EMBECOSM®

Non 8-bit byte support in Clang and LLVM

Ed Jones, Simon Cook

CHAR_BIT

- From C99 standard §6.2.6.1

  "A byte contains CHAR_BIT bits, and the values of type unsigned char range from 0 to 2^CHAR_BIT – 1"
- CHAR_BIT is normally the number of bits in a native machine byte
  - Doesn't *have* to be though
- Also according to C99, CHAR_BIT >= 8

- But:
- POSIX says CHAR_BIT == 8
- Lots of people assume bytes are 8 bits, and that CHAR_BIT == 8

EMBECOSM®

Why does it matter?

- Not all processors have 8-bit bytes

  For example DSPs, and other domain specific processors
  - Often 16-bit bytes
  - Sometimes a bit weirder (24-bit bytes)
  - Sometimes a lot weirder (10/12-bit bytes)
- Have to have some support for *char*
- Useful to use a native machine type for *char*
  - String manipulation, legacy code, generic C tests and benchmarks
- Nice if it’s efficient too (no performance gotchas)

Support in Clang and LLVM

- LLVM IR is really nice
  - All bit based, doesn’t make any assumptions about byte size
  - IR optimizations generally just operate on bits, and don’t bake in assumptions about size of bytes
- DataLayout string specifies sizes and alignments in bits
- Clang has CharWidth, and targets can set char to be whatever size they want
- However...

Support in Clang and LLVM

- i8 pointers are often materialized in both Clang and LLVM (partly because there is no void* type)
  - For example SROA will create i8 pointers sometimes
- There are a lot of hardcoded “/8” and “*8”s all over the place.
  - Breaking this assumption triggers assertions + crashes
- Intrinsics used fixed width types
  - llvm.memcpy, llvm.memset, llvm.memmove all take i8 pointers

EMBECOSM®

Previous discussions

- Discussion has appeared on the mailing list a few time
  - http://lists.llvm.org/pipermail/llvm-dev/2017-January/108901.html
  - http://lists.llvm.org/pipermail/llvm-dev/2017-January/109335.html
  - http://lists.llvm.org/pipermail/llvm-dev/2009-September/026027.html
  - http://lists.llvm.org/pipermail/llvm-dev/2014-September/076543.html
  - http://lists.llvm.org/pipermail/llvm-dev/2015-March/083177.html
- Some groups seem to be maintaining out of tree patches
  - DCPU16, Ericsson, others (?)
- Various solutions have been mentioned
- We’ve tried some of them:

Solutions

- Set CharWidth = 16 in Clang
- Set CharWidth = 8, and CharAlign = 16
- Fat Pointers!
- Add ByteBitWidth to DataLayout in LLVM

Set CharWidth = 16 in Clang

- Simplest solution
- Clang has CharWidth which defines CHAR_BIT, and affect code generated by Clang
- Change CharWidth and it should just work...
- Clang still materializes Int8PtrTy pointers
- Only helps with Clang. LLVM has no equivalent
  - And LLVM makes assumptions that bytes are 8-bits in many places
- Doesn’t work out of the box, needs some changes to LLVM

Solutions

- Set CharWidth = 16 in Clang
- Set CharWidth = 8, and CharAlign = 16
- Fat Pointers!
- Add ByteBitWidth to DataLayout in LLVM

Set CharWidth = 8, CharAlign = 16

- To minimize changes to LLVM, use 8-bit bytes
- BUT, always align to the native byte size of 16-bits
- LLVM still sees an 8-bit byte, so should just work
- In the backend, halve all addresses to get word addresses
- For load/store, do a word operation and then mask off the top half

Set CharWidth = 8, CharAlign = 16

- Inefficient, need to halve addresses in the backend
  - Free for global addresses
  - BUT occasionally need to halve an address
  - Must be done late, or DAGCombiner folding may break things
- Must also mask off unwanted part during load or store
- Padding causes problems
  - i8 used for padding, but i8s also need to be padded
- If you somehow end up with an unaligned address, you get crashes (if you're lucky), or miscompilations

Solutions

- Set CharWidth = 16 in Clang
- Set CharWidth = 8, and CharAlign = 16
- Fat Pointers!
- Add ByteBitWidth to DataLayout in LLVM

Fat pointers

- Make i8* a fat pointer
  - http://lists.llvm.org/pipermail/llvm-dev/2015-March/083555.html
- Use 8-bit bytes, CHAR_BIT == 8
- Use a fat pointer for every i8*
  - Word address plus offset into the word

    struct { int word_address : 16; int byte_select : 1};
- During load/store through an i8*, mask off correct byte
- David Chisnall has been working on fat pointers for the Cheri backend
- Might be bad for performance
  - To address 64kW need 17-bits of pointer for i8*

Solutions

- Set CharWidth = 16 in Clang
- Set CharWidth = 8, and CharAlign = 16
- Fat Pointers!
- Add ByteBitWidth to DataLayout in LLVM

Add ByteBitWidth to DataLayout in LLVM

- What we eventually settled on
- Set CharWidth = 16 in Clang
- Add DataLayout::ByteBitWidth in LLVM
  - Allow target to specify in DataLayout string
- Fix places where Clang and LLVM assume 8-bit bytes to query the DataLayout instead
- Use bits instead of bytes if possible
- Requires a lot of small changes all over the place

Add ByteBitWidth to DataLayout in LLVM

Stumbling points:

- i8* being generated by Clang and LLVM
- Intrinsics have fixed width types
- Hardcoded *8 and /8 in the source code
- May want to keep ELF/DWARF using 8-bit bytes

Add ByteBitWidth to DataLayout in LLVM

i8* being generated by Clang and LLVM

- Type::getInt8PtrTy
- Replace calls to Type::getInt8PtrTy with Type::getIntNPtrTy, feed in the appropriate size from DataLayout
- Alternatively
  - Allow i8*, but legalize to a wider load?

Add ByteBitWidth to DataLayout in LLVM

Intrinsics use fixed-width types

- llvm.memset, llvm.memcpy, llvm.memmove
- Add a new pseudo ‘byte’ type used in the intrinsic definitions
- When the intrinsic is created, query DataLayout::getByteBitWidth to determine the correct type appropriate to a char
- Alternatively
  - Use llvm_anyptr_ty for intrinsics, choose the correct pointer type when the intrinsics is created

Add ByteBitWidth to DataLayout in LLVM

Hardcoded *8 and /8 in the source code

- Some important headers hardcode 8-bit bytes
  - EVT::getStoreSize, EVT::getStoreSizeInBits
  - MVT::getStoreSize, MVT::getStoreSizeInBits
  - DataLayout::getPointerTypeSize
  - DataLayout::getTypeStoreSize
  - StructLayout::getSizeInBits
  - StructLayout::getElementOffsetInBits
- In simple header files, with no access to DataLayout
- Need to feed in DataLayout, or somehow get the byte size in there

Add ByteBitWidth to DataLayout in LLVM

```
unsigned getStoreSize() const {
  return (getSizeInBits() + 7) / 8;
}

unsigned getStoreSizeInBits() const {
  return getStoreSize() * 8;
}
```

Add ByteBitWidth to DataLayout in LLVM

May want to keep ELF/DWARF using 8-bit bytes

- Machine can have a different byte-size to the on-disk representation in ELF/DWARF
- Easier to keep ELF/DWARF in 8-bit byte world
- Have to convert from machine byte width, to 8-bit bytes at some point

Add ByteBitWidth to DataLayout in LLVM

Advantages:

- CHAR_BIT matches the size of the smallest addressable unit
- No performance penalty

Disadvantages

- Maintaining a huge patch set isn’t fun
- Breaks commonly held assumption that byte == 8 bits

Status

- Changes implemented in a production compiler
- No patches for generic
  - Need to rebase against top of tree (internal screaming)
  - Need to tidy up and build into a patch set
- Only looked at CHAR_BIT == 16
  - No attempt to solve weirder CHAR_BIT sizes yet
  - Probably won’t work where CHAR_BIT % 8 != 0
- Need to write targeted tests

Future Work

- Submit patches for scrutiny and for general use
- Transplant patches into AAP (our experimental target)
  - Gives us something to test against
  - Could be in-tree target for this feature (?)
- Handle CHAR_BIT % 8 != 0

Thank You

www.embecosm.com