Architecture

The ExVM Virtual Processor is a portable interpreter that is intended to be embedded into a host application in order to permit that application to expose scriptable behaviours. Whereas applications like web browsers provide a similar host platform for the direct execution of high level languages like JavaScript, ExVM presents a much lower level programming interface more akin to a physical CPU.

Motivation

The motivation behind ExVM was to provide an embeddable mechanism that would allow an application to have scriptable behaviours. The Amiga Operating System provided the ARexx Script Host (RexxMast) that, through the use of embedded command ports, permitted scripts to be written that could interact with multiple applications at once. While extremely flexible and powerful, the language was comparatively slow.

ExVM began on the Amiga as an experiment in a low (read machine) level script processor that could be used to implement more performance demanding scripting for a single host application.

Overview

ExVM is intended to provide access to the virtual processor via a shared library on the target platform. A host application incorporates this library and is then able to load, link and execute code for the virtual processor that in turn affects the behaviour of the host application.

It is intended that the host application and the virtual code written for it will be coupled to some extent. The host application will expose various resources to the virtual code, which in turn must be able to know about them and interact with them.

Interaction with the host application is achieved through the use of a dedicated instruction that allows the virtual program to invoke a native function call provided by the host. These native calls have access to the virtual processor instance state and can thus read / write registers, stack etc.

A loaded and linked virtual program may expose many virtual code entry points to the host application to call and likewise the host application can expose many native code entry points for the virtual code to call. A typical example would be a game engine in which the host application calls behaviours defined in virtual code for the game itself, such as AI or collision behaviours. The virtual game code would in turn call host native functions for tracing lines of sight, playing sound effects or saving the current player progress to a saved game file.

Similarities to a physical CPU

- Executes low level binary code compiled from a higher level language.
- Highly orthogonal instruction set based on 16-bit words.
- Deals with basic data types, signed, unsigned integers and floating point.
- Load/Store Register based architecture with many register to register operations.
- Stack
- Program Counter

Key differences from a physical CPU

- No bitwise structure to instruction opcodes, opcodes are a straightforward enumeration.
    - The individual bits in the opcodes in many real CPU have a specific interpretation.

- No condition codes.
    - Uses explicit compare and branch opcodes instead, much closer to C style if/else logic.

- No supervisor state.
    - Real processors tend to implement privilege levels in which the most privileged levels are accessible only by the kernel.
    - As ExVM runs entirely in the context of a userspace application, no such privilege levels are required.

- No direct access access to host native addresses.
    - Linked object code for real processors typically contains direct pointer references. For example, the runtime address of the string literal in the typical “hello world” example will often be presented as a literal inlined address in the instruction stream on CPU that support such constructs.
    - ExVM instead uses unique, runtime enumerated Symbol ID values that are mapped to the host native address.
    - This solves problems caused by, for example, different pointer sizes on 32 and 64-bit hosts but also introduces some limitations, in particular on the number of symbols that are available.

Other noteworthy points

- Separate stacks for register backup, function return and general data.
- No separate registers for floating point values, the main General Purpose Registers can hold any supported integer or floating point value.#

There are 16 64-bit general purpose registers r0-r15. These can contain integer, floating point or address values. The interpretation of register contents is always inferred by the instruction being executed.

Most instructions take two general purpose registers as arguments, a source and a destination and are of the form:

destination = source <operator> destination

Many operations support multiple fundamental types. For example, multiplication supports all unsigned and signed integers from 8 to 64 bits as well as 32 and 64-bit IEEE floating point.

Implementation Levels

ExVM defines 4 Implementation Levels for a host interpreter:

- Level 0 supports operations on 8, 16 and 32-bit integer types only.
- Level 1 adds support for operations on 32-bit floating point types.
- Level 2 adds support for operations on 64-bit integer and floating point types.
- Level 3 adds the requirement for 64-bit memory support.

The intent of this is to allow some degree of scaling across platforms and hosts where support for floating point or larger integer sizes is impractical or simply undesirable. Object code will indicate which level a host must implement in order to be able to run it correctly.

There is no explicit requirement to support a 64-bit host memory except in Level 3. However it is reasonable to assume that a 64-bit host should support the full Level 2 specification rather than some smaller subset.

Register Model

General Purpose

ExVM has 16 general purpose registers that can hold any of the supported elemental types:

- The Level 2 and 3 specification requires that these are 64-bit wide.
- The Level 0 and 1 specification only requires that these are 32-bit wide although a 64-bit host may define them as 64-bit so that native addresses can be used for references.
- Registers are arranged so that operations on elements smaller than the register width are always applied to the least significant portion, irrespective of the host architecture.

Level	**63 (MSB)**	**31**	**15**	**7**
0	Not Applicable	u32, s32	u16, s16	u8, s8
1	Not Applicable	f32		
2, 3	u64, s64, f64			

When a register contains an address, it is assumed to occupy all bits of the register, even if the host only has a 32-bit memory model.

Stacks

Three separate stacks are maintained by the interpreter for general data, register backup and return addresses. These stacks are not directly addressable from the virtual code and are free to be realised as host native pointers.

Call Stack

The call stack is used to track the return address when a function is called. The implied type is pointer to instruction word, e.g. uint16*.

- Only calls to virtual code push a return address onto the stack.
- Calls to host native code do not affect the virtual machine program counter and therefore do not require any call stack interaction.
- Overflow is an error condition.
- There is no underflow condition possible for the call stack since the implication of the stack being empty is that the entry function has been returned from.

Data Stack

The data stack is used to push and pop data from general purpose registers. There is no implicit alignment of data on the data stack.

- Overflow and underflow of the data stack are error conditions.

Register

The register stack is used to temporarily save and restore entire registers. The implied type is the machine word that represents the full width of a register.

- Overflow and underflow of the register stack are error conditions.

Other

Status

Referencing Model

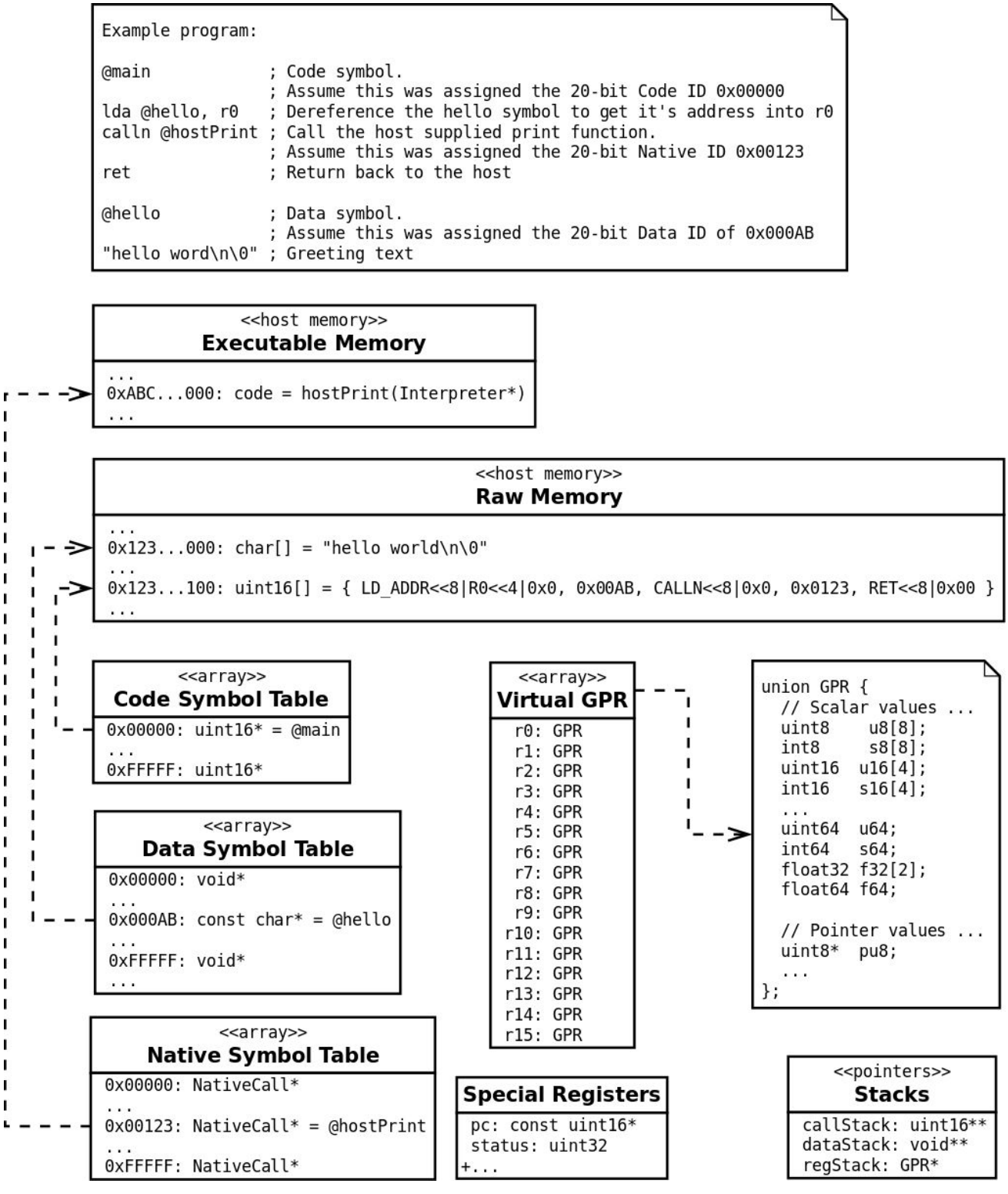
Irrespective of the pointer size of the host, ExVM does not allow static references to native addresses within object code. Instead, a platform agnostic ID based mechanism is used to refer to code, data and host-native functions. This approach has the following consequences:

- Identifiers in the instruction stream have a fixed size, independent of host pointer size.
- Identifiers can be readily validated during the linking phase.
- The actual addresses that identifiers point to can be changed at runtime, if necessary, without any self-modifying code behaviour requirements.
- Only one extension word is used for the symbol ID. This is combined with 4 unused bits of the operand byte to give a 20-bit Symbol ID at runtime. Therefore in the final linked object code there can only be 1048576 unique function symbols, 1048576 unique data symbols and 1048576 unique host-native function symbols.
- Resolving a symbol ID at runtime requires a ID to address lookup.

Note that while there can only be 1048576 unique data symbols, there is no restriction on how large or complex the data referenced by each symbol is.

Runtime Representation

The host layout for the canonical “hello world” program, once loaded and linked is depicted below. In this example, the host provides a native print function that expects the address of the string in r0. The virtual program loads the greeting text address into r0, invokes the native call then returns. The static references to the host native call and greeting string are decoupled from the underlying host addresses (and pointer sizes) by 20-bit symbol ID values that index tables of pointers to the actual entities.



Instruction Set

The ExVM Instruction set is based on a varying length 16-bit word format. Instruction opcodes are organised first by ascending Implementation Level, then by behavioural group.

- Organising by Implementation Level first ensures that the opcode range is always contiguous. This means that a Level 0 interpreter using a function table or switch case will not have to worry about gaps left by unimplemented Level 1 and 2 instructions.

Extension Words

Some instructions require additional operands, such as branch offsets, symbol identifiers and non-global literals. These are encoded into the instruction stream as extension words. For literals, the largest allowed type is 32-bit, which is assumed to be in host-native endian format (resolved during the loading and linking phase).

Base Instructions

Base instructions cover the majority of commonly used logic, arithmetic, branching, load and store operations. Base instructions consist of an opcode definition in the upper byte and typically a register pair operand in the lower byte.

- Additional data may be present in subsequent extension words.
- Most common dyadic operations are single 16-bit instructions.

Extended Instructions

Extended instructions provide direct access to more complex operations that would otherwise require host-native functions to implement, eg:

- Indexed load/store
- Transcendental floating point calculations
- Complex / Vector / Matrix arithmetic calculations
- Streaming vector (array) operations

Extended instructions consist of a class identifier in the upper byte and the extended opcode in the lower byte.

- Operands are encoded in the first extension word, often up to four register identifiers.