CREXX
VM Specification

The CREXX team

July 26, 2026

THE REXX LANGUAGE ASSOCIATION
CREXX Programming Series
ISBN 978-94-039116-6-3

Publication Data

The responsible publisher of this edition is identified as *IBizz IT Services and Consultancy*, Amsteldijk 14, 1074 HR Amsterdam, a registered company governed by the laws of the Kingdom of The Netherlands.

This edition is registered under ISBN 978-94-039116-6-3

Content is up to date with version *crexx-1.0.0-beta.3+local.g78c83e4a0b88*

Contents

The CREXX Publication Series

This book is part of a library, the *CREXX Programming Series*, documenting the CREXX programming language and its use and applications. This section lists the other publications in this series, and their roles. These books can be ordered in convenient hardcopy and electronic formats from the Rexx Language Association.

Programming Guide	The Programming Guide explains the working of the tools and has examples for building programs on the platforms it supports.
Language Reference	Referred to as the CRL, this is meant as the formal definition for the language, documenting its syntax and semantics, and prescribing minimal functionality for language implementers.
Library Reference	A complete reference of all included functionality in the CREXX distribution.
VM Specification	The CREXX VM Specification documents the CREXX Assembly Language and its execution by the CREXX Virtual Machine. It also contains low level virtual machine and ABI specifications.

Typographical conventions

In general, the following conventions have been observed in the CREXX publications:

- Body text is in this font
- Examples of language statements are in a `keyword` or **bold** type
- Items that are introduced, or emphasized, are in an *italic* type
- Included program fragments are listed in this fashion:

```
/* salute the reader */
say 'hello reader'
```

Console output generated from programs contained in the text is shown in this form:

```
hello reader
```

1

About This Book

This book documents the CREXX virtual machine, RXAS assembly language, and RXBIN bytecode format used by the Release 1 beta line.

The programming guide describes how to build and run programs. This book is for readers who need the lower-level model: RXAS authors, compiler and linker contributors, VM embedders, native plugin authors, and anyone debugging the generated bytecode.

The VM is the execution target for the current Level B compiler. Level B is a typed REXX-family systems language; it is not a complete Classic REXX compatibility layer.

The main pipeline is:

1. `rxc` compiles `.rexx` source to `.rxas` assembly.
2. `rxas` assembles `.rxas` to `.rxbin` bytecode.
3. `rxlink` optionally combines modules into a linked image with one shared constant pool.
4. `rxvm` and related interpreters execute RXBIN bytecode.

Where this book discusses implementation structures, it describes the current C implementation as a guide to the ABI and runtime model. Internal structures can change; the public contract is the RXAS/RXBIN behaviour and the embedding APIs exposed by the project headers.

2

Low-Level System Information

2.1 Register-Based Virtual Machine

CREXX uses a register-based virtual machine. Instructions name their source and target registers directly, and logical data flow is written from right to left in RXAS instruction forms.

The threaded `rxvm` interpreter is the base VM executable. `rxbvm` provides the bytecode-dispatch variant. `rxvme` and `rxbvme` are extended interpreters that include the standard library image used by common Level B programs.

2.2 Machine Interface

RXAS is the assembly language for the RXVM instruction set. The assembler emits RXBIN records containing constant-pool data, module records, instruction streams, and metadata records.

There are no privileged RXVM instructions in the current release. Native and platform integration is provided through VM instructions, the embedding API, and the plugin architecture.

2.3 Registers and Values

The number of VM registers is limited by memory and by instruction encoding, not by a small hardware-style register file. Source-level variables, temporary values, arguments, object attributes, and VM integration records are mapped to registers by the compiler or by hand-written RXAS.

At runtime, a register holds a `value` structure. The exact C structure is an implementation detail, but the current model contains:

- a `value_type` status
- integer, floating-point, decimal, string, binary, and object storage
- string and binary length/capacity fields
- native payload hooks used by the plugin/runtime integration layer
- an immutable object type descriptor and attribute storage used by the class/interface runtime

The current implementation lives in `interpreter/rxvalue.h`. A simplified view is:

```
typedef struct value {
    value_type status;
    rxinteger int_value;
    double float_value;
    void *decimal_value;
    size_t decimal_value_length;
    char *string_value;
    size_t string_length;
    char *binary_value;
    size_t binary_length;
    void *native_payload;
    const struct RxGraphTypeRef *object_type;
    struct value **attributes;
} value;
```

Do not rely on older documentation that showed `value_type` as a set of object, string, decimal, float, and int bit flags. That layout is not the current runtime contract.

2.4 Conversions

RXVM instructions are typed. Conversions are explicit in generated or hand written RXAS. For example, an integer value that must be printed as text is converted by an integer-to-string instruction before a string-oriented instruction consumes it.

This explicit conversion model is one of the reasons Level B can be checked by the compiler and still expose useful assembler-level control.

3

Architectural Facilities

The CREXX Virtual Machine is organized as a collection of architectural facilities. Each facility provides a coherent set of operations that implement a particular aspect of program execution. The instruction set is the programmer-visible interface to these facilities.

This organization separates the architectural capabilities of the virtual machine from the individual instructions that implement them. New instructions may be introduced in future releases without altering the architectural model, provided they extend existing facilities or introduce new ones in a compatible manner.

The following sections describe each architectural facility at a conceptual level. The detailed semantics, operands, exceptions, and examples for individual instructions are described in the Instruction Reference chapter.

3.1 Fixed-Point Arithmetic

The Fixed-Point Arithmetic facility provides the fundamental integer computation capabilities of the CREXX Virtual Machine. These facilities operate on signed integer values and are used for counters, loop variables, indexing, address calculations, and general-purpose arithmetic. Since integer operations produce exact results whenever the mathematical result is representable, they form the basis of most compiler-generated code.

The facility supports arithmetic operations, comparisons, sign manipulation, in-

crement and decrement operations, and conversions between integer values and the other numeric representations supported by the virtual machine. Representative instructions include `iadd`, `isub`, `imul`, `idiv`, `isex`, `inc`, and `dec`.

Arithmetic operations perform the elementary mathematical functions on integer operands. Comparison operations establish ordering relationships that are examined by the Control Flow facility. Conversion operations translate integer values into floating-point, decimal, binary, and string representations.

3.2 Floating-Point Arithmetic

The Floating-Point Arithmetic facility provides efficient computation on approximate real numbers. It is intended for scientific, engineering, statistical, and graphical applications where numerical range is more important than exact decimal representation.

The facility supports arithmetic operations, comparisons, mathematical transformations, and conversions between floating-point values and the other numeric representations supported by the virtual machine. Representative instructions include `fadd`, `fsub`, `fmul`, `fdiv`, floating-point comparison operations, and the various conversion instructions.

Unlike fixed-point arithmetic, floating-point operations may involve rounding, overflow, underflow, and exceptional values such as infinities and NaNs. These behaviors are defined by the floating-point model implemented by the virtual machine.

3.3 Decimal Arithmetic

The Decimal Arithmetic facility provides exact computation on decimal values. Unlike binary floating-point arithmetic, decimal operations preserve decimal precision throughout a computation, making them particularly suitable for financial, commercial, and accounting applications.

The facility supports decimal arithmetic, comparison, scaling, rounding, and conversions between decimal values and the other numeric representations supported by the virtual machine. Representative instructions include decimal ad-

dition, subtraction, multiplication, division, comparison, scaling, and rounding operations.

By providing decimal arithmetic directly within the instruction set, the virtual machine ensures predictable and portable behavior independent of the decimal facilities provided by the host processor. There are two implementations for every decimal instruction, one following the traditional Rexx unlimited precision model, and the other a high-performance hardware variant.

3.4 String Processing

The String Processing facility provides the text manipulation capabilities of the crexx Virtual Machine. Character strings are represented as immutable UTF-8 values, allowing efficient storage while supporting the complete Unicode character repertoire.

The facility includes string construction, concatenation, extraction, searching, comparison, formatting, and conversion between strings and the other data types supported by the virtual machine. Representative instructions include concatenation, substring extraction, lexical comparison, string search, and conversions between strings and numeric or binary values.

Because text processing occupies a central role in the Rexx language family, the virtual machine provides dedicated string operations rather than relying entirely on runtime library routines.

3.5 Binary Memory

The Binary Memory facility provides operations on uninterpreted sequences of bytes. Unlike the String Processing facility, binary memory assigns no character encoding or semantic meaning to stored data. These facilities are intended for implementing binary file formats, communication protocols, serialization, cryptographic algorithms, and interfaces to external systems.

The facility includes storage management, copying, updating, typed field access, memory movement, text-field extraction, and binary comparison. Representative instructions include `blen`, `bresize`, `bcopy`, `bappend`, `bupdate`, `bgetu32`, `bseti64`,

`bgets`, `bsets`, `bmove`, `bmemmove`, `bcmpb`, and `bcmps`.

Storage management operations create, resize, and initialize binary objects. Copying and movement operations efficiently relocate byte sequences without interpreting their contents. Typed access operations interpret selected byte ranges as fixed-width integers or floating-point values, while text-field operations provide controlled interchange between binary objects and UTF-8 strings.

3.6 Logical Operations

The Logical Operations facility manipulates Boolean values and individual bits without interpreting them as numeric quantities. These operations provide the building blocks for condition evaluation, bit masking, flag manipulation, and compact binary encoding.

The facility supports Boolean operations, bit testing, shifting, rotation, and other bit-oriented manipulations. Representative instructions include `and`, `or`, `xor`, `not`, logical and arithmetic shift operations, rotation operations, and bit-test instructions.

Logical operations frequently cooperate with the Fixed-Point Arithmetic and Control Flow facilities to implement efficient decision-making and low-level algorithms.

3.7 Control Flow

The Control Flow facility governs the order in which instructions are executed. Every programming construct, including conditional statements, loops, procedure invocation, recursion, exception handling, and program termination, is ultimately implemented by this facility.

The facility includes unconditional transfers, conditional branches, procedure invocation, procedure return, and exception transfer. Representative instructions include `branch`, the conditional branch instructions, `call`, and `ret`.

Branch operations examine execution conditions established by previous comparison operations. This separation between comparison and branching allows a single comparison result to be examined by multiple subsequent branches and

simplifies compiler optimization.

Procedure invocation operations preserve sufficient execution context to support nested procedure calls and recursive execution while maintaining machine independence.

3.8 Object and Runtime Facilities

The Object and Runtime Facilities provide services that manage the execution environment rather than application data. These facilities support the object-oriented execution model of the CREXX Virtual Machine and expose selected aspects of the runtime environment to executing programs.

The facility includes object creation, runtime type inspection, reflection, exception handling, metadata access, dynamic loading, stack inspection, and runtime service invocation. Representative instructions include the object construction, type inquiry, exception, and runtime metadata operations.

These facilities isolate application programs from the internal organization of the virtual machine while providing the flexibility required by dynamic language features.

3.9 Input/Output

The Input/Output facility provides controlled access to files, streams, devices, and operating-system services while insulating application programs from host-specific interfaces.

The facility includes operations for opening and closing files, reading and writing streams, positioning within files, querying stream status, and performing related operating-system services. Representative instructions include the file, stream, console, and operating-system interface operations. The Input/Output Facility also includes the low-level network instructions which form the base for TCP/IP Socket handling in the runtime library.

By presenting a uniform interface to external resources, the Input/Output facility enables programs to execute consistently across different host platforms.

3.10 Time Services

The Time Services facility provides access to temporal information maintained by the execution environment. Programs may obtain the current date and time, measure elapsed execution intervals, and access high-resolution timing facilities.

Representative instructions obtain calendar dates, clock values, elapsed times, processor timing information, and related temporal measurements.

These facilities support scheduling, profiling, benchmarking, timeout processing, logging, and other time-dependent algorithms while remaining independent of the host operating system.

3.11 Debugging Facilities

The Debugging Facilities provide architectural support for debugging, instrumentation, tracing, and program analysis. They allow execution to be suspended at well-defined locations so that execution state can be examined or modified by external development tools.

The facility includes breakpoint operations, execution tracing, state reporting, and controlled resumption of execution. Representative instructions include the various breakpoint and debugging operations.

Although these facilities have no direct effect on program semantics, they provide a standardized interface for debuggers, profilers, test frameworks, and other software development tools.

PART I

Reference

4 Virtual Machine Instruction Set

This chapter describes RXVM instructions by functional characteristic. It is a reference for RXAS authors and for readers inspecting compiler output.

Read the assembler guide before writing non-trivial RXAS by hand; many details that look obvious at the instruction level are normally handled by the compiler and linker.

4.1 Characteristics of RXVM Instructions

Most RXVM instructions share these properties:

- logical data flow is written from right to left
- comparisons write their boolean result into an explicit register
- there is no separate condition-code register
- constants and symbols are represented through the RXBIN constant pool
- modules carry metadata records for source mapping, imports, classes, interfaces, factories, methods, and linker/runtime support
- typed conversion instructions make representation changes explicit

4.2 Instruction Argument Types

Instruction definitions use compact argument-type names. Common ones include:

Type	Description
`REG`	VM register
`ID`	label or symbolic identifier
`FUNC`	global or local callable reference
`INT`	integer literal or constant-pool integer
`FLOAT`	floating-point literal or constant-pool float
`DECIMAL`	decimal literal or constant-pool decimal
`STRING`	string literal or constant-pool string
`BINARY`	binary literal or constant-pool binary payload where supported

Related instructions often share a mnemonic and differ by opcode and operand types. The generated instruction tables later in this book are the detailed source for individual opcode forms.

4.3 Fixed Point Arithmetic

`DEC` - Decrement Register

`Syntax` - form

Name	OP1
DEC	REG

`Operation` The `dec` instruction decrements the register specified in Operand 1. Assume `r1` holds the value 43, after the execution of the instruction `r1` holds the number 42.

`0x001d` Decrement Int (op1--) REG

```
/* dec example */
main()  .locals=2
    say  "dec example:"
    load r1,43 * load the integer 43 into register 1
    dec  r1    * decrement register 1
    itos r1    * integer to string for display
    say  r1    * display the string, 42
    ret        * return to caller
```

```
dec example:
42
```

DEC0 - Decrement Register 0

Syntax - form

Name	OP1
DEC0	no operand

Operation The `dec0` instruction decrements the `r0` register, and is a shortcut that allows a performance increase in the microcode implementation.

0x001f Decrement R0-- Int no

```
/* dec0 example */
main()  .locals=2
    say  "dec0 example:"
    load r0,43 * load the integer 43 into register 0
    dec0       * decrement register 0
    itos r0    * integer to string for display
    say  r0    * display the string, 42
    ret        * return to caller
```

```
dec0 example:
42
```

DEC1 - Decrement Register 1

Syntax - form

Name	OP1
DEC1	no operand

Operation The `dec1` instruction decrements the `r1` register, and is a shortcut that allows a performance increase in the microcode implementation.

0x0021 Decrement R1-- Int no

```
/* dec1 example */
main()  .locals=2
    say  "dec1 example:"
    load r1,43 * load the integer 43 into register 0
    dec1       * decrement register 0
    itos r1    * integer to string for display
    say  r1    * display the string, 42
    ret        * return to caller
```

```
dec1 example:
42
```

DEC2 - Decrement Register 2

Syntax - form

Name	OP1
DEC2	no operand

Operation The dec2 instruction decrements the r2 register, and is a shortcut that allows a performance increase in the microcode implementation.

0x0023 Decrement R2-- Int no

```
/* dec2 example */
main()  .locals=3
    say  "dec2 example:"
    load r2,43 * load the integer 43 into register 0
    dec2       * decrement register 0
    itos r2    * integer to string for display
    say  r2    * display the string, 42
    ret        * return to caller
```

```
dec2 example:
42
```

IADD - Integer Add

Syntax - variants

Name	OP1	OP2	OP3
IADD	REG	REG	REG
IADD	REG	REG	INT

Operation

0x000f Integer Add (op1=op2+op3) REGREGREG

0x0010 Integer Add (op1=op2+op3) REGREGINT

IAND - Integer AND

Syntax - variants

Name	OP1	OP2	OP3
IAND	REG	REG	REG
IAND	REG	REG	INT

Operation

0x00a8 bit wise and of 2 integers (op1=op2&op3) REGREGREG

0x00a9 bit wise and of 2 integers (op1=op2&op3) REGREGINT

ICOPY - Integer Copy

Syntax - form

Name	OP1	OP2
ICOPY	REG	REG

Operation

0x0009 Copy Integer op2 to op1 REGREG

IDIV - Integer Divide

Syntax - variants

Name	OP1	OP2	OP3
IDIV	REG	REG	REG
IDIV	REG	REG	INT
IDIV	REG	INT	REG

Operation

0x0016 Integer Divide (op1=op2/op3) REGREGREG

0x0017 Integer Divide (op1=op2/op3) REGREGINT

0x0018 Integer Divide (op1=op2/op3) REGINTREG

IEQ - Integer Equals

Syntax - variants

Name	OP1	OP2	OP3
IEQ	REG	REG	REG
IEQ	REG	REG	INT

Operation

0x0064 Int Equals op1=(op2==op3) REGREGREG

0x0065 Int Equals op1=(op2==op3) REGREGINT

IGT - Integer Greater Than

Syntax - variants

Name	OP1	OP2	OP3
IGT	REG	REG	REG
IGT	REG	REG	INT
IGT	REG	INT	REG

Operation

0x0068 Int Greater than op1=(op2>op3) REGREGREG

```
/* igt example */
main()  .locals=6
    load r3,0
    load r4,""
    load r5,"\\#"
loop:
    igt   r1,r3,11
    brt   endloop,r1
    concat r4,r4,r5
    inc   r3
    br    loop
endloop:
    say r4
    itos r3
    load r2,"we ran this "
    concat r3,r2,r3
    concat r3,r3," times."
    say r3
    ret
```

```
############
we ran this 12 times.
```

0x0069 Int Greater than op1=(op2>op3) REGREGINT

0x006a Int Greater than op1=(op2>op3) REGINTREG

IGTE - Integer Greater Than or Equal

Syntax - variants

Name	OP1	OP2	OP3
IGTE	REG	REG	REG
IGTE	REG	REG	INT
IGTE	REG	INT	REG

Operation

0x006b Int Greater than equals op1=(op2>=op3) REGREGREG

0x006c Int Greater than equals op1=(op2>=op3) REGREGINT

0x006d Int Greater than equals op1=(op2>=op3) REGINTREG

ILT - Integer Less Than

Syntax - variants

Name	OP1	OP2	OP3
ILT	REG	REG	REG
ILT	REG	REG	INT
ILT	REG	INT	REG

Operation

0x006e Int Less than op1=(op2<op3) REGREGREG

0x006f Int Less than op1=(op2<op3) REGREGINT

0x0070 Int Less than op1=(op2<op3) REGINTREG

ILTE - Integer Less Than or Equal

Syntax - variants

Name	OP1	OP2	OP3
ILTE	REG	REG	REG
ILTE	REG	REG	INT
ILTE	REG	INT	REG

Operation

0x0071 Int Less than equals op1=(op2<=op3) REGREGREG

0x0072 Int Less than equals op1=(op2<=op3) REGREGINT

0x0073 Int Less than equals op1=(op2<=op3) REGINTREG

IMOD - Integer Modulo

Syntax - variants

Name	OP1	OP2	OP3
IMOD	REG	REG	REG
IMOD	REG	REG	INT
IMOD	REG	INT	REG

Operation

0x0019 Integer Modulo (op1=op2
0x001a Integer Modulo (op1=op2
0x001b Integer Modulo (op1=op2&op3) REGINTREG

IMULT - Intger Multiply

Syntax - variants

Name	OP1	OP2	OP3
IMULT	REG	REG	REG
IMULT	REG	REG	INT

Operation

0x0014 Integer Multiply (op1=op2*op3) REGREGREG

0x0015 Integer Multiply (op1=op2*op3) REGREGINT

INC - Increment Integer Register

Syntax - form

Name	OP1
INC	REG

Operation

0x001c Increment Int (op1++) REG

INC0 - Increment Register R0

Syntax - form

Name	OP1
INC0	no operand

Operation

0x001e Increment R0++ Int no

INC1 - Increment Register R1

Syntax - form

Name	OP1
INC1	no operand

Operation

0x0020 Increment R1++ Int no

INC2 - Increment Register R2

Syntax - form

Name	OP1
INC2	no operand

Operation

0x0022 Increment R2++ Int no

INE - Integer Not Equal

Syntax - variants

Name	OP1	OP2	OP3
INE	REG	REG	REG
INE	REG	REG	INT

Operation

0x0066 Int Not equals op1=(op2!=op3) REGREGREG

0x0067 Int Not equals op1=(op2!=op3) REGREGINT

IPOW - Power of Integer

Syntax - variants

Name	OP1	OP2	OP3
IPOW	REG	REG	REG
IPOW	REG	REG	INT
IPOW	REG	INT	REG

Operation

0x0153 op1=op2**op3 REGREGREG

0x0154 op1=op2**op3 REGREGINT

0x0155 op1=op2**op3 REGINTREG

ISEX - Integer Sign Exchange

Syntax - form

Name	OP1
ISEX	REG

Operation Sign EXchange, this instructions inverts the sign of the integer in the register in OP1.

0x0156 dec op1 = -op1 (sign change) REG

```
/* isex example */
main()  .locals=2
    say  "isex example:"
    load r1,1 * load 1 into register 1
    isex r1   * exchange the sign
    itos r1   * integer to string for display
    say  r1   * display the number, -1
    isex r1   * sign exchange again
    itos r1   * need to convert again
    say  r1   * display the string, 1
    ret       * return to caller
```

```
isex example:
-1
1
```

ISHL - Integer Shift Left

Syntax - variants

Name	OP1	OP2	OP3
ISHL	REG	REG	REG
ISHL	REG	REG	INT

Operation

0x00ae bit wise shift logical left of integer (op1=op2<<op3) REGREGREG

0x00af bit wise shift logical left of integer (op1=op2<<op3) REGREGINT

ISHR - Integer Shift Right

Syntax - variants

Name	OP1	OP2	OP3
ISHR	REG	REG	REG
ISHR	REG	REG	INT

Operation

0x00b0 bit wise shift logical right of integer (op1=op2>>op3) REGREGREG

0x00b1 bit wise shift logical right of integer (op1=op2>>op3) REGREGINT

ISUB - Integer Subtract

Syntax - variants

Name	OP1	OP2	OP3
ISUB	REG	REG	REG
ISUB	REG	REG	INT
ISUB	REG	INT	REG

Operation

0x0011 Integer Subtract (op1=op2-op3) REGREGREG

0x0012 Integer Subtract (op1=op2-op3) REGREGINT

0x0013 Integer Subtract (op1=op2-op3) REGINTREG

ITOF - Integer to Float

Syntax - form

Name	OP1
ITOF	REG

Operation

0x00e7 Set register float value from its int value REG

LOAD - Load

Syntax - variants

Name	OP1	OP2
LOAD	REG	INT
LOAD	REG	FLOAT
LOAD	REG	STRING
LOAD	REG	REG
LOAD	REG	DECIMAL
LOAD	INT	INT
LOAD	INT	REG
LOAD	REG	BINARY

Operation `load rDst,0x...` materializes a whole binary constant into the binary slot of `rDst`. It is useful for small values and tests, but large lookup tables should be read directly through the constant-source forms below.

0x0001 Load op1 with op2 REGINT

0x0002 Load op1 with op2 REGFLOAT

0x0003 Load op1 with op2 REGSTRING

0x0004 Load op1 with op2 REGREG

0x0005 Load op1 with op2 REGDECIMAL

```
main() .locals=6
   load r3,1.23456e3d
```

```
        dtos r3
        say r3
        ret
```

```
1234.56
```

0x0006 Load op1 with op2 (non symbolic registers) INTINT

0x0007 Load op1 with op2 (non symbolic registers) INTREG

0x00b7 Load binary literal op2 into op1 REGBINARY

LOADSETTP - Load and Set Type

Syntax - variants

Name	OP1	OP2	OP3
LOADSETTP	REG	INT	INT
LOADSETTP	REG	FLOAT	INT
LOADSETTP	REG	STRING	INT

Operation

0x0206 load register and sets externally writable status flags load op1=op2 (op1.flags = op3) REGINTINT

0x0207 load register and sets externally writable status flags load op1=op2 (op1.flags = op3) REGFLOATINT

0x0208 load register and sets externally writable status flags load op1=op2 (op1.flags = op3) REGSTRINGINT

4.4 Floating Point Arithmetic

FADD - Add Float

Syntax - variants

Name	OP1	OP2	OP3
FADD	REG	REG	REG
FADD	REG	REG	FLOAT

Operation

0x012c Float Add (op1=op2+op3) REGREGREG

0x012d Float Add (op1=op2+op3) REGREGFLOAT

FCOPY - Copy Float

Syntax - form

Name	OP1	OP2
FCOPY	REG	REG

Operation

0x000a Copy Float op2 to op1 REGREG

FDIV - Divide Float

Syntax - variants

Name	OP1	OP2	OP3
FDIV	REG	REG	REG
FDIV	REG	REG	FLOAT
FDIV	REG	FLOAT	REG

Operation

0x0133 Float Divide (op1=op2/op3) REGREGREG

0x0134 Float Divide (op1=op2/op3) REGREGFLOAT

0x0135 Float Divide (op1=op2/op3) REGFLOATREG

FEQ - Float Equals

Syntax - variants

Name	OP1	OP2	OP3
FEQ	REG	REG	REG
FEQ	REG	REG	FLOAT

Operation

0x013c Float Equals op1=(op2==op3) REGREGREG

0x013d Float Equals op1=(op2==op3) REGREGFLOAT

FFORMAT - Format String from Float

Syntax - form

Name	OP1	OP2	OP3
FFORMAT	REG	REG	REG

Operation

0x0106 DEPRECATED use fextr. Set string value from float value using a format string REGREGREG

FGT - Float Greater Than

Syntax - variants

Name	OP1	OP2	OP3
FGT	REG	REG	REG
FGT	REG	REG	FLOAT
FGT	REG	FLOAT	REG

Operation

0x0140 Float Greater than op1=(op2>op3) REGREGREG

0x0141 Float Greater than op1=(op2>op3) REGREGFLOAT

0x0142 Float Greater than op1=(op2>op3) REGFLOATREG

FGTE - Float Greater Than or Equal

Syntax - variants

Name	OP1	OP2	OP3
FGTE	REG	REG	REG
FGTE	REG	REG	FLOAT
FGTE	REG	FLOAT	REG

Operation

0x0143 Float Greater than equals op1=(op2>=op3) REGREGREG

0x0144 Float Greater than equals op1=(op2>=op3) REGREGFLOAT

0x0145 Float Greater than equals op1=(op2>=op3) REGFLOATREG

FLT - Float Less Than

Syntax - variants

Name	OP1	OP2	OP3
FLT	REG	REG	REG
FLT	REG	REG	FLOAT
FLT	REG	FLOAT	REG

Operation

0x0146 Float Less than op1=(op2<op3) REGREGREG

0x0147 Float Less than op1=(op2<op3) REGREGFLOAT

0x0148 Float Less than op1=(op2<op3) REGFLOATREG

FLTE - Float Less Than or Equal

Syntax - variants

Name	OP1	OP2	OP3
FLTE	REG	REG	REG
FLTE	REG	REG	FLOAT
FLTE	REG	FLOAT	REG

Operation

0x0149 Float Less than equals op1=(op2<=op3) REGREGREG

0x014a Float Less than equals op1=(op2<=op3) REGREGFLOAT

0x014b Float Less than equals op1=(op2<=op3) REGFLOATREG

FMULT - Float Multiply

Syntax - variants

Name	OP1	OP2	OP3
FMULT	REG	REG	REG
FMULT	REG	REG	FLOAT

Operation

0x0131 Float Multiply (op1=op2*op3) REGREGREG

0x0132 Float Multiply (op1=op2*op3) REGREGFLOAT

FNE - Float Not Equal

Syntax - variants

Name	OP1	OP2	OP3
FNE	REG	REG	REG
FNE	REG	REG	FLOAT

Operation

0x013e Float Not equals op1=(op2!=op3) REGREGREG

0x013f Float Not equals op1=(op2!=op3) REGREGFLOAT

FPOW - Power of Float

Syntax - variants

Name	OP1	OP2	OP3
FPOW	REG	REG	REG
FPOW	REG	REG	FLOAT
FPOW	REG	FLOAT	REG

Operation

0x014e op1=op2**op3 REGREGREG

0x014f op1=op2**op3 REGREGFLOAT

0x0150 op1=op2**op3 REGFLOATREG

FSEX - Sign Exchange Float

Syntax - form

Name	OP1
FSEX	REG

Operation Sign EXchange, this instructions inverts the sign of the floating point number in the register in OP1.[1]

0x0151 float op1 = -op1 (sign change) REG

```
/* fsex example */
main()  .locals=2
    say  "fsex example:"
    load r1,1.0 * load 1 into register 1
    fsex r1     * exchange the sign
    ftos r1     * integer to string for display
    say  r1     * display the string, -1
    fsex r1     * sign exchange again
    ftos r1     * need to convert again
    say  r1     * display the string, 1
    ret         * return to caller
```

```
fsex example:
-1
1
```

[1] Named, like `isex` in honour of the PDP11 instruction that nearly got out in an assembler, but was stopped by management at the last moment.

FSUB - Subtract Float

Syntax - variants

Name	OP1	OP2	OP3
FSUB	REG	REG	REG
FSUB	REG	REG	FLOAT
FSUB	REG	FLOAT	REG

Operation

0x012e Float Subtract (op1=op2-op3) REGREGREG

0x012f Float Subtract (op1=op2-op3) REGREGFLOAT

0x0130 Float Subtract (op1=op2-op3) REGFLOATREG

FTOB - Float to Boolean

Syntax - form

Name	OP1
FTOB	REG

Operation

0x00e9 Set register boolean (int 1 or 0) value from its float value REG

FTOI - Float to Int

Syntax - form

Name	OP1
FTOI	REG

Operation

0x00e8 Set register int value from its float value REG

FTOS - Float to String

Syntax - form

Name	OP1
FTOS	REG

Operation

0x00e6 Set register string value from its float value REG

LOAD - Load

Syntax - variants

Name	OP1	OP2
LOAD	REG	INT
LOAD	REG	FLOAT
LOAD	REG	STRING
LOAD	REG	REG
LOAD	REG	DECIMAL
LOAD	INT	INT
LOAD	INT	REG
LOAD	REG	BINARY

Operation `load rDst,0x...` materializes a whole binary constant into the binary slot of `rDst`. It is useful for small values and tests, but large lookup tables should be read directly through the constant-source forms below.

0x0001 Load op1 with op2 REGINT

0x0002 Load op1 with op2 REGFLOAT

0x0003 Load op1 with op2 REGSTRING

0x0004 Load op1 with op2 REGREG

0x0005 Load op1 with op2 REGDECIMAL

```
main() .locals=6
   load r3,1.23456e3d
```

```
        dtos r3
        say r3
        ret
```

```
1234.56
```

0x0006 Load op1 with op2 (non symbolic registers) INTINT

0x0007 Load op1 with op2 (non symbolic registers) INTREG

0x00b7 Load binary literal op2 into op1 REGBINARY

4.5 Logical Operations

AND - Logical AND

Syntax - form

Name	OP1	OP2	OP3
AND	REG	REG	REG

Operation

0x00b4 Logical (int) and op1=(op2 && op3) REGREGREG

COPY - Copy Register

Syntax - form

Name	OP1	OP2
COPY	REG	REG

Operation

0x0008 Copy op2 to op1 REGREG

DCALL - Dynamic Call Procedure

Syntax - form

Name	OP1	OP2	OP3
DCALL	REG	REG	REG

Operation

0x002f Dynamic call procedure (op1=op2(op3...)) REGREGREG

EXIT - Exit program

Syntax - variants

Name	OP1
EXIT	no operand
EXIT	REG
EXIT	INT

Operation

0x0035 Exit no

0x0036 Exit op1 REG

0x0037 Exit op1 INT

IAND - Integer AND

Syntax - variants

Name	OP1	OP2	OP3
IAND	REG	REG	REG
IAND	REG	REG	INT

Operation

0x00a8 bit wise and of 2 integers (op1=op2&op3) REGREGREG

0x00a9 bit wise and of 2 integers (op1=op2&op3) REGREGINT

INOT - Invert All Bits of Integer

Syntax - variants

Name	OP1	OP2
INOT	REG	REG
INOT	REG	INT

Operation

0x00b2 inverts all bits of an integer (op1= op2) REGREG

0x00b3 inverts all bits of an integer (op1= op2) REGINT

IOR - Bitwise OR of two integers

Syntax - variants

Name	OP1	OP2	OP3
IOR	REG	REG	REG
IOR	REG	REG	INT

Operation

0x00aa bit wise or of 2 integers (op1=op2|op3) REGREGREG

0x00ab bit wise or of 2 integers (op1=op2|op3) REGREGINT

IRAND - Random Integer

Syntax - variants

Name	OP1	OP2
IRAND	REG	REG
IRAND	REG	INT

Operation

0x01d0 random number random, op1=irand(op2) REGREG

0x01d1 random number random, op1=irand(op2) REGINT

IXOR - Bitwise Exclusive OR of two Integers

Syntax - variants

Name	OP1	OP2	OP3
IXOR	REG	REG	REG
IXOR	REG	REG	INT

Operation

0x00ac bit wise exclusive OR of 2 integers (op1=op2ôp3) REGREGREG

0x00ad bit wise exclusive OR of 2 integers (op1=op2ôp3) REGREGINT

LOADSETTP - Load and Set Type

Syntax - variants

Name	OP1	OP2	OP3
LOADSETTP	REG	INT	INT
LOADSETTP	REG	FLOAT	INT
LOADSETTP	REG	STRING	INT

Operation

0x0206 load register and sets externally writable status flags load op1=op2 (op1.flags = op3) REGINTINT

0x0207 load register and sets externally writable status flags load op1=op2 (op1.flags = op3) REGFLOATINT

0x0208 load register and sets externally writable status flags load op1=op2 (op1.flags = op3) REGSTRINGINT

NOT - Not

Syntax - form

Name	OP1	OP2
NOT	REG	REG

Operation

0x00b6 Logical (int) not op1=!op2 REGREG

OR - Logical OR

Syntax - form

Name	OP1	OP2	OP3
OR	REG	REG	REG

Operation

0x00b5 Logical (int) or op1=(op2 || op3) REGREGREG

SETORTP - OR the Register Type Flag

Syntax - form

Name	OP1	OP2
SETORTP	REG	INT

Operation

0x0209 or externally writable register status flags (op1.flags = op1.flags | op2)
REGINT

SWAP - Swap Registers

Syntax - form

Name	OP1	OP2
SWAP	REG	REG

Operation

0x01fe Swap op1 and op2 REGREG

4.6 Branching

BCF - Branch on Count if False

Syntax - variants

Name	OP1	OP2	OP3
BCF	ID	REG	
BCF	ID	REG	REG

Operation

0x015e if op2=0 goto op1(if false) else dec op2 IDREG

0x015f if op2=0 goto op1(if false) else dec op2 and inc op3 IDREGREG

BCT - Branch on Count if True

Syntax - variants

Name	OP1	OP2	OP3
BCT	ID	REG	
BCT	ID	REG	REG

Operation

0x0159 dec op2; if op2>0; goto op1(if true) IDREG

0x015a dec op2; inc op3, if op2>0; goto op1(if true) IDREGREG

BCTNM - Branch on Count to Name

Syntax - variants

Name	OP1	OP2	OP3
BCTNM	ID	REG	
BCTNM	ID	REG	REG

Operation

0x015b dec op2; if op2>=0; goto op1(if true) IDREG

0x015c dec op2; inc op3, if op2>=0; goto op1(if true) IDREGREG

BEQ - Branch on Equal

Syntax - variants

Name	OP1	OP2	OP3
BEQ	ID	REG	REG
BEQ	ID	REG	INT

Operation

0x0028 if op2==op3 then goto op1 IDREGREG

0x0029 if op2==op3 then goto op1 IDREGINT

BGE - Branch on Greater Than or Equal

Syntax - variants

Name	OP1	OP2	OP3
BGE	ID	REG	REG
BGE	ID	REG	INT

Operation

0x0162 if op2>=op3 then goto op1 IDREGREG

```
/*
 * rexx TESTBGT branch if op2>op3 to op1
 */
.globals=0
main()  .locals=8

   load r1,"Test BGE with reg reg"
   say r1
   load r2,0
   load r3,10
loop:
   bge  endloop,r2,r3
   inc  r2
   itos r2
   say  r2
   br   loop
endloop:
   load r1,"Endloop reached"
   say r1
   ret
```

```
Test BGE with reg reg
1
2
3
```

```
4
5
6
7
8
9
10
Endloop reached
```

0x0163 if op2>=op3 then goto op1 IDREGINT

```
/*
 * rexx TESTBGT branch if op2>op3 to op1
 */
.globals=0
main()  .locals=8
   load r1,"Test BGE with int"
   say r1
   load r2,0
loop2:
   bge  endloop2,r2,5
   inc  r2
   itos r2
   say  r2
   br   loop2
endloop2:
   load r1,"Endloop reached"
   say r1

   ret
```

```
Test BGE with int
1
2
3
4
5
Endloop reached
```

BGT - Branch on Greater Than

Syntax - variants

Name	OP1	OP2	OP3
BGT	ID	REG	REG
BGT	ID	REG	INT

Operation

0x0160 if op2>op3 then goto op1 IDREGREG

0x0161 if op2>op3 then goto op1 IDREGINT

```
/* bgt example */
main()  .locals=6
    load r3,0
    load r4,""
    load r5,"bgt "
loop:
    bgt   endloop,r3,11
    concat r4,r4,r5
    inc   r3
    br    loop
endloop:
    say r4
    itos r3
    load r2,"we ran this "
    concat r3,r2,r3
    concat r3,r3," times."
    say r3
    ret
```

```
bgt bgt bgt bgt bgt bgt bgt bgt bgt bgt bgt bgt
we ran this 12 times.
```

BLE - Branch on Less Than or Equal

Syntax - variants

Name	OP1	OP2	OP3
BLE	ID	REG	REG
BLE	ID	REG	INT

Operation

0x0166 if op2<=op3 then goto op1 IDREGREG

0x0167 if op2<=op3 then goto op1 IDREGINT

BLT - Branch on Less Than

Syntax - variants

Name	OP1	OP2	OP3
BLT	ID	REG	REG
BLT	ID	REG	INT

Operation

0x0164 if op2<op3 then goto op1 IDREGREG

```
/*
 * rexx TESTBLT branch if op2>op3 to op1
 */
.globals=0
main()  .locals=8

   load r1,"Test BLT with reg reg"
   say r1
   load r2,12
   load r3,10
loop:
   blt  endloop,r2,r3
   dec  r2
   itos r2
   say  r2
   br   loop
endloop:
   load r1,"Endloop reached"
   say r1
   ret
```

```
Test BLT with reg reg
11
10
9
```

```
Endloop reached
```

0x0165 if op2<op3 then goto op1 IDREGINT

```
/*
 * rexx TESTBLT branch if op2>op3 to op1
 */
.globals=0
main()  .locals=8
   load r1,"Test BLT with int"
   say r1
   load r2,7
loop2:
   blt  endloop2,r2,5
   dec  r2
   itos r2
   say  r2
   br   loop2
endloop2:
   load r1,"Endloop reached"
   say r1
   ret
```

```
Test BLT with int
6
5
4
Endloop reached
```

BNE - Branch on Not Equal

Syntax - variants

Name	OP1	OP2	OP3
BNE	ID	REG	REG
BNE	ID	REG	INT

Operation

0x002a if op2!=op3 then goto op1 IDREGREG

0x002b if op2!=op3 then goto op1 IDREGINT

BR - Branch Unconditionally

Syntax - form

Name	OP1
BR	ID

Operation

0x0024 Branch to op1 ID

```
/* load example */
main()  .locals=6
    load r3,0
    load r4,""
    load r5,"load str "
loop:
    igt   r1,r3,11
    brt   endloop,r1
    concat r4,r4,r5
    inc   r3
    br    loop
endloop:
    say r4
    itos r3
    load r2,"we ran this "
    concat r3,r2,r3
    concat r3,r3," times."
    say r3
    ret
```

```
load str load str load str load str load str load str load str load
str load str load str load str load str
we ran this 12 times.
```

BRF - Branch on False

Syntax - form

Name	OP1	OP2
BRF	ID	REG

Operation

0x0026 Branch to op1 if op2 false IDREG

BRT - Branch on True

Syntax - form

Name	OP1	OP2
BRT	ID	REG

Operation

0x0025 Branch to op1 if op2 true IDREG

BRTF - Branch on True

Syntax - form

Name	OP1	OP2	OP3
BRTF	ID	ID	REG

Operation

0x0027 Branch to op1 if op3 true, otherwise branch to op2 IDIDREG

BRTPANDT - Branch on Type And True

Syntax - form

Name	OP1	OP2	OP3
BRTPANDT	ID	REG	INT

Operation

0x020c if op2 readable status flags & op3 true then goto op1 IDREGINT

BRTPT - Branch on Type

Syntax - form

Name	OP1	OP2
BRTPT	ID	REG

Operation

0x020b if op2 has externally writable status flags then goto op1 IDREG

CALL - Call Procedure

Syntax - variants

Name	OP1	OP2	OP3
CALL	FUNC		
CALL	REG	FUNC	
CALL	REG	FUNC	REG

Operation A call instruction calls a procedure, which can be local, from a library or from a statically or dynamically linked plugin or VM plugin. In the example, the returncode is deposited by the called produre in the third operand register, the called procedure is specified in the second operand, FUNC field, as a procedure (ending in parentheses). The third operand register contains the register name of the first argument passed to the procedure, as per the CREXX linking convention. The other arguments are in adjacent higher numbered registers. The called procedure must be linked in with an `.expose` statement, as on line 17 of the example.

0x002c Call procedure (op1()) FUNC

0x002d Call procedure (op1=op2()) REGFUNC

0x002e Call procedure (op1=op2(op3...)) REGFUNCREG

```
.globals=0

main() .locals=4
   .meta "words.main"="b" ".void" main() ""
   .srcstep 1 1 17 "words.rexx" 5 1 57 "say words('The quick brown
       fox jumps over the lazy dog')"
   load r1,1
   load r2,"The quick brown fox jumps over the lazy dog"
   settp r2,512
```

```
    call r3,words(),r1
    itos r3
    say r3
    .srcstep 2 2 1 "words.rexx" 5 57 57 "say words('The quick brown
        fox jumps over the lazy dog')"
    ret

words() .expose=rxfnsb.words
    .meta "rxfnsb.words"="b" ".int" words() "string1=.string"
```

```
9
```

CNOP - No Operation

Syntax - variants

Name	OP1
CNOP	no operand

Operation

0x0057 no operation with nine register inputs REGREGREGREGREGREGREGREGREG

```
.globals=0

main() .locals=9
    load r0,0
    load r1,1
    load r2,2
    load r3,3
    load r4,4
    load r5,5
    load r6,6
    load r7,7
    load r8,8
    cnop r0,r1,r2,r3,r4,r5,r6,r7,r8
    say "wide cnop accepted"
    ret 0
```

```
wide cnop accepted
```

0x021c no operation no

`LINK` - Link

`Syntax` - form

Name	OP1	OP2
LINK	REG	REG

`Operation`

`0x00d4` Link op2 to op1 REGREG

NULL - Null

Syntax - form

Name	OP1
NULL	REG

Operation

0x000e Null (clear) op1 REG

RET - Return

Syntax - variants

Name	OP1
RET	no operand
RET	REG
RET	INT
RET	FLOAT
RET	STRING

Operation

0x0030 Return VOID no

0x0031 Return op1 REG

0x0032 Return op1 INT

0x0033 Return op1 FLOAT

0x0034 Return op1 STRING

4.7 I/O operations

READLINE - Read a Line

Syntax - form

Name	OP1
READLINE	REG

Operation

0x01c9 Read Line to op1 REG

SAY - Write on Console

Syntax - variants

Name	OP1
SAY	REG
SAY	INT
SAY	FLOAT
SAY	STRING
SAY	CHAR

Operation

0x01c2 Say op1 REG

0x01c5 Say op1 INT

0x01c6 Say op1 FLOAT

0x01c7 Say op1 STRING

0x01c8 Say op1 CHAR

SAYX - Write on Console without Linefeed

Syntax - variants

Name	OP1
SAYX	REG
SAYX	STRING

Operation

0x01c3 Say op1 without line feed REG

```
main() .locals=3
   load r1,"this is line1"
   load r2,"this is line2"
   say r1
   sayx "this is line2"
   sayx r2
   say "end"
   ret
```

```
this is line1
this is line2this is line2end
```

0x01c4 Say op1 (as string) without line feed STRING

4.8 Time Instructions

`MTIME` - Microseconds Time

`Syntax` - form

Name	OP1
MTIME	REG

`Operation`

`0x01cb` Put time in microseconds into op1 REG

`TIME` - Get Time

`Syntax` - form

Name	OP1
TIME	REG

`Operation`

`0x01ca` Put time into op1 REG

XTIME - Put extended Time properties into OP1

Syntax - form

Name	OP1	OP2
XTIME	REG	STRING

Operation

0x01cc put special time properties into op1 REGSTRING

4.9 Meta Instructions

GETTP - Get Type

Syntax - form

Name	OP1	OP2
GETTP	REG	REG

Operation

0x0204 gets the readable register status flags (op1 = op2.flags) REGREG

LINKATTR - Link Attributes

Syntax - variants

Name	OP1	OP2	OP3
LINKATTR	REG	REG	REG
LINKATTR	REG	REG	INT

Operation

0x00c8 Link attribute op3 of op2 to op1 REGREGREG

0x00c9 Link attribute op3 of op2 to op1 REGREGINT

METADECODEINST - Metadecodeinst

Syntax - form

Name	OP1	OP2
METADECODEINST	REG	REG

Operation

0x0212 Decode opcode (op1 decoded op2) REGREG

METALINKPREG - Metalinkpreg

Syntax - form

Name	OP1	OP2
METALINKPREG	REG	REG

Operation

0x0218 Link parent-frame-register[op2] to op1 REGREG

METALOADCALLERADDR - Metaloadcalleraddr

Syntax - form

Name	OP1
METALOADCALLERADDR	REG

Operation

0x0219 Load caller address object to op1 REG

METALOADDATA - Metaloaddata

Syntax - form

Name	OP1	OP2	OP3
METALOADDATA	REG	REG	REG

Operation

0x0217 Load Metadata (op1 = (metadata)op2[op3]) REGREGREG

METALOADEDMODULES - Metaloadedmodules

Syntax - form

Name	OP1
METALOADEDMODULES	REG

Operation

0x020e Loaded Modules (op1 = array loaded modules) REG

METALOADEDPROCS - Metaloadedprocs

Syntax - form

Name	OP1	OP2
METALOADEDPROCS	REG	REG

Operation

0x020f Loaded Procedures (op1 = array procedures in module op2) REGREG

METALOADFOPERAND - Metaloadfoperand

Syntax - form

Name	OP1	OP2	OP3
METALOADFOPERAND	REG	REG	REG

Operation

0x0214 Load Float Operand (op1 = (float)op2[op3]) REGREGREG

METALOADINST - Metaloadinst

Syntax - form

Name	OP1	OP2	OP3
METALOADINST	REG	REG	REG

Operation

0x0211 Load Instruction Code (op1 = (inst)op2[op3]) REGREGREG

METALOADIOPERAND - Metaloadioperand

Syntax - form

Name	OP1	OP2	OP3
METALOADIOPERAND	REG	REG	REG

Operation

0x0213 Load Integer/Index Operand (op1 = (int)op2[op3]) REGREGREG

METALOADMODULE - Metaloadmodule

Syntax - form

Name	OP1	OP2
METALOADMODULE	REG	REG

Operation

0x020d Load Module (op1 = module num of last loaded module in rxbin op2)
REGREG

METALOADPOPERAND - Metaloadpoperand

Syntax - form

Name	OP1	OP2	OP3
METALOADPOPERAND	REG	REG	REG

Operation

0x0216 Load Procedure Operand (op1 = (proc)op2[op3]) REGREGREG

METALOADSOPERAND - Metaloadsoperand

Syntax - form

Name	OP1	OP2	OP3
METALOADSOPERAND	REG	REG	REG

Operation

0x0215 Load String Operand (op1 = (string)op2[op3]) REGREGREG

MOVE - Move

Syntax - form

Name	OP1	OP2
MOVE	REG	REG

Operation

0x01fd Move op2 to op1 REGREG

SETTP - Set Register Type Flag

Syntax - form

Name	OP1	OP2
SETTP	REG	INT

Operation

0x0205 sets externally writable register status flags, preserving VM-private flags
REGINT

UNLINK - Unlink Register

Syntax - form

Name	OP1
UNLINK	REG

Operation

0x00d5 Unlink op1 REG

4.10 Breakpoint Instructions

BPOFF - Breakpoint Off

Syntax - form

Name	OP1
BPOFF	no operand

Operation

0x01ef Disable Breakpoints no

BPON - Breakpoint On

Syntax - variants

Name	OP1
BPON	FUNC
BPON	no operand

Operation

0x01ed Enable Breakpoints with op1 handler FUNC

0x01ee Enable Breakpoints with existing handler no

4.11 String Instructions

APPEND - Append to String

Syntax - form

Name	OP1	OP2
APPEND	REG	REG

Operation

0x008f String Append (op1=op1||op2) REGREG

APPENDCHAR - Append Character

Syntax - form

Name	OP1	OP2
APPENDCHAR	REG	REG

Operation

0x008c Append Concat Char op2 (as int) on op1 REGREG

CONCAT - String Concat

Syntax - variants

Name	OP1	OP2	OP3
CONCAT	REG	REG	REG
CONCAT	REG	REG	STRING
CONCAT	REG	STRING	REG

Operation

0x0086 String Concat (op1=op2||op3) REGREGREG

```
/* concat example */
main()  .locals=6
    load r3,0
    load r4,""
    load r5,"conc "
loop:
    igt   r1,r3,11
    brt   endloop,r1
    concat r4,r4,r5
    inc   r3
    br    loop
endloop:
    say r4
    itos r3
    load r2,"we ran this "
    concat r3,r2,r3
    concat r3,r3," times."
    say r3
    ret
```

```
conc conc conc conc conc conc conc conc conc conc conc conc
we ran this 12 times.
```

0x0087 String Concat (op1=op2||op3) REGREGSTRING

0x0088 String Concat (op1=op2||op3) REGSTRINGREG

CONCCHAR - Concatenate Characters

Syntax - form

Name	OP1	OP2	OP3
CONCCHAR	REG	REG	REG

Operation

0x008d Concat Char op1 from op2 position op3 REGREGREG

DROPCHAR - Drop Character

Syntax - form

Name	OP1	OP2	OP3
DROPCHAR	REG	REG	REG

Operation

0x00a0 set op1 from op2 after dropping all chars from op3 REGREGREG

FNDBLNK - Find Blank

Syntax - form

Name	OP1	OP2	OP3
FNDBLNK	REG	REG	REG

Operation

0x00a5 op1 = find next blank in op2[op3] and behind REGREGREG

```
/* word
 * word: procedure = .string
 * arg string1 = .string, wordnum = .int
*/
         .globals=0
main()   .locals=13
    load r1,"The quick brown fox jumps over the lazy dog."
    load r2,4
    load  r3,0          /* offset in string                    */
    strlen r4,r1
    ieq   r12,r4,0      /* break out if zero length            */
    brt   break,r12
    load  r6,0          /* word count                          */
    load  r10,""        /* extracted word                      */
loop:
    fndnblnk r3,r1,r3   /* find first/next non blank offset    */
    ilt r5,r3,0         /* if <0, nothing found, end search    */
    brt break,r5
    inc r6              /* else increase word count            */
                        /* offset of word is in R3             */
    copy r8,r3          /* save offset of word                 */
    fndblnk r3,r1,r3    /* from offset find next blank offset  */
    ieq r7,r6,r2        /* is this the word we are looking for?*/
    brt wordf,r7        /* go and fetch it                     */
    ilt r5,r3,0         /* if <0, nothing found, end search    */
    brt break,r5        /* word not found                      */
    bct loop,r4,r3      /* cont. to look for next non blank    */
wordf:
    igt r5,r3,0         /* if <0 eos reached = -length(string) */
```

```
    brt isplus,r5       /* length of string is offset +1       */
    imult r3,r3,-1      /* make it positive                    */
isplus:
    isub r5,r3,r8       /* calculate length to extract         */
wordc:
    concchar r10,r1,r8  /* extract word char by char           */
    bct  wordc,r5,r8    /* ... until length counter is 0       */
break:
    say r10
    ret        * return to caller
```

```
fox
```

FNDNBLNK - Find Nth Blank

Syntax - form

Name	OP1	OP2	OP3
FNDNBLNK	REG	REG	REG

Operation

0x00a6 op1 = find next next non blank in op2[op3] and behind REGREGREG

GETSTRPOS - Get String Position

Syntax - form

Name	OP1	OP2
GETSTRPOS	REG	REG

Operation getstrpos gets the character pointer of the register. Each register has a string and a current position in that string, which is returned by this instruction.

0x009b Get String (op2) charpos into op1 REGREG

HEXCHAR - Get Hex Char from String

Syntax - form

Name	OP1	OP2	OP3
HEXCHAR	REG	REG	REG

Operation

0x0098 op1 (as hex) = op2[op3] REGREGREG

ITOS - Integer to String

Syntax - form

Name	OP1
ITOS	REG

Operation

0x00e5 Set register string value from its int value REG

LOAD - Load

Syntax - variants

Name	OP1	OP2
LOAD	REG	INT
LOAD	REG	FLOAT
LOAD	REG	STRING
LOAD	REG	REG
LOAD	REG	DECIMAL
LOAD	INT	INT
LOAD	INT	REG
LOAD	REG	BINARY

Operation `load rDst,0x...` materializes a whole binary constant into the binary slot of `rDst`. It is useful for small values and tests, but large lookup tables should be read directly through the constant-source forms below.

0x0001 Load op1 with op2 REGINT

0x0002 Load op1 with op2 REGFLOAT

0x0003 Load op1 with op2 REGSTRING

0x0004 Load op1 with op2 REGREG

0x0005 Load op1 with op2 REGDECIMAL

```
main() .locals=6
   load r3,1.23456e3d
```

```
        dtos r3
        say r3
        ret
```

```
1234.56
```

0x0006 Load op1 with op2 (non symbolic registers) INTINT

0x0007 Load op1 with op2 (non symbolic registers) INTREG

0x00b7 Load binary literal op2 into op1 REGBINARY

PADSTR - Pad a String

Syntax - form

Name	OP1	OP2	OP3
PADSTR	REG	REG	REG

Operation

0x00a3 set op1=op2[repeated op3 times] REGREGREG

POSCHAR - Position of Char

Syntax - form

Name	OP1	OP2	OP3
POSCHAR	REG	REG	REG

Operation

0x0099 op1 = position of op3 in op2 REGREGREG

RSEQ - Rseq

Syntax - variants

Name	OP1	OP2	OP3
RSEQ	REG	REG	REG
RSEQ	REG	REG	STRING

Operation

0x0076 non strict String Equals op1=(op2=op3) REGREGREG

0x0077 non strict String Equals op1=(op2=op3) REGREGSTRING

SAPPEND - Append a String

Syntax - form

Name	OP1	OP2
SAPPEND	REG	REG

Operation

0x008e String Append with space (op1=op1||op2) REGREG

SCONCAT - Concatenate String

Syntax - variants

Name	OP1	OP2	OP3
SCONCAT	REG	REG	REG
SCONCAT	REG	REG	STRING
SCONCAT	REG	STRING	REG

Operation

0x0089 String Concat with space (op1=op2||op3) REGREGREG

0x008a String Concat with space (op1=op2||op3) REGREGSTRING

0x008b String Concat with space (op1=op2||op3) REGSTRINGREG

SCOPY - Copy String

Syntax - form

Name	OP1	OP2
SCOPY	REG	REG

Operation

0x000b Copy String op2 to op1 REGREG

SEQ - String Equals

Syntax - variants

Name	OP1	OP2	OP3
SEQ	REG	REG	REG
SEQ	REG	REG	STRING

Operation

0x0074 String Equals op1=(op2==op3) REGREGREG

0x0075 String Equals op1=(op2==op3) REGREGSTRING

SETSTRPOS - Set String Charpos

Syntax - form

Name	OP1	OP2
SETSTRPOS	REG	REG

Operation setstrpos sets the character pointer of the register. Each register has a string and a current position in that string, which is set by this instruction.

0x009a Set String (op1) charpos set to op2 REGREG

SGT - String Greater Than

Syntax - variants

Name	OP1	OP2	OP3
SGT	REG	REG	REG
SGT	REG	REG	STRING
SGT	REG	STRING	REG

Operation

0x007a String Greater than op1=(op2>op3) REGREGREG

0x007b String Greater than op1=(op2>op3) REGREGSTRING

0x007c String Greater than op1=(op2>op3) REGSTRINGREG

SGTE - String Greater Than or Equal

Syntax - variants

Name	OP1	OP2	OP3
SGTE	REG	REG	REG
SGTE	REG	REG	STRING
SGTE	REG	STRING	REG

Operation

0x007d String Greater than equals op1=(op2>=op3) REGREGREG

0x007e String Greater than equals op1=(op2>=op3) REGREGSTRING

0x007f String Greater than equals op1=(op2>=op3) REGSTRINGREG

SLT - String Less Than

Syntax - variants

Name	OP1	OP2	OP3
SLT	REG	REG	REG
SLT	REG	REG	STRING
SLT	REG	STRING	REG

Operation

0x0080 String Less than op1=(op2<op3) REGREGREG

0x0081 String Less than op1=(op2<op3) REGREGSTRING

0x0082 String Less than op1=(op2<op3) REGSTRINGREG

SLTE - String Less Than or Equal

Syntax - variants

Name	OP1	OP2	OP3
SLTE	REG	REG	REG
SLTE	REG	REG	STRING
SLTE	REG	STRING	REG

Operation

0x0083 String Less than equals op1=(op2<=op3) REGREGREG

0x0084 String Less than equals op1=(op2<=op3) REGREGSTRING

0x0085 String Less than equals op1=(op2<=op3) REGSTRINGREG

SNE - String Not Equal

Syntax - variants

Name	OP1	OP2	OP3
SNE	REG	REG	REG
SNE	REG	REG	STRING

Operation

0x0078 String Not equals op1=(op2!=op3) REGREGREG

0x0079 String Not equals op1=(op2!=op3) REGREGSTRING

STOF - String to Float

Syntax - form

Name	OP1
STOF	REG

Operation

0x00ec Set register float value from its string value REG

STOI - String to Int

Syntax - form

Name	OP1
STOI	REG

Operation

0x00ed Set register int value from its string value REG

STRCHAR - String to Char

Syntax - variants

Name	OP1	OP2	
STRCHAR	REG	REG	REG
STRCHAR	REG	REG	

Operation

0x0096 op1 (as int) = op2[op3] REGREGREG

0x0097 op1 (as int) = op2[charpos] REGREG

STRLEN - String Length

Syntax - form

Name	OP1	OP2
STRLEN	REG	REG

Operation

0x0095 String Length op1 = length(op2) REGREG

STRLOWER - Lowercase String

Syntax - form

Name	OP1	OP2
STRLOWER	REG	REG

Operation

0x009d Set string to lower case value REGREG

STRUPPER - Uppercase String

Syntax - form

Name	OP1	OP2
STRUPPER	REG	REG

Operation

0x009e Set string to upper case value REGREG

SUBSTCUT - Set Substring and Cut Off

Syntax - form

Name	OP1	OP2
SUBSTCUT	REG	REG

Operation

0x00a2 set op1=substr(op1,,op2) cuts off op1 after position op3 REGREG

SUBSTR - Substring

Syntax - form

Name	OP1	OP2	OP3
SUBSTR	REG	REG	REG

Operation

0x009c op1 = op2[charpos]...op2[charpos+op3-1] REGREGREG

SUBSTRING - Set remaining String

Syntax - form

Name	OP1	OP2	OP3
SUBSTRING	REG	REG	REG

Operation

0x00a1 set op1=substr(op2,op3) remaining string REGREGREG

TRANSCHAR - Translate Characters

Syntax - form

Name	OP1	OP2	OP3
TRANSCHAR	REG	REG	REG

Operation

0x009f replace op1 if it is in op3-list by char in op2-list REGREGREG

TRIML - Trim String from Left

Syntax - variants

Name	OP1	OP2	OP3
TRIML	REG	REG	
TRIML	REG	REG	REG

Operation

0x0090 Trim String (op1) from Left by (op2) Chars REGREG

0x0092 Trim String (op2) from Left by (op3) Chars into op1 REGREGREG

TRIMR - Trim String from Right

Syntax - variants

Name	OP1	OP2	OP3
TRIMR	REG	REG	
TRIMR	REG	REG	REG

Operation

0x0091 Trim String (op1) from Right by (op2) Chars REGREG

0x0093 Trim String (op2) from Right by (op3) Chars into op1 REGREGREG

TRUNC - Truncate

Syntax - form

Name	OP1	OP2	OP3
TRUNC	REG	REG	REG

Operation

0x0094 Trunc String (op2) to (op3) Chars into op1 REGREGREG

4.12 Decimal Arithmetic

DADD - Dadd

Syntax - variants

Name	OP1	OP2	OP3
DADD	REG	REG	DECIMAL
DADD	REG	REG	REG

Operation

0x0168 Decimal Add (op1=op2-op3) REGREGREG

0x0169 Decimal Add (op1=op2-op3) REGREGDECIMAL

```
main() .locals=6
   load r3,1.23456e3d
   dadd r4,r3,1e3
   dtos r4
   say r4
   ret
```

```
2234.56
```

DCOPY - Dcopy

Syntax - form

Name	OP1	OP2
DCOPY	REG	REG

Operation

0x000c Copy Decimal op2 to op1 REGREG

```
main() .locals=6
   load r3,1.23456e3d
   dcopy r4,r3
   dtos r4
   say r4
   ret
```

```
1234.56
```

DDIV - Ddiv

Syntax - variants

Name	OP1	OP2	OP3
DDIV	REG	REG	REG
DDIV	REG	REG	DECIMAL
DDIV	REG	DECIMAL	REG

Operation

0x016f Decimal Divide (op1=op2/op3) REGREGREG

```
/* ddiv example */
main()  .locals=4
   say  "ddiv example 100/81 in 100 decimals:"
   setnumdgts  75
   load  r1,100
   load r2,81
   itod  r1
   itod  r2
   ddiv  r3,r1,r2
   dtos  r1
   dtos  r2
   dtos  r3
   say   r1
   say   r2
   say   r3
   ret
```

```
ddiv example 100/81 in 100 decimals:
100
81
1.23456790123456790123456790123456790123456790123456790123456790123456790123
```

0x0170 Decimal Divide (op1=op2/op3) REGREGDECIMAL

`0x0171` Decimal Divide (op1=op2/op3) REGDECIMALREG

DECPLNM - Decplnm

Syntax - form

Name	OP1	OP2	OP3
DECPLNM	REG	REG	REG

Operation

0x0190 Get Decimal Plugin Name op1=name op2=description op3=version REGREGREG

```
main() .locals=6
    decplnm r3,r4,r5
    load r0," Name="
    concat r0,r0,r3
    concat r0,r0," Description="
    concat r0,r0,r4
    concat r0,r0," Version="
    concat r0,r0,r5
    say r0
    ret
```

```
Name=mcdecimal Description=Decimal Plugin using Mike Cowlishaw's dec-
Number library Version=Plugin:0.1 Library:3.64
```

DEQ - Decimal Equals

Syntax - variants

Name	OP1	OP2	OP3
DEQ	REG	REG	REG
DEQ	REG	REG	DECIMAL

Operation

0x0178 Decimal Equals op1=(op2==op3) REGREGREG

0x0179 Decimal Equals op1=(op2==op3) REGREGDECIMAL

```
main() .locals=6
   load r3,1.23456e3d
   deq r4,r3,1234.56d
   brt equal,r4
   say "not equal"
   ret
equal:
   say "equal"
   ret
```

```
equal
```

DEQBR - Deqbr

Syntax - form

Name	OP1	OP2	OP3
DEQBR	ID	REG	REG

Operation

0x018a Decimal Equal if (op2=op3) goto op1 IDREGREG

```
main() .locals=6
   load r3,1.23456e3d
   load r4,1.23456e3d
   deqbr succeed,r3,r4
   say "failed "
   ret
succeed:
   say "succeed"
   ret
```

```
succeed
```

DFORMAT -

Syntax - variants

Name	OP1	OP2	OP3
DFORMAT	REG	REG	REG

Operation

DGT - Decimal Greater Than

Syntax - variants

Name	OP1	OP2	OP3
DGT	REG	REG	DECIMAL
DGT	REG	REG	REG
DGT	REG	DECIMAL	REG

Operation

0x017c Decimal Greater than op1=(op2>op3) REGREGREG

```
main() .locals=6
   load r3,1.23457e3d
   load r4,1.23456e3d
   dgt r5,r3,r4
   brt succeed,r5
   say "failed "
   ret
succeed:
   say "succeed"
   ret
```

```
succeed
```

0x017d Decimal Greater than op1=(op2>op3) REGREGDECIMAL

0x017e Decimal Greater than op1=(op2>op3) REGDECIMALREG

DGTBR - Dgtbr

Syntax - form

Name	OP1	OP2	OP3
DGTBR	ID	REG	REG

Operation

0x0188 Decimal Greater than if (op2>op3) goto op1 IDREGREG

```
main() .locals=5
   load r3,1.23457e3d
   load r4,1.23456e3d
   dgtbr succeed,r3,r4
   say "failed "
   ret
succeed:
   say "succeed"
   ret
```

```
succeed
```

DGTE - Decimal Greater Than or Equal

Syntax - variants

Name	OP1	OP2	OP3
DGTE	REG	REG	REG
DGTE	REG	REG	DECIMAL
DGTE	REG	DECIMAL	REG

Operation

0x017f Decimal Greater than equals op1=(op2>=op3) REGREGREG

```
main() .locals=6
   load r3,1.23456e3d
   load r4,1.23456e3d
   dgte r5,r3,r4
   brt succeed,r5
   inc r1
   say "failed"
succeed:
   itos r5
   say r5
   ret
```

```
1
```

0x0180 Decimal Greater than equals op1=(op2>=op3) REGREGDECIMAL

0x0181 Decimal Greater than equals op1=(op2>=op3) REGDECIMALREG

```
main() .locals=5
   load r3,1234.56d
   dgte r4,1234.56d,r3
   brt succeed,r4
```

```
    say "failed"
    ret
succeed:
    itos r4
    say r4
    ret
```

```
1
```

DIVF -

Syntax - variants

Name	OP1	OP2	OP3
DIVF	REG	REG	REG
DIVF	REG	REG	FLOAT
DIVF	REG	FLOAT	REG

Operation

DIVI -

Syntax - variants

Name	OP1	OP2	OP3
DIVI	REG	REG	REG
DIVI	REG	REG	INT

Operation

DLT - Decimal Less Than

Syntax - variants

Name	OP1	OP2	OP3
DLT	REG	REG	REG
DLT	REG	REG	DECIMAL
DLT	REG	DECIMAL	REG

Operation

0x0182 Decimal Less than op1=(op2<op3) REGREGREG

0x0183 Decimal Less than op1=(op2<op3) REGREGDECIMAL

0x0184 Decimal Less than op1=(op2<op3) REGDECIMALREG

```
main() .locals=5
   load r3,1234.57d
   dlt r4,1234.56d,r3
   brt succeed,r4
   say "failed"
   ret
succeed:
   say "succeed"
   ret
```

```
succeed
```

DLTBR - Dltbr

Syntax - form

Name	OP1	OP2	OP3
DLTBR	ID	REG	REG

Operation

0x0189 Decimal Less than if (op2<op3) goto op1 IDREGREG

DLTE - Decimal Less Than or Equal

Syntax - variants

Name	OP1	OP2	OP3
DLTE	REG	REG	REG
DLTE	REG	REG	DECIMAL
DLTE	REG	DECIMAL	REG

Operation

0x0185 Decimal Less than equals op1=(op2<=op3) REGREGREG

```
main() .locals=6
   load r3,1.23456e3d
   load r4,1.23456e3d
   dlte r5,r3,r4
   brt succeed,r5
   say "failed"
   ret
succeed:
   say "succeed"
   ret
```

```
succeed
```

0x0186 Decimal Less than equals op1=(op2<=op3) REGREGDECIMAL

```
main() .locals=5
   load r3,1.23456e3d
   dlte r4,r3,1234.56d
   brt succeed,r4
   say "failed"
succeed:
   say "succeed"
   ret
```

```
succeed
```

0x0187 Decimal Less than equals op1=(op2<=op3) REGDECIMALREG

```
main() .locals=5
   load r3,1234.56d
   dlte r4,1234.56d,r3
   brt succeed,r4
   say "failed"
   ret
succeed:
   say "succeed"
   ret
```

```
succeed
```

DMOD - Decimal Modulo

Syntax - variants

Name	OP1	OP2	OP3
DMOD	REG	REG	REG

Operation

0x0175 Decimal Modulo (op1=op2
0x0176 Decimal Modulo (op1=op2
0x0177 Decimal Modulo (op1=op2&op3) REGDECIMALREG

DMULT - Decimal Multiply

Syntax - variants

Name	OP1	OP2	OP3
DMULT	REG	REG	REG
DMULT	REG	REG	DECIMAL

Operation

0x016d Decimal Multiply (op1=op2*op3) REGREGREG

```
main() .locals=6
   load r3,1.23456e3d
   load r4,5.4321e3d
   dmult r5,r3,r4
   dtos r5
   say r5
   ret
```

```
6706253.376
```

0x016e Decimal Multiply (op1=op2*op3) REGREGDECIMAL

```
main() .locals=5
   load r3,1.23456e3d
   dmult r4,r3,2d
   dtos r4
   say r4
   ret
```

```
2469.12
```

DNE - Decimal Not Equal

Syntax - variants

Name	OP1	OP2	OP3
DNE	REG	REG	REG
DNE	REG	REG	DECIMAL

Operation

0x017a Decimal Not equals op1=(op2!=op3) REGREGREG

```
main() .locals=6
   load r3,1.23456e3d
   load r4,1.23457e3d
   dne r5,r3,r4
   brt succeed,r5
   say "failed"
   ret
succeed:
   say "succeed"
   ret
```

```
succeed
```

0x017b Decimal Not equals op1=(op2!=op3) REGREGDECIMAL

```
main() .locals=5
   load r3,1.23456e3d
   dne r4,r3,1234.57d
   brt succeed,r4
   say "failed"
   ret
succeed:
   say "succeed"
   ret
```

```
succeed
```

DPOW - Decimal Power

Syntax - variants

Name	OP1	OP2	OP3
DPOW	REG	REG	REG
DPOW	REG	REG	DECIMAL
DPOW	REG	DECIMAL	REG

Operation

0x018b op1=op2**op3 REGREGREG

```
main() .locals=6
   load r3,2d
   load r4,3d
   dpow r5,r3,r4
   dtos r5
   say r5
   ret
```

```
8
```

0x018c op1=op2**op3 REGREGDECIMAL

```
main() .locals=5
   load r3,2d
   dpow r4,r3,3d
   dtos r4
   say r4
   ret
```

```
8
```

0x018d op1=op2**op3 REGDECIMALREG

```
main() .locals=5
   load r3,3d
   dpow r4,2d,r3
   dtos r4
   say r4
   ret
```

```
8
```

DSEX - Decimal Sign Exchange

Syntax - form

Name	OP1
DSEX	REG

Operation

0x018e Decimal op1 = -op1 (sign change) REG

```
main() .locals=4
   load r3,1.23456e3d
   dsex r3
   dtos r3
   say r3
   ret
```

```
-1234.56
```

DSUB - Decimal Subtract

Syntax - variants

Name	OP1	OP2	OP3
DSUB	REG	REG	REG
DSUB	REG	REG	DECIMAL
DSUB	REG	DECIMAL	REG

Operation

0x016a Decimal Subtract (op1=op2-op3) REGREGREG

```
main() .locals=5
   load r3,1.23456e3d
   dsub r4,r3,1e3
   dtos r4
   say r4
   ret
```

```
234.56
```

0x016b Decimal Subtract (op1=op2-op3) REGREGDECIMAL

0x016c Decimal Subtract (op1=op2-op3) REGDECIMALREG

```
main() .locals=5
   load r3,1.23456e3d
   dsub r4,1e3,r3
   dtos r4
   say r4
   ret
```

```
-234.56
```

DTOF - Decimal to Float

Syntax - form

Name	OP1
DTOF	REG

Operation

0x00f4 Convert Decimal Number to Float op1=f2dec(op2) REG

```
main() .locals=4
   load r3,1.23456e3d
   dtof r3
   ftos r3
   say r3
   ret
```

```
1234.56
```

DTOI - Decimal to Integer

Syntax - form

Name	OP1
DTOI	REG

Operation

0x00f0 Convert Decimal Number to Integer op1=dec2s(op2) REG

DTOS - Decimal to String

Syntax - form

Name	OP1
DTOS	REG

Operation

0x00ef Convert Decimal Number to Decimal String op1=dec2s(op2) REG

```
main() .locals=4
   load r3,"1e3"
   stod r3
   dtos r3
   say r3
   ret
```

```
1000
```

4.13 Binary Memory

BAPPEND - Binary Append

Syntax - form

Name	OP1	OP2
BAPPEND	REG	REG

Operation `bappend rDst,rSrc` appends the binary payload of `rSrc` to `rDst`. This instruction operates on raw bytes and do not validate UTF-8.

0x00bb Append binary op2 to op1 REGREG

BCHECKRANGE - Binary Range Check

Syntax - form

Name	OP1	OP2	OP3
BCHECKRANGE	REG	REG	REG

Operation `bcheckrange rBin,rOffset,rLen` checks that the byte range `rOffset..rOffset+rLen` is inside the logical binary length. It does not mutate any operand. It is an explicit assertion; target-sized copy, typed reads, typed writes, text field instructions, and compare instructions still perform their own range checks.

0x0257 Check binary range op1[op2..op2+op3) REGREGREG

BCLEAR - Binary Clear

Syntax - form

Name	OP1
BCLEAR	REG

Operation `bclear rBin` sets the logical binary length to zero.

0x0247 Clear binary op1 length REG

BCMPB - Binary Byte Compare

Syntax - variants

Name	OP1	OP2	OP3
BCMPB	REG	REG	REG
BCMPB	REG	BINARY	REG
BCMPB	REG	REG	BINARY
BCMPB	REG	BINARY	BINARY

Operation `bcmpb rCmp,rSrc,rNeedle` compares bytes from binary memory without copying. On entry, `rCmp` contains the byte offset into `rSrc`. `rSrc` may be a binary register or binary constant. `rNeedle` may be a binary register or binary constant. The compare length is the whole length of `rNeedle`. On success, `rCmp` is overwritten with `-1`, `0`, or `1` for unsigned-byte lexicographic ordering.

0x026f Compare binary register op2 slice at offset op1 with binary register op3
REGREGREG

0x0270 Compare binary constant op2 slice at offset op1 with binary register op3
REGBINARYREG

0x0271 Compare binary register op2 slice at offset op1 with binary constant op3
REGREGBINARY

```
/* Binary zero-copy compare example */
main() .locals=4
    load r1,0x6162630061626400
    load r2,0x616263

    load r3,0
    bcmpb r3,r1,r2
```

```
itos r3
say r3

load r3,4
bcmpb r3,r1,0x616263
itos r3
say r3

load r3,0
bcmps r3,r1,"abc"
itos r3
say r3

load r3,4
bcmps r3,r1,"abe"
itos r3
say r3
ret
```

```
0
1
0
-1
```

0x0272 Compare binary constant op2 slice at offset op1 with binary constant op3 REGBINARYBINARY

BCMPS - Binary String Compare

Syntax - variants

Name	OP1	OP2	OP3
BCMPS	REG	REG	REG
BCMPS	REG	REG	STRING
BCMPS	REG	BINARY	REG
BCMPS	REG	BINARY	STRING

Operation `bcmps rCmp,rSrc,rString` compares a zero-terminated UTF-8 field in binary memory with a string register or string constant. On entry, `rCmp` contains the byte offset into `rSrc`; on success it is overwritten with `-1`, `0`, or `1`. The source field is validated exactly as `bgets` would validate it, but no temporary string is allocated.

0x0273 Compare zero-terminated UTF-8 string from binary register op2 at offset op1 with string register op3 REGREGREG

0x0274 Compare zero-terminated UTF-8 string from binary register op2 at offset op1 with string constant op3 REGREGSTRING

0x0275 Compare zero-terminated UTF-8 string from binary constant op2 at offset op1 with string register op3 REGBINARYREG

0x0276 Compare zero-terminated UTF-8 string from binary constant op2 at offset op1 with string constant op3 REGBINARYSTRING

BCONCAT - Binary Concatenate

Syntax - form

Name	OP1	OP2	OP3
BCONCAT	REG	REG	REG

Operation `bconcat rDst,rLeft,rRight` concatenates two binary registers into `rDst`.

This instruction operates on raw bytes and do not validate UTF-8.

0x00ba Binary concat op1 = op2 || op3 REGREGREG

BCOPY - Binary Copy

Syntax - variants

Name	OP1	OP2	OP3
BCOPY	REG	REG	
BCOPY	REG	REG	REG
BCOPY	REG	BINARY	REG

Operation `bcopy rDst,rSrc` copies the whole binary payload from one register to another.
`bcopy rDst,rSrc,rOffset` copies exactly `blen(rDst)` bytes from `rSrc` starting at `rOffset` into `rDst`. `rSrc` may be a binary register or a binary constant. The destination must already have the required length.

0x0245 Copy Binary op2 to op1 REGREG

0x0259 Copy op1 length bytes from binary op2 at byte offset op3 into op1 REGREGREG

0x025a Copy op1 length bytes from binary constant op2 at byte offset op3 into op1 REGBINARYREG

```
/* Binary target-sized copy example */
.const table binary 0xaabbccdd
main() .locals=8
    load r1,0x0011223344556677
    load r2,3
    bresize r3,r2
    load r4,2
    bcopy r3,r1,r4
    load r4,0
    bgetu8 r5,r3,r4
    itos r5
```

```
say r5
load r4,2
bgetu8 r6,r3,r4
itos r6
say r6

load r4,1
bcopy r3,table,r4
load r4,0
bgetu8 r5,r3,r4
itos r5
say r5

blen r7,0x0102030405
itos r7
say r7
ret
```

```
34
68
187
5
```

BFILL - Binary Fill

Syntax - form

Name	OP1	OP2
BFILL	REG	REG

Operation `bfill rBin,rByte` fills the current logical byte range with `rByte`. The byte value must be in `0..255`.

0x0248 Fill binary op1 with byte op2 REGREG

BGETF32 - Binary Float32 Read

Syntax - variants

Name	OP1	OP2	OP3
BGETF32	REG	REG	REG
BGETF32	REG	BINARY	REG

Operation

0x0264 Float32 little-endian binary read op1=op2[op3] REGREGREG

0x0265 Float32 little-endian binary constant read op1=op2[op3] REGBINARYREG

BGETF64 - Binary Float64 Read

Syntax - variants

Name	OP1	OP2	OP3
BGETF64	REG	REG	REG
BGETF64	REG	BINARY	REG

Operation

0x024f Float64 little-endian binary read op1=op2[op3] REGREGREG

0x0261 Float64 little-endian binary constant read op1=op2[op3] REGBINARYREG

BGETI16 - Binary Signed 16-bit Read

Syntax - variants

Name	OP1	OP2	OP3
BGETI16	REG	REG	REG
BGETI16	REG	BINARY	REG

Operation

0x024c Signed 16-bit little-endian binary read op1=op2[op3] REGREGREG

0x025e Signed 16-bit little-endian binary constant read op1=op2[op3] REGBINARYREG

BGETI32 - Binary Signed 32-bit Read

Syntax - variants

Name	OP1	OP2	OP3
BGETI32	REG	REG	REG
BGETI32	REG	BINARY	REG

Operation

0x024e Signed 32-bit little-endian binary read op1=op2[op3] REGREGREG

0x0260 Signed 32-bit little-endian binary constant read op1=op2[op3] REGBINARYREG

BGETI64 - Binary Signed 64-bit Read

Syntax - variants

Name	OP1	OP2	OP3
BGETI64	REG	REG	REG
BGETI64	REG	BINARY	REG

Operation `bgetu8`, `bgeti8`, `bgetu16`, `bgeti16`, `bgetu32`, `bgeti32`, `bgeti64`, `bgetf32`, and `bgetf64` read fixed-width fields from a binary register or binary constant. Multi-byte values use little-endian storage. Signed integer forms sign-extend into the destination integer register. Unsigned forms raise `OUT_OF_RANGE` if the result cannot be represented by the active VM integer type.

`.int` maps to signed 64-bit storage through `bgeti64`. `.f32` is IEEE binary32 storage and widens to the VM float register. `.float` maps to the current VM float representation, which is binary64 for Release 1.

0x0262 Signed 64-bit little-endian binary read op1=op2[op3] REGREGREG

0x0263 Signed 64-bit little-endian binary constant read op1=op2[op3] REGBINARYREG

BGETI8 - Binary Signed 8-bit Read

Syntax - variants

Name	OP1	OP2	OP3
BGETI8	REG	REG	REG
BGETI8	REG	BINARY	REG

Operation

0x024a Signed 8-bit little-endian binary read op1=op2[op3] REGREGREG

0x025c Signed 8-bit little-endian binary constant read op1=op2[op3] REGBINARYREG

BGETS - Binary UTF-8 String Read

Syntax - variants

Name	OP1	OP2	OP3
BGETS	REG	REG	REG
BGETS	REG	BINARY	REG

Operation `bintos rReg` validates the register's current binary bytes as UTF-8 and copies them into its string slot. Invalid UTF-8 raises `UNICODE_ERROR`.

0x0268 Zero-terminated UTF-8 binary string read op1=op2[op3] REGREGREG

0x0269 Zero-terminated UTF-8 binary constant string read op1=op2[op3] REG-BINARYREG

BGETU16 - Binary Unsigned 16-bit Read

Syntax - variants

Name	OP1	OP2	OP3
BGETU16	REG	REG	REG
BGETU16	REG	BINARY	REG

Operation

0x024b Unsigned 16-bit little-endian binary read op1=op2[op3] REGREGREG

0x025d Unsigned 16-bit little-endian binary constant read op1=op2[op3] REG-BINARYREG

BGETU32 - Binary Unsigned 32-bit Read

Syntax - variants

Name	OP1	OP2	OP3
BGETU32	REG	REG	REG
BGETU32	REG	BINARY	REG

Operation

0x024d Unsigned 32-bit little-endian binary read op1=op2[op3] REGREGREG

0x025f Unsigned 32-bit little-endian binary constant read op1=op2[op3] REG-BINARYREG

BGETU8 - Binary Unsigned 8-bit Read

Syntax - variants

Name	OP1	OP2	OP3
BGETU8	REG	REG	REG
BGETU8	REG	BINARY	REG

Operation `bgetu8`, `bgeti8`, `bgetu16`, `bgeti16`, `bgetu32`, `bgeti32`, `bgeti64`, `bgetf32`, and `bgetf64` read fixed-width fields from a binary register or binary constant. Multi-byte values use little-endian storage. Signed integer forms sign-extend into the destination integer register. Unsigned forms raise `OUT_OF_RANGE` if the result cannot be represented by the active VM integer type.

`.int` maps to signed 64-bit storage through `bgeti64`. `.f32` is IEEE binary32 storage and widens to the VM float register. `.float` maps to the current VM float representation, which is binary64 for Release 1.

0x0249 Unsigned 8-bit little-endian binary read op1=op2[op3] REGREGREG

0x025b Unsigned 8-bit little-endian binary constant read op1=op2[op3] REGBINARYREG

`BINTOS` - Binary to String

`Syntax` - form

Name	OP1
BINTOS	REG

`Operation` `bintos rReg` validates the register's current binary bytes as UTF-8 and copies them into its string slot. Invalid UTF-8 raises `UNICODE_ERROR`.

`0x00c1` Convert binary register bytes to string bytes after UTF-8 validation REG

BLEN - Binary Length

Syntax - variants

Name	OP1	OP2
BLEN	REG	REG
BLEN	REG	BINARY

Operation `blen rOut,rBin` stores the logical byte length of a binary register in `rOut`. `blen rOut,bConst` does the same for a binary constant without materializing the constant in a register.

0x00b8 Binary byte length op1 = length(op2) REGREG

0x0258 Binary constant length op1=len(op2) REGBINARY

`BMEMMOVE` - Binary Memory Move

`Syntax` - form

Name	OP1	OP2	OP3
BMEMMOVE	REG	REG	REG

`Operation` `bmemmove rBin,rDstOffset,rLen` copies `rLen` bytes within one binary register. The source byte offset is read from `rBin`'s integer slot and the destination offset is the explicit `rDstOffset` operand. Overlapping ranges are safe and behave like C `memmove`.

`0x026e` Move op3 bytes within binary op1 from source offset op1.int to destination offset op2.int REGREGREG

BMOVE - Binary Move

Syntax - form

Name	OP1	OP2	OP3
BMOVE	REG	REG	REG

Operation `bmove rDst,rSrc,rLen` copies `rLen` bytes between different binary registers. The destination byte offset is read from `rDst`'s integer slot; the source byte offset is read from `rSrc`'s integer slot. The registers must be different.

0x026d Copy op3 bytes from binary op2 offset op2.int to binary op1 offset op1.int REGREGREG

```
/* Binary memory move example */
main() .locals=6
    load r1,0x001122334455
    load r2,0xaabbccdd
    load r1,1
    load r2,2
    load r3,2
    bmove r1,r2,r3
    load r5,1
    getbyte r4,r1,r5
    itos r4
    say r4

    load r1,0
    load r2,2
    load r3,4
    bmemmove r1,r2,r3
    load r5,2
    getbyte r4,r1,r5
    itos r4
    say r4
    ret
```

```
204
0
```

BRESIZE - Binary Resize

Syntax - form

Name	OP1	OP2
BRESIZE	REG	REG

Operation `bresize rBin,rLen` changes the logical binary length. Existing bytes are preserved up to the new length and growth is zero-filled. Negative lengths raise `OUT_OF_RANGE`.

0x0246 Resize binary op1 to op2 bytes, zero-fill growth REGREG

```
/* Binary fixed-width field example */
main() .locals=8
    load r1,0x
    load r2,16
    bresize r1,r2

    load r2,0
    load r3,-42
    bseti64 r1,r2,r3
    bgeti64 r4,r1,r2
    itos r4
    say r4

    load r2,8
    load r5,1.5
    bsetf32 r1,r2,r5
    bgetf32 r6,r1,r2
    ftos r6
    say r6

    load r2,0
    bgetu32 r7,0x78563412,r2
    itos r7
    say r7
    ret
```

```
-42
1.5
305419896
```

BSETF32 - Binary Float32 Write

Syntax - form

Name	OP1	OP2	OP3
BSETF32	REG	REG	REG

Operation `bgetu8`, `bgeti8`, `bgetu16`, `bgeti16`, `bgetu32`, `bgeti32`, `bgeti64`, `bgetf32`, and `bgetf64` read fixed-width fields from a binary register or binary constant. Multi-byte values use little-endian storage. Signed integer forms sign-extend into the destination integer register. Unsigned forms raise `OUT_OF_RANGE` if the result cannot be represented by the active VM integer type.

`.int` maps to signed 64-bit storage through `bgeti64`. `.f32` is IEEE binary32 storage and widens to the VM float register. `.float` maps to the current VM float representation, which is binary64 for Release 1.

0x0267 Float32 little-endian binary write op1[op2]=op3 REGREGREG

BSETF64 - Binary Float64 Write

Syntax - form

Name	OP1	OP2	OP3
BSETF64	REG	REG	REG

Operation `bgetu8`, `bgeti8`, `bgetu16`, `bgeti16`, `bgetu32`, `bgeti32`, `bgeti64`, `bgetf32`, and `bgetf64` read fixed-width fields from a binary register or binary constant. Multi-byte values use little-endian storage. Signed integer forms sign-extend into the destination integer register. Unsigned forms raise `OUT_OF_RANGE` if the result cannot be represented by the active VM integer type.

`.int` maps to signed 64-bit storage through `bgeti64`. `.f32` is IEEE binary32 storage and widens to the VM float register. `.float` maps to the current VM float representation, which is binary64 for Release 1.

0x0256 Float64 little-endian binary write op1[op2]=op3 REGREGREG

BSETI16 - Binary Signed 16-bit Write

Syntax - form

Name	OP1	OP2	OP3
BSETI16	REG	REG	REG

Operation

0x0253 Signed 16-bit little-endian binary write op1[op2]=op3 REGREGREG

BSETI32 - Binary Signed 32-bit Write

Syntax - form

Name	OP1	OP2	OP3
BSETI32	REG	REG	REG

Operation `bgetu8`, `bgeti8`, `bgetu16`, `bgeti16`, `bgetu32`, `bgeti32`, `bgeti64`, `bgetf32`, and `bgetf64` read fixed-width fields from a binary register or binary constant. Multi-byte values use little-endian storage. Signed integer forms sign-extend into the destination integer register. Unsigned forms raise `OUT_OF_RANGE` if the result cannot be represented by the active VM integer type.

`.int` maps to signed 64-bit storage through `bgeti64`. `.f32` is IEEE binary32 storage and widens to the VM float register. `.float` maps to the current VM float representation, which is binary64 for Release 1.

0x0255 Signed 32-bit little-endian binary write op1[op2]=op3 REGREGREG

BSETI64 - Binary Signed 64-bit Write

Syntax - form

Name	OP1	OP2	OP3
BSETI64	REG	REG	REG

Operation

0x0266 Signed 64-bit little-endian binary write op1[op2]=op3 REGREGREG

BSETI8 - Binary Signed 8-bit Write

Syntax - form

Name	OP1	OP2	OP3
BSETI8	REG	REG	REG

Operation `bgetu8`, `bgeti8`, `bgetu16`, `bgeti16`, `bgetu32`, `bgeti32`, `bgeti64`, `bgetf32`, and `bgetf64` read fixed-width fields from a binary register or binary constant. Multi-byte values use little-endian storage. Signed integer forms sign-extend into the destination integer register. Unsigned forms raise `OUT_OF_RANGE` if the result cannot be represented by the active VM integer type.

`.int` maps to signed 64-bit storage through `bgeti64`. `.f32` is IEEE binary32 storage and widens to the VM float register. `.float` maps to the current VM float representation, which is binary64 for Release 1.

0x0251 Signed 8-bit little-endian binary write op1[op2]=op3 REGREGREG

BSETS - Binary UTF-8 String Write

Syntax - variants

Name	OP1	OP2	OP3
BSETS	REG	REG	REG
BSETS	REG	REG	STRING

Operation `bintos rReg` validates the register's current binary bytes as UTF-8 and copies them into its string slot. Invalid UTF-8 raises `UNICODE_ERROR`.

0x026a Write string register op3 as UTF-8 plus NUL to binary op1[op2] REGREGREG

0x026b Write string constant op3 as UTF-8 plus NUL to binary op1[op2] REGREGSTRING

BSETU16 - Binary Unsigned 16-bit Write

Syntax - form

Name	OP1	OP2	OP3
BSETU16	REG	REG	REG

Operation `bgetu8`, `bgeti8`, `bgetu16`, `bgeti16`, `bgetu32`, `bgeti32`, `bgeti64`, `bgetf32`, and `bgetf64` read fixed-width fields from a binary register or binary constant. Multi-byte values use little-endian storage. Signed integer forms sign-extend into the destination integer register. Unsigned forms raise `OUT_OF_RANGE` if the result cannot be represented by the active VM integer type.

`.int` maps to signed 64-bit storage through `bgeti64`. `.f32` is IEEE binary32 storage and widens to the VM float register. `.float` maps to the current VM float representation, which is binary64 for Release 1.

0x0252 Unsigned 16-bit little-endian binary write op1[op2]=op3 REGREGREG

BSETU32 - Binary Unsigned 32-bit Write

Syntax - form

Name	OP1	OP2	OP3
BSETU32	REG	REG	REG

Operation `bgetu8`, `bgeti8`, `bgetu16`, `bgeti16`, `bgetu32`, `bgeti32`, `bgeti64`, `bgetf32`, and `bgetf64` read fixed-width fields from a binary register or binary constant. Multi-byte values use little-endian storage. Signed integer forms sign-extend into the destination integer register. Unsigned forms raise `OUT_OF_RANGE` if the result cannot be represented by the active VM integer type.

`.int` maps to signed 64-bit storage through `bgeti64`. `.f32` is IEEE binary32 storage and widens to the VM float register. `.float` maps to the current VM float representation, which is binary64 for Release 1.

0x0254 Unsigned 32-bit little-endian binary write op1[op2]=op3 REGREGREG

BSETU8 - Binary Unsigned 8-bit Write

Syntax - form

Name	OP1	OP2	OP3
BSETU8	REG	REG	REG

Operation

0x0250 Unsigned 8-bit little-endian binary write op1[op2]=op3 REGREGREG

BSLICE - Legacy Binary Cursor Slice

Syntax - form

Name	OP1	OP2	OP3
BSLICE	REG	REG	REG

Operation Deprecated: These legacy cursor instructions remain available for cursor-style byte processing. New Release 1 source lowering should prefer target-sized `bcopy` for binary field extraction.

`bslice rDst,rSrc,rLen` copies up to `rLen` bytes from `rSrc` starting at its cursor into `rDst`. It truncates at end of buffer; generated strict reads should use `bcheckrange` or target-sized `bcopy` instead.

0x00be Binary slice op1 = op2[cursor..cursor+op3) REGREGREG

BUPDATE - Binary Overlay Update

Syntax - form

Name	OP1	OP2	OP3
BUPDATE	REG	REG	REG

Operation `bupdate rDst,rOffset,rSrc` overlays the whole binary payload of `rSrc` into `rDst` at `rOffset`. The overlay must fit in the existing destination length.

This instruction operates on raw bytes and do not validate UTF-8.

0x00bf Overlay binary op3 into op1 at byte offset op2 REGREGREG

GETBINPOS - Legacy Binary Cursor Read

Syntax - form

Name	OP1	OP2
GETBINPOS	REG	REG

Operation Deprecated: These legacy cursor instructions remain available for cursor-style byte processing. New Release 1 source lowering should prefer target-sized `bcopy` for binary field extraction.
`getbinpos rOut,rBin` reads the current binary cursor.

0x00bd Get binary byte cursor from op2 into op1 REGREG

`GETBYTE` - Get Byte

`Syntax` - form

Name	OP1	OP2	OP3
GETBYTE	REG	REG	REG

`Operation` `getbyte rOut,rBin,rOffset` is the legacy tolerant byte read. It stores the byte at `rOffset`, or `-1` when the offset is outside the logical binary length. It has no binary-constant form; use `bgetu8` for strict constant byte reads.

`0x00a7` get byte (op1=op2(op3) REGREGREG

LOAD - Load

Syntax - variants

Name	OP1	OP2
LOAD	REG	INT
LOAD	REG	FLOAT
LOAD	REG	STRING
LOAD	REG	REG
LOAD	REG	DECIMAL
LOAD	INT	INT
LOAD	INT	REG
LOAD	REG	BINARY

Operation `load rDst,0x...` materializes a whole binary constant into the binary slot of `rDst`. It is useful for small values and tests, but large lookup tables should be read directly through the constant-source forms below.

0x0001 Load op1 with op2 REGINT

0x0002 Load op1 with op2 REGFLOAT

0x0003 Load op1 with op2 REGSTRING

0x0004 Load op1 with op2 REGREG

0x0005 Load op1 with op2 REGDECIMAL

```
main() .locals=6
   load r3,1.23456e3d
```

```
        dtos r3
        say r3
        ret


    1234.56
```

0x0006 Load op1 with op2 (non symbolic registers) INTINT

0x0007 Load op1 with op2 (non symbolic registers) INTREG

0x00b7 Load binary literal op2 into op1 REGBINARY

SETBINPOS - Legacy Binary Cursor Set

Syntax - form

Name	OP1	OP2
SETBINPOS	REG	REG

Operation Deprecated: These legacy cursor instructions remain available for cursor-style byte processing. New Release 1 source lowering should prefer target-sized `bcopy` for binary field extraction.
`setbinpos rBin,rOffset` sets the binary cursor after clamping the offset to `0..blen(rBin)`.

0x00bc Set binary byte cursor on op1 from op2 REGREG

`SETBYTE` - Set Byte

`Syntax` - form

Name	OP1	OP2	OP3
SETBYTE	REG	REG	REG

`Operation` `setbyte rBin,rOffset,rByte` writes one byte into mutable binary memory. The offset must be in range and the byte value must be in `0..255`.

`0x00b9` Set binary byte op1[op2] = op3 REGREGREG

`SGET` - String Constant Extract

`Syntax` - form

Name	OP1	OP2	OP3
SGET	REG	STRING	REG

`Operation` `sget rString,sConst,rOffset` extracts from a `STRING_CONST`, not from binary memory. `rOffset` is a byte offset into the UTF-8 string constant and must be a codepoint boundary. The current codepoint length of `rString` is the requested copy length. The VM scans that many codepoints, copies the exact bytes, sets the destination string byte length and codepoint count, and writes the safety NUL outside the logical value.

`0x026c` Copy op1 codepoints from string constant op2 at byte offset op3 into op1 REGSTRINGREG

STOBIN - String to Binary

Syntax - form

Name	OP1
STOBIN	REG

Operation `stobin rReg` copies the register's current string bytes exactly into its binary slot. It does not append a NUL terminator.

0x00c0 Convert string register bytes to binary bytes REG

5

Decimal instruction implementation variants

The preceding set of decimal instructions is implemented twice, in VM Plugins which can be switched out at runtime to determine the requested behaviour. One set implements the classic Rexx ANSI X3.274-1996 unlimited precision decimal floating point, which corresponds to the IEEE 754-2008 revision. The other set of instruction implementations is new for crexx and contains the implementation of the rxvmplugin plugin using (platform-instruction set architecture specific) long double precision floating point numbers.

5.1 md decimal

5.2 db decimal

table 1: db decimal precision.

Arch	OS #	`long double` Size	Format Used	Precision (bits)	Decimal Precision
x86_64	1	16 bytes	x87 80-bit extended	~64	~19--20 digits
x86_64	2	16 bytes	IEEE 128-bit	113	~33--34 digits
x86_64	3	16 bytes	IEEE 128-bit (emulated)	106--113	~31--33 digits
x86_64	4	8 bytes	IEEE 64-bit (double)	53	~15--17 digits
ARM64	5	16 bytes	IEEE 128-bit	113	~33--34 digits

Arch	OS #	`long double` Size	Format Used	Precision (bits)	Decimal Precision
ARM64	6	8 bytes	IEEE 64-bit (double)	53	~15--17 digits
s390x	5	16 bytes	IEEE 128-bit	113	~33--34 digits

TABLE 2: db decimal precision: OS legenda.

OS #	OS Description
1	Linux (glibc, e.g. Ubuntu, Debian, Fedora)
2	Linux (musl, e.g. Alpine)
3	macOS (x86_64)
4	Windows (x86_64, MSVC)
5	Linux (glibc, including s390x and ARM64)
6	macOS (ARM64)

TABLE 3: db decimal precision: architecture notes.

Arch/OS#	Notes
x86_64/1	True x87 80-bit precision; ABI aligns to 16 bytes; used by GCC
x86_64/2	long double uses IEEE binary128 via software (libquadmath)
x86_64/3	Clang supports IEEE 128-bit emulation; not hardware backed
x86_64/4	long double is just an alias for double in MSVC
ARM64/5	GCC maps long double to binary128 if supported
ARM64/6	No __float128 support in Clang; long double equals double
s390x/5	Hardware-backed IEEE 128-bit (z13 and newer)

On macOS ARM64 (Arch/OS `ARM64/6`), `db decimal` is therefore limited to about 15-17 significant decimal digits. Programs that require higher precision, such as `numeric digits 1000`, must use the `mc decimal` backend instead of `db decimal`.

PART II

Appendices

A

cRexx Architecture

crexx is a custom Rexx-to-bytecode toolchain that translates Classic Rexx semantics into an optimized bytecode format executed by a specialized VM. The core bytecode path happens through four main binaries:

1. rxc - The Compiler
2. rxas - The Assembler
3. rxlink - The Linker
4. rxvm - The Virtual Machine (Interpreter)

rxpp is a first-class preprocessor component for .rxpp sources. It runs before rxc when the wrapper or a build step asks for preprocessing, but rxc does not run RXPP internally.

A.1 The Compilation Pipeline

The pipeline of transforming Rexx source code into executable bytecode is structured as follows:

0. **RXPP Preprocessor (optional)**
    - Lives in the root preprocessor/ component and builds the rxpp tool plus its native precomp helper.
    - Expands .rxpp macro/directive source into generated CREXX source.
    - Emits options ... srcmap and raw @ source-map markers by default so

`rxc` diagnostics and source-step metadata can point back to the original `.rxpp` file. `##CFLAG nosrcmap` is the explicit legacy escape hatch for plain generated CREXX.

1. **re2c (Lexical Analyzer)**
   - Used to generate the scanner/lexer from `.re` rules (e.g., `compiler/rxcpbscn.re` and `assembler/rxasscan.re`).
   - Converts the raw Rexx source text into a stream of discrete tokens (`Token` structs).
2. **Lemon (Parser Generator)**
   - The token stream is parsed using a grammar defined in `Lemon` (e.g., `compiler/rxcpbgmr.y` and `compiler/rxcpopgr.y`).
   - Lemon applies the grammar rules to recognize the syntactic structures of the Rexx language and translates them into an Abstract Syntax Tree (AST).
   - `DO ... END` is overloaded: statement-leading `DO` remains the normal grouped/loop form, while expression-position `DO ... END` becomes `BLOCK_EXPR`. The grammar resolves the command-start ambiguity by routing top-level command expressions through a restricted `command_expression` spine while leaving the general expression grammar free to accept block expressions.
3. **AST (Abstract Syntax Tree) & Compiler Exits**
   - The primary data structure bridging the parser and the code emitter.
   - Built using a hierarchical structure of `ASTNode` C structs that capture operations, scopes, typing, and tree associations.
   - Standard comparison operators (`=`, `<>`, `>`, `<`, `>=`, `<=`) retain loose Rexx comparison semantics for string-targeted operands. The compiler emits the `r*` opcode family (`req`, `rne`, `rgt`, `rlt`, `rgte`, `rlte`), and the VM compares numerically only when both runtime string forms parse as numbers; otherwise it performs blank-padded string comparison. This prevents dynamic nonnumeric text such as `"a" > 1` from raising `CONVERSION_ERROR`.
   - String comparison operators (`==`, `>>`, `<<`, `>>=`, `<<=`) are carried as a distinct `OP_COMPARE_S_*` family. During type validation both operands are retargeted to `TP_STRING`, but the optimiser still preserves intrinsic numeric constant types long enough to stringify from value rather than source

spelling. That keeps folded behaviour aligned with runtime cases like `01 == 1`.

- Inline cloning must preserve the callee scope's numeric context when it builds replacement scopes. Without that, folded float/decimal string comparisons can silently drift from runtime behaviour under local `NUMERIC DIGITS` / `FORM` / `CASE` settings.
- Expression-level control flow is supported through `BLOCK_EXPR` (`DO ... END` used as an expression) and `LEAVE_WITH` (`LEAVE WITH expr`). The `association` pointer links each `LEAVE_WITH` back to its owning `BLOCK_EXPR`, similar to how loop `LEAVE` / `ITERATE` link to `DO`.
- Contains the **Exit Bridge Framework** (`rxcp_exit.c`), which intercepts unrecognized `IMPLICIT_CMD` nodes, invokes user-provided `rxplugin` macros to generate replacement source code, parses the interpolated strings (preserving literal quotes), and surgically grafts the resulting AST back into the main tree without violating return-type constraints. Compiler-exit dispatch is keyed by the first source token in the instruction, not by the first marshalled AST node; this matters for command-position member calls such as `x.add(...)`, whose AST node token is the member name but whose command head is `x`. A non-implicit exit that returns `REJECT` falls through to the certified implicit ADDRESS exit; `ERROR` remains a compiler error.
- **Implicit Main Argument Bridge**: when the compiler synthesizes the file-level `main()` wrapper, that procedure is marked `is_implicit_main`. Later typing and emission use that marker to interpret `arg[]` / `arg[n]` access against the hidden command-line `.string[]` that the VM already passes to `main`. Ordinary procedures still use normal vararg semantics, and explicit zero-argument `main()` does not gain accidental source-level visibility of the hidden VM argv payload.
- **Exit Fragment Scope Lifecycle**: replacement fragments from exits are parsed and structurally normalized before grafting, but any new lexical block scopes created by structured replacements are finalized later during symbol structuring/build. Nested `DO` / `IF` / `INSTRUCTIONS` emitted by exits are therefore a supported shape, and debug validation is staged after that scope rebuild so the validator sees the stabilized tree rather than the transient pre-scope fragment form.

- **Expose Mechanics**: Implements automatic scope resolution that allows `namespace ... expose` global variables to implicitly bind into local `PROCEDURE` scopes. Procedure-level `name: procedure [= .type] expose var ...` remains the local form for selected private module state shared by specific procedures.
- **Automatic Register Allocation**: Within `rxcp_val_sym.c` (Step 3 - Pass 3), the compiler walks the AST (`build_symbols_walker`) to identify explicit `NODE_REGISTER` allocations via the `with register.N[.view]` clause on class attributes. `register.0` is the source-level convention for a typed view of the containing value itself; `register.1` and above are one-based child attribute slots. `register.0` and duplicate typed views of the same physical `register.N` slot are complex attributes: emitted reads copy the linked physical payload view into a local register before expression manipulation, and writes copy the selected payload view back through the physical slot. The compiler does not copy status flags as hidden cache maintenance for these typed views; runtime classes such as `RexxValue` own their library/user flag protocol explicitly. Register flag views are direct masked status-word views, with VM/compiler/readable partitions validated as read-only source views and writable source partitions lowered through `settpmask`. The compiler automatically maps any remaining unmapped attributes of a class to unused VM registers (`r1`, `r2`, etc.) by synthesizing implicit `NODE_REGISTER` AST nodes. For normal classes, prefer this implicit allocation and keep callers on factories/methods; explicit `with register...` mappings should be reserved for genuine physical interop where a fixed layout is required.

4. **Emitter (IR -> Assembly)**
   - AST walkers (e.g., `rxcp_ast_walk.c`, `rxcp_emit_*.c`) traverse the tree.
   - Outputs intermediate string fragments representing the `rxas` Assembly instructions.
   - The optimiser performs opportunity-based AST inlining before emission. A callable may be structurally available for inlining, but each call site is still validated against the supported rewrite shapes. The release slice uses a 300 AST-node body cutoff as a profitability and metadata-size policy, not as a semantic safety boundary. Unsupported semantic gates still fail closed, including procedure-level `expose`, assembler alias-

ing, dynamic-index varargs, receiver/copyback shapes without proof, and interface/member dispatch that cannot be proved monomorphic. When a right-hand eager-operator call is inlineable but the left operand is already a computed value, the inliner rewrites the whole operator to a `BLOCK_EXPR`: first capture the left operand in a compiler temp, then `LEAVE WITH temp <op> call(...)`. The next fixed-point pass can inline the call with a stable left operand, avoiding BLOCK_EXPR result-register aliasing without giving up the inline opportunity.

- Inlining is implemented as AST tree surgery, not textual substitution. The cloned body must preserve callee-local scopes, remap symbols, bind arguments exactly as a real call would, preserve source/debug metadata, and leave the downstream emitter with an ordinary validated tree. This is why new inline cases are opened by specific parent/operand shape instead of by globally deciding that a callable is "always inlineable".
- Cross-file inlining uses compiler-owned `META_INLINE` payloads alongside normal callable metadata. The current `I6` payload begins with a versioned callable summary containing formal read/write/escape and exact-shape facts, result/control/context facts, and structural cost. The reader reconstructs those facts from the body and checks the result shape against the separately parsed callable declaration. Only exact agreement opens the imported template and summary-backed binding; older, missing, malformed, or mismatched summaries retain the ordinary call. This is an evidence gate, not a permanent exclusion: when a currently closed transformation gains a complete mathematical proof and regression coverage, open that case narrowly rather than leaving the conservative fallback in place. Import attachment also distinguishes the explicit callable from any compiler-created implicit `main`; class-factory evidence is attached to the synthesized `FACTORY` contract node rather than its semantically different generic registry procedure. Libraries preserve this metadata for downstream `rxc` optimisation; final linked images strip it by default.
- Suitable `SELECT` and equality ladders are lowered through a dedicated dispatch AST to RXAS packed jump tables. Eligibility, semantic gates, profitability thresholds, and regression invariants are documented in

`RXC_DISPATCH_OPTIMIZATION.md`.

5. **Assembler (`rxas`)**
   - Parses the generated `rxas` Assembly instructions.
   - Translates human-readable IR assembly into packed binary format (`rxbin` bytecode).
   - Generates the final executable bytecode file.
6. **Linker (`rxlink`, optional)**
   - Combines one or more `.rxbin` modules into a single linked image with one shared constant-pool record and one shared-backed module record per selected module.
   - Resolves imports and interface-provider relationships up front, while preserving the module boundaries needed by the VM loader.
   - Can strip source-level debug metadata (`META_SOURCE_STEP` and `META_TRACE_EVENT`) for smaller deployable artifacts without removing runtime contract metadata.
7. **Interpreter (`rxvm`)**
   - `rxvm` reads and executes the `rxbin` bytecode.
   - Modules are loaded via `rxldmod`.
   - The execution loop happens inside the `rxvm_run` function (e.g., in `rxvmmain.c` / `rxvmintp.c`).

A.2 Text, UTF-8, and Binary Data

In normal UTF builds, `.string` register data is stored as UTF-8 bytes while the VM exposes character operations as codepoint operations. A VM `value` tracks `string_length` as the byte length and, in UTF builds, also tracks `string_chars` plus a byte/codepoint cursor pair (`string_pos` and `string_char_pos`). Instructions such as `strlen`, `strchar`, `setstrpos`/`getstrpos`, `substr`, `strpos`, `appendchar`, `fndblnk`, and `fndnblnk` operate in codepoint space and use helpers such as `string_set_byte_pos()`, `string_slice_from_cursor()`, and `string_concat_char()` to walk or synthesize the underlying UTF-8 bytes. Some scan paths have ASCII fast paths when byte length and codepoint count match. These string instructions assume valid UTF-8 in the register payload. NUTF8 builds collapse this model back to byte positions and byte lengths.

Numeric-to-string promotion opcodes such as `itos`, `btos`, `ftos`, and `dtos` may mutate the target VM value to materialize its string representation. That is valid even when the target register is linked to caller-visible storage, such as a class attribute, because this is representation materialization rather than a source-level assignment. The compiler must therefore emit normal type promotion for linked expression values too. The VM does not currently set or consume a "string representation is valid" cache flag, and the compiler does not reuse a previously materialized string form across multiple uses; this remains a potential performance improvement rather than current behaviour.

Hex and binary suffixed source strings now split by decoded byte content. `ast_fstr()` validates the hex or binary digit syntax and decodes the bytes. `ast_fstr_chain()` handles adjacent string literal tokens separated by a physical line break, joining the raw chunk bodies before normal decoding; only the final chunk may carry an `x` or `b` suffix, and the line break contributes no byte or character. When the decoded span is valid UTF-8, the literal remains a `STRING` AST node and follows the normal escaped RXAS string path. When the decoded span is not valid UTF-8, the literal becomes a `BINARY` AST node with canonical `0x...` RXAS text. That keeps arbitrary bytes out of string-only character opcodes: using such a byte literal in a text context is rejected as `CANNOT_CAST_BINARY`, while assigning it to `.binary`, or writing `literal as .binary`, emits an RXAS binary load. A first untyped assignment such as `x = 'ffff'x` is deliberately treated as a text context, so invalid UTF-8 byte literals do not silently infer `.binary`. Ordinary strings can still be converted to `.binary`; the conversion stores the exact current UTF-8 byte sequence with no normalization. Constant strings in an explicitly binary target may be folded into a binary load, and runtime string expressions use the VM `stobin` instruction. The reverse conversion is explicit-only: `.binary as .string` validates the bytes as UTF-8. Valid constant binary values may be folded back into string literals, while invalid constant binary-to-string casts are rejected as `CANNOT_CAST_BINARY` and invalid runtime binary values raise `UNICODE_ERROR` through `bintos`.

`.binary` is present in the Level B surface and compiler metadata as `TP_BINARY`. The VM `value` has a separate `binary_value`, `binary_length`, and `binary_pos`, and `binary_buffer_length` slot. Copies and moves preserve that slot, socket and file byte operations use it (`socksendb`, `sockrecvb`, `freadb`, `fwriteb`), and native payloads reuse it with `rxvm_native_payload_ops`. The VM has shared binary buffer helpers for reserve/set/append/concat/slice operations. `GETBYTE` reads a

zero-based byte index and returns `-1` when the requested byte is outside the current binary length. `FREADB` and socket binary receive reuse the binary buffer growth machinery instead of reallocating to exact byte counts.

At the Level B source surface, `||` is byte concatenation when either operand is `.binary`; the compiler targets both operands as `.binary` and emits `bconcat`. This means a string operand in a binary concat is converted to its exact UTF-8 bytes with `stobin`, while binary bytes are never routed through `.string` or `bintos`. Blank concatenation (`OP_SCONCAT`) remains a text-only operation for now and binary use is a type mismatch.

The public byte-helper BIFs live in `lib/rxfnsb/rexx/binary.crexx`: `binlength`, `binbyte`, `binsetbyte`, `binsubstr`, `binconcat`, `binoverlay`, `bininsert`, `bindelstr`, `binpos`, `bincompare`, `bin2x`, and `x2bin`. They are thin Level B wrappers over the RXAS byte-buffer instructions plus small loops for search and hex conversion.

At the RXAS level, `BINARY_CONST` and `OP_BINARY` support exist and `0x...` operands are constant-pool binary records. `load rN,0x...` lowers to `LOAD_REG_BINARY` and populates the register's binary slot rather than the string slot. Binary literals are byte-paired hex (`0x00ff` is two bytes); `0x`/`0X` is the empty binary literal, and disassembly canonicalizes as lowercase `0x...`. RXAS also exposes byte-buffer instructions for length, single-byte update, concat, append, byte-cursor get/set, cursor-based slice, fixed-size overlay update, `stobin` string-byte conversion, and `bintos` binary-to-string conversion. Binary-buffer instructions never validate UTF-8 and clear VM-private UTF cache flags on the destination. `bintos` is the exception: it validates the source bytes and raises `UNICODE_ERROR` in UTF builds when they are not valid text. Most character and string opcodes still take string operands and assume valid UTF-8 in UTF builds.

Level C text and binary behavior should be treated as design space, not as settled current compiler behavior. Classic Rexx is byte-oriented and commonly stores binary data in the same text values used for strings, while current Level B separates the intended surfaces as `.string` and `.binary`. Any Level C compatibility mode therefore has to choose where Classic byte-text semantics map: to UTF-8 `.string` semantics, to `.binary`, or to an explicit option such as `bytetext`. Classic Rexx BIFs will need to be audited against that decision. Level G and library work have a separate Unicode extension path for grapheme, word, and sentence boundaries, normalization, and case operations through the Unicode plugin hooks; those fea-

tures sit above the core codepoint-level VM string contract.

A.2.1 Locked Direction

The architecture direction is:

- `.string` means valid UTF-8 text in normal UTF builds.
- `.binary` means arbitrary bytes.
- Level B keeps those surfaces strict and typed; invalid byte streams should not silently become `.string` values.
- Level C may provide Classic Rexx byte-text compatibility through an explicit compatibility mode such as `bytetext`, but that mode must not weaken the Level B/G `.string` contract.
- Level G should build richer Unicode services through the existing Unicode plugin hooks. `utf8proc` is the preferred first implementation candidate for normalization, case folding, Unicode property checks, and grapheme/word/sentence segmentation because it is a small C library under MIT expat plus Unicode data license terms.

Trust boundaries for `.string` validation are compiler/assembler string constants, RXVML string setters, CREXXSAA ADDRESS variable setters, RXPA native function returns and updated argument trees, text file and socket reads, ADDRESS callbacks, and any explicit byte-to-text conversion API. Internal operations that preserve validity, such as copying, concatenating two already-valid strings, slicing on codepoint boundaries, and appending a valid Unicode scalar, should propagate cached validity/count state rather than rescanning.

The VM now maintains the first two VM-private status bits as a UTF-8 cache: `RXFLAG_VM_UTF8_VALID` and `RXFLAG_VM_UTF8_COUNT_VALID`. Trusted string constants set both bits, bounded native setters validate and count before setting them, and operations that preserve validity copy or propagate the bits. Internal raw setters can still clear the cache when they are used to materialize bytes, but Level B user-facing text boundaries reject invalid UTF-8 instead of letting those bytes become `.string` values. RXAS string constants, source literals in text context, RXVML setters, CREXXSAA variable setters, RXPA native return/argument trees, command-line arguments, ADDRESS callback text/result copying, `freadline`, `freadcdpt`, socket text receive, and explicit `.binary as .string` all

validate. RXPA validation is recursive over child attributes and is enabled by default; developers can temporarily opt out with `CREXX_RXPA_DISABLE_UTF8_CHECKS=1` while migrating plugins. Invalid byte input must stay on the binary path (`.binary`, `freadb`, `fwriteb`, `sockrecvb`, and `socksendb`) until it is decoded explicitly.

The VM register/value status word is a `uint32_t` field partitioned in `binutils/include/rxflags.h` instead of adding a second flag field:

- `0x000000FF`: VM-private, externally readable but not writable through RXAS flag instructions. This band is reserved first for UTF-8 validity/count and Unicode normalization-form cache bits.
- `0x0000FF00`: compiler call ABI flags. The current bits are `REGTP_VAL` (`0x00000100`) and `REGTP_NOTSYM` (`0x00000200`).
- `0x00FF0000`: stable library/runtime ABI flags.
- `0x7F000000`: user/experimental flags.
- `0x80000000`: reserved to avoid signed integer ambiguity.

`SETTP`, `SETORTP`, and `LOADSETTP` mask external writes so VM-private bits are preserved or cleared only by VM internals. `SETTP` is partition-aware for the public bands: a non-zero compiler-band write replaces only compiler flags, a non-zero library-band write replaces only library flags, and `SETTP reg,0` clears all public flags. This lets compiler call-ABI setup update `REGTP_*` without destroying runtime/library cache flags stored on the same value. `GETTP`, `GETANDTP`, and explicit `BRTPANDT` masks may observe readable VM-private bits; unmasked `BRTPT` only tests public/external flag bands so VM cache bits do not change old branch semantics.

RXAS/RXBIN integer operands remain `rxinteger`. The canonical definition is `platform/rxinteger.h`, and Release 1 fixes it to signed 64-bit across the compiler, assembler, VM, and RXPA ABI. Host pointer width is not the language integer width. Status instructions cast masks to the 32-bit flag word before applying the partition. Level B flag-view assignments use `SETTPMASK`, a masked replacement operation restricted to the source-writable library/user bands, so `.flags.compiler` remains read-only to source code while generated call setup can still maintain the compiler ABI flags.

Regression coverage for the partition and UTF cache contract lives in

- `interpreter/tests/tests_register_flags.rxas`,

- `interpreter/tests/tests_utf_flags.rxas`, and
- `interpreter/tests/ts_regvalue_tester.c`.

Compiler and runtime regression coverage for `.string`/`.binary` coexistence lives in

- `compiler/tests/rexx_src/binary_literal_load.crexx`,
- `compiler/tests/rexx_src/scalar_type_casts.crexx`,

and the generated negative tests in `compiler/tests/CMakeLists.txt`.

Boundary regressions include direct RXAS invalid string constants,

- `compiler/tests/src/test_rxvml_utf_boundaries.c`,
- `interpreter/tests/tests_utf_freadline_boundary.rxas` /
- `interpreter/tests/tests_utf_freadcdpt_boundary.rxas`.

The completed UTF baseline is:

1. Bounded UTF-8 validate-and-count support exists through `utf8nvalid_count()` and is used by VM string cache refresh paths.
2. Compiler source hex/binary literals are decoded as bytes, stay `.string` only when valid UTF-8, and otherwise require explicit `.binary`.
3. The register status word is partitioned in `rxflags.h`; VM-private UTF cache bits are preserved from external writes.
4. `.binary` has growable buffer helpers plus RXAS/VM literal load, length, byte get/set, append, concat, cursor, slice, and overlay instructions.
5. Explicit `.string`/`.binary` coexistence is first class in Level B: `as .binary` stores exact UTF-8 bytes through `stobin`, `as .string` validates bytes through `bintos`, and valid constant casts fold in both directions. Invalid constant byte-to-text casts fail as `CANNOT_CAST_BINARY`; invalid runtime byte-to-text conversions raise `UNICODE_ERROR`.
6. Direct RXAS string constants are part of the same contract: hand-written RXAS `STRING_CONST` / `OP_STRING` creation validates and rejects invalid UTF-8 with guidance to use binary literals.
7. External Level B text ingress validates. The public `rxvml_set_str()` returns non-zero for invalid UTF-8, RXVML run arguments and ADDRESS native callback text are checked, CREXXSAA rejects invalid variable setter values

immediately, RXPA recursively validates returned values and updated arguments after native calls, `freadline`/`freadcdpt` raise `UNICODE_ERROR`, and socket text receive reports an invalid text status. Character-walking opcodes require valid UTF cache state before using codepoint iterators.

The open work has moved to its owning levels:

- Level G owns normalization cache semantics and richer Unicode services. The VM-private status band reserves space for normalization knowledge, but NFC, NFD, NFKC, and NFKD bits should only become meaningful when Level G normalization APIs set and consume them. The preferred first Unicode plugin candidate is `utf8proc`, subject to vendoring/build work and carrying its MIT expat plus Unicode data license notices. Initial coverage should target normalization, case folding, Unicode property checks, and grapheme/word/ sentence segmentation. There is also room for a Level B CREXX proof of concept of UTF helper libraries while Level G remains design work.
- Level C owns Classic REXX migration and byte-text compatibility. Classic byte-text behavior should be isolated behind an explicit compatibility option such as `bytetext`; Classic BIFs then need auditing so users can choose UTF text semantics, byte semantics, or explicit `.binary` operations predictably.

A.3 Compiler Import Discovery

`rxc` does not treat every import location as both source and binary space anymore. Import discovery is now split into two root classes:

- source roots for `.crexx`, `.crx`, `.rexx`, and the arbitrary extension used by the initial source file, if any
- binary roots for `.rxbin`, optional `.rxas`, and `.rxplugin`

The primary source root is the directory containing the source file being compiled. Additional source roots come from `-s`. Binary roots come from any `-i` paths and the compiler executable directory. Repeated `-i` and `-s` options are accumulated in order. Search order keeps project source files ahead of deployed binary artifacts.

An extensionless initial `rxc` input falls back to `.crexx`. Headerless `.crexx`, `.crx`,

and arbitrary-extension sources default to Level G; headerless `.rexx` defaults to Level C. `.rxpp` is reserved for the preprocessor and is not scanned as an import source extension.

For source discovery the compiler now performs a lightweight header scan before any full parse. That scan reads the leading `options`, `namespace`, and `import` clauses so namespace-invisible files can be rejected before `rexbpars()` and `rxcp_val()` are invoked. Full source parsing is still used once a source file is actually selected as an import candidate.

Within a binary root, same-stem artifacts are collapsed to the freshest candidate. If timestamps tie, `.rxbin` is preferred over `.rxas`.

Compiler-generated consumer `.rxas` treats imported declaration blocks as a runtime dependency snapshot, not as a copy of the provider's full public surface. The provider artifact's exports and metadata remain definitive. When `rxc` emits a consumer `.rxas`, it re-emits only imported callable declarations that the generated instruction stream still needs for linking or runtime lookup. If optimization or inlining removes every runtime reference to an imported file, the imported declaration block is suppressed entirely.

A.4 Level B Classes and Interfaces

Level B interface support is now implemented across the compiler, assembler metadata path, and VM.

A.4.1 Compiler model

The source surface includes:

- `interface`
- `class implements .iface ...`
- interface methods with or without bodies
- interface default `*` factories and named factories
- class-side same-named factory implementations
- optional class-side same-named `match`
- checked casts with `expr as .type`

- boolean type tests with `expr is .type`
- concrete type introspection with `<typeof>(expr)`
- namespace-qualified contracts such as `.pkg..thing()`

Interface methods with bodies are emitted as final/default methods. The class must still implement abstract members, but it may not override a final/default interface member.

Qualified references use `namespace..symbol`; the left side must be an imported namespace, not a class or interface name. `namespace::symbol` remains accepted as a compatibility alias.

A.4.2 Metadata and import

The contract model is carried through normal `.rxbin` metadata with:

- `META_INTERFACE`
- `META_IMPLEMENTS`
- `META_MEMBER`

That metadata is sufficient for import reconstruction of class/interface headers without parsing procedure bodies. Imported stubs are not re-exported as new local contracts, and richer imported stubs replace poorer duplicates.

A.4.3 Runtime dispatch

Created objects carry their concrete class identity. The VM then resolves contract calls through load/link-time registries:

- a method registry keyed by concrete class plus method descriptor
- a factory-provider registry keyed by interface plus factory member name

`srcmethodsel` resolves the effective method for an interface/class receiver. The registry prefers a concrete class method and otherwise falls back to a final interface default method. The selector is a callable descriptor (`rxsig1|name|return_type|args`), and the resolved procedure metadata must match it.

`srcfprocsel` resolves interface factories from the same descriptor form. Every candidate provider is evaluated through its effective `match`; omitted `match` behaves

as score 1, scores <= 0 reject, highest positive score wins, and tied scores are broken alphabetically by concrete class name.

A.5 Source Tree and Parser Mode

CREXX now has an explicit split between the user-facing source model and the mutable compiler tree.

- After the early source-shaping and source-location work, the compiler builds an immutable SourceNode tree in compiler/rxcp_source_tree.c.
- context->source_tree is the canonical user-facing tree for authored structure, diagnostics, semantic sidecars, metadata anchors, and editor projection.
- Compiler diagnostics carry a stable RxcpDiagnostic payload: a message code plus named parameters. node_string and SourceDiagnostic.message are rendered fallbacks, not the diagnostic identity.
- Diagnostic rendering defaults to localized text. CREXX_DIAGNOSTICS=raw selects the machine-readable CODE name="value" form used by golden tests. CREXX_DIAGNOSTIC_LOCALE overrides locale selection; otherwise the renderer uses the platform locale environment and falls back to en_GB. Message catalogs are UTF-8 files in messages/, currently including en_GB, en_US, de_DE, and nl_NL, and are loaded lazily per process.
- context->ast / work_ast remains the mutable compiler tree for import loading, exit dispatch, fixed-point rewrites, optimization, and emission.
- ASTNode instances keep explicit links back to the source tree so later rewritten nodes can still report against authored source.

Parser mode (rxc --syntaxhighlight) uses the same parser and early source preparation, but it routes through compiler/rxcp_highlight_controller.c and serializes DSLSH from source_tree, not from the later rewritten work tree. The controller also keeps retained parser-mode cache state for imports and exit discovery across requests.

For the compiler-side build order and tree-split details, see Parsing Pipeline Anatomy.

For the DSLSH/editor mapping and parser-mode contract, see cREXX DSLSH Integration.

For RXPP build shape, wrapper role, and source-map marker rules, see RXPP Preprocessor.

A.6 Core Data Structures

A.6.1 `ASTNode` (from `compiler/rxcp_ast.h`)

The `ASTNode` forms the backbone of the compilation process, maintaining context, tree structure (parent/child/sibling relations), value/target typing details, code generation fragments, and parser token details.

```
struct ASTNode {
    Context *context;
    int node_number;
    NodeType node_type;
    char* file_name;
    ValueType value_type;    /* Value type */
    size_t value_dims;       /* Value dimensions */
    int *value_dim_base;
    int *value_dim_elements;
    char* value_class;       /* Value class name */
    int *target_dim_base;
    int *target_dim_elements;
    ValueType target_type;   /* Target type */
    size_t target_dims;      /* Target dimensions */
    char* target_class;      /* Target class name */
    int high_ordinal; /* Order of node after validation but before
        optimisations */
    int low_ordinal;  /* lowest in this tree root */
    int register_num;
    char register_type;
    int additional_registers;
    int num_additional_registers;
    char is_ref_arg;
    char is_opt_arg;
    char is_const_arg;
    char is_varg;
    ASTNode *free_list;
    ASTNode *parent, *child, *sibling;
    ASTNode *association; /* E.g. for LEAVE / ITERATE relevant DO node
        or LEAVE_WITH relevant BLOCK_EXPR */
    Token *token;
    Scope *scope;
```

```
    char *node_string;
    size_t node_string_length;
    char free_node_string;
    rxinteger int_value;
    int bool_value;
    double float_value;
    char* decimal_value; /* Decimal value as a string */
    int exit_obj_reg; /* VM register index of the attached Exit object
        */
    /* These are only valid after the set_source_location walker has
        run */
    Token *token_start, *token_end;
    char *source_start, *source_end;
    int line, column;
    SymbolNode *symbolNode;
    /* These are used by the code emitters */
    OutputFragment *output;          /* Primary node output or loop
        assign / init instruction */
    OutputFragment *cleanup;         /* Clean up logic */
    OutputFragment *loopstartchecks; /* Begin Loop exit checks */
    OutputFragment *loopinc;         /* Loop increments */
    OutputFragment *loopendchecks;   /* End Loop exit checks */
};
```

A.7 Abstract Syntax Tree: Scope & Block Construction

A.7.1 Mapping `DO` loops from Lemon Parser to AST

In the compiler, block scopes such as `DO` loops are strictly translated from Lemon grammar tokens into a parent-child AST topology.

Example from `compiler/rxcpbgmr.y`:

```
tk_doloop(D)  ::= TK_DO(T).
                  { D = ast_f(context, DO, T); }
do(G)         ::= tk_doloop(T) dorep(R) TK_EOC instruction_list(I)
    TK_END.
                  { G = T; add_ast(G,R); add_ast(G,I); }
```

1. **Root Creation**: When a `TK_DO` token is identified, a new `ASTNode` of type `DO` is instantiated (via `ast_f`).
2. **Repetition and Conditions**: Sub-rules process the `dorep` (like `TO`, `BY`, `FOR` attributes) into a `REPEAT` AST node or `docond` (like `WHILE` / `UNTIL`).

3. **Block Body**: The `instruction_list(I)` contains all expressions and assignments defined within the loop body.
4. **Tree Assembly**: The `REPEAT` clause, `WHILE/UNTIL` conditions, and the actual loop body instructions are iteratively appended as **child nodes** to the parent `DO` node using the `add_ast(parent, child)` and `add_sbtr(older_sibling, younger_sibling)` C functions.
5. **Association**: The AST node also uses the `association` pointer to link commands like `LEAVE` or `ITERATE` directly back to the target enclosing `DO` loop node.

A.8 Execution and the VM

Once compilation via `rxc` and `rxas` is complete, `rxvm` handles the execution. Modules are ingested into memory mapping via functions like `rxldmod`. The VM spins up its contexts, loading dynamically or statically linked extensions, and invokes `rxvm_run` to march through and execute the virtual CPU instructions matching the loaded byte sequence.

B

Platform Considerations

B.1 Instructions

All VM instructions work in exactly the same way on all supported platforms - for numerical algorithms, within the limitations of the available hardware.

B.2 Bytecode and layout of binary modules

The binary file layout containing the VM bytecode instructions is identical over all supported platforms. This means that the `.rxbin` files can be transported to other platforms, for example from Linux on arm64 to Windows on X86_64 and to Linux on arm64, z390x or RISC-V, and work unchanged.

B.3 Native executables

This portability does not hold for native executables. These are `.rxbin` files which are preprocessed by the `rxcpack` command into `.c` source which are then compiled and linked by platform specific compiler toolchains. These are usable on only one instruction set - operating system combination.

The same goes for `.rxplugin` files; these are also native executables.

B.4 64-bit limitation

CREXX is not intended for, or is expected to work on 32-bit architectures. Certain IBM legacy architectures, like s/370, might work when explicitly indicated.

C

Notices

D

Instructions by Mnemonic

Opcode	Instruction	parameters
18	ADDF	REG,REG,REG
19	ADDF	REG,REG,FLOAT
03	ADDI	REG,REG,REG
04	ADDI	REG,REG,INT
90	AMAP	REG,REG
91	AMAP	REG,INT
88	AND	REG,REG,REG
47	APPEND	REG,REG
44	APPENDCHAR	REG,REG
D5	BCF	ID,REG
D6	BCF	ID,REG,REG
D1	BCT	ID,REG
D2	BCT	ID,REG,REG
D3	BCTNM	ID,REG
D4	BCTNM	ID,REG,REG
E1	BEQ	ID,REG,REG
E2	BEQ	ID,REG,INT
D9	BGE	ID,REG,REG
DA	BGE	ID,REG,INT
D7	BGT	ID,REG,REG
D8	BGT	ID,REG,INT
DD	BLE	ID,REG,REG
DE	BLE	ID,REG,INT
DB	BLT	ID,REG,REG
DC	BLT	ID,REG,INT
DF	BNE	ID,REG,REG
E0	BNE	ID,REG,INT

A4	BR	ID
A6	BRF	ID,REG
A5	BRT	ID,REG
A7	BRTF	ID,ID,REG
EE	BRTPANDT	ID,REG,INT
ED	BRTPT	ID,REG
9C	CALL	REG,FUNC
9D	CALL	REG,FUNC,REG
9B	CALL	FUNC
CE	CNOP	NO OPERAND
41	CONCAT	REG,REG,REG
42	CONCAT	REG,REG,STRING
43	CONCAT	REG,STRING,REG
45	CONCCHAR	REG,REG,REG
AA	COPY	REG,REG
2B	DEC	REG
2D	DEC0	NO OPERAND
2F	DEC1	NO OPERAND
31	DEC2	NO OPERAND
27	DIVF	REG,REG,REG
28	DIVF	REG,REG,FLOAT
29	DIVF	REG,FLOAT,REG
11	DIVI	REG,REG,REG
12	DIVI	REG,REG,INT
C9	DROPCHAR	REG,REG,REG
BB	EXIT	NO OPERAND
BC	EXIT	REG
BD	EXIT	INT
16	FADD	REG,REG,REG
17	FADD	REG,REG,FLOAT
AC	FCOPY	REG,REG
24	FDIV	REG,REG,REG
26	FDIV	REG,FLOAT,REG
25	FDIV	REG,REG,FLOAT
67	FEQ	REG,REG,FLOAT
66	FEQ	REG,REG,REG
C5	FFORMAT	REG,REG,REG
6A	FGT	REG,REG,REG
6B	FGT	REG,REG,FLOAT
6C	FGT	REG,FLOAT,REG
6E	FGTE	REG,REG,FLOAT
6D	FGTE	REG,REG,REG
6F	FGTE	REG,FLOAT,REG
71	FLT	REG,REG,FLOAT
70	FLT	REG,REG,REG

```
72      FLT             REG,FLOAT,REG
75      FLTE            REG,FLOAT,REG
73      FLTE            REG,REG,REG
74      FLTE            REG,REG,FLOAT
20      FMULT           REG,REG,REG
21      FMULT           REG,REG,FLOAT
E3      FNDBLNK         REG,REG,REG
E4      FNDNBLNK        REG,REG,REG
68      FNE             REG,REG,REG
69      FNE             REG,REG,FLOAT
E6      FSEX            REG
1A      FSUB            REG,REG,REG
1B      FSUB            REG,REG,FLOAT
1C      FSUB            REG,FLOAT,REG
C2      FTOB            REG
C1      FTOI            REG
BF      FTOS            REG
CD      GETBYTE         REG,REG,REG
54      GETSTRPOS       REG,REG
E7      GETTP           REG,REG
94      GMAP            REG,REG
95      GMAP            REG,STRING
51      HEXCHAR         REG,REG,REG
01      IADD            REG,REG,REG
02      IADD            REG,REG,INT
32      IAND            REG,REG,REG
33      IAND            REG,REG,INT
AB      ICOPY           REG,REG
0E      IDIV            REG,REG,REG
0F      IDIV            REG,REG,INT
10      IDIV            REG,INT,REG
56      IEQ             REG,REG,REG
57      IEQ             REG,REG,INT
5A      IGT             REG,REG,REG
5B      IGT             REG,REG,INT
5C      IGT             REG,INT,REG
5D      IGTE            REG,REG,REG
5E      IGTE            REG,REG,INT
5F      IGTE            REG,INT,REG
60      ILT             REG,REG,REG
61      ILT             REG,REG,INT
62      ILT             REG,INT,REG
63      ILTE            REG,REG,REG
64      ILTE            REG,REG,INT
65      ILTE            REG,INT,REG
```

13	IMOD	REG,REG,REG
15	IMOD	REG,INT,REG
14	IMOD	REG,REG,INT
0A	IMULT	REG,REG,REG
0B	IMULT	REG,REG,INT
2A	INC	REG
2C	INC0	NO OPERAND
2E	INC1	NO OPERAND
30	INC2	NO OPERAND
58	INE	REG,REG,REG
59	INE	REG,REG,INT
3C	INOT	REG,REG
3D	INOT	REG,INT
34	IOR	REG,REG,REG
35	IOR	REG,REG,INT
CF	IPOW	REG,REG,REG
D0	IPOW	REG,REG,INT
E5	ISEX	REG
38	ISHL	REG,REG,REG
39	ISHL	REG,REG,INT
3A	ISHR	REG,REG,REG
3B	ISHR	REG,REG,INT
05	ISUB	REG,REG,REG
06	ISUB	REG,REG,INT
07	ISUB	REG,INT,REG
C0	ITOF	REG
BE	ITOS	REG
36	IXOR	REG,REG,REG
37	IXOR	REG,REG,INT
AE	LINK	REG,REG
B1	LOAD	REG,INT
B4	LOAD	REG,CHAR
B3	LOAD	REG,STRING
B2	LOAD	REG,FLOAT
EB	LOADSETTP	REG,STRING,INT
EA	LOADSETTP	REG,FLOAT,INT
E9	LOADSETTP	REG,INT,INT
8E	MAP	REG,REG
8F	MAP	REG,STRING
A8	MOVE	REG,REG
8C	MTIME	REG
22	MULTF	REG,REG,REG
23	MULTF	REG,REG,FLOAT
0C	MULTI	REG,REG,REG
0D	MULTI	REG,REG,INT

```
8A      NOT             REG,REG
98      NSMAP           REG,STRING,STRING
96      NSMAP           REG,REG,REG
99      NSMAP           REG,STRING,REG
97      NSMAP           REG,REG,STRING
B0      NULL            REG
89      OR              REG,REG,REG
CC      PADSTR          REG,REG,REG
92      PMAP            REG,REG
93      PMAP            REG,STRING
52      POSCHAR         REG,REG,REG
9E      RET             NO OPERAND
9F      RET             REG
A0      RET             INT
A1      RET             FLOAT
A2      RET             CHAR
A3      RET             STRING
78      RSEQ            REG,REG,REG
79      RSEQ            REG,REG,STRING
46      SAPPEND         REG,REG
B5      SAY             REG
B7      SAY             INT
B8      SAY             FLOAT
BA      SAY             CHAR
B9      SAY             STRING
3E      SCONCAT         REG,REG,REG
3F      SCONCAT         REG,REG,STRING
40      SCONCAT         REG,STRING,REG
AD      SCOPY           REG,REG
76      SEQ             REG,REG,REG
77      SEQ             REG,REG,STRING
EC      SETORTP         REG,INT
53      SETSTRPOS       REG,REG
E8      SETTP           REG,INT
7C      SGT             REG,REG,REG
7D      SGT             REG,REG,STRING
7E      SGT             REG,STRING,REG
7F      SGTE            REG,REG,REG
80      SGTE            REG,REG,STRING
81      SGTE            REG,STRING,REG
82      SLT             REG,REG,REG
83      SLT             REG,REG,STRING
84      SLT             REG,STRING,REG
85      SLTE            REG,REG,REG
86      SLTE            REG,REG,STRING
```

```
87      SLTE          REG,STRING,REG
7B      SNE           REG,REG,STRING
7A      SNE           REG,REG,REG
B6      SSAY          REG
C3      STOF          REG
C4      STOI          REG
50      STRCHAR       REG,REG
4F      STRCHAR       REG,REG,REG
4E      STRLEN        REG,REG
C6      STRLOWER      REG,REG
C7      STRUPPER      REG,REG
1D      SUBF          REG,REG,REG
1E      SUBF          REG,REG,FLOAT
1F      SUBF          REG,FLOAT,REG
08      SUBI          REG,REG,REG
09      SUBI          REG,REG,INT
CB      SUBSTCUT      REG,REG
55      SUBSTR        REG,REG,REG
CA      SUBSTRING     REG,REG,REG
A9      SWAP          REG,REG
8B      TIME          REG
C8      TRANSCHAR     REG,REG,REG
48      TRIML         REG,REG
4B      TRIML         REG,REG,REG
49      TRIMR         REG,REG
4C      TRIMR         REG,REG,REG
4A      TRUNC         REG,REG
4D      TRUNC         REG,REG,REG
AF      UNLINK        REG
9A      UNMAP         REG
8D      XTIME         REG,STRING
```

E

Instructions by Opcode

Opcode	Instruction	parameters
01	IADD	REG,REG,REG
02	IADD	REG,REG,INT
03	ADDI	REG,REG,REG
04	ADDI	REG,REG,INT
05	ISUB	REG,REG,REG
06	ISUB	REG,REG,INT
07	ISUB	REG,INT,REG
08	SUBI	REG,REG,REG
09	SUBI	REG,REG,INT
0A	IMULT	REG,REG,REG
0B	IMULT	REG,REG,INT
0C	MULTI	REG,REG,REG
0D	MULTI	REG,REG,INT
0E	IDIV	REG,REG,REG
0F	IDIV	REG,REG,INT
10	IDIV	REG,INT,REG
11	DIVI	REG,REG,REG
12	DIVI	REG,REG,INT
13	IMOD	REG,REG,REG
14	IMOD	REG,REG,INT
15	IMOD	REG,INT,REG
16	FADD	REG,REG,REG
17	FADD	REG,REG,FLOAT
18	ADDF	REG,REG,REG
19	ADDF	REG,REG,FLOAT
1A	FSUB	REG,REG,REG
1B	FSUB	REG,REG,FLOAT

```
1C      FSUB          REG,FLOAT,REG
1D      SUBF          REG,REG,REG
1E      SUBF          REG,REG,FLOAT
1F      SUBF          REG,FLOAT,REG
20      FMULT         REG,REG,REG
21      FMULT         REG,REG,FLOAT
22      MULTF         REG,REG,REG
23      MULTF         REG,REG,FLOAT
24      FDIV          REG,REG,REG
25      FDIV          REG,REG,FLOAT
26      FDIV          REG,FLOAT,REG
27      DIVF          REG,REG,REG
28      DIVF          REG,REG,FLOAT
29      DIVF          REG,FLOAT,REG
2A      INC           REG
2B      DEC           REG
2C      INC0          NO OPERAND
2D      DEC0          NO OPERAND
2E      INC1          NO OPERAND
2F      DEC1          NO OPERAND
30      INC2          NO OPERAND
31      DEC2          NO OPERAND
32      IAND          REG,REG,REG
33      IAND          REG,REG,INT
34      IOR           REG,REG,REG
35      IOR           REG,REG,INT
36      IXOR          REG,REG,REG
37      IXOR          REG,REG,INT
38      ISHL          REG,REG,REG
39      ISHL          REG,REG,INT
3A      ISHR          REG,REG,REG
3B      ISHR          REG,REG,INT
3C      INOT          REG,REG
3D      INOT          REG,INT
3E      SCONCAT       REG,REG,REG
3F      SCONCAT       REG,REG,STRING
40      SCONCAT       REG,STRING,REG
41      CONCAT        REG,REG,REG
42      CONCAT        REG,REG,STRING
43      CONCAT        REG,STRING,REG
44      APPENDCHAR    REG,REG
45      CONCCHAR      REG,REG,REG
46      SAPPEND       REG,REG
47      APPEND        REG,REG
48      TRIML         REG,REG
```

```
49      TRIMR         REG,REG
4A      TRUNC         REG,REG
4B      TRIML         REG,REG,REG
4C      TRIMR         REG,REG,REG
4D      TRUNC         REG,REG,REG
4E      STRLEN        REG,REG
4F      STRCHAR       REG,REG,REG
50      STRCHAR       REG,REG
51      HEXCHAR       REG,REG,REG
52      POSCHAR       REG,REG,REG
53      SETSTRPOS     REG,REG
54      GETSTRPOS     REG,REG
55      SUBSTR        REG,REG,REG
56      IEQ           REG,REG,REG
57      IEQ           REG,REG,INT
58      INE           REG,REG,REG
59      INE           REG,REG,INT
5A      IGT           REG,REG,REG
5B      IGT           REG,REG,INT
5C      IGT           REG,INT,REG
5D      IGTE          REG,REG,REG
5E      IGTE          REG,REG,INT
5F      IGTE          REG,INT,REG
60      ILT           REG,REG,REG
61      ILT           REG,REG,INT
62      ILT           REG,INT,REG
63      ILTE          REG,REG,REG
64      ILTE          REG,REG,INT
65      ILTE          REG,INT,REG
66      FEQ           REG,REG,REG
67      FEQ           REG,REG,FLOAT
68      FNE           REG,REG,REG
69      FNE           REG,REG,FLOAT
6A      FGT           REG,REG,REG
6B      FGT           REG,REG,FLOAT
6C      FGT           REG,FLOAT,REG
6D      FGTE          REG,REG,REG
6E      FGTE          REG,REG,FLOAT
6F      FGTE          REG,FLOAT,REG
70      FLT           REG,REG,REG
71      FLT           REG,REG,FLOAT
72      FLT           REG,FLOAT,REG
73      FLTE          REG,REG,REG
74      FLTE          REG,REG,FLOAT
75      FLTE          REG,FLOAT,REG
```

```
76      SEQ         REG,REG,REG
77      SEQ         REG,REG,STRING
78      RSEQ        REG,REG,REG
79      RSEQ        REG,REG,STRING
7A      SNE         REG,REG,REG
7B      SNE         REG,REG,STRING
7C      SGT         REG,REG,REG
7D      SGT         REG,REG,STRING
7E      SGT         REG,STRING,REG
7F      SGTE        REG,REG,REG
80      SGTE        REG,REG,STRING
81      SGTE        REG,STRING,REG
82      SLT         REG,REG,REG
83      SLT         REG,REG,STRING
84      SLT         REG,STRING,REG
85      SLTE        REG,REG,REG
86      SLTE        REG,REG,STRING
87      SLTE        REG,STRING,REG
88      AND         REG,REG,REG
89      OR          REG,REG,REG
8A      NOT         REG,REG
8B      TIME        REG
8C      MTIME       REG
8D      XTIME       REG,STRING
8E      MAP         REG,REG
8F      MAP         REG,STRING
90      AMAP        REG,REG
91      AMAP        REG,INT
92      PMAP        REG,REG
93      PMAP        REG,STRING
94      GMAP        REG,REG
95      GMAP        REG,STRING
96      NSMAP       REG,REG,REG
97      NSMAP       REG,REG,STRING
98      NSMAP       REG,STRING,STRING
99      NSMAP       REG,STRING,REG
9A      UNMAP       REG
9B      CALL        FUNC
9C      CALL        REG,FUNC
9D      CALL        REG,FUNC,REG
9E      RET         NO OPERAND
9F      RET         REG
A0      RET         INT
A1      RET         FLOAT
A2      RET         CHAR
```

```
A3      RET         STRING
A4      BR          ID
A5      BRT         ID,REG
A6      BRF         ID,REG
A7      BRTF        ID,ID,REG
A8      MOVE        REG,REG
A9      SWAP        REG,REG
AA      COPY        REG,REG
AB      ICOPY       REG,REG
AC      FCOPY       REG,REG
AD      SCOPY       REG,REG
AE      LINK        REG,REG
AF      UNLINK      REG
B0      NULL        REG
B1      LOAD        REG,INT
B2      LOAD        REG,FLOAT
B3      LOAD        REG,STRING
B4      LOAD        REG,CHAR
B5      SAY         REG
B6      SSAY        REG
B7      SAY         INT
B8      SAY         FLOAT
B9      SAY         STRING
BA      SAY         CHAR
BB      EXIT        NO OPERAND
BC      EXIT        REG
BD      EXIT        INT
BE      ITOS        REG
BF      FTOS        REG
C0      ITOF        REG
C1      FTOI        REG
C2      FTOB        REG
C3      STOF        REG
C4      STOI        REG
C5      FFORMAT     REG,REG,REG
C6      STRLOWER    REG,REG
C7      STRUPPER    REG,REG
C8      TRANSCHAR   REG,REG,REG
C9      DROPCHAR    REG,REG,REG
CA      SUBSTRING   REG,REG,REG
CB      SUBSTCUT    REG,REG
CC      PADSTR      REG,REG,REG
CD      GETBYTE     REG,REG,REG
CE      CNOP        NO OPERAND
CF      IPOW        REG,REG,REG
```

D0	IPOW	REG,REG,INT
D1	BCT	ID,REG
D2	BCT	ID,REG,REG
D3	BCTNM	ID,REG
D4	BCTNM	ID,REG,REG
D5	BCF	ID,REG
D6	BCF	ID,REG,REG
D7	BGT	ID,REG,REG
D8	BGT	ID,REG,INT
D9	BGE	ID,REG,REG
DA	BGE	ID,REG,INT
DB	BLT	ID,REG,REG
DC	BLT	ID,REG,INT
DD	BLE	ID,REG,REG
DE	BLE	ID,REG,INT
DF	BNE	ID,REG,REG
E0	BNE	ID,REG,INT
E1	BEQ	ID,REG,REG
E2	BEQ	ID,REG,INT
E3	FNDBLNK	REG,REG,REG
E4	FNDNBLNK	REG,REG,REG
E5	ISEX	REG
E6	FSEX	REG
E7	GETTP	REG,REG
E8	SETTP	REG,INT
E9	LOADSETTP	REG,INT,INT
EA	LOADSETTP	REG,FLOAT,INT
EB	LOADSETTP	REG,STRING,INT
EC	SETORTP	REG,INT
ED	BRTPT	ID,REG
EE	BRTPANDT	ID,REG,INT

List of Tables

Index

ISBN 978-94-039116-6-3