let's add 2 to $-\frac{1}{3}$, and then add 2 again. The procedure is as follows. We load $-\frac{1}{3}$ and cluster add 2 to the loaded $-\frac{1}{3}$. Then, we reload the output of that computation to REG1. Finally, we cluster add 2 to the new REG1-loaded operand. In ARIXA, it is written as follows:

LOAD $-1/3$
RELOAD ADDC REG1 2
ADDC REG1 2

The MRIXA command for RELOAD ADDC REG1 is 11100. Thus, in MRIXA, the instructions are written as follows:

10000 00001 00001 00000
– – – – – – – – – – – –
11100 00000 00010 00000
– – – – – – – – – – – –
01100 00000 00010 00000

In the case of multiple instructions, the compiler sends them to an instruction memory unit, which are then sent via a parallel instruction bus to the 2AALU one at a time. (In this respect, the Hensel CPU architecture resembles the Harvard architecture.)

Each time a LOAD or RELOAD is executed, the FC-2-2025 instruction for the (re)load is stored in a queue in the instruction memory unit. Thus, REG1 refers to the FC-2-2025 instruction first in the queue, and so on.

1.5 Multiplication: ARIXA and MRIXA

Multiplication and division operations, when given to the compiler, are converted to ARIXA as lists of RELOAD ADDC commands. For instance, the ARIXA command for $-\frac{1}{3} \times 4$ is:

LOAD $-1/3$
RELOAD ADDC REG1 $-1/3$
RELOAD ADDC REG1 $-1/3$
ADDC REG1 $-1/3$

In MRIXA, the list of instructions is given as follows:

```
10000 00001 00001 00000
- - - - - - - - - - - --
11100 00001 00001 00000
- - - - - - - - - - - --
11100 00001 00001 00000
- - - - - - - - - - - --
01100 00001 00001 00000
```

Just as, in the case of addition, subtraction is treated merely as a matter of adding a negative number, so too is division simply treated as multiplication by a number with a non-1 denominator. $-\frac{1}{3} \times 4$ is the same as $\frac{4}{-3}$. Thus, there is no `SUBTRACT`, `MULTIPLY`, or `DIVIDE` command in ARIXA; when commands for such operations are given to the compiler, they are converted into a list of instructions involving only addition. Next, let's consider the case where the numerators and denominators are more complicated.

1.6 Numerators and Denominators: ARIXA and MRIXA

Next, let's consider a case in which the numerators and denominators of both operands are nontrivial, such as $\frac{3}{15} \times \frac{7}{21}$. In this case, the Hensel CPU performs multiplication on the numerator and denominator separately and sends the result of each to memory, and then retrieves from memory the PFC-3-2025 encoding of the operand with the output numerator and denominator. Thus, in keeping with the RISC philosophy, the entire computation is still no more than a series of addition operations.

The command for sending the numerator result to memory is `MEMN`, and, for the denominator, `MEMD`. The command `LOAD MEM` loads the operand in memory with the `MEMN` numerator and `MEMD` denominator. In ARIXA, the list of commands is given as follows:

```
                LOAD 3
RELOAD ADDC REG1 3
RELOAD ADDC REG1 3
RELOAD ADDC REG1 3
RELOAD ADDC REG1 3
RELOAD ADDC REG1 3
RELOAD ADDC REG1 3
          MEMN REG1
             LOAD 15
RELOAD ADDC REG2 15
RELOAD ADDC REG2 15
RELOAD ADDC REG2 15
RELOAD ADDC REG2 15
RELOAD ADDC REG2 15
RELOAD ADDC REG2 15
RELOAD ADDC REG2 15
RELOAD ADDC REG2 15
RELOAD ADDC REG2 15
RELOAD ADDC REG2 15
RELOAD ADDC REG2 15
RELOAD ADDC REG2 15
RELOAD ADDC REG2 15
RELOAD ADDC REG2 15
RELOAD ADDC REG2 15
RELOAD ADDC REG2 15
RELOAD ADDC REG2 15
RELOAD ADDC REG2 15
          MEMD REG2
          LOAD MEM
```

The MRIXA command for `MEMN` is `00100`. The command for `MEMD` is `00010`. In both cases, one need only include these commands and specify the register. The `LOAD MEM` command is `10001`. Thus, the MRIXA commands are as follows:

```
10000 00000 00011 00000
_ _ _ _ _ _ _ _ _ _ _ _
11100 00000 00011 00000
_ _ _ _ _ _ _ _ _ _ _ _
11100 00000 00011 00000
_ _ _ _ _ _ _ _ _ _ _ _
11100 00000 00011 00000
_ _ _ _ _ _ _ _ _ _ _ _
11100 00000 00011 00000
_ _ _ _ _ _ _ _ _ _ _ _
11100 00000 00011 00000
_ _ _ _ _ _ _ _ _ _ _ _
11100 00000 00011 00000
_ _ _ _ _ _ _ _ _ _ _ _
00100 00001 00000 00000
_ _ _ _ _ _ _ _ _ _ _ _
10000 00000 01111 00000
_ _ _ _ _ _ _ _ _ _ _ _
11010 00000 01111 00000
_ _ _ _ _ _ _ _ _ _ _ _
11010 00000 01111 00000
_ _ _ _ _ _ _ _ _ _ _ _
11010 00000 01111 00000
_ _ _ _ _ _ _ _ _ _ _ _
11010 00000 01111 00000
_ _ _ _ _ _ _ _ _ _ _ _
11010 00000 01111 00000
_ _ _ _ _ _ _ _ _ _ _ _
11010 00000 01111 00000
_ _ _ _ _ _ _ _ _ _ _ _
11010 00000 01111 00000
_ _ _ _ _ _ _ _ _ _ _ _
11010 00000 01111 00000
_ _ _ _ _ _ _ _ _ _ _ _
11010 00000 01111 00000
_ _ _ _ _ _ _ _ _ _ _ _
11010 00000 01111 00000
_ _ _ _ _ _ _ _ _ _ _ _
11010 00000 01111 00000
_ _ _ _ _ _ _ _ _ _ _ _
11010 00000 01111 00000
_ _ _ _ _ _ _ _ _ _ _ _
11010 00000 01111 00000
_ _ _ _ _ _ _ _ _ _ _ _
11010 00000 01111 00000
_ _ _ _ _ _ _ _ _ _ _ _
11010 00000 01111 00000
_ _ _ _ _ _ _ _ _ _ _ _
11010 00000 01111 00000
_ _ _ _ _ _ _ _ _ _ _ _
11010 00000 01111 00000
_ _ _ _ _ _ _ _ _ _ _ _
11010 00000 01111 00000
_ _ _ _ _ _ _ _ _ _ _ _
11010 00000 01111 00000
_ _ _ _ _ _ _ _ _ _ _ _
00010 00010 00000 00000
_ _ _ _ _ _ _ _ _ _ _ _
10001 00000 00000 00000
```

1.7 Review

Given below are the ARIXA and MRIXA commands considered herein:

`LOAD`	10000
`ADD`	01000
`ADDC REG1`	01100
`ADDC REG2`	01010
`RELOAD ADDC REG1`	11100
`RELOAD ADDC REG2`	11010
`LOAD MEM`	10001

Looking ahead, it may be the case that a second opcode block is necessary in order to allow for `REG` commands beyond, say, `REG1` and `REG2`. Given the distributed nature of Hensel load-store, one could in principle allow for `REG` queuing of FC-2-2025 instructions to be rather long.

SciSci Research

ｷｬﾄｷｬﾄ・ﾆｷｰﾙ

Published by SciSci Press