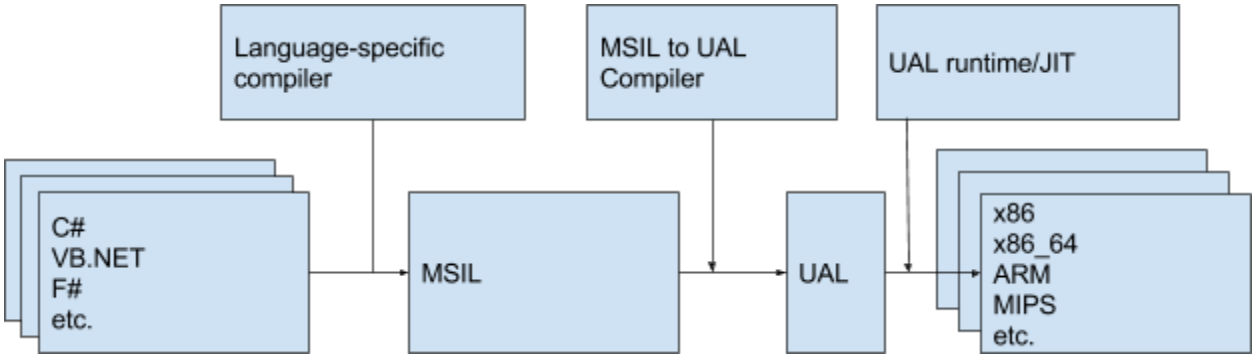
A simplified, universal instruction set for N-bit CPU architectures for N >= 32



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 (https://github.com/IDWMaster/CompilersProject), and the code for the Android port can be found here (https://github.com/IDWMaster/CompilersProject/tree/android).

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, passing the path to the UAL assembly file as a parameter, and any parameters you wish to pass to UAL after the name of the UAL file. 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

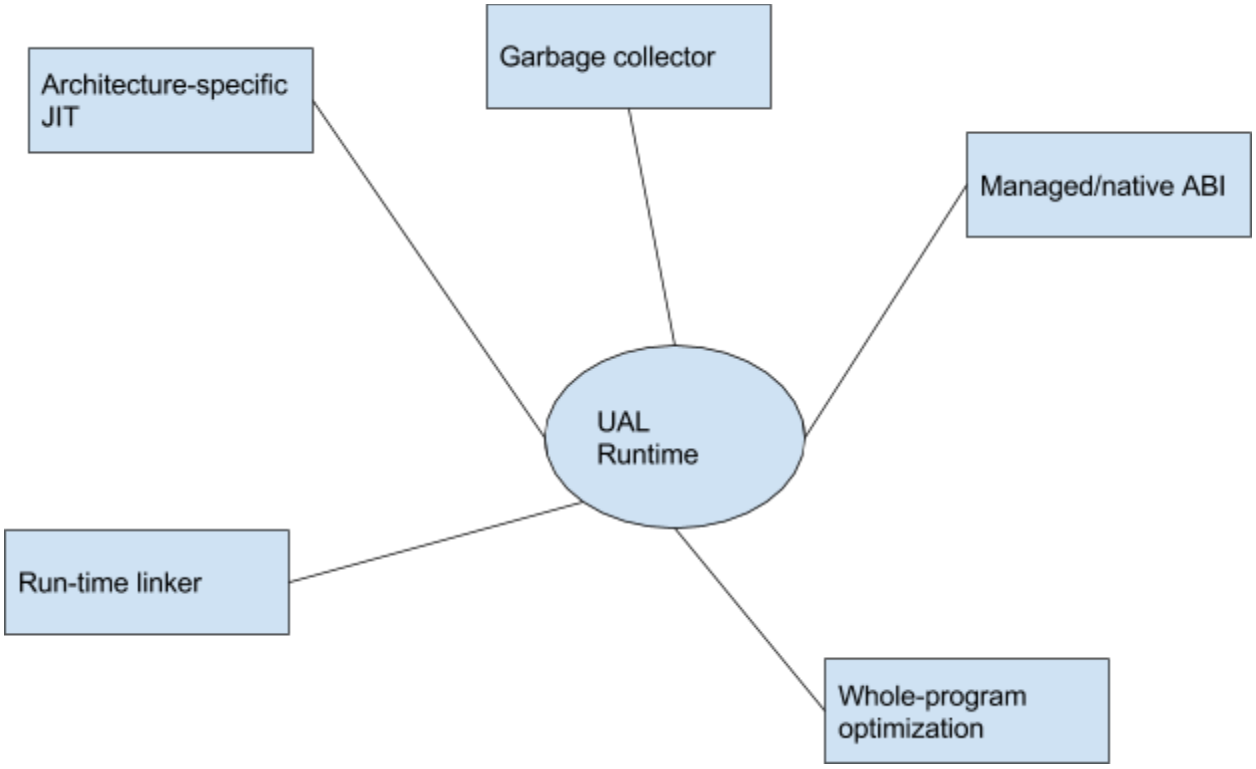
UAL does not have native support for arrays. However; literal Buffers can be instantiated using OPCODE 26 in order to simulate an array. A Buffer is effectively a byte array, represented as an opaque reference. A Buffer is encoded in UAL with a 32-bit integer representing the length of the array, followed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

First stage compilation

The first stage compilation takes place in the UALCompiler submodule. UALCompiler takes as input the name of the file to be compiled, and produces an output file named ual.out. The whole compilation process is implemented in a single C# file; ualspec.cs. UALSpec.cs processes the input file using Mono.Cecil, and converts the file, one instruction at a time into a UAL output file. Since UAL was designed to be compatible with all .NET functionality; there is typically a one-to-one mapping between IL and UAL instructions. This part of the project was implemented in the previous semester, along with a UAL interpreter.

JIT compilation

JIT compilation was implemented in the second semester of the project. The JIT compiler currently only works on the x86_64 architecture, with experimental (not functioning) support for Android devices. The JIT compiler is a component built directly into the UAL runtime, which is composed of the parts described in the diagram below.



Garbage collector

The garbage collector is a multi-generational, compacting collector. Managed objects are stored in multiple heaps (generations), and allocation in a given heap is as simple as incrementing a pointer. Each heap has two regions of memory -- in use memory, and free memory.

If a heap becomes too full, a garbage collection cycle occurs. During a garbage collection cycle, the heap is scanned for objects that are not referenced by any other objects. During this collection; the object graph is traversed, and each object discovered during the traversal process is marked as in use. Objects which have not been marked are then freed, by moving them over to the left. At the end of a collection cycle, the marker pointer is moved to the position denoting the new boundary between free and in-use memory.

Architecture-specific JIT

The architecture-specific JIT works by reading in the UAL file, creating a parse tree, optimizing the code, and emitting assembly code for the specific platform in question, and mapping it into memory. On all platforms where a JIT is supported; the code is JITted one class at a time, starting from the class containing the main method. For each class; it traverses all methods, and emits code; one method at a time. As each method is JITted; the method signature and address is inserted into a global linker table, allowing for fast resolution of functions during JIT code generation of other methods.

Parse tree components

During compilation of a method within a module, a parse tree is generated by the Parse function of UALMethod (see main.cpp in the Runtime folder of the CompilersProject repository).

- Module -- The UAL module containing assembly code. Each program can be linked with multiple UAL modules.
    - UALType -- A type contained in a Module. A type can contain multiple methods and fields.
        - Node -- A parse tree node, containing information about executable code
            - ConstantInt -- A constant integer expression
            - ConstantDouble -- A constant double expression
            - ConstantString -- A constant string expression
            - ConstantBuffer -- Support for array intrinsics and data literals embedded in a UAL assembly.
            - LdLoc -- An expression which loads a value from a local variable
            - StLoc -- An expression which stores a value into a local variable

        - LdArg -- An expression which loads an argument passed into a function
        - Ret -- An expression which returns from a function; optionally providing an output value.
        - Branch -- An expression which branches based on a comparison between a left-hand argument (Democrat), and a right-hand argument (Republican) -- to a specific UAL byte offset in a method.
            - UnconditionalSurrender -- Always branch (a complete and unconditional branch)
            - Ble -- Branch on less than or equal to
            - Blt -- Branch on less than
            - Bgt -- Branch on greater than
            - Bge -- Branch on greater than or equal to
            - Beq -- Branch on equal to
            - Bne -- Branch on not equal to
        - CallNode -- Invokes a function; either managed or native. Passes in a list of expressions evaluating to arguments which the function takes in.
        - BinaryExpression -- An expression which takes a Democrat and a Republican, and performs some operation on them.
            - Addition
            - Subtraction
            - Multiplication
            - Division
            - Remainder
            - Shift left
            - Shift right
            - AND
            - OR
            - NOT (only needs a Democrat)
            - XOR
    - Local -- A local variable used within a function
    - Argument -- An argument passed into a function
    - MethodSignature -- The signature of a method
    - Assembly -- The UAL module in which the method resides
    - Type -- The UAL type in which the method resides (may be 0 if native code)
    - isManaged -- Whether or not the function is managed (generated by the JIT); otherwise the function refers to native code.

Code generation

Code generation occurs within the Emit() function in main.cpp, located in the Runtime folder of the CompilersProject GIT repository. The Emit() function traverses the parse tree in a recursive manner, and generates the corresponding platform-specific assembly code for each platform. Currently; only x86_64 is supported, with experimental support for Android. On x86_64, I use a third-party assembler, called asmjit. Asmjit is an assembler which contains a built-in register allocator, as well as the ability to display the assembly code passed into it in a human-readable string format, which was very useful for debugging errors in my compiler. Rather than passing in assembly in an intermediate assembler language, the JIT assembler has a corresponding function for each x86_64 instruction, and calling the function generates x86_64 bytecode directly. This approach offers a tremendous performance advantage over traditional compilation models, in which the compiler emits assembler code in an intermediate; human-readable text form, and is then processed by a separate assembler. Asmjit allows for the JIT compilation, and generation of machine instructions to take place within the same process, without the need for an intermediate text representation.