
RFC - ENDIAN AGNOSTIC IR REPRESENTATION.

MOTIVATION:

In our current compilation model, a compiler will compile from a source language to an intermediate IR representation, perform optimizations, and then use the LLVM code generation infrastructure to target a specific backend. This approach works well for homogenous devices, but compilation models of compute languages, like OpenCL, now take place with heterogeneous devices (e.g. x86 core + gpu) as the targets. In some cases the devices have varying pointer sizes and byte ordering and this causes a problem for cross-device binary compatibility that a source language provides.

The major problem is that when creating a program for a heterogeneous system, different compilation paths are required for different devices for each source compilation. OpenCL, an Open Standard for compute on various types of devices, defines a source language that is portable across devices (i.e. same source on GPU's and CPUs). While compilation from source is portable, compilation from binary is not, and diverging compilation paths have issues with both maintenance and testing.

PROBLEM QUESTION:

How does a vendor simplify the compiler stack across multiple target devices by removing endianness from the IR representation?

PROPOSAL:

I am proposing an extension to LLVM[1] that abstracts away all endian related IR operations with a series of intrinsic calls. These intrinsic calls allow consumers of the IR to quickly reconstruct either the original IR, or an equivalent IR, with respect to the byte ordering of the target device. This IR representation provides an abstraction layer similar to the `hton[s]()` series of function calls with network programming.

OTHER SYSTEMS:

While this approach is similar to Google's PNaCl[3] which attempts to provide an ISA neutral representation. The goals of this proposal are slightly different in that we not only want ISA neutral representation, but also endian-neutral representation. Therefore, where PNaCl represents LLVM-IR before codegen, see figure 1 from [3], this approach provides a portable representation after the frontend and before LLVM-bitcode generation. The PNaCl representation makes assumptions on address space, data types, byte-order, concurrency and runtime system. This proposal inherits the assumptions on data types, address sizes, concurrency and runtime systems from OpenCL[4].

DEFINITIONS:

Global Memory - Memory that is visible to all threads in a process/program, e.g. video ram. This includes all read-only, write-only and read-write memories on the system that are visible to all threads.

INTRINSICS:

This proposal introduces new sets of intrinsics, two load intrinsics and two store intrinsics.

The sets are as follows:

```
declare <type> @llvm.portable.load.e.<type>(<type>* ptr, , i32 alignment, i1 host, i1 atomic, i1 volatile, i1 nontemporal, i1 singlethread) // little endian load
```

```

declare <type> @llvm.portable.load.E.<type><type>* ptr, i32 alignment, i1 host, i1 atomic, i1 volatile, i1
nontemporal, i1 singlethread) // big endian load
declare void @llvm.portable.store.e.<type><type> data, <type>* ptr, , i32 alignment, i1 host, i1 atomic, i1
volatilei1 nontemporal, i1 singlethread) // little endian store
declare void @llvm.portable.store.E.<type><type> data, <type>* ptr, i32 alignment, i1 host, i1 atomic, i1 volatile,
i1 nontemporal, i1 singlethread) // big endian store

```

A second smaller set could be:

```

declare <type> @llvm.portable.load.<type><type>* ptr, i32 alignment, i1 host, i1 littleEndian, i1 atomic, i1
volatile, i1 nontemporal, i1 singlethread)
declare void @llvm.portable.store.<type><type> data, <type>* ptr, i32 alignment, i1 host, i1 littleEndian, i1
atomic, i1 volatile, i1 nontemporal, i1 singlethread)

```

Valid values for type are scalar sizes i8, i16, i32, i64, f16, f32, f64 and vector versions with sizes of 2, 3, 4, 8 and 16 elements. Only pointers to the global address space, designated to separate it from the default address space in LLVM which is 0, with the pointer address space 1, are valid pointer values. The reason for the different address space is that requirement in OpenCL that the default address space is private memory, which conflicts with LLVM's default memory going to globally visible memory. For brevity, all possible combinations are not enumerated here. Another issue is with the data layout. A third option to the endianness is added to the LLVM reference manual that is defined as follows.

"p Specifies that the IR is in endian-portable form, i.e. code produced by little- and big-endian target back ends will be functionally equivalent (in their affect on global memory). The IR must be converted to a target format before the IR is valid LLVM-IR."

Using this data layout option will allow the compiler to quickly determine if the IR is in endian portable form.

PARAMETERS:

host – True when the load/store is from the host machine and false when the load/store is from the device.

atomic/volatile/nontemporal/singlethread – Follows the same semantics as the arguments to the load/store instructions in the LLVM-IR with the same names. See the LLVM Lang Ref.

POINTER ATTRIBUTES:

In OpenCL, a pointer can have attributes attached, and this information needs to be encoded. In LLVM, the method of encoding extra information is via metadata nodes and this is used so that the intrinsic do not need to be modified to add extra information. One example of this is the endian(host) attribute that can be attached to a pointer argument(see 6.10.3 of OpenCL 1.1 spec). This information can be encoded in a metadata node which is attached to the intrinsic. An example encoding of this information is as follows:

```

!0 = metadata !{
  i32, ;; Tag = <OpenCL version number> using the official OpenCL version macro
  i1, ;; Boolean value to specify that load is from host on true, device on false
  metadata ;; List of attributes for this intrinsic instruction
}

```

CONSTRAINTS:

Except for the data and ptr arguments, all arguments must be compile time constants.

Optimizations that rely on the byte ordering of memory or that modify the programs interactions with global memory are illegal to be performed on the IR when in the portable form.

All accesses to global memory must be done through these intrinsic calls.

LINKS:

1. <http://llvm.org/docs/LangRef.html>
2. <http://llvm.org/docs/Atomics.html>
3. <http://nativeclient.googlecode.com/svn/data/site/pnacl.pdf>
4. <http://www.khronos.org/opengl/>