A simplified, universal instruction set for N-bit CPU architectures for N >= 32

Introduction

Previously; I had proposed and developed an instruction set and a compiler framework for a universal assembly language. The design and implementation of this system is now complete, and as a demonstration; features a JIT (just-in-time) compiler for the x86-64 architecture, as well as an experimental JIT for Android processors (including support for x86, ARM, ARM64, and MIPS). The code for the x86_64 release can be found here (TODO insert link), and the code for the Android port can be found here (TODO insert link).

Initial compilation

In order to develop tests for my bytecode format, I have created a project called UALCompiler, which contains an MSIL (Microsoft Intermediate Language) to UAL compiler. This allows users to write code in a .NET language, and cross-compile to UAL; as long as no dependencies on .NET framework types are introduced in the program. To run the cross-compiler; simply open the UALCompiler.sln file in MonoDevelop, enter your C# code in the Program.cs file, and hit run. If compilation succeeds, a list of MSIL instructions will be produced, and an output file will be produced in the UALCompiler/bin/Debug folder named ual.out. This ual.out file contains an intermediate representation of the program in UAL format. The UAL file cannot be ran as a standalone executable; as it depends on a UAL runtime (much like an MSIL file depends on the .NET framework or Mono runtime).

Running the UAL file

To run the UAL file; it is necessary to compile and run the UALRunner program. The UALRunner program performs a JIT of the code to native assembly, and then runs the compiled program in a sandboxed environment. If the UAL file is corrupted, an uncaught runtime error occurs, or a security exception occurs; the software will abort and dump its core; allowing for easy debugging with a program such as GDB or LLDB. UALRunner is also designed to output debug information during both runtime and compilation, to assist in debugging any errors.

Overview of execution of UAL files

When a UAL file is first opened, it is mapped into memory with the permission flag PROT_READ, and MAP_SHARED. The UAL header of the file will be parsed, and classes that are immediately relevant to execution will be compiled. Classes are scanned in the order in which they appear in UAL bytecode, but compilation is deferred until a method (typically Main) is invoked. Once Main is invoked; it may invoke other methods; including those in other classes. If a class that is referred to by Main is not yet compiled when a function is called, that function will be compiled as a dependency when the function is invoked. Once a class is compiled, it stays mapped in pageable memory with PROT_EXEC and PROT_READ permissions, and is freed

when the program exits. On desktop architectures; the JIT is not currently capable of sharing generated code between multiple instances of a process; so if multiple instances of a program are running, they may each have unique assembly code. On Android devices, the code is stored in the device's internal memory, and persists between multiple executions of a program.

Application Binary Interface for Native Code

Once the code is JITTed or loaded from a static file; it is possible for the native environment to use Reflection to invoke UAL methods directly as function pointers (the managed ABI will always be compatible with the native execution environment on each supported JIT platform). If you are writing native extensions to a UAL runtime, care should be taken to call GC_Mark and GC_Unmark; appropriately for managed objects. GC_Mark and GC_Unmark calls are essential to allow the garbage collector to trace all references to a given object. Note that the memory address of a managed object may unexpectedly change as you are working with it; so it should never be assumed that a managed memory address is static.

UAL supported data types

The UAL runtime natively supports 3 data types, Doubles, Words, and Objects. Additional data types can be built from these core types, or supported through intrinsics or native function calls.

Intrinsics for Strings

Strings can be loaded with the Load String Immediate (OPCODE 2) instruction. Opcode 2 will push a value of type System.String to the stack, which is a managed Object. UAL strings must be NULL-terminated with a \0 character. However; in cases where a \0 may appear in the middle of a string, and not represent the end of a string, this may not be the desired behavior. See the Array intrinsics below for an alternative. No other intrinsics are supported at the UAL level for strings.

Intrinsics for arrays

UAL does not have native support for arrays. However; literal Buffers can be instantiated using OPCODE 26 in order to simulate an array. A Buffer is effectively a byte array, represented as an opaque reference. A Buffer is encoded in UAL with a 32-bit integer representing the length of the array, followed by the raw bytes of the object. Buffers will have a data type of System.Blob.