Fixing BRexx on VM/370
(Or REXX and Bytecode)
(Or CREXX - A REXX Architecture)

The 31st Annual Rexx Symposium

Adrian Sutherland • 29.09.2020
(Final)
Fixing BRexx on VM/370

VM/370 DevOps

GCCLIB - The C library

BREXX - Internals and changes

The CREXX Project - A REXX Architecture

REXX - how can it be improved while keeping its essence, “to make programming easier than before”?

Next Up - The VM/370 interface to Rexx with Bob Bolch

WARNING: We are going to cover this but in a non-linear narrative structure (!)
The LLVM Project is a collection of modular and reusable compiler and toolchain technologies.

https://llvm.org/

The core of LLVM is the intermediate representation (IR), a low-level programming language similar to assembly. IR is a strongly typed reduced instruction set computing (RISC) instruction set which abstracts away most details of the target. For example, the calling convention is abstracted through call and ret instructions with explicit arguments. Also, instead of a fixed set of registers, IR uses an infinite set of temporaries of the form %0, %1, etc.
Pipelining

Conceptually a CPU executes instructions one after the other, however to get a performance boost instructions overlap in a "production line" (pipeline).

Pipeline lengths are currently typically 15 odd stages (Intel/ARM)

There are hazards that cause a pipeline stall (e.g. accessing information not yet written and conditional branching)

This is one reason why it is very hard to write modern assembler and efficient backends for different architectures (hence the use of LLVM). Another is CPU memory caching

However a journeyman programmer can do somethings!

Diagram from a Aharon Kupershtok lecture
Typical Bytecode Optimisation

1. Threaded code
2. Super-instructions / inlining
3. Top-of-stack in a register
4. Scheduling the dispatch of the next VM instruction

In all about 2x faster than classic bytecode

We should be aiming for performance of only 2-5 times slower than native code

NOTE - We could be talking about any language ...

```
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;
        ... 
    }
}
```

```
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);
... 
```
REXX Assembler

This is where optimisations become REXX specific ...

BREXX

- Stack Based
- Leaves work to the interpreter

CREXX

- Register Based
- Trying to handle REXXisms at the low level

We need to get this right for LLVM ...

```rexx
/* SIMPLE */
A = 10
B = 5
SAY A + B
```

```crexx
.NEWCLAUSE
CREATE     "A"
PUSH       10
COPY
.NEWCLAUSE
CREATE     "B"
PUSH       5
COPY
.NEWCLAUSE
PUSHTMP
LOAD       "A"
LOAD       "B"
ADD
SAY
.NEWCLAUSE
IEXIT
```

```brexx
.def main: locals=3 {r1="A", r2="B"}
.ILOAD  r1,10
.ILOAD  r2,5
.IADD  r3,r1,r2
.ISAY  r3
.HALT
```
REXX Variable Types

1. Rexx is typeless ... and more than that conceptually all variables are strings
2. Rexx stems provide a flexible and arbitrary index scheme
3. VALUE(), INTERPRET(), and REXXSAA/EXECOMM all require dynamic variable name resolution
4. Performance requires compile time resolution of variable names and types, wherever possible

Note: BREXX has a single shared buffer
REXX Assembler

This is where optimisations become REXX specific ...

**BREXX**
- Stack Based
- Leaves work to the interpreter

**CREXX**
- Register Based
- Trying to handle REXXisms at the low level

We need to get this right for LLVM ...

```
/* SIMPLE2 */
ARG INDEX VAL
DATA.INDEX = VAL
SAY DATA.INDEX

> SIMPLE2 10 TEST
```

```cpp
.NEWCLAUSE
LOADARG
COPY2TMP
UPPER
PARSE
TR_SPACE
CREATE "INDEX"
PVAR
TR_END
CREATE "VAL"
PVAR
POP
NEWCLAUSE
CREATE "DATA.INDEX"
LOAD "VAL"
COPY
NEWCLAUSE
LOAD "DATA.INDEX"
SAY
NEWCLAUSE
IEXIT

.BREXX
.def main: locals=4 {r1="INDEX", r2="VAL"}
ARGUPPER r4
TR_SPACE r1,r4
TR_END r2,r4

.SCLOAD r4,"DATA." * DATA. in const pool
.SCONCAT r4,r4,r1 * r4 is now "DATA.10"
.SRMAP1 r3,r4 * r3 is var DATA.10
.SLOAD r3,r2 * = VAL
.SSAY r3
.SHALT r3

.CREXX
```
REXX Assembler

This is where optimisations become REXX specific...

BREXX
- Stack Based
- Leaves work to the interpreter

CREXX
- Register Based
- Trying to handle REXXisms at the low level

We need to get this right for LLVM...

/* SIMPLE2 */
ARG INDEX VAL
DATA.INDEX = VAL
SAY DATA.INDEX

> SIMPLE2 10 TEST

NEWCLAUSE
LOADARG
COPY2TMP
UPPER
PARSE
TR_SPACE
CREATE     "INDEX"
PVAR
TR_END
CREATE     "VAL"
PVAR
POP
NEWCLAUSE
CREATE     "DATA.INDEX"
LOAD       "VAL"
COPY
NEWCLAUSE
LOAD       "DATA.INDEX"
SAY
NEWCLAUSE
IEXIT

.BREXX
.def main: locals=4 {r1="INDEX", r2="VAL"}
ARGUPPER  r4
TR_SPACE  r1,r4
TR_END    r2,r4

.CREXX
REXX Assembler Variable Mapping
With Constant Pool
* Map RegisterN to DATA
SCRMAP rN,"DATA"

* Map RegisterN to DATA."registerA"
SCRMAP1 rN,"DATA",rA

* Map RegisterN to DATA."registerA"."registerB"
SCRMAP2 rN,"DATA",rA, rB

... etc.

Without Constant Pool
* Map RegisterN to "registerA"
SRMAP1 rN,rA

* Map RegisterN to "registerA"."registerB"
SRMAP2 rN,rA,rB

... etc.

REXX Assembler Variable Type Control
IREADY rN
FREADY rN * TBD - how to control precision ...
SREADY rN
DREADY rN * Date ... why not?!
ADD rN,RM * Readies both registers as integers (unlike IADD) ...

... etc.

... etc.

... etc.
High Level Components & Implementation Leverageable Tools

- **CREXX**
  - Lexer & Parser
    - ANTLR 3
    - PackCC
  - AST & Static Optimisation
    - ANTLR 3
  - Compiler
    - VMGEN
  - ByteCode Interpreter
    - VMGEN
  - native backends (Compiled / JIT)
    - LLVM
  - Variables
    - BREXX
  - Maths
    - Regina
  - Strings
    - Regina
  - ADDRESS
    - External Calls
    - ShellSpawn
  - BIF
    - Built In Functions
    - Regina
  - IO
    - Linux, Windows, Mac OS, VM/370 Drivers
    - POSIX
  - LIB
    - LLVM
Other REXX Considerations

1. Global Variables vs. EXPOSEd Variables
2. PARSE vs Regular Expressions
3. Shared variables across modules
4. Mixed-case lexing (and other funnies)
5. OOREXX - How to make it “easier”. E.g.
   a. Static or Dynamic
   b. Class Hierarchies or Interfaces?
6. What should “easier” IO look like? For example are pipes a more “REXX” IO metaphor?

   Is this accusation fair: just bolting on functions is rather “C” ...
DevOps - "a set of practices intended to reduce the time between committing a change to a system and the change being placed into normal production, while ensuring high quality"

Len Bass, Ingo Weber, and Liming Zhu
Technology

- Hercules S/370 Emulator
- VM/370 “Six Pack”
- Docker Containers
- Docker Hub
- GitHub Repository
- GitHub Actions
- Google Cloud Repository
- Google Cloud Container
- Optimised OS for hosting VM/370 Host
- YATA & Herccontrol
- REXX & C Test Suites
Adrian Sutherland

- CTO of Jumar Technology, specialists in legacy modernisation
- Journeyman Architect
- Keeps “hands-on” through numerous projects, from Raspberry PI toys and Domain Specific Languages to open architectural papers and other assets.

adrian@sutherlandonline.org

adrian.sutherland@jumar-technology.com
Thanks to ...

- **Bob Bolch** - for all the hard work on VM/370, help with my very poor S/370 Assembler, and always debating to make things better! Without his help this would not have been possible
- **René Jansen** - for all his encouragement, and bringing together the REXX test Suite

And everyone else - whose ideas I have misused!
Questions

adrian@sutherlandonline.org
adrian.sutherland@jumar-technology.com