A Tale of Two ABIs: ILP32 on AArch64

Tim Northover
Apple Inc

Introduction

- We regularly tell people that LLVM IR is target dependent.
- But what if you could design your targets?
- Meet armv7k and arm64_32.

Outline

- Generic target-dependence of LLVM IR
- Some of the steps we took to make our chosen ABIs work together.
- How AArch64 addressing modes work in ILP32 mode on LLVM.

ABI and IR Compatibility

IR is Target Dependent: Structs

- DataLayout specifies alignment of fields in structs & arrays.
- sizeof bakes this into the IR.
- There are at many different ways to handle bitfields.
    - Does the type of a field affect alignment?
    - Do zero-width fields force alignment?
    - Do they force 4-byte alignment regardless of type?

IR is Target Dependent: Function Calls

- Clang decides how to pass all function parameters, with intimate knowledge of the ABI and the backend.
- Largely involves input C or C++ type, and output an LLVM type.
- Can also insert unnamed padding types, for example to satisfy alignment types.
- Also various flags: indirect, “byval”, “inreg”.

Function Call Examples

```
struct Foo { int64_t a; };
```

- Clang chooses `i64`, LLVM uses register or 64-bit aligned stack slot.

```
struct Foo { int a, b; };
```

- Clang chooses `i64` since it has the same requirements.

```
struct Foo { int64_t arr[4]; };
```

- Too big, clang chooses `%struct Foo*`, in space allocated by calle

```
struct Foo { double arr[4]; };
```

- “Homogeneous Floating Aggregate”. Same size, but Clang chooses `[4 x double]`.

IR is Target Dependent: Miscellaneous

- C++ name mangling reveals types of `int32_t`: is it int or long?
- NEON SIMD intrinsics map to `@llvm.arm.*` calls.
- Inline assembly is right out.
- Headers produce platform specific type definitions: how much room for registers is there in `jmpbuf_t`?

The Solution: Work Backwards

- Design both ABIs at the same time to carefully sidestep these issues.
- Both must be ILP32 or LP64, and ILP32 makes more sense.
- Start with reasonably sensible 64-bit ABI and try to produce 32-bit ABI that produces IR which compiles in the same way. Some compromises have to be made on 32-bit side.

Implications for armv7k

- Closer to AArch64 ABI, even when it doesn't necessarily make sense for 32-bit ARM on its own.
- Types bigger than 16-bytes must be passed indirectly (compared to 4 on normal ARM).
- HFAs must be passed as such to IR, and use AArch64 rules.

Inevitable impact on arm64_32

- IR passes to translate ARM intrinsics to AArch64 equivalents.
- Special handling for array types.

Passing Structs

```
ct Foo { int a, b; };
ct Bar { long long a };
 takeFoo(int r0, Foo r1_r2);
 takeBar(int r0, Bar r2_r3);
```

arm64 →

```
declare void @takeFoo(i32 %r0, i64 %r1_r2)
declare void @takeBar(i32 %r0, i64 %r2_r3)
```

armv7k →

```
declare void @takeFoo(i32 %r0, i64 %r1_r2)
declare void @takeBar(i32 %r0, i64 %r2_r3)
```

%r1_r2 would go in r2 and r3 on ARM.

armv7k →

```
declare void @takeFoo(i32 %r0, i64 %r1_r2)
declare void @takeBar(i32 %r0, i32, i64 %r2
```

%r2_r3 would go in x2 on AArch64.

armv7k →

```
declare void @takeFoo(i32 %r0, [2 x i32] %
declare void @takeBar(i32 %r0, i64 %r2_r3)
```

%r1_r2 would go in x1 and x2 on AArch64.
But that's only convention

Avoidable impact on

Avoidable impact on arm64_32

- Still got bitfields wrong.
- Didn’t anticipate Swift assuming i64 is returned the same way as [2 x i32]. Used one variant in Swift CodeGen, another in C++ implementation.
- .NET used grey area calling conventions that aren’t really supported.

ILP32 Implementation

ILP32 in Panglossia

```
define i32 @load(i32* %base, i32 %n) {
  %elt = getelementptr inbounds i32, i32* %base, i32 %n
  %val = load i32, i32* %elt
  ret i32 %val
}
```

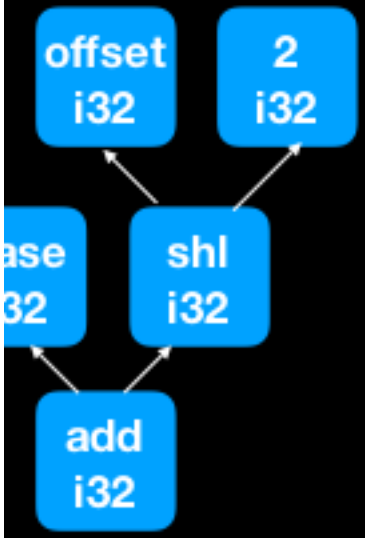
In principle this is perfect for AArch64's addressing modes

```
ldr w0, [x0,      w1,          sxtw                #2]
```

```
base + offset, sign extended & shifted left by 2
```

```
(add i64:$base, (shl (sext i32:$offset), 2)
```

ILP32 in LLVM



Q. If base is in a 32-bit register, what are in the high bits?

A. Usually nothing, but very difficult to prove it.

Q. If we do know the high bits of base are clear, should we zext or sext the shifted value before adding?

A. Impossible to say without knowing (dynamic) wrapping behaviour. Even nuw and nsw are no help.

```
0xffff_ffff + 0xffff_ffff => sext
0x0000_0000 + 0xffff_ffff => zext
```

ILP32 in LLVM

The Solution

- Make pointers i64 when in the DAG.
- Zero-extend at every load, truncate at every store.
- Inbounds GEPs get lowered to plain 64-bit arithmetic.
- Wrapping GEPs have to mask off high bits after that arithmetic.
- Signed comparisions special.

Verification

Verification

- Not just implementing new target, had to verify compatibility.
- Built test-suite with $RANDOM target (arm64_32 or armv7k -> arm64_32).
- Mixed and matched system frameworks and libraries.
- Direct App testing found the Swift issue.

Questions?