How to cross compile with LLVM based tools

Peter Smith, Linaro

Introduction and assumptions

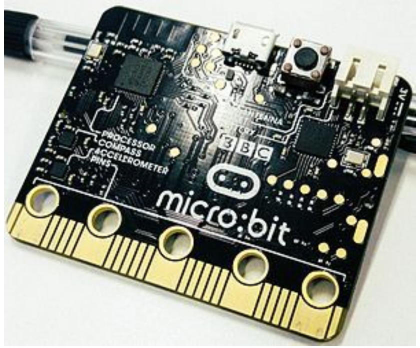
- What we are covering Today
  - What is cross compilation?
  - How does cross compilation work with Clang and LLVM?
  - The extra bits you need to build a Toolchain.
  - Building applications and running them on qemu.
- About me
  - Working in the Linaro TCWG team on LLVM.
  - Wrote the LLVM doc “How to cross compile Builtins on Arm”
    - After I found it significantly harder than I had expected.

Definitions

- **Host**
  - The system that we run the development tool on.
- **Target**
  - The system that we run the generated program on.
- **Native** compilation
  - Host is the same as Target.
- **Cross** compilation
  - Host is different to Target.

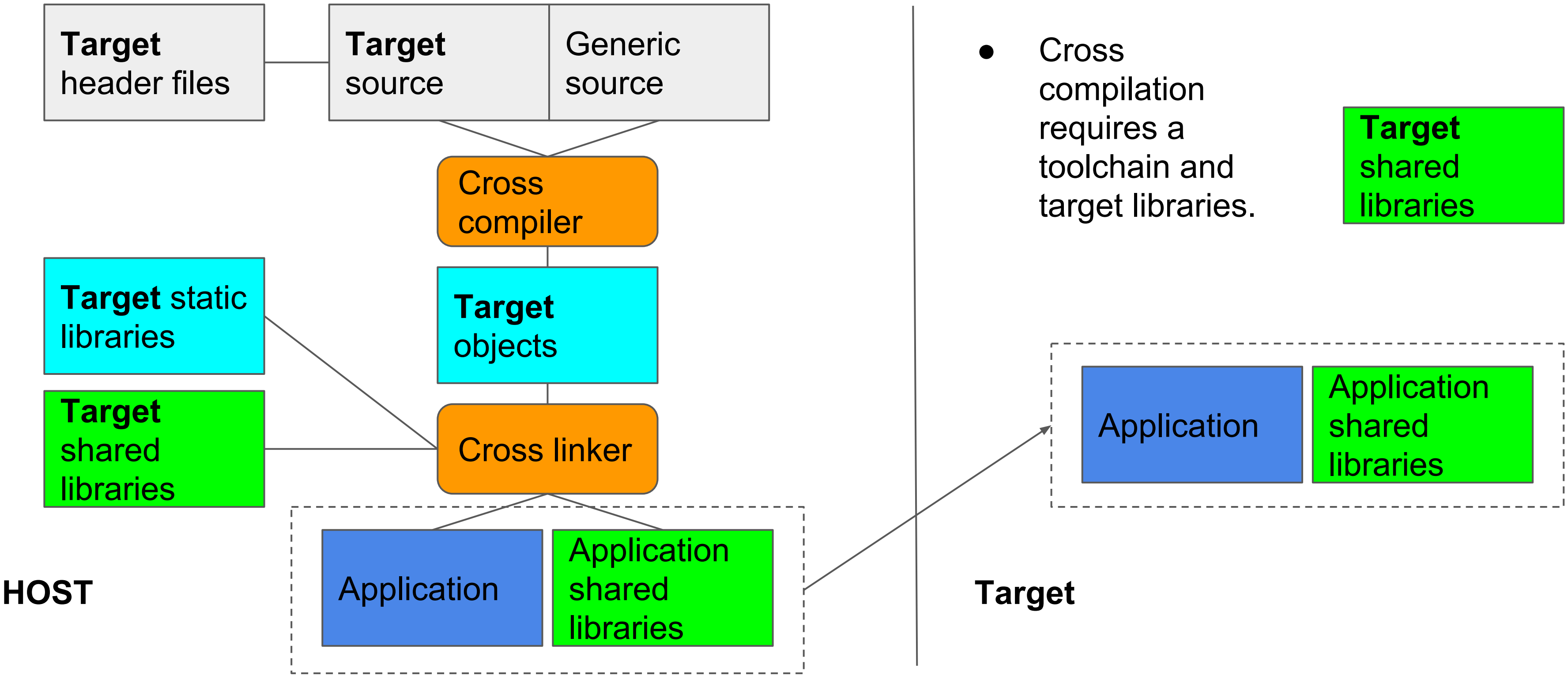
Motivation for cross compilation

- Can be a significant productivity boost
  - Host is faster than target.
- The only option available
  - Target can't run C/C++ compiler.
- Building an application for many platforms on the same machine.
- Bootstrapping a compiler on a new architecture.



A cross compiled application

- Cross compilation requires a toolchain and target libraries.



Complications

- Building an application requires more than just a compiler and linker
  - Header files for the target.
  - Target specific source code needs to be compiled for the right target.
  - Static and shared library dependencies for the target.
- Build system on the host needs to be configured.
- Shared library dependencies on target must be compatible with host.

Cross compilation and the LLVM toolchain

- Clang and other LLVM tools can work with multiple targets from same binary.
- Clang and LLD drivers can emulate drivers of other toolchains.
- Target controlled by the target triple.
- LLVM project does not have implementations of all the parts of toolchain.
- LLVM project includes some but not all of the library dependencies.

Toolchain components

Component	LLVM	GNU
C/C++ Compiler	clang	gcc
Assembler	clang integrated assembler	as
Linker	ld.lld	ld.bfd ld.gold
Runtime	compiler-rt	libgcc
Unwinder	libunwind	libgcc_s
C++ library	libc++abi, libc++	libsupc++ libstdc++
Utils such as archiver	llvm-ar, llvm-objdump etc.	ar, objdump etc.
C library		libc

Toolchain component choices

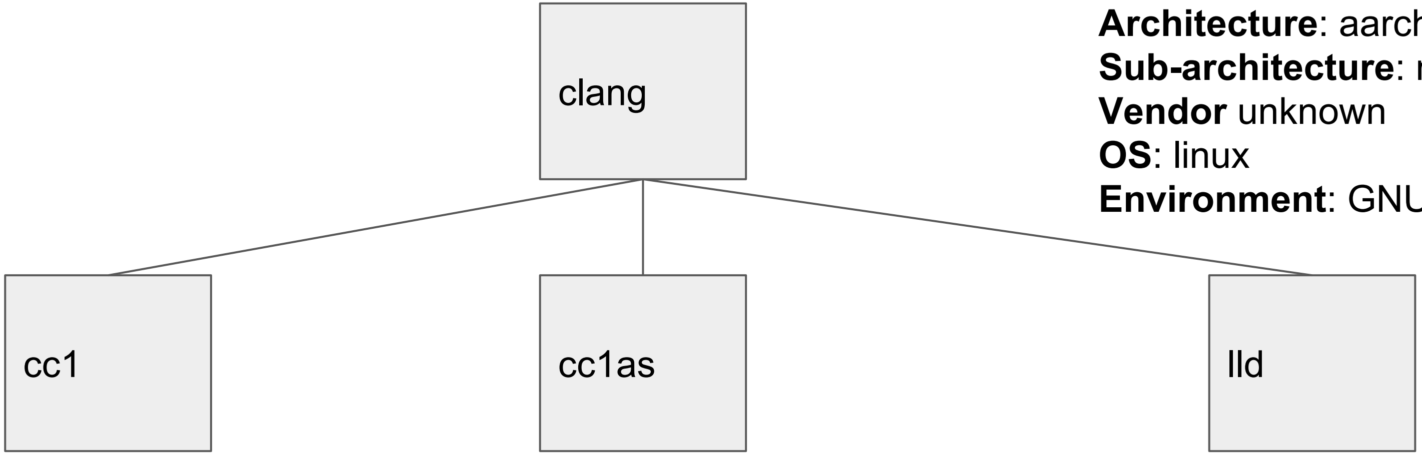
- Clang defaults chosen at build time, usually favour GNU libraries.
- Compiler runtime library
  - `--rtlib=compiler-rt`, `--rtlib=libgcc.`
  - Compiler-rt needed for sanitizers but these are separate from builtins provided by libgcc.
- C++ library
  - `--stdlib=libc++`, `--stdlib=libstdc++.`
  - No run-time option to choose C++ ABI library, determined at C++ library build time.
- Linker
  - `-fuse-ld=lld`, `-fuse-ld=bfd`, `-fuse-ld=gold.`
  - Driver calls `ld.lld`, `ld.bfd`, `ld.gold` respectively.
- C-library choice can affect target triple
  - For example arm-linux-gnueabi, arm-linux-musleabi.

Target Triple

- General format of <**Arch**><**Sub-arch**>-<**Vendor**>-<**OS**>-<**Environment**>
  - **Arch** is the architecture that you want to compile code for
    - Examples include arm, aarch64, x86_64, mips.
  - **Sub-arch** is a refinement specific to an architecture
    - Examples include armv7a armv7m.
  - **Vendor** captures differences between toolchain vendors
    - Examples include Apple, PC, IBM.
  - **OS** is the target operating system
    - Examples include Darwin, Linux, OpenBSD.
  - **Environment** includes the ABI and object file format
    - Examples include android, elf, gnu, gnueabihf.
- Missing parts replaced with defaults.

Clang Driver

`clang --target=aarch64-linux-gnu func1.s hello.c -o hello`



Architecture: aarch64
Sub-architecture: not applicable
Vendor unknown
OS: linux
Environment: GNU

```
clang-7.0 -cc1 -triple
aarch64-linux-gnu
-target-cpu=generic
-target-feature +neon
-target-abi aapcs
-Isystem /path/to/includes
...
```

```
clang-7.0 -cc1as -triple
aarch64-linux-gnu
-target-cpu=generic
-target-feature +neon
...
```

```
Ld.lld -m aarch64linux
-dynamiclinker
/lib/ld-linux-aarch64.so
-L /path/to/system/libraries
-lc
...
```

Clang driver and toolchains

- Driver mode
  - gcc, g++, cpp (preprocessor), cl (MSVC).
  - Set with option or inferred from filename clang, clang++, clang-cl.
- Target triple used to instantiate a ToolChain derived class
  - `arm-linux-gnueabihf` instantiates the Linux ToolChain.
  - `arm-none-eabi` instantiates the bare metal driver.
- Toolchain class has knowledge of how to emulate the native toolchain
  - Include file locations.
  - Library locations.
  - Constructing linker and non integrated assembler options.
  - Includes cross-compilation emulation.
- Not all functionality is, or could realistically be, documented.

Building a simple AArch64 Linux OS

- Choose to use compiler-rt, with undefined behaviour sanitizer with LLD as linker.
- Shopping list
  - AArch64 C library includes and library dependencies.
  - Clang, LLD.
  - AArch64 Compiler-rt sanitizer library.
  - qemu-aarch64 user mode emulator to test our application.

Obtaining toolchain components

- x86_64 host builds of clang and lld
  - Built from source on the host system.
  - The x86_64 stable release.
- Compiler-rt AArch64 libraries
  - Built from source (cross compiled) on the host system.
  - The aarch64 stable release
    - Prebuilt shared libraries have dependencies on libstdc++.
- C library and other library dependencies
  - Install AArch64 multiarch support.
  - Use a Linaro Binary Toolchain release.
    - Compilers, binutils, glibc...

Using a Linaro gcc toolchain from a directory

- Download and install the gcc toolchain for your target
    - https://releases.linaro.org/components/toolchain/binaries/
    - Should closely match your target triple. We will use is `aarch64-linux-gnu`.
    - Unpacks to a dir we'll call `installdir` containing:
        - `aarch64-linux-gnu, bin, include, lib, libexec, share`
- Clang needs to be given the toolchain location and the sysroot location
    - `--gcc-toolchain=/path/to/installdir`
        - Clang is looking for `lib/gcc/<target-triple>` subdir.
    - `--sysroot=/path/to/installdir/<target-triple>/libc`
- **Warning** other gcc toolchains may have small differences in directory layout.
- **Warning** without `--gcc-toolchain` clang will use heuristics to find tools
    - Will often find the host ld in `/usr/bin/ld` as the linker.

Location of runtime libraries

- Clang looks for a “resource directory” in a location relative to the binary for:
  - Compiler specific includes such as `stddef.h`
  - Target specific includes such as `arm_acle.h` and `arm_neon.h`
  - sanitizer includes in a subdirectory.
  - Runtime libraries such as compiler-rt.
- Print out location with `--print-resource-dir`
  - `../lib/clang/<release>/`
- AArch64 compiler-rt sanitizer libraries need to be in the `lib/linux` subdirectory of the resource directory.
  - If you have downloaded the LLVM release the x86 package will only contain x86 runtime libraries.
  - If you build compiler-rt yourself, you’ll need to install to the resource directory.

Building and running the application

- Test is a slightly modified version of the ubsan example
  - Modified to throw an exception.
- We want to use as much of the LLVM libraries as possible
  - `compiler-rt`
  - `libc++`, `libc++abi`, `libunwind`

```
#include <iostream>
#include <string>

int func(void) {
    throw std::string("Hello World\n");
    return 0;
}

int main(int argc, char** argv) {
    try {
        func();
    } catch (std::string& str) {
        std::cout << str;
    }
    int k = 0x7fffffff;
    k += argc; //signed integer overflow
    return 0;
}
```

Building and running the application

```
prompt$ root=/path/to/clang/install_dir
prompt$ sysroot=/path/to/linarogcc/aarch64-linux-gnu/libc
prompt$ ${root}/bin/clang++ --target=aarch64-linux-gnu -fsanitize=undefined \
                            --rtlib=compiler-rt --stdlib=libc++ \
                            -nostdinc++ -I${root}/include/c++/v1 \
                            -Wl,-L${root}/lib \
                            --sysroot ${sysroot} \
                            --gcc-toolchain=/path/to/linarogcc \
                            -rpath ${root}/lib \
                            example.cpp -o example

prompt$ qemu-aarch64 -L ${sysroot} example
Hello World
example.cpp:16:7: runtime error: signed integer overflow: 2147483647 + 1 cannot be
represented in type 'int'
```

Cross compiling the LLVM libraries yourself

- The home pages for the libraries have build instructions for standalone builds.
- Extra cmake options are required for cross-compilation.
- Build defaults to GNU library dependencies.
- Guide to cross compiling compiler-rt for arm available in LLVM docs.
- Similar principles can be used for libc++, libc++abi and libunwind.
- Need to be careful with the order that you build the libraries in.
    - Compiler-rt builtins do not depend on anything.
    - Libunwind needs c++ includes but not binary libraries.
    - Libc++abi needs c++ includes but not binary libraries and an unwinder such as libunwind.
    - Libc++ needs an abi library such as libc++abi.
    - Compiler-rt non builtin libraries need a c++ library.

Cross compilation hints and tips

- Name of clang binary is significant
  - `<target-triple>-clang -> clang --target=<target-triple>`
  - `<config-filename>-clang -> clang --config <config-filename>`
  - `clang<driver-mode> -> clang --driver-mode=<driver-mode>`
    - Most important is using `clang++` for C++ files.
- The `clang -v` option will show the gcc toolchain selected.
- The `--gcc-toolchain <path>` option can be pointed at a gcc cross toolchain
  - For example the Linaro Binary Toolchain releases.
- The `--sysroot` option can be used as root path for includes and libraries.
- Clang can use cross compilation support from multi-arch Linux distributions
  - A chroot or container is useful to maintain consistency of builds.

Cross compilation hints and tips

- When using shared libraries make sure the host and target have the same libraries
  - Be careful that any `rpath` option applies on the target system.
- For cmake cross compilation
  - The trycompile step may not pick options that set the target, sysroot and gcc-toolchain
    - Can pass these in with `-DCFLAGS` and `-DCXXFLAGS`
  - For compiling bare-metal static libraries the option `-DCMAKE_TRY_COMPILE_TARGET_TYPE=STATIC_LIBRARY` will skip the link step.
  - Use standalone builds with separate cmake invocations when cross-building the llvm libraries.
- You can use clang option `--print-search-dirs` to see effect of options
  - For example `--target`, `--sysroot`, `--gcc-toolchain`

Conclusions

- Clang can work well as a cross-compiler for a Linux environment if you have a cross compiling gcc installation available.
- Path of least resistance is to use the same default libraries as your target system.
- Clang bare-metal driver requires a lot of manual configuration.

Resources

- For general information: https://clang.llvm.org/docs/CrossCompilation.html
- How to cross compile clang itself: https://llvm.org/docs/HowToCrossCompileLLVM.html
- My own experience in cross-compiling compiler-rt for Arm https://llvm.org/docs/HowToCrossCompileBuiltinsOnArm.html
- How to assemble a toolchain using llvm components http://clang.llvm.org/docs/Toolchain.html

The End

Thanks for listening and good luck!

Backup

Building a bare-metal Arm application

- Target a Cortex-M3 with no RTOS, with newlib, compiler-rt, LLD linker.
    - --target=arm-none-eabi.
    - Use the GNU Arm Embedded Toolchain.
    - Clang will use the BareMetal Toolchain driver.
    - A linker script is needed to separate the Flash and Ram.
- Adapt the semihosting sample program from the GNU toolchain.
- Test with qemu-system-arm with -machine lms3811evb using semihosting
    - System mode emulates a development board and not just a CPU.

Complications

- No prebuilt binary for compiler-rt builtins on Cortex-M3
  - The compiler-rt builtins sources for armv7m include assembler files with floating point.
- The GNU ARM embedded toolchain sample uses specs files for configuration
  - Clang doesn't support specs files.
- The clang BareMetal driver doesn't support multilib
  - We will have to select the library that we need.
- LLD doesn't support the section type COPY in linker scripts
  - Have to place the heap and stack using a different method.
- LLD always adds .a suffix for -L<library> even if <library> already has one
- Clang integrated assembler, LLD don't handle the startup_CM3.S
  - Contains a .section with code but no "ax" flags.

Building an application for a Bare Metal device

- Building newlib with clang
  - Possible but there may be some source changes. For example on Arm
    - `__attribute__((naked))` on clang only allows assembler.
    - Integrated assembler does not support some of the syntax used.
- Building libc++, libc++abi and libunwind with newlib
  - Newlib 2.4 has some incompatibilities with libc++ locale
  - Depending on configuration Newlib may not define clock_gettime, needed by chrono
  - Some extra defines may be necessary:
    - `__GNU_VISIBLE`, `_GNU_SOURCE`, `LIBCPP_HAS_NO_ALIGNED_ALLOCATION`
  - Consider disabling threads and monotonic clock
  - Cmake option `-DCMAKE_TRY_COMPILE_TARGET_TYPE=STATIC_LIBRARY` skips trycompile link step.