Troubleshooting for Generating Bitcode Using Clang & WLLVM

A naive idea for generating bitcode for the `neuralert` project is to use clang (instead of gcc) to compile the project so that we can generate bitcode when compiling. The `wllvm` tool does exactly this stuff: it uses a clang (or gcc with dragonegg plugin) as its core, and after compilation, it takes an extra step to generate bitcode for that file. As long as you use `wllvm` as C compiler, we can get bitcode for analysis or symbolic execution.

The most difficult point is that clang and gcc are quite different compilers in essence, at least more than it appears to us. A code that can be compiled smoothly by gcc doesn't necessarily mean that clang can compile it as well (esp. for bottom-level code like this project), which we need to deal with manually.

To start with, we first need to add the following environment variables for `wllvm`:

```
# Use Clang as the core of wllvm
export LLVM_COMPILER=Clang
# Add target prefix for wllvm cross compilation
export BINUTILS_TARGET_PREFIX=arm-none-eabi
```

1. Clang always tries to link `compiler-rt` library, which doesn't exist during cross-compilation

- Error info: When trying to execute `cmake ..`, it will stop when checking C compiler and report it can find `compiler-rt` library.

As described above, the simplest idea is to use `wllvm` as the C compiler instead of the given gcc arm toolchain. But clang tries to visit its own `compiler-rt` library for arm, which doesn't exist in the host machine. This stops clang from compiling, since it will throw an error if it can't find `compiler-rt`.

We don't find a feasible way to disable this, so we brutely disable the default linking with the `-nodefaultlibs` argument, and manually handle linking issues later.

2. Disable unrecognizable compile options for clang

- Error info: clang reports that it can't recognize the argument `-fno-move-loop-invariants`.

  gcc and clang has different compile options since they have different backends. Thus, some compile options for gcc can't be recognized by clang.

  Thus, we comment the argument `fno-move-loop-invariants` in `da16200_sdk/cmake/da16200-toolchain.cmake`.

3. Manually link libraries for clang in `CMakeLists.txt`

- Error info: When trying to execute `make`, clang reports that it can't find many libraries such as `stdio.h`.

We need to manually find, include and link all C libs in arm gcc toolchains. Using `-sysroot=` option doesn't work properly as expected, since the actual toolchains is a little messy so clang can't find all includes/libs. For instance, `libc`, `libm` and `libnosys` are in one directory, while `libgcc` is in another. Moreover, clang links soft-float library by default despite the arguments.

After a day of finding the correct libs to link, we add the following lines to the `CMakeLists.txt`:

```
# Part 1. Import sysroot so CMake finds correct linker and other utils
set(CMAKE_SYSROOT "${TOOLCHAIN_PATH}/arm-none-eabi")

# Part 2. Set correct arm toolchains so clang can find them
# Set the correct place of headers included
include_directories(${TOOLCHAIN_PATH}/arm-none-eabi/include)
# Set the correct place of static-link libraries for c, m and nosys
set(CMAKE_LIBRARY_PATH "${TOOLCHAIN_PATH}/arm-none-eabi/lib/arm/v5te/hard")
# Link library for gcc exists in another place, so add it
link_directories("${TOOLCHAIN_PATH}/lib/gcc/arm-none-eabi/10.3.1/arm/v5te/hard")
# Set linker flags
set(CMAKE_EXE_LINKER_FLAGS "${CMAKE_EXE_LINKER_FLAGS} -L${TOOLCHAIN_PATH}/arm-
none-eabi/lib/arm/v5te/hard -L${TOOLCHAIN_PATH}/lib/gcc/arm-none-
eabi/10.3.1/arm/v5te/hard")

# Part 3. Manually link all libraries needed
target_link_libraries(${MAIN_TARGET} DA16200_SDK_MAIN c m gcc nosys)
```

The above codes ensure Clang finds and links the right libraries.

4. Adjust SDK codes so that Clang compiles all files smoothly

- Error info: Some files may fail to compile during `make` procedure.

This error is mainly caused by different built-in types or functions. For instance, `u32_t` and `__uint32_t` are different types in Clang(`unsigned int` and `unsigned long` respectively), but are the same type in gcc, which leads to compile error.

We fix these codes manually so that they can be compiled by Clang. Modifications include:

- Clang does not support floating point register `vfpcc`. We referenced strategy used by Chromium to fix it.

```
// cmsis_gcc.h
// Before modification
__ASM volatile ("VMSR fpscr, %0" : : "r" (fpscr) : "vfpcc");
// After modification
__builtin_arm_set_fpscr(0xdeadbeef);
```

- Clang don't have the same `typedef` macros as gcc, esp. about unsigned 32-bit integer (which can either be `unsigned int` or `unsigned long`). We have to manually fix certain macros to avoid redefinition/wrong definition.

```
// inet.h
// Before modification
typedef u32_t in_addr_t;
// After modification
typedef __uint32_t in_addr_t;

// cc_pal_interrupt_ctrl.c
// Before modification
typedef unsigned long uint32_t;
// After modification
typedef __uint32_t uint32_t;

// da16x_crypto_sbrom.c
// Before modification
typedef unsigned long uint32_t;
// After modification
typedef __uint32_t uint32_t;
```

- Clang doesn't use `UINT8` as the default `enum` type.

```
// app_common_support.h
// Before modification
typedef enum {/* some enums */}
// After modification
typedef enum: UINT8 {/* some enums */}
```

- Clang doesn't have function `spi_flash_open`, so we use `flash_open` instead. **Warning: It needs to check whether the two function do the exactly same thing.**

```
// user_command.c
// Before modification
SPI = spi_flash_open(SPI_MASTER_CLK, SPI_MASTER_CS);
// After modification
SPI = flash_open(SPI_MASTER_CLK, SPI_MASTER_CS);
```

After the above modification, the project can at least compile for one commit.

5. Using `ld` in the toolchain as linker

- Error info: Clang reports that the address of two sections overlap during linking.

During cross compilation, we need to use the `ld` in the toolchain instead of `ld.lld` in host machine to link so that all sections appear in the correct place.

We use `-fuse-ld=-fuse-ld=${TOOLCHAIN_PATH}/arm-none-eabi/bin/ld` to enable `ld` in the arm toolchain.

6. Clang needs `ld` scripts during linking

- Error info: Clang reports that `mem.ld` or `sections_xip.ld`

`ld` scripts are needed during linking, so we have to manually locate and link the above scripts to the project.

So we add `-L${CMAKE_CURRENT_SOURCE_DIR}/da16200_sdk/core/bsp/ldscripts` to the C compiler flags.

7. Built-in subroutine `__aeabi_l2f` does not exist in the toolchain

- Error info: Clang reports that the symbol `__aeabi_l2f` is undefined.

It is a weird error since this subroutine should exist in the static-link libraries. But we didn't actually find it.

As a result, we wrote a fake `__aeabi_l2f` function to trick Clang into believing the subroutine is defined. **Warning: The function `aeabi_l2f` always returns `0.0f`, which needs to be manually fixed in symbolic execution.**

In `placebo.h/placebo.cpp` we add function `__aeabi_l2f`, which always return `0.0f`.

8. Post-compilation linking errors

- Error info: After the progress of `make` reaches 100%, post-compilation phase can't find the correct static-link libraries.

This is caused by linking libraries for only the main targets. Some other targets in sdk may also needs to be linked.

After adding the following `target_link_libraries` command for `bootloader.cmake` in sdk, this problem is also resolved.

```
target_link_libraries(${SDK_BOOT_IMG} PUBLIC
    cryptopkg
    ${TOOLCHAIN_PATH}/arm-none-eabi/lib/arm/v5te/hard/libc.a
    ${TOOLCHAIN_PATH}/lib/gcc/arm-none-eabi/10.3.1/arm/v5te/hard/libgcc.a
    nosys
)
```