CREXX
VM Specification

The CREXX team

June 30, 2024

THE REXX LANGUAGE ASSOCIATION
CREXX Programming Series
ISBN 978-90-819090-1-3

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 978-90-819090-1-3

ISBN 978-90-819090-1-3

9 789081 909013 >

Contents

The CREXX Programming 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.

User Guide	This guide is meant for an audience that has done some programming and wants to start quickly. It starts with a quick tour of the language, and a section on installing the CREXX translator and how to run it. It also contains help for troubleshooting if anything in the installation does not work as designed, and states current limits and restrictions of the open source reference implementation.
Application Programming Guide	The Application 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.
The CREXX 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 REXX publications:

- Body text is in this font
- Examples of language statements are in a `keyword` or **bold** type
- Variables or strings as mentioned in source code, or things that appear on the console, are in a `typewriter` 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'
```

About This Book

The crexx language is a further development, and variant of the Rexx language[1]. This book aims to document the workings of this implementation and serves as reference for users and implementors alike.

History

mvp This version documents the Minimally Viable Product release, Q1 2022 and is intended for developers only. It documents the RXAS instruction set and crexx level B, which is a typed subset of Classic Rexx.

f0047 First version, Git feature [F0047]

[1] Cowlishaw, 1979

1

Building CREXX

1.1 Building the CREXX toolchain

This paragraph aims to show all you need to know about how to build cRexx from scratch, and then run it. It will show what you need, what to do and when it is build, how to make working programs with it.

1.2 Requirements

Currently cRexx is known to build on Windows, macOS and Linux[2]. You need one of those and:

Tool	Function
Git	source code version management
CMake	build tool
Rexx	used during build process (brexx, ooRexx, Regina will all do)
C compiler	gcc, clang (install g++ as some C++ elements are used)
Bison	parser generator
Make	conventional build tool **or**
Ninja	fast build tool

1.3 Platform specific info

On Linux and macOS, this instruction is identical. For macOS, Xcode batch tools need to be installed, which will provide you with git, make and the compiler. Brew will give easy access to regina-rexx[3] and Ninja-build.

On Windows, you will need a compatibility layer like msys - installing this has the additional advantage of easy access to git, gcc, cmake and the rest of the necessary tools. On more modern Windows, the WSL[4] and Ubuntu is not a bad choice.

[2] There is a separate instruction for VM/370 (and later) mainframe operating systems.

[3] ooRexx and brexx will also work, one of those needs to be on the path. For Linux, you will need to install git (which will be there on most distributions), cmake and gcc or clang.

[4] WSL: Windows subsystem for Linux.

1.4 Process

Here it is assumed that all tools are installed and working, and available on your PATH environment variable.

Choose or make a suitably named directory on your system to contain the source code. Note that the cRexx source is kept in a different directory on you system than where it is built in, or will run from. Now run this command:

```
git clone https://github.com/adesutherland/CREXX.git
```

This will give you a CREXX subdirectory in the current directory, containing the source of cRexx and its dependencies. This is the 'develop' branch, which is the one you would normally want to use. All of these are written in the C99 version of the C programming language, which should be widely supported by C compilers on most platforms.

Make a new subdirectory in the current directory (not in CREXX, but in the one that contains it), like 'crexx-build'.

```
mkdir crexx-build
```

and cd into that directory. Now issue the following command (we assume that you installed ninja, otherwise subsitute 'make' for the two instances of 'ninja'):

```
cmake -G Ninja -DCMAKE_BUILD_TYPE=Release ../CREXX && ninja && ctest
```

This will do a lot of things. In fact, if all goes well, you will have a built and tested cRexx system.

1.5 Explaining the build process

Let's zoom in a little on what we did. The first step is to tell CMake to validate your system environment and generate a build script (and a test script) for the chosen build tool. Cmake will read the file CmakeLists.txt and validate that your system can do what it asks it to do. This can yield error messages, for example if the C compiler lacks certain functions or header files. (When that happens, open an issue and someone will have a look at it - or peruse stack overflow which is what we probably also will do).

After CMake has successfully validated the build environment, it will generate a build script (a Makefile in the case of Make and a build.ninja file in the case of Ninja). This is specified after the '-G' flag. The '-DCMAKE_BUILD_-TYPE=Release' flag makes sure we do an optimized build, which means we specify an '-o3' flag to the C compiler, which then will spend some time optimizing the executable modules, which makes them run faster (they do!). The alternative is a 'debug' build which will yield slower executables, but with more debugging information in them.

The two ampersands (&&) mean we do the next part only if the previous step was successful. This is a 'ninja' statement, which will build everything in the

build.ninja specification file. These are a lot of parts, and the good news is, when they are built once, only the changed source will be built, which will be fast.

After this, the generated test suite is run with the 'ctest' command. This knows what to do, and will show you successes and failures. If what you checked out if git is not a released version, there is a small change that some test cases fail, but generally these should indicate success.

1.6 Use of Rexx to build cRexx

Rexx code is used twice during the build process. The first time is in an early stage: as the Rexx VM 'machine code' instructions are generated from a file, a Rexx script needs to work on these to generate two C sources. (The Rexx engine for this needs to be in working order on the building system - this is one of the things CMake checks, and it can flag down the build if it cannot locate a working 'rexx' executable).

The second time Rexx is used, it is already our working cRexx instance: the library of built-in functions is written in Rexx (and Rexx Assembler) and needs to be compiled (carefully observing the dependencies on other Rexx built-in functions) before it is added to the library and the cRexx executables.

1.7 What do we have after a successful build process

When all went well, we have a set of native executables for the platform we built cRexx on. These are

Name	Function
rxc	cRexx compiler
rxas	cRexx assembler
rxdas	cRexx disassembler
rxvm	cRexx virtual machine
rxbvm	cRexx virtual machine, non-threaded version
rxvme	cRexx virtual machine with linked-in Rexx library
rxdb	cRexx debugger
rxcpack	cRexx C-generator for native executables

You read that right, cRexx already can compile your Rexx script into an executable file, that can be run standalone, for example, on a computer that has no cRexx and/or C compiler installed.

Another salient fact from the above table is that 'rxdb', the cRexx debugger, is entirely written in Rexx and compiled and packaged into an executable file.

The executables in the above table need to be on the PATH environment variable. These are to be found in the compiler, assembler, disassembler, debugger and cpacker directories of the crexx-build directory we created earlier. It is up to you

to add all these separately to the PATH environment, or to just collect them all into one directory that is already on the PATH.

2

Low-Level System Information

2.1 Register based Virtual Machine

CREXX uses a register based virtual machine. A uses registers, which are small, fast, on-chip memory locations, to store and manipulate data. In a register-based virtual machine, the instructions explicitly reference the registers by number, and the values are loaded into and out of the registers as needed. This approach can provide faster performance because accessing registers is typically faster than accessing memory. In the *rxvm* virtual machine, the compiler of the underlying C implementation assigns the hardware registers.

In contrast, a uses a stack to store and manipulate data. In a stack-based virtual machine, the operands are pushed onto the stack, and the operators pop them off the stack, perform the operation, and push the result back onto the stack. The stack-based approach is simpler than the register-based approach because there are no explicit references to registers in the instructions, and the operands are automatically managed by the stack. However, the stack-based approach can be slower than the register-based approach because accessing memory is typically slower than accessing registers.

Some other differences between a register-based virtual machine and a stack-based virtual machine include:

- Register-based virtual machines typically use more memory for the registers than stack-based virtual machines use for the stack.
- Register-based virtual machines may require more complex instruction encoding and decoding logic than stack-based virtual machines, which can impact the size and complexity of the virtual machine implementation.
- Stack-based virtual machines can be easier to implement and optimize for certain types of operations, such as function calls and loops, where the data can be easily pushed onto and popped off the stack.
- Register-based virtual machines can provide more flexibility in terms of register allocation and optimization, which can be particularly important for high-performance applications.

The execution environment of a CREXX program is a threaded[5] virtual machine that is designed for optimal performance. This virtual machine, implemented in the *rxvm* executable, executes machine instructions produced by the *rxc* CREXX compiler, or written by hand, assembled into an .rxbin binary file by the *rxas*

[5] an alternative, non-threaded executable is available under the name

assembler.

2.2 REXX VM Specification

Page Status: This page is work in progress, incomplete, inconsistent and full of errors ... read with care!

2.2.1 Acknowledgement

Much of this work is based on the excellent "Language Implementation Patterns" by Terence Parr.

https://pragprog.com/titles/tpdsl/language-implementation-patterns/

2.2.2 REXX VM Components - Summary

The REXX interpreter simulates a computer with the following components:

- Code memory: This 64 or 32-bit unsigned integer array holds a program's "bytecode" instructions (bytecodes plus operands). Addresses are integers.
- ip register: The instruction pointer is a special-purpose "register" that points into code memory at the next instruction to execute.
- CPU: To execute instructions, the interpreter has a simulated CPU that conceptually amounts to a loop around a giant "switch on bytecode" statement. This is called the instruction dispatcher. We will use a Threaded VM instead.
- Constant pool: Anything that we can't store as an integer operand goes into the constant pool. This includes strings, arbitrary precision numbers, and function symbols. Instructions like `say [STRING]` and use an index into the constant pool instead of the actual operand.
- Function call stack: The interpreter has a stack to hold function call return addresses as well as parameters and local variables.
- fp register: A special-purpose register called the frame pointer that points to the top of the function call stack. StackFrame represents the information we need to invoke functions.
- An infinite and regular register set per function call. Functions can access any element in the register array, whereas a stack can only access elements at the end. Registers hold integer, float, string and object values, plus flags that can be used to dynamically store which value formats are valid.

The following capabilities will be implemented in REXX Level B - REXX called by the virtual machine

- Variable Pool: Holds slots for variables, each function/procedure has its own pool, and a function/procedure can "link" to variables from the parents pool (to support dynamic expose scenarios). The memory slots can point

at Integer, Float, String, and struct instances. Variables are accessed via a linked register. This linking can be done statically or dynamically via a variable search capability
- Runtime libraries that care used as "exits"; for example, REXX code to format error messages.

2.2.3 REXX Machine Architecture

cREXX will target a bytecode interpreter tuned for the specific needs of the REXX language. This has two advantages

- Provides platform independence and portability
- Provides a much simpler target for the cREXX compile compared to real CPU's instruction sets (which after all have to be implemented in hardware)

A bytecode interpreter is like a real processor. The biggest difference is that bytecode instruction sets are much simpler. Also, for example, we can assume there are an infinite number of registers.

The cREXX VM is a register based bytecode interpreter that uses Threading and Super-Instructions, meaning:

Register Based

A Register Based VM instructions use registers. The cREXX VM gives each stack frame an “infinite” set of registers.

Threading

Threading means that the "bytecode" loader converts instruction opcodes to the address of the subroutine that emulates the instruction. This reduces an indirection to each instruction (i.e. instead of `switch(opcode) {case ...}`) we can do `goto opcode_address`).

In addition, the code to dispatch to the next instruction is repeated at the end of each instruction subroutine rather than having a single dispatcher. This allows the real CPU's pipelining logic to work better.

The following two examples demonstrate the concept.

Traditional Interpreter

```
char code[] = {
  ICONST_1, ICONST_2,
  IADD, ...
}
char *pc = code;

/* dispatch loop */
```

```
while(true) {
  switch(*pc++) {
    case ICONST_1: *++sp = 1; break;
    case ICONST_2: *++sp = 2; break;
    case IADD:
      sp[-1] += *sp; --sp; break;
   ...
  }
}
```

Equivalent Threaded Interpreter

```
void *code[] = {
  &&ICONST_1, &&ICONST_2,
  &&IADD, ...
}
void **pc = code;

/* implementations */
goto **(pc);

ICONST_1: pc++; *++sp = 1; goto **(pc);
ICONST_2: pc++; *++sp = 2; goto **(pc);
IADD:
  pc++; sp[-1] += *sp; --sp; goto **(pc);
...
```

Note how the next instruction pointer `pc` is calculated up front (`pc++`) before the instruction logic. This is to allow the CPU pipeline to work; by the time the `goto` is being decoded the value of `pc` *should* have completed.

Super Instructions

In any interpreter the dispatch to the next instruction is an overhead. The fewer instructions needed to execute logic then the less dispatching overhead. Moreover a complex instruction's native implementation code running linearly maximises the effectiveness of the real CPU's pipeline.

We therefore will have a large number of instructions to cater for common sequences, for example a decrement (decr) instruction might often be followed by a branch if not zero instruction (brnz), e.g. for a loop - a super-instruction combines these two instructions to one (e.g. decrbrnz).

Low-level Function Instructions

The VM provides low-level functions as instruction, for example covering aspects like variable manipulation (like substr()), IO functions, Environment access, and virtual hardware like the timer. This ensures that:

- Platform independence / platform drivers / porting etc. is achieved only by implementing these instructions for each platform.
- Higher level functionality can all be implemented in REXX.

Instruction coding

There is a balance here - memory usage verses performance, made more complicated because reducing memory usage also has performance benefits. Tests will be needed across different CPUs types to validate our optimisations.

Compiled Mode REXX VM binary code can be generated for storage (i.e. compiled) to be loaded and run later. In this case the code will have to be loaded and then threaded.

The threading process converts the op_codes into addresses, branches to addresses constant indexes to addresses - everything is converted into real addresses of the machine running the REXX VM, so that when executed (for example) data is accessed via its real address with no lookups/indexes needed. This needs to be all done at load time as the exact real addresses will be different on each machine and for each run (modern OS's randomise load addresses).

We will investigate if there is value in compressing the saved REXX VM binary to speed load times. Java uses ZIP for this and cREXX could do the same - and/or we could pack the instruction coding.

In the short term we will save it as 32 or 64 bit instructions (see following).

This means that the threading process can simply replace the opcode with the opcode address - the 32 or 64 bit memory address can fit in the 32 or 64 bit int instruction code location, we overwrite the opcode integer with the opcode function address.

Interpreter Mode When interpreting a REXX program the compiler will emit code that will be executed there and then. In principle all the real addresses will be known and this means that the compiler can generate threaded code from the get-go. This way the loading / threading stage can be skipped.

Initial versions will only use the Compiler Mode while the Interpreter Mode is checked for feasibility and to see if there is a noticeable performance gain. One issue we may have is that the loader/threader may be needed anyway for other reasons like late binding/linking of libraries.

Endianness Where the compiler and runtime are known to be on the same machine then the Endianness and float format will be machine specific.

Where the REXX VM is to be saved then big-endianness / "network order" will be used. And for floats we will use big-endianness double (IEEE 754 / binary64).

The loader / threader will need to convert as need be.

32 / 64 bit Although we are only expecting a few hundred opcodes (if that!!), we need to store them in an integer of the same size as the machine address pointer size because of the need to overwrite the op_code with its implementation address.

For a 64bit OS we therefore need to store all opcodes and register numbers as 64-bit integers.

Note: If not for threading, there could have been many schemes to compress this, for example a 64bit integer could have held a 16-bit opcode and 3 16-bit register numbers as parameters - very efficient. Therefore we will confirm that the threaded interpreter is still faster than a classic interpreter using this more efficient scheme. Modern CPU's may be much smarter at pipelining!

The 64 bit machine code format is therefore

- A 64bit Opcode, followed by
- Zero or more (depending on opcode) 64bit parameters. Each parameter (depending on the opcode) can be one of:
  - An integer constant
  - An double float constant
  - Index to an item in the constant pool (e.g. a String, High-precision Number or Object Constant)
  - Register Number
  - An instruction Number (i.e. a virtual address for a jump)

For 32-bit machines we can use two options

1. Have an alternative but equivalent format using 32bit integers/floats/addresses. This reduces the size of binaries but reduces compatability between 64-bit and 32-bit machines. 2 versions of binaries may be needed, or the loader could covert on load. An issue is that Float constants would need to be put in the Constant pool.
2. Use the 64-bit format for 32 bit machines. This means that 50% of the program space will be wasted but will provide compatibility. Note that only 4,294,967,295 registers per stack frame will be supported!

We will explore the different options here including looking at program size and performance.

2.2.4 Standards/Rules for VM Instruction Implementation

The following are the rules that should be followed when implementing VM Instructions.

Key Principles

1. To maximise CPU performance with respect to speculative execution (including simple pipelining) branches should be avoided and if required (e.g. when

dispatching to the next instruction) the target address should be calculated and assigned as early as possible.
2. The compiler (or human writing the assembler code) is responsible for producing valid assembler including ensuring that assembler registers are initiated and have the required type, and ensuring (for example) then bound checks have been done. Run time validation at the VM Instruction levels should be avoided except for

- When in debug mode
- When "safe" versions of instructions are used by the compiler - see later.

Badly formed REXX Assembler code will have undefined runtime behaviour.

Justification for this approach is that SPECTRE attacks mean that it is possible for a program to access all process memory in a VM despite any access checks implemented by the VM (see Google's SPECTRE JavaScript demonstrator).

For information, a SPECTRE attack uses the non-functional performance improvement of a data read associated with a speculative execution of an invalid instruction that moved data into the L1 cache. My assessment (and that of security advice) is that designers should assume untrusted code can read across an entire process memory space (and across a CPU core too) and that countermeasures will have limited impact because of the fundamental need for the performance boost provided by speculative execution and caches (we are not giving those up!)

Therefore when deploying cREXX code in a secure manner each VM instance will need to be in its own process space - and that is the way we will need to protect against nefarious code. All the cleaver work done in the Java and .NET VMs to prove code is safe might very well be proved to be pointless :-(

Rules

1. Calculate the next instruction address as early as possible.
2. Do not use "if" statements and validation
3. Conversion errors should cause SIGNALS ("goto SIGNAL;")
4. When needed by the compiler we will define large instructions that combine several instructions into one - this avoid multiple instruction dispatches.
5. Use DEBUG Macros to store debug validation - these will be not included in production/release compiled code.

2.3 cREXX Virtual Machine instructions

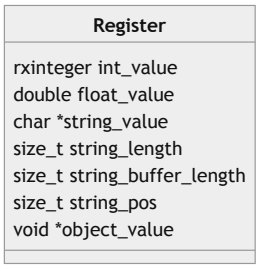
This section describes the processor-specific information for . The instruction set can be seen as the ISA (instruction set architecture) for an RXVM processor, of which the microcode is implemented in the C99 language.

Programs intended to execute directly on the processor use the RXVM instruction set and the instruction encoding and semantics of this architecture.

An application program can assume that all instructions defined by the architecture and that are neither privileged nor optional, exist and work as documented.

To be ABI (application binary interface) conforming, the processor must implement the instructions of the architecture, perform the specified operations, and produce the expected results. The ABI neither places performance constraints on systems nor specifies what instructions must be implemented in hardware. A software implementation of the architecture conforms to the ABI; likewise, the architecture could be implemented in hardware, e.g. an FPGA.

2.3.1 The Register



CREXX `rxvm` is a Register based virtual machine, as opposed to a Stack based VM. The number of registers is only limited by memory and, for practical purposes, can be considered unlimited. The address size of the fields in a register is, for the virtual machine implementations, implied by the address size that the host OS can handle. For hardware the size is undefined and can follow the hardware address generation capacity.

```
struct value {
    /* bit field to store value status - these are explicitly set */
    value_type status;

    /* Value */
    rxinteger int_value;
    double float_value;
    void *decimal_value; /* TODO */
    char *string_value;
    size_t string_length;
    size_t string_buffer_length;
    size_t string_pos;
#ifndef NUTF8
    size_t string_chars;
    size_t string_char_pos;
#endif
    void *object_value;
```

Conversions of register values

The status field determines for instructions that expect a data type, which field is the field to act upon. Conversions are possible but never implicit. So for example, when the register contains an int value in field `int_value`, but it needs to be printed with the rxas `say` instruction, the `0` instruction takes care of the conversion and the population of the memory area the `*string_value` points to.

```
typedef union {
```

```
    struct {
        unsigned int type_object : 1;
        unsigned int type_string : 1;
        unsigned int type_decimal : 1;
        unsigned int type_float : 1;
        unsigned int type_int : 1;
    };
    unsigned int all_type_flags;
} value_type;
```

3

CREXX Virtual Machine Instruction Set

Most CREXX Virtual machine instructions share the following characteristics:

- Logical data flow is from right to left
- There is no separate CC (Condition Code) or status register; a register specified in the instruction is used for these results; for example all comparisons leave their result in a register, that subsequently can be used for a conditional branch
- The assembler does not have any data definition directives; data is entered directly into registers, which then are used by the instructions. In `.rxbin` modules this data is represented in the Constant Pool.

3.1 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.

0x0039 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.

0x003b 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.

0x003d 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.

0x003f 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

0x0040 bit wise and of 2 integers (op1=op2&op3) REGREGREG

0x0041 bit wise and of 2 integers (op1=op2&op3) REGREGINT

ICOPY - Integer Copy

Syntax - form

Name	OP1	OP2
ICOPY	REG	REG

Operation

0x00ba 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

0x001c Integer Divide (op1=op2/op3) REGREGREG

0x001d Integer Divide (op1=op2/op3) REGREGINT

0x001e 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

0x0021 Integer Modulo (op1=op2
0x0022 Integer Modulo (op1=op2
0x0023 Integer Modulo (op1=op2&op3) REGINTREG

IMULT - Intger Multiply

Syntax - variants

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

Operation

0x0018 Integer Multiply (op1=op2*op3) REGREGREG

0x0019 Integer Multiply (op1=op2*op3) REGREGINT

INC - Increment Integer Register

Syntax - form

Name	OP1
INC	REG

Operation

0x0038 Increment Int (op1++) REG

INC0 - Increment Register R0

Syntax - form

Name	OP1
INC0	no operand

Operation

0x003a Increment R0++ Int no

INC1 - Increment Register R1

Syntax - form

Name	OP1
INC1	no operand

Operation

0x003c Increment R1++ Int no

INC2 - Increment Register R2

Syntax - form

Name	OP1
INC2	no operand

Operation

0x003e 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

0x00e2 op1=op2**op3 REGREGREG

0x00e3 op1=op2**op3 REGREGINT

0x00e4 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.

0x00fb 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

0x0046 bit wise shift logical left of integer (op1=op2<<op3) REGREGREG

0x0047 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

0x0048 bit wise shift logical right of integer (op1=op2>>op3) REGREGREG

0x0049 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

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

0x0014 Integer Subtract (op1=op2-op3) REGREGINT

0x0015 Integer Subtract (op1=op2-op3) REGINTREG

ITOF - Integer to Float

Syntax - form

Name	OP1
ITOF	REG

Operation

0x00d3 Set register float value from its int value REG

LOAD - Load

Syntax - variants

Name	OP1	OP2
LOAD	REG	INT
LOAD	REG	FLOAT
LOAD	REG	CHAR
LOAD	REG	STRING

Operation

0x00c2 Load op1 with op2 REGINT

0x00c3 Load op1 with op2 REGFLOAT

0x00c4 Load op1 with op2 REGSTRING

0x00c5 Load op1 with op2 REGCHAR

LOADSETTP - Load and Set Type

Syntax - variants

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

Operation

0x00ff load register and sets the register type flag load op1=op2 (o REGINTINT

0x0100 load register and sets the register type flag load op1=op2 (o REGFLOATINT

0x0101 load register and sets the register type flag load op1=op2 (o REGSTRINGINT

3.2 Floating Point Arithmetic

FADD - Add Float

Syntax - variants

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

Operation

0x0024 Float Add (op1=op2+op3) REGREGREG

0x0025 Float Add (op1=op2+op3) REGREGFLOAT

FCOPY - Copy Float

Syntax - form

Name	OP1	OP2
FCOPY	REG	REG

Operation

0x00bb 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

0x0032 Float Divide (op1=op2/op3) REGREGREG

0x0033 Float Divide (op1=op2/op3) REGREGFLOAT

0x0034 Float Divide (op1=op2/op3) REGFLOATREG

FEQ - Float Equals

Syntax - variants

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

Operation

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

0x0075 Float Equals op1=(op2==op3) REGREGFLOAT

FFORMAT - Format String from Float

Syntax - form

Name	OP1	OP2	OP3
FFORMAT	REG	REG	REG

Operation

0x00d8 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

0x0078 Float Greater than op1=(op2>op3) REGREGREG

0x0079 Float Greater than op1=(op2>op3) REGREGFLOAT

0x007a 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

0x007b Float Greater than equals op1=(op2>=op3) REGREGREG

0x007c Float Greater than equals op1=(op2>=op3) REGREGFLOAT

0x007d 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

0x007e Float Less than op1=(op2<op3) REGREGREG

0x007f Float Less than op1=(op2<op3) REGREGFLOAT

0x0080 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

0x0081 Float Less than equals op1=(op2<=op3) REGREGREG

0x0082 Float Less than equals op1=(op2<=op3) REGREGFLOAT

0x0083 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

0x002e Float Multiply (op1=op2*op3) REGREGREG

0x002f 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

0x0076 Float Not equals op1=(op2!=op3) REGREGREG

0x0077 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

Operation

0x00e5 op1=op2**op3 REGREGREG

0x00e6 op1=op2**op3 REGREGFLOAT

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.[6]

0x00fc 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
```

[6]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

0x0028 Float Subtract (op1=op2-op3) REGREGREG

0x0029 Float Subtract (op1=op2-op3) REGREGFLOAT

0x002a Float Subtract (op1=op2-op3) REGFLOATREG

FTOB - Float to Boolean

Syntax - form

Name	OP1
FTOB	REG

Operation

0x00d5 Set register boolean (int 1 or 0) value from its float value REG

FTOI - Float to Int

Syntax - form

Name	OP1
FTOI	REG

Operation

0x00d4 Set register int value from its float value REG

FTOS - Float to String

Syntax - form

Name	OP1
FTOS	REG

Operation

0x00d2 Set register string value from its float value REG

LOAD - Load

Syntax - variants

Name	OP1	OP2
LOAD	REG	INT
LOAD	REG	FLOAT
LOAD	REG	CHAR
LOAD	REG	STRING

Operation

0x00c2 Load op1 with op2 REGINT

0x00c3 Load op1 with op2 REGFLOAT

0x00c4 Load op1 with op2 REGSTRING

0x00c5 Load op1 with op2 REGCHAR

3.3 Logical Operations

AND - Logical AND

Syntax - form

Name	OP1	OP2	OP3
AND	REG	REG	REG

Operation

0x0096 Logical (int) and op1=(op2 && op3) REGREGREG

COPY - Copy Register

Syntax - form

Name	OP1	OP2
COPY	REG	REG

Operation

0x00b9 Copy op2 to op1 REGREG

DCALL - Dynamic Call Procedure

Syntax - form

Name	OP1	OP2	OP3
DCALL	REG	REG	REG

Operation

0x00ac Dynamic call procedure (op1=op2(op3...)) REGREGREG

DLLPARMS - DLL Parameter Fetch

Syntax - form

Name	OP1	OP2	OP3
DLLPARMS	REG	REG	REG

Operation

0x0106 fetches parms for DLL call REGREGREG

EXIT - Exit program

Syntax - variants

Name	OP1
EXIT	no operand
EXIT	REG
EXIT	INT

Operation

0x00ce Exit no

0x00cf Exit op1 REG

0x00d0 Exit op1 INT

IAND - Integer AND

Syntax - variants

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

Operation

0x0040 bit wise and of 2 integers (op1=op2&op3) REGREGREG

0x0041 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

0x004a inverts all bits of an integer (op1= op2) REGREG

0x004b 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

0x0042 bit wise or of 2 integers (op1=op2|op3) REGREGREG

0x0043 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

0x0107 random number random, op1=irand(op2) REGREG

0x0108 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

0x0044 bit wise exclusive OR of 2 integers (op1=op2ôp3) REGREGREG

0x0045 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

0x00ff load register and sets the register type flag load op1=op2 (o REGINTINT

0x0100 load register and sets the register type flag load op1=op2 (o REGFLOATINT

0x0101 load register and sets the register type flag load op1=op2 (o REGSTRINGINT

NOT - Not

Syntax - form

Name	OP1	OP2
NOT	REG	REG

Operation

0x0098 Logical (int) not op1=!op2 REGREG

OR - Logical OR

Syntax - form

Name	OP1	OP2	OP3
OR	REG	REG	REG

Operation

0x0097 Logical (int) or op1=(op2 || op3) REGREGREG

SETORTP - OR the Register Type Flag

Syntax - form

Name	OP1	OP2
SETORTP	REG	INT

Operation

0x0102 or the register type flag (op1.typeflag = op1.typeflag || op2 REGINT

SWAP - Swap Registers

Syntax - form

Name	OP1	OP2
SWAP	REG	REG

Operation

0x00b8 Swap op1 and op2 REGREG

3.4 Branching

BCF - Branch on Count if False

Syntax - variants

Name	OP1	OP2	OP3
BCF	ID	REG	
BCF	ID	REG	REG

Operation

0x00eb if op2=0 goto op1(if false) else dec op2 IDREG

0x00ec 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

0x00e7 dec op2; if op2>0; goto op1(if true) IDREG

0x00e8 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

0x00e9 dec op2; if op2>=0; goto op1(if true) IDREG

0x00ea 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

0x00f7 if op2==op3 then goto op1 IDREGREG

0x00f8 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

0x00ef 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
```

0x00f0 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

0x00ed if op2>op3 then goto op1 IDREGREG

0x00ee 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

0x00f3 if op2<=op3 then goto op1 IDREGREG

0x00f4 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

0x00f1 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
```

0x00f2 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

0x00f5 if op2!=op3 then goto op1 IDREGREG

0x00f6 if op2!=op3 then goto op1 IDREGINT

BR - Branch Unconditionally

Syntax - form

Name	OP1
BR	ID

Operation

0x00b3 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

0x00b5 Branch to op1 if op2 false IDREG

BRT - Branch on True

Syntax - form

Name	OP1	OP2
BRT	ID	REG

Operation

0x00b4 Branch to op1 if op2 true IDREG

BRTF - Branch on True

Syntax - form

Name	OP1	OP2	OP3
BRTF	ID	ID	REG

Operation

0x00b6 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

0x0104 if op2.typeflag && op3 true then goto op1 IDREGINT

BRTPT - Branch on Type

Syntax - form

Name	OP1	OP2
BRTPT	ID	REG

Operation

0x0103 if op2.typeflag true then goto op1 IDREG

`CALL` - Call Procedure

`Syntax` - variants

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

`Operation`

`0x00a9` Call procedure (op1()) FUNC

`0x00aa` Call procedure (op1=op2()) REGFUNC

`0x00ab` Call procedure (op1=op2(op3...)) REGFUNCREG

CNOP - No Operation

Syntax - form

Name	OP1
CNOP	no operand

Operation

0x00e1 no operation no

LINK - Link

Syntax - form

Name	OP1	OP2
LINK	REG	REG

Operation

0x00bf Link op2 to op1 REGREG

NULL - Null

Syntax - form

Name	OP1
NULL	REG

Operation

0x00c1 Null op1 REG

RET - Return

Syntax - variants

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

Operation

0x00ad Return VOID no

0x00ae Return op1 REG

0x00af Return op1 INT

0x00b0 Return op1 FLOAT

0x00b1 Return op1 CHAR

0x00b2 Return op1 STRING

3.5 I/O operations

OPENDLL - Open Dynamic Link Library

Syntax - form

Name	OP1	OP2	OP3
OPENDLL	REG	REG	REG

Operation

0x0105 open DLL REGREGREG

READLINE - Read a Line

Syntax - form

Name	OP1
READLINE	REG

Operation

0x00cd Read Line to op1 REG

SAY - Write on Console

Syntax - variants

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

Operation

0x00c6 Say op1 REG

0x00c9 Say op1 INT

0x00ca Say op1 FLOAT

0x00cb Say op1 STRING

0x00cc Say op1 CHAR

SAYX - Write on Console without Linefeed

Syntax - variants

Name	OP1
SAYX	REG
SAYX	STRING

Operation

0x00c7 Say op1 without line feed REG

0x00c8 Say op1 (as string) without line feed STRING

3.6 Time Instructions

MTIME - Microseconds Time

Syntax - form

Name	OP1
MTIME	REG

Operation

0x009a Put time in microseconds into op1 REG

TIME - Get Time

Syntax - form

Name	OP1
TIME	REG

Operation

0x0099 Put time into op1 REG

XTIME - Put extended Time properties into OP1

Syntax - form

Name	OP1	OP2
XTIME	REG	STRING

Operation

0x009b put special time properties into op1 REGSTRING

3.7 Meta Instructions

AMAP - Map to Arg

Syntax - variants

Name	OP1	OP2
AMAP	REG	REG
AMAP	REG	INT

Operation

0x009e Map op1 to arg register index in op2 REGREG

0x009f Map op1 to arg register index op2 REGINT

GETTP - Get Type

Syntax - form

Name	OP1	OP2
GETTP	REG	REG

Operation

0x00fd gets the register type flag (op1 = op2.typeflag) REGREG

GMAP - Map Global

Syntax - variants

Name	OP1	OP2
GMAP	REG	REG
GMAP	REG	STRING

Operation

0x00a2 Map op1 to global var name in op2 REGREG

0x00a3 Map op1 to global var name op2 REGSTRING

LINKATTR - Link Attributes

Syntax - variants

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

Operation

0x00bd Link attribute op3 of op2 to op1 REGREGREG

0x00be Link attribute op3 of op2 to op1 REGREGINT

MAP - Map to Var Name

Syntax - variants

Name	OP1	OP2
MAP	REG	STRING
MAP	REG	REG

Operation

0x009c Map op1 to var name in op2 REGREG

0x009d Map op1 to var name op2 REGSTRING

METADECODEINST -

Syntax - variants

Name	OP1	OP2
METADECODEINST	REG	REG

Operation

METALINKPREG -

Syntax - variants

Name	OP1	OP2
METALINKPREG	REG	REG

Operation

METALOADCALLERADDR -

Syntax - variants

Name	OP1
METALOADCALLERADDR	REG

Operation

METALOADDATA -

Syntax - variants

Name	OP1	OP2	OP3
METALOADDATA	REG	REG	REG

Operation

METALOADEDMODULES -

Syntax - variants

Name	OP1
METALOADEDMODULES	REG

Operation

METALOADEDPROCS -

Syntax - variants

Name	OP1	OP2
METALOADEDPROCS	REG	REG

Operation

METALOADFOPERAND -

Syntax - variants

Name	OP1	OP2	OP3
METALOADFOPERAND	REG	REG	REG

Operation

METALOADINST -

Syntax - variants

Name	OP1	OP2	OP3
METALOADINST	REG	REG	REG

Operation

METALOADIOPERAND -

Syntax - variants

Name	OP1	OP2	OP3
METALOADIOPERAND	REG	REG	REG

Operation

METALOADMODULE -

Syntax - variants

Name	OP1	OP2
METALOADMODULE	REG	REG

Operation

METALOADPOPERAND -

Syntax - variants

Name	OP1	OP2	OP3
METALOADPOPERAND	REG	REG	REG

Operation

METALOADSOPERAND -

Syntax - variants

Name	OP1	OP2	OP3
METALOADSOPERAND	REG	REG	REG

Operation

NSMAP - Map to Namespace

Syntax - variants

Name	OP1	OP2	OP3
NSMAP	REG	REG	REG
NSMAP	REG	STRING	REG
NSMAP	REG	REG	STRING
NSMAP	REG	STRING	STRING

Operation

0x00a4 Map op1 to namespace in op2 var name in op3 REGREGREG

0x00a5 Map op1 to namespace in op2 var name op3 REGREGSTRING

0x00a6 Map op1 to namespace op2 var name op3 REGSTRINGSTRING

0x00a7 Map op1 to namespace op2 var name in op3 REGSTRINGREG

PMAP - Map to Parent Var

Syntax - variants

Name	OP1	OP2
PMAP	REG	REG
PMAP	REG	STRING

Operation

0x00a0 Map op1 to parent var name in op2 REGREG

0x00a1 Map op1 to parent var name op2 REGSTRING

SETTP - Set Register Type Flag

Syntax - form

Name	OP1	OP2
SETTP	REG	INT

Operation

0x00fe sets the register type flag (op1.typeflag = op2) REGINT

UNLINK - Unlink Register

Syntax - form

Name	OP1
UNLINK	REG

Operation

0x00c0 Unlink op1 REG

UNMAP - Unmap Register

Syntax - form

Name	OP1
UNMAP	REG

Operation

0x00a8 Unmap op1 REG

3.8 Breakpoint Instructions

BPOFF - Breakpoint Off

Syntax - form

Name	OP1
BPOFF	no operand

Operation

0x0002 Disable Breakpoints no

BPON - Breakpoint On

Syntax - form

Name	OP1
BPON	no operand

Operation

0x0001 Enable Breakpoints no

3.9 String Instructions

APPEND - Append to String

Syntax - form

Name	OP1	OP2
APPEND	REG	REG

Operation

0x0055 String Append (op1=op1||op2) REGREG

APPENDCHAR - Append Character

Syntax - form

Name	OP1	OP2
APPENDCHAR	REG	REG

Operation

0x0052 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

0x004f 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.
```

0x0050 String Concat (op1=op2||op3) REGREGSTRING

0x0051 String Concat (op1=op2||op3) REGSTRINGREG

CONCCHAR - Concatenate Characters

Syntax - form

Name	OP1	OP2	OP3
CONCCHAR	REG	REG	REG

Operation

0x0053 Concat Char op1 from op2 position op3 REGREGREG

DROPCHAR - Drop Character

Syntax - form

Name	OP1	OP2	OP3
DROPCHAR	REG	REG	REG

Operation

0x00dc 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

0x00f9 op1 = find next blank in op2[op3] and behind REGREGREG

FNDNBLNK - Find Nth Blank

Syntax - form

Name	OP1	OP2	OP3
FNDNBLNK	REG	REG	REG

Operation

0x00fa op1 = find next next non blank in op2[op3] and behind REGREGREG

GETBYTE - Get Byte

Syntax - form

Name	OP1	OP2	OP3
GETBYTE	REG	REG	REG

Operation

0x00e0 get byte (op1=op2(op3) REGREGREG

GETSTRPOS - Get String Position

Syntax - form

Name	OP1	OP2
GETSTRPOS	REG	REG

Operation

0x0062 Get String (op2) charpos into op1 REGREG

GMAP - Map Global

Syntax - variants

Name	OP1	OP2
GMAP	REG	REG
GMAP	REG	STRING

Operation

0x00a2 Map op1 to global var name in op2 REGREG

0x00a3 Map op1 to global var name op2 REGSTRING

HEXCHAR - Get Hex Char from String

Syntax - form

Name	OP1	OP2	OP3
HEXCHAR	REG	REG	REG

Operation

0x005f op1 (as hex) = op2[op3] REGREGREG

ITOS - Integer to String

Syntax - form

Name	OP1
ITOS	REG

Operation

0x00d1 Set register string value from its int value REG

LOAD - Load

Syntax - variants

Name	OP1	OP2
LOAD	REG	INT
LOAD	REG	FLOAT
LOAD	REG	CHAR
LOAD	REG	STRING

Operation

0x00c2 Load op1 with op2 REGINT

0x00c3 Load op1 with op2 REGFLOAT

0x00c4 Load op1 with op2 REGSTRING

0x00c5 Load op1 with op2 REGCHAR

PADSTR - Pad a String

Syntax - form

Name	OP1	OP2	OP3
PADSTR	REG	REG	REG

Operation

0x00df set op1=op2[repeated op3 times] REGREGREG

POSCHAR - Position of Char

Syntax - form

Name	OP1	OP2	OP3
POSCHAR	REG	REG	REG

Operation

0x0060 op1 = position of op3 in op2 REGREGREG

RSEQ -

Syntax - variants

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

Operation

0x0086 non strict String Equals op1=(op2=op3) REGREGREG

0x0087 non strict String Equals op1=(op2=op3) REGREGSTRING

SAPPEND - Append a String

Syntax - form

Name	OP1	OP2
SAPPEND	REG	REG

Operation

0x0054 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

0x004c String Concat with space (op1=op2||op3) REGREGREG

0x004d String Concat with space (op1=op2||op3) REGREGSTRING

0x004e String Concat with space (op1=op2||op3) REGSTRINGREG

SCOPY - Copy String

Syntax - form

Name	OP1	OP2
SCOPY	REG	REG

Operation

0x00bc Copy String op2 to op1 REGREG

SEQ - String Equals

Syntax - variants

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

Operation

0x0084 String Equals op1=(op2==op3) REGREGREG

0x0085 String Equals op1=(op2==op3) REGREGSTRING

SETSTRPOS - Set String Charpos

Syntax - form

Name	OP1	OP2
SETSTRPOS	REG	REG

Operation

0x0061 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

0x008a String Greater than op1=(op2>op3) REGREGREG

0x008b String Greater than op1=(op2>op3) REGREGSTRING

0x008c 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

0x008d String Greater than equals op1=(op2>=op3) REGREGREG

0x008e String Greater than equals op1=(op2>=op3) REGREGSTRING

0x008f 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

0x0090 String Less than op1=(op2<op3) REGREGREG

0x0091 String Less than op1=(op2<op3) REGREGSTRING

0x0092 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

0x0093 String Less than equals op1=(op2<=op3) REGREGREG

0x0094 String Less than equals op1=(op2<=op3) REGREGSTRING

0x0095 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

0x0088 String Not equals op1=(op2!=op3) REGREGREG

0x0089 String Not equals op1=(op2!=op3) REGREGSTRING

STOF - String to Float

Syntax - form

Name	OP1
STOF	REG

Operation

0x00d6 Set register float value from its string value REG

STOI - String to Int

Syntax - form

Name	OP1
STOI	REG

Operation

0x00d7 Set register int value from its string value REG

STRCHAR - String to Char

Syntax - variants

Name	OP1	OP2	OP3
STRCHAR	REG	REG	
STRCHAR	REG	REG	REG

Operation

0x005d op1 (as int) = op2[op3] REGREGREG

0x005e op1 (as int) = op2[charpos] REGREG

STRLEN - String Length

Syntax - form

Name	OP1	OP2
STRLEN	REG	REG

Operation

0x005c String Length op1 = length(op2) REGREG

STRLOWER - Lowercase String

Syntax - form

Name	OP1	OP2
STRLOWER	REG	REG

Operation

0x00d9 Set string to lower case value REGREG

STRUPPER - Uppercase String

Syntax - form

Name	OP1	OP2
STRUPPER	REG	REG

Operation

0x00da Set string to upper case value REGREG

SUBSTCUT - Set Substring and Cut Off

Syntax - form

Name	OP1	OP2
SUBSTCUT	REG	REG

Operation

0x00de 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

0x0063 op1 = op2[charpos]...op2[charpos+op3-1] REGREGREG

SUBSTRING - Set remaining String

Syntax - form

Name	OP1	OP2	OP3
SUBSTRING	REG	REG	REG

Operation

0x00dd set op1=substr(op2,op3) remaining string REGREGREG

TRANSCHAR - Translate Characters

Syntax - form

Name	OP1	OP2	OP3
TRANSCHAR	REG	REG	REG

Operation

0x00db 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

0x0056 Trim String (op1) from Left by (op2) Chars REGREG

0x0059 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

0x0057 Trim String (op1) from Right by (op2) Chars REGREG

0x005a Trim String (op2) from Right by (op3) Chars into op1 REGREGREG

TRUNC - Truncate

Syntax - variants

Name	OP1	OP2	OP3
TRUNC	REG	REG	
TRUNC	REG	REG	REG

Operation

0x0058 Trunc String (op1) to (op2) Chars REGREG

0x005b Trunc String (op2) to (op3) Chars into op1 REGREGREG

4

cREXX Assembler

4.1 REXX Assembler Specification

4.1.1 Overview

[Work in Progress]
Additional details of the RXVM virtual machine

4.1.2 Features Required to Support cREXX Languages

The following table maps REXX features to RXAS capabilities to explain the motivation for RXAS capabilities and approach for implementing REXX features. Details of the REXX Features themselves are documented in the REXX specifications, and more details of the RXAS capabilities follow.

REXX Features	RXAS Capabilities	Implementation Approach	Available
Program Flow	Branches	Generate loops in RX	Yes
Simple Variables	Local Registers		Yes
STEM Variables	See Classes	STEMs implemented as class	N/A
PROCEDURES	Procedures		Yes
SIGNAL / Labels	branches / labels	may need duplicate code	Yes
Static EXPOSE	Pass by Reference args	EXPOSED vars by reference	Yes
Dynamic EXPOSE	No Additional	Pool implemented in REXX	N/A
VALUE()	No Additional	Pool implemented in REXX	N/A
INTERPRET	Dynamically created rxbin	Complex[7]	No

[7] Version 1 will require the compiler / assembler linked into the runtime. Version 2 (REXX on REXX) will require advanced REXX parsing.

REXX Features	RXAS Capabilities	Implementation Approach	Available
REXXb Classes	Regs contain sub-regs	[8]	No
REXXc Singleton Classes	global regs	[9]	Yes
REXXb Interface	Regs hold function pointer	No	
Dynamic / Late binding	footnote	by REXXb class(s)	No
Arbitrary Precision Maths	No Additional	in REXXb class(s)	N/A
ADDRESS	Platform specific		No
External Functions	Platform specific	[10]	No
Error Reporting	Annotation and debug pool	[11]	No

4.1.3 RXAS Source Syntax and Structure

Structure

1. Global Variable definition or Declaration
2. Procedure Definition or Declaration (repeated)

Comments

```
/* Block Comment */

* Line Comment
```

Instructions

These have the format OP_CODE [ARG1[,ARG2[,ARG3]]] where each argument can be a

[8] Class Attributes (always private) are stored in sub-registers. Member functions are statically linked and use a naming convention "class.member". The first argument to the member is the object

[9] Singleton Classes replace static classes / members in other languages. The singleton is stored in a global register and exposed with a naming convention "class.@1". Note that "@" cannot be used from REXX programs directly

[10] Dynamic discovery and linking, call-backs to access variable pool. Classic REXX implementation will need to ensure all required variables are available in the variable pool

[11] Source line information stored in a "debug" pool to allow source line error reporting. Run time error conditions (signals) call REXX Exit functions

- Register - e.g. r0...1 (for locals), a0...n (for arguments) or g0...n (for globals)
- String - e.g. “hello”
- Integer - e.g. 5
- Float - e.g. 5.0
- Function - e.g. proc()
- Identifiers / Label - e.g. label1

Directives

RXAS supports the following directives.

Directive	Description
.globals = {INT}	Defines the number of globals defined for the file
.locals = {INT}	Defines the number of local registers in a procedure
.expose = {ID}	Defines the exposed index of a global register [12]

File Scope Global Registers

```
.globals={int}
```

Defines {int} global variable g0 ... gn. These can be used within any procedure in the file.

Global Registers

Any global register marked as exposed is available to any file which also has the corresponding exposed index/name.

File 1

```
.globals=2            * 2 Global Registers
g0 .expose=namespace.var_name   *
```

File 2

```
.globals=3            * 3 Global Registers
g2 .expose=namespace.var_name   *
```

In this case file 1 g0 is mapped to file 2 g2 under the index/name of “namespace.var_-name”.

[12] or Procedure. These are used for linking between files/modules

File Scope Procedure

The locals define how many local registers, r0 to r(locals-1), are needed by the procedure.

```
* The ".locals" shows the procedure is defined in here
file_scope_proc() .locals=3
...
ret
```

Global Procedures

Global Procedures can be called between file/modules.

File 1

```
* The ".locals" shows the procedure is defined in here
proc() .locals=3 .expose=namespace.proc
ret
```

File 2

```
* No ".locals" here! Showing that the procedure is only being declared
rproc() .expose=namespace.proc

main() .locals=3
call rproc()
ret
```

In this case main() in File 2, calls rproc() which is globally provided under the index/name of "namespace.proc". In File 1, proc() is exposed under this name and hence called from File 1.

Note: that "namespace" hints at the use of namespaces as part of exposed names; this facility is used by the compiler to define classes.

Also, as shown names can be mapped - they don't have to be the same in the source and in the target.

4.1.4 RXAS Capabilities (alphabetical)

Branching and Labels

Within a procedure labels can be defined as branch targets. Conditional and unconditional branch instructions can target these labels. The following example shows a loop structure.

```
   ...                  * Code before loop
l75:                    * Loop start label
```

```
   igt r0,r1,r4          * Does a integer compare of r1 and r4 - puts true (1) or fals
   brt l37,r0            * Branch if true to l37 (i.e. branch out of the loop)
   ...                   * Instructions in the loop
   inc r1                * Increment r1 (the loop counter)
   br l75                * Unconditional Branch to the start of the loop
l37:                     * Loop end Label
   ...                   * Instructions following the loop
```

Code Annotation and Debug Pool

NOT IMPLEMENTED

cREXX needs to support appropriate error messages (including source line number / text), breakpoints, and tracing. RXAS directives (.file and .line) allow the source file, source line to be defined. The .clause directive allow clause boundaries to be defined.

```
.file = "testfile.rexx"   * Source file name
proc()   .locals=3
   .regname r1,a          * Maps r1 to an id for debugging purposes
   .regname r2,b
   .line 9 "a=5; b=6"     * The line number and source string
   .clause                * REXX clause boundary
   load r1,5
   .clause
   load r2,6
   .line 10 "say a+b"
   .clause
   iadd r0,r1,r2
   itos r0
   say r0
   .line 11 "return"
   .clause
   ret
```

The directives are processed at "build time" and the debug constant pool is created which allow assembler instructions to be mapped back to source lines. In this way error messages can be generated as if the source file was being interpreted "classically" but with no run-time overhead.

In addition, tracing/debugging is implemented by the VM machine using the clause boundaries stored in the debug pool. The VM can set a breakpoint by replacing the instruction at the appropriate address with a breakpoint instruction. When the breakpoint is reached it uses the clause boundary information to determine where the next breakpoint should be set. Tools can be made available to allow a REXX programmer to set a breakpoint at a REXX source code line number.

The debug pool also contains the information to display the rexx variable name stored in a register.

Note that accessing debug information is a significant overhead as the debug pool will need repeated searching, and will only be used for debugging/tracing (or creating an error message) where performance is not critical. The reason this approach is used is that when there is no debugging in action there is no runtime performance overhead at all (obviously the size of the rxbin binary file is made larger with the debug pool).

Constant Pool

Each File/Module has a constant pool that stores: * String Constants * Procedure Details * PTable information (mapping class to interface procedures for a class and objects)

Dynamic Access to Registers (including Arguments)

Dynamic access to a register is enabled by additional members of the link family of instructions these allow a register to point to the same value as a dynmically number primary register.

```
alink secondary_reg, arg_reg_num * Links secondary_reg to the argument register with n
glink secondary_reg, global_reg_num * Links secondary_reg to the global register with
```

Dynamic Type Instructions

The compiler will be able to manage which registers have what values most of the time but there will be certain dynamic situations where the value type or status is not known. To handle this the compiler use the registers type flag:

```
gettp - gets the register type flag (op1 = op2.typeflag)
settp - sets the register type flag (op1.typeflag = op2)
setortp - or the register type flag (op1.typeflag = op1.typeflag || op2)
brtpt - if op2.typeflag true then goto op1
brtpandt - if op2.typeflag && op3 true then goto op1
```

The typeflag is a 64bit integer and its usage is defined by convention only see cREXX Calling Convention.

Dynamic Procedure Pointers

This capability is to support interfaces. Where the compiler knows the object's class it can link statically to the correct member by using the procedure name (i.e. class_name.member_name(), however when the object is only known to implement an interface (i.e. its class is not known) then the VM needs to dynamically link interface members to the object's class specific implementation.

Each register, that contains an object whose call implements an interface, has a pointer to a entry in the constant pool. This entry allows the the interface name and member number to be searched at runtime, returning its implementation

procedure pointer. This is known as the register's static ptable. The static ptable also stores the name of the objects class.

Note: Where an object members are dynamically assigned at runtime (not an immediately required capability) the dynamic mapping from member name to procedure pointer will be done within the REXX runtime library (i.e. not applicable to RXAS).

A directive defines the entry

```
.ptable class_name interface1_name(impl1_1(), impl1_2(), ...) interface2_name(impl2_1(
```

This creates the entry into the constant pool with links to the procedure implementing an interfaces members #1,#2 etc.

The object is linked to the entry with an instruction:

```
setptable r1,class_name
```

This sets the register r1 ptable to the entry "class_name" in the constant pool.

Finally the entry can be used at runtime:

```
srcptable r2,"interface_name",3
```

In this example r2 is a class instance (object) implementing interface "interface_-name". This instruction looks up the object's procedure implementing member #3 of interface "interface_name" and sets the procedure pointer of r2 to this. Then

```
dyncall r0, r2, r3
```

calls the procedure in r2, with arguments from r3 and puts the result in r0.

External Functions

Injecting Dynamic Code

Instructions

Type coding Instructions have prefix to determine type: s=string, i=integer, f=float, o=object

Maths As an example, the add family will have * iadd reg,reg,reg
* fadd reg,reg,reg * etc.

Each function just uses the corresponding registers value (int, float, etc).

Load

- [s/i/f/d]load - lost loads the corresponding type value only
- load (i.e. with no prefix) copies all values and the type flag to the target register

Conversion Converting means setting a value for a type based on the value on another type in the same register, e.g. * itos reg - sets the string value to the string representation of the integer value of reg * ftos reg * stof reg - This converts the string to a float, or triggers a signal if it can't

Note: this replaces prime/master.

SAY / ADDRESS etc. Where an instruction needs a string it will only have a string "version". For clarity we will have

- say
- address

but there will not be a isay etc. Instead the compiler might need to do a "itos" first.

Procedures and Arguments

A procedures registers are independent to the caller's registers. What happens is that the VM maps its registers to the registers in the caller.

Each time a procedure/function is called a new "stack frame" is provided. This means that the called function has its own set of registers.

The function header defines how many registers (called locals) the function can access - for practical purposes we can consider that any number of registers can be assigned to a function.

In addition, each file defines a number of global registers that can be shared between procedures.

In a function with 'a' arguments, 'n' locals, and 'm' globals: * R0 ... R(n-1) - are local registers to be used by the function * R(n) ... R(n+m-1) - are the global registers, i.e. g(0) ... g(m-1) * R(n+m) - holds the number of arguments (a) * R(n+m+1) ... R(n+m+a) holds the arguments, i.e. a(1) ... a(a)

This ordering allows a dynamic numbers of arguments.

cREXX Calling Convention All arguments within RXAS are pass by reference, therefore arguments needs to be copied to another register if pass by reference is not wanted. This approach is a way to support moves rather than copies - especially important to avoid slow object and string copies.

It is mandatory to use this calling convention between REXX and RXAS procedures. Although not necessary, it is recommended to also use this convention between RXAS procedures.

In this convention the caller is responsible for setting argument registers' typeflag. This is used to indicate if an optional argument is present, and if a pass-by-value string or object argument needs preserving.

The callee (procedure) is responsible for applying default values for optional arguments, and for ensuring that pass-by-value arguments are kept constant

(so they are not changed, effecting the caller logic) if required. The callee uses the typeflag for this.

Register Type Flag Byte Values

The register typeflag is used to optimise function arguments.

- Bit 1 - REGTP_VAL - ONLY used for optional arguments; setting (1) means the register has a specified value
- Bit 2 - REGTP_NOTSYM - ONLY used for "pass be value" and ONLY large (strings, objects) registers; setting (2) means that it is not a symbol so does not need copying as, even if it is changed, the caller will not use its original value. Note: Small registers (int, float) are always copied as this is faster than setting and checking this flag; the REGTP_NOTSYM flag is not set or read for integers and floats.

The following examples demonstrate the calling convention.

Basic Call by Reference

REXX Program

Annotated Generated RXAS

Optional Call by Reference

REXX Program

Annotated Generated RXAS

Call By Value Integer and Optimisations

In this example, REGTP_NOTSYM is not used as the parameter is an integer.

REXX Program

Annotated Generated RXAS

Call By Value Strings and Optimisations (optional arguments)

In this example, REGTP_NOTSYM is used as the parameter is an string in optional arguments.

REXX Program

Annotated Generated RXAS

Arbitrary number of arguments with ... **TO BE IMPLEMENTED** (requires arrays)

EXPOSE **TO BE IMPLEMENTED**

Syntax candy to provide familiar (but not the same) EXPOSE experience for REXX programmers.

In this example:

```
exp = 100

say test( "hello")
exit
```

```
test: procedure = .string expose exp = .int
  arg message = .string
  return message || exp
```

Is converted by the compiler to:

```
exp = 100

say test(exp, "hello")
exit

test: procedure = .string
  arg expose exp = .int, message = .string
  return message || exp
```

This is designed to provide a familiar (but not the same) experience for REXX programmers

Procedure Lookup Tables

Registers

Register Data Each register holds 5 values - String, float, integer and object, and a type flag which is used to indicate which values are valid. Note that arbitrary position maths is handled as objects.

In most cases it is for the compiler to decide what values to use, and how/if to set the type flag. Only a few dynamic scenarios will need some extended functions (see following). The type values are 0=unset, 1=string, 2=float, 4=integer, 8=object, 16=interface. Each register can have multiple types set as valid. The compiler sets the valid types explicitly with instructions - this is not an automatic runtime capability. At runtime the VM has no need nor the *ability* to validate data. Any caching has to be achieved by compiler logic.

An Object has an pointer to its static ptable entry in the constant pool as well as an array of sub-registers. These sub-registers are the private attributes of the object.

Register Initialisation All registers are initialized on entry to a procedure on the register "stack". The rationale is that all the memory can be malloced at once which is faster/safer. The pointers to globals and arguments are also setup.

In addition a shadow set of pointers to the procedure's registers are setup. These are used by the unlink instruction, they holding the base/initial register pointer, and unlink sets the register pointer to the pointer held in the respective shadow value.

Registers hold references to their parent/owner for memory freeing purposes. The owner can change, for example the owner of a returned register is set to the caller. When an object, stack frame, global pool is being deleted the registers

are also freed/deleted if the registers owner is the same as the container being deleted.

Register Re-Mapping Facilities There are a few scenarios where the contents of a register is needed in another register number: a call requires the arguments to be in consecutive registers, object sub-registers need to be copied to registers, or accessing an arbitrary argument registers (i.e. when the number of arguments is only known at runtime).

Copying the contents of registers to achieve this would be slow (and for large strings or objects, very slow). Also it is inconsistent because an integer copy and an object pointer copy (which is a copy by reference) behave differently (integers become independent but a change to the object changes it in each "copy").

We provide 2 facilities to allow very fast and safe register moves:

- SWAP. This swaps two registers. This is very fast as it requires only 6 pointer copies (3 swapping the two register pointers and 3 swapping the two shadow pointers). It allows the programmer to swap two registers (arguments, globals, registers) to get the register into a convenient register number (perhaps for a call). Doing the swap again restores the register numbers.
- LINK/UNLINK. The link instruction makes two register numbers point to the same register (one is primary, the other secondary). Unlink makes the *secondary* register revert back to it original state by using the shadow register pointer. Each instruction only requires one pointer copy.

The behaviour should normally be quite simple - however swapping or linking already linked or swapped registers may cause complex outcomes. Developers should consider the above behaviour descriptions to untangle this!

Runtime Error Conditions and Exit Functions

Shell Instructions

Sub-Registers

Super Instructions

Once we see what the code generated by the compiler looks like we may combine some of these instructions in to super instructions for performance reasons.

4.1.5 Variable Pool in REXX

In the current code for the VM we have registers which point to a variable structure (that can contain string, integers etc). These can be considered to be

“anonymous variables” – the compiler will assign a register to hold a variable at compile time.

In this way “80%” of the needs for REXX programs will be handled – but not all. Some aspects of REXX needs dynamic variable names – e.g. Some EXPOSE scenarios, INTERPRET, VALUE, and the REXXCOMM / SAA API. When these are needed the compiler needs to work in what I am calling “Pedantic” mode. In this mode variables are also given a string index in a variable pool, this index can be searched for dynamically at runtime.

Each stack frame will also include its own variable pool. This is a name/value index (via a HASH or TREE) whereby a variable can be found via the index string. *This will be implemented in REXXb.*

- When a procedure exits and the stack frame is torn down, the corresponding variable pool and variables also need freeing.
- To facilitate EXPOSE, a variable pool index can be linked to a parent pool variable.
- A register can point to an anonymous variable (as implemented today in the code) or mapped to a variable in the variable pool.

A

Platform Considerations

B

Notices

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
```

D

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

ISBN 978-90-819090-1-3