# Supplementary A: Guide to LLVM-based C/C++ Cross Compiler to Target ARM Cortex-M0 Microprocessor

\*\*Parent Project\*\*: Innovative Hardware Accelerator Architecture for FPGA-Based General-Purpose RISC Microprocessors (Article)

\*\*Author\*\*: Dr. Ehsan Ali

\*\*Email\*\*: <ehssan.aali@gmail.com>

\*\*Last update\*\*: 26th October, 2024.

\*\*URL: https://github.com/ehsan-ali-th/cortex\_m0\_MA/tree/master/Supplementaries

## Getting LLVM Source Code

1. Download the LLVM project source code (version = 18.1.8):

```console

wget https://github.com/llvm/llvm-project/releases/download/llvmorg-18.1.8/llvm-project-18.1.8.src.tar.xz

```

```console

tar xvf llvm-project-18.1.8.src.tar.xz

```

2. Apply patches:

Copy [./src/patches/CMakePolicy.cmake.patch](./src/patches/CMakePolicy.cmake.patch) into lvm-project-18.1.8.src/patches folder, then:

```console

cd llvm-project-18.1.8.src/

```

```console

mkdir patches

```

```console

cp ../src/patches/CMakePolicy.cmake.patch ./patches

```

```console

patch -p1 < ./patches/CMakePolicy.cmake.patch

```

## LLVM Stage-1 Build: Use host native compiler (e.g., GCC) to build LLVM Clang/Clang++ and LLD

\*\*Clang\*\*: A C language family frontend for LLVM.

\*\*Clang++\*\*: A C++ language family frontend for LLVM.

\*\*LLD\*\*: A linker from the LLVM project.

1. Create a build folder:

```console

cd llvm-project-18.1.8.src/

```

```console

mkdir build\_stage1 && cd build\_stage1/

```

2. Create [./src/stage1/stage1.sh](./src/stage1/stage1.sh) script:

```bash

#!/bin/bash

homedir=/home/durian

arminstallprefix=$homedir/arm-unknown-eabi

LLVMinstallprefix=$homedir/llvm

cmake -G Ninja ../llvm \

-DLLVM\_INSTALL\_UTILS=ON \

-DLLVM\_ENABLE\_PIC=ON \

-DLLVM\_ENABLE\_BINDINGS=OFF \

-DCMAKE\_BUILD\_TYPE=Release \

-DCMAKE\_INSTALL\_PREFIX=$LLVMinstallprefix \

-DLLVM\_TARGETS\_TO\_BUILD="Native" \

-DLLVM\_ENABLE\_PROJECTS="clang;lld" \

-DLLVM\_ENABLE\_RUNTIMES="compiler-rt;libcxx;libcxxabi;libunwind" \

-DLLVM\_BUILD\_DOCS=OFF \

-DLLVM\_ENABLE\_DOXYGEN=OFF

```

3. Run the CMAKE configure script:

```console

chmod +x stage1.sh

```

```console

./stage1.sh

```

4. Build and install:

```console

ninja

```

```console

ninja install

```

5. Verify that the stage-1 Clang is installed:

```console

/usr/local/bin/clang –version

```

Output:

```text

clang version 18.1.8

Target: x86\_64-unknown-linux-gnu

Thread model: posix

InstalledDir: /usr/local/bin

```

```console

/usr/local/bin/clang -print-target-triple

```

```console

x86\_64-unknown-linux-gnu

```

```console

/usr/local/bin/clang -print-targets

```

Output:

```text

Registered Targets:

x86 - 32-bit X86: Pentium-Pro and above

x86-64 - 64-bit X86: EM64T and AMD64

```

\*\*Note:\*\* According to: https://llvm.org/docs/AdvancedBuilds.html the Clang CMake build system supports bootstrap (aka multi-stage) builds. `-DCLANG\_ENABLE\_BOOTSTRAP=On` does this bootstrap mode with multiple stages. The `-DCLANG\_BOOTSTRAP\_PASSTHROUGH` allows passing arguments between stages.

Although using the above two arguments are possible, But we are going to perform the multi-stage build manually for the learning purposes. This allows the reader to see the details of every step and reduces confusion.

6. Compile a test C++ program:

We can now compile a C++ program on our local host X86\_64 machine using the freshly installed stage-1 LLVM Clang++ compiler (installed at local folder /home/esi/llvm/bin):

```console

sudo ldconfig /home/esi/llvm/lib/x86\_64-unknown-linux-gnu/

```

```console

/home/esi/llvm/bin/clang++ main.cpp -o main2 -I/home/esi/llvm/include/c++/v1/ -I/home/esi/llvm/include/x86\_64-unknown-linux-gnu/c++/v1/ --target=x86\_64-unknown-linux-gnu -fuse-ld=lld -stdlib=libc++ -L/home/esi/llvm/lib/x86\_64-unknown-linux-gnu/

```

## LLVM Stage-2 Build: Use the stage-1 Clang/Clang++ compilers to build LLVM Clang/Clang++ and LLD

Now we are going to re-compile the LLVM project, this time with the already built Clang/Clang++ in the stage-1:

1. Create a build folder:

```console

cd llvm-project-18.1.8.src/

```

```console

mkdir build\_stage2 && cd build\_stage2/

```

2. Create [./src/stage2/stage2.sh](./src/stage2/stage2.sh) script:

```bash

#!/bin/bash

homedir=/home/durian

arminstallprefix=$homedir/arm-unknown-eabi

LLVMinstallprefix=$homedir/llvm

LD\_FOR\_TARGET=$LLVMinstallprefix/bin/ld.lld cmake -G Ninja ../llvm \

-DLLVM\_INSTALL\_UTILS=ON \

-DLLVM\_ENABLE\_BINDINGS=OFF \

-DLLVM\_ENABLE\_PIC=ON \

-DLLVM\_USE\_LINKER="lld" \

-DCMAKE\_BUILD\_WITH\_INSTALL\_RPATH=ON \

-DCMAKE\_C\_COMPILER=$LLVMinstallprefix/bin/clang \

-DCMAKE\_CXX\_COMPILER=$LLVMinstallprefix/bin/clang++ \

-DCMAKE\_C\_FLAGS="-I$homedir/esi/llvm/include/c++/v1/ -I$homedir/llvm/include/x86\_64-unknown-linux-gnu/c++/v1/" \

-DCMAKE\_CXX\_FLAGS="-I$homedir/esi/llvm/include/c++/v1/ -I$homedir/llvm/include/x86\_64-unknown-linux-gnu/c++/v1/ -stdlib=libc++" \

-DCMAKE\_BUILD\_TYPE=Release \

-DCMAKE\_INSTALL\_PREFIX=$LLVMinstallprefix \

-DLLVM\_TARGETS\_TO\_BUILD="Native" \

-DLLVM\_ENABLE\_PROJECTS="clang;clang-tools-extra;lld;lldb;compiler-rt" \

-DLLVM\_ENABLE\_RUNTIMES="libcxx;libcxxabi;libunwind" \

-DCLANG\_ENABLE\_BOOTSTRAP=OFF \

-DLLVM\_BUILD\_DOCS=OFF \

-DLLVM\_ENABLE\_DOXYGEN=OFF

```

3. Run the CMAKE configure script:

```console

chmod +x stage2.sh

```

```console

./stage2.sh

```

4. Build and install:

```console

ninja

```

```console

ninja install

```

Now we have a fully functional LLVM Clang/Clang++ compilers and LLD linker with compiler builtins (Compiler-rt) and C++ standard libraries (libcxx).

## Built LLVM Clang/Clang++ Compilers for ARM Architecture

1. Create a build folder:

```console

cd llvm-project-18.1.8.src

```

```console

mkdir build\_ARM && cd build\_ARM

```

2. Create [./src/stage\_ARM/stage\_ARM.sh](./src/stage\_ARM/stage\_ARM.sh) script:

```bash

#!/bin/bash

homedir=/home/durian

arminstallprefix=$homedir/arm-unknown-eabi

LLVMinstallprefix=$homedir/llvm

LD\_FOR\_TARGET=$LLVMinstallprefix/bin/ld.lld cmake -G Ninja ../llvm \

-DLLVM\_INSTALL\_UTILS=ON \

-DLLVM\_ENABLE\_BINDINGS=OFF \

-DLLVM\_ENABLE\_PIC=ON \

-DLLVM\_USE\_LINKER="lld" \

-DCMAKE\_BUILD\_WITH\_INSTALL\_RPATH=ON \

-DCMAKE\_C\_COMPILER=$LLVMinstallprefix/bin/clang \

-DCMAKE\_CXX\_COMPILER=$LLVMinstallprefix/bin/clang++ \

-DCMAKE\_C\_FLAGS="-I$homedir/esi/llvm/include/c++/v1/ -I$homedir/llvm/include/x86\_64-unknown-linux-gnu/c++/v1/" \

-DCMAKE\_CXX\_FLAGS="-I$homedir/esi/llvm/include/c++/v1/ -I$homedir/llvm/include/x86\_64-unknown-linux-gnu/c++/v1/ -stdlib=libc++" \

-DCMAKE\_BUILD\_TYPE=Release \

-DCMAKE\_INSTALL\_PREFIX=$arminstallprefix \

-DLLVM\_TARGETS\_TO\_BUILD="ARM" \

-DLLVM\_TARGET\_ARCH="ARM" \

-DLLVM\_DEFAULT\_TARGET\_TRIPLE="armv6m-unknown-unknown-eabi" \

-DLLVM\_ENABLE\_PROJECTS="clang;clang-tools-extra;lld;lldb" \

-DCLANG\_ENABLE\_BOOTSTRAP=OFF \

-DLLVM\_BUILD\_DOCS=OFF \

-DLLVM\_ENABLE\_DOXYGEN=OFF

```

3. Run the CMAKE configure script:

```console

chmod +x stage\_ARM.sh

```

```console

./stage\_ARM.sh

```

4. Build and install:

```console

ninja

```

```console

ninja install

```

```console

ls /home/durian/arm-unknown-eabi/ -ls

```

Output:

```text

otal 32

4 drwxr-xr-x 2 durian durian 4096 Oct 28 18:46 bin

4 drwxr-xr-x 9 durian durian 4096 Oct 28 18:46 include

12 drwxr-xr-x 7 durian durian 12288 Oct 28 18:46 lib

4 drwxr-xr-x 2 durian durian 4096 Oct 28 18:46 libexec

4 drwxr-xr-x 3 durian durian 4096 Oct 28 18:46 local

4 drwxr-xr-x 7 durian durian 4096 Oct 28 18:46 share

```

Inside the /home/durian/arm-unknown-eabi/bin we have the following programs installed that can be executed on an x86\_64 machine but produce ARM code (cross-compiling):

- clang -> clang-18

- clang++ -> clang

- clang-18

- llc

- lld

\*\*Note\*\*: At this point we still cannot compile C/C++ programs as there are no standard library that provides standard I/O (e.g., stdio.h) or builtins (e.g. those functions provided by compiler-rt). Therefore, we need to compile the \*compiler-rt\* and \*newlib\* for ARM architecture.

5. Verify that the ARM Clang is installed:

```console

/home/esi/arm-unknown-eabi/bin/clang++ -print-targets

```

Output:

```text

Registered Targets:

arm - ARM

armeb - ARM (big endian)

thumb - Thumb

thumbeb - Thumb (big endian)

```

## Compile LLVM Compiler-RT Project for Baremetal ARM Cortex-M0 (Triple: armv6m-unknown-eabi)

To cross-compile compiler-rt for ARM architecture eon an x86\_64 host machine we need the following items:

- An x84\_64 build of LLVM/clang for the llvm-tools and llvm-config: We built this already in our stage-2 built step [See Stage-2](#llvm-stage-2-build-use-the-stage-1-clangclang-compilers-to-build-llvm-clangclang-and-lld).

- A clang executable with support for the ARM target: We built this already in our ARM built [See LLVM ARM](#built-llvm-clangclang-compilers-for-arm-architecture)

- Compiler-rt sources: The subfoler \*compiler-rt\* inside LLVM source code folder.

- An ARM sysroot: One can build this sysroot from scratch but it is too complicated. The easier option is to download an already built sysroot from the official developer section of ARM website: https://developer.arm.com/downloads/-/gnu-a

The LLVM compiler-rt project consists of:

- \*\*Builtins\*\*: a simple library that provides an implementation of the low-level target-specific hooks required by code generation and other runtime components.

- \*\*Sanitizer runtimes\*\*: runtime libraries that are required to run the code with sanitizer instrumentation.

- \*\*Profile\*\*: library which is used to collect coverage information.

- \*\*BlocksRuntime\*\*: a target-independent implementation of Apple "Blocks" runtime interfaces.

We need to compile the compiler-rt project for ARM Cortext-M0 Microprocessor. Before that one more concept should be explained: \*Triple\*.

### LLVM Target Triple

It is a dash separated string (text) that specifies the architecture which the generated machine code is compiled for. The word \*triple\* because it originally had exactly three parts (today it can have between two and five parts). The triple format is shown below:

\<arch>\<sub\_arch>-\<vendor>-\<sys>-\<env>

Therefore, if we would like to form the target triple text for ARM Cortex-M0 microprocessor we should have:

- The ARM architecture (`arm`)

- Cortex-M0 processor is ARMSubArch\_v6m so sub-architecture is (`v6m`)

- From any vendor (`unknown`)

- Possible values for sys = none, linux, win32, darwin, cuda, etc. (`unknown`)

- Environment type is embedded application binary interface (`eabi`)

Combining all the above and put them together yields the correct target triple: `armv6m-unknown-unknown-eabi`

\*\*Note\*\*: For the Cortex-M0 core, instruction memory and private peripheral bus (PPB) accesses are always little-endian. For the data memory, it is dependent on the implementation. So if we face an option that allows us to set the endianness we will opt for little-endian.

1. Download and copy an ARM sysroot to a folder

1. Either visit the ARM website https://developer.arm.com/downloads/-/gnu-a or download the ARM compiler using wget:

```console

wget https://developer.arm.com/-/media/Files/downloads/gnu-a/10.3-2021.07/binrel/gcc-arm-10.3-2021.07-x86\_64-aarch64-none-linux-gnu.tar.xz

```

\*\*Note\*\*: We need to navigate to "x86\_64 Linux hosted cross compilers" section and download the compiler for AArch64 ELF bare-metal target (aarch64-none-linux-gnu). Do not download the (arm-none-linux-gnueabihf compiler as it is compiled with hard floating point unit. The ARM Cortex-M0 has no hard floating point unit and we need to use software to emulate the floating point unit (soft-float).

2. Unzip:

```console

tar xvf

```

3. Copy the ARM sysroot to any folder (in my case `/home/esi/backup\_data/workspace/sysroot`):

```console

cp -r gcc-arm-10.3-2021.07-x86\_64-aarch64-none-elf/aarch64-none-elf/\* /home/esi/backup\_data/workspace/sysroot/

```

2. Create a build folder:

```console

cd llvm-project-18.1.8.src/

```

```console

mkdir build\_compiler\_rt && cd build\_compiler\_rt

```

3. Create [./src/stage\_ARM\_crt/stage\_AMR\_compiler-rt.sh](./src/stage\_ARM\_crt/stage\_AMR\_compiler-rt.sh) script:

```bash

#!/bin/bash

homedir=/home/durian

arminstallprefix=$homedir/arm-unknown-eabi

LLVMinstallprefix=$homedir/llvm

cmake ../compiler-rt -G Ninja \

-DLLVM\_CMAKE\_DIR=$homedir/backup\_data/workspace/llvm-project-18.1.8.src \

-DLLVM\_ENABLE\_PIC=ON \

-DCMAKE\_AR=$homedir/arm-unknown-eabi/bin/llvm-ar \

-DCMAKE\_ASM\_COMPILER\_TARGET="armv6m-unknown-unknown-eabi" \

-DCMAKE\_ASM\_FLAGS="--target=armv6m-unknown-unknown-eabi --sysroot=$homedir/backup\_data/workspace/sysroot -march=armv6m -mcpu=cortex-m0 -mfpu=none -mfloat-abi=soft" \

-DCMAKE\_C\_COMPILER=$homedir/arm-unknown-eabi/bin/clang \

-DCMAKE\_C\_COMPILER\_TARGET="armv6m-unknown-unknown-eabi" \

-DCMAKE\_C\_FLAGS="--sysroot=$homedir/backup\_data/workspace/sysroot -march=armv6m -mcpu=cortex-m0 -mfpu=none -mfloat-abi=soft -I$homedir/backup\_data/workspace/sysroot/aarch64-none-linux-gnu/libc/usr/include" \

-DCMAKE\_CXX\_FLAGS="--sysroot=$homedir/backup\_data/workspace/sysroot -march=armv6m -mcpu=cortex-m0 -mfpu=none -mfloat-abi=soft" \

-DCMAKE\_EXE\_LINKER\_FLAGS="-fuse-ld=lld" \

-DCMAKE\_NM=$homedir/arm-unknown-eabi/bin/llvm-nm \

-DCMAKE\_RANLIB=$homedir/arm-unknown-eabi/bin/llvm-ranlib \

-DCOMPILER\_RT\_BUILD\_BUILTINS=ON \

-DCOMPILER\_RT\_BUILD\_LIBFUZZER=OFF \

-DCOMPILER\_RT\_BUILD\_MEMPROF=OFF \

-DCOMPILER\_RT\_BUILD\_PROFILE=OFF \

-DCOMPILER\_RT\_BUILD\_SANITIZERS=OFF \

-DCOMPILER\_RT\_BUILD\_XRAY=OFF \

-DCOMPILER\_RT\_DEFAULT\_TARGET\_ONLY=ON \

-DLLVM\_CONFIG\_PATH=$homedir/arm-unknown-eabi/bin/llvm-config \

-DCMAKE\_TRY\_COMPILE\_TARGET\_TYPE=STATIC\_LIBRARY \

-DCOMPILER\_RT\_OS\_DIR="baremetal" \

-DCOMPILER\_RT\_DEFAULT\_TARGET\_ONLY=ON \

-DCMAKE\_BUILD\_TYPE=Release \

-DCMAKE\_INSTALL\_PREFIX=$arminstallprefix \

-DLLVM\_TARGETS\_TO\_BUILD="ARM"

```

4. Run the CMAKE configure script:

```console

chmod +x stage\_AMR\_compiler-rt.sh

```

```console

./stage\_AMR\_compiler-rt.sh

```

4. Build and install:

```console

ninja

```

```console

ninja install

```

Output:

```text

0/1] Install the project...

-- Install configuration: "Release"

-- Installing: /home/durian/arm-unknown-eabi/include/orc/c\_api.h

-- Installing: /home/durian/arm-unknown-eabi/lib/baremetal/libclang\_rt.builtins-arm.a

-- Installing: /home/durian/arm-unknown-eabi/lib/baremetal/clang\_rt.crtbegin-arm.o

-- Installing: /home/durian/arm-unknown-eabi/lib/baremetal/clang\_rt.crtend-arm.o

```

Now we have the compiler-rt library for ARM architeture installed.

## Cross Compile the Newlib for ARM Architecture

[Newlib](https://sourceware.org/newlib/) is a C library intended for use on embedded systems

1. Download the newlib source code:

```console

wget ftp://sourceware.org/pub/newlib/newlib-4.2.0.20211231.tar.gz

```

2. Unzip the newlib source code:

```console

tar xvf newlib-4.2.0.20211231.tar.gz

```

3. Copy the newlib patches into the patches folder:

```console

mkdir newlib-4.4.0.20231231/patches

```

```console

cp -r ./src/newlib/patches/\* newlib-4.4.0.20231231/patches

```

```console

ls newlib-4.4.0.20231231/patches

```

Outputs:

```text

crt0.S.patch linux-crt0.c.patch README.md syscalls.c.patch

```

4. Apply the 3 patches to the newlib source code:

```console

patch -p0 < patches/crt0.S.patch

```

```console

patch -p0 < patches/linux-crt0.c.patch

```

```console

patch -p0 < patches/syscalls.c.patch

```

5. Compile the newlib source code:

```console

cd newlib-4.4.0.20231231

```

```console

mkdir build && cd build

```

```console

CC=/home/durian/llvm/bin/clang CFLAGS\_FOR\_TARGET="--target=armv6m-unknown-eabi -mcpu=cortex-m0 -mfloat-abi=soft -g -O2 -ffunction-sections -fdata-sections" CXX=/home/durian/llvm/bin/clang++ CXXFLAG\_FOR\_TARGETS="--target=armv6m-unknown-eabi -mcpu=cortex-m0 -mfloat-abi=soft -g -O2 -ffunction-sections -fdata-sections" CC\_FOR\_TARGET=/home/durian/arm-unknown-eabi/bin/clang CXX\_FOR\_TARGET=/home/durian/arm-unknown-eabi/bin/clang++ AR\_FOR\_TARGET=/home/durian/arm-unknown-eabi/bin/llvm-ar AS\_FOR\_TARGET=/home/durian/arm-unknown-eabi/bin/llvm-as RANLIB\_FOR\_TARGET=/home/durian/arm-unknown-eabi/bin/llvm-ranlib NM\_FOR\_TARGET=/home/durian/arm-unknown-eabi/bin/llvm-nm LIPO\_FOR\_TARGET=/home/durian/arm-unknown-eabi/bin/llvm-lipo OBJDUMP\_FOR\_TARGET=/home/durian/arm-unknown-eabi/bin/llvm-objdump DLLTOOL\_FOR\_TARGET=/home/durian/arm-unknown-eabi/bin/llvm-dlltool LD\_FOR\_TARGET=/home/durian/arm-unknown-eabi/bin/ld.lld WINDRES\_FOR\_TARGET=/home/durian/arm-unknown-eabi/bin/llvm-windres STRIP\_FOR\_TARGET=/home/durian/arm-unknown-eabi/bin/llvm-strip READELF\_FOR\_TARGET=/home/durian/arm-unknown-eabi/bin/llvm-readelf ../configure --prefix=/home/durian/arm-unknown-eabi --target=armv6m-unknown-eabi --enable-newlib-io-pos-args --enable-newlib-io-c99-formats --enable-newlib-io-long-long --enable-newlib-io-long-double --enable-newlib-register-fini --enable-newlib-retargetable-locking --disable-newlib-supplied-syscalls --enable-bootstrap=no

```

```console

make && make install

```

Now the newlib is installed inside /home/durian/arm-unknown-eabi/armv6m-unknown-eabi/:

```console

ls /home/durian/arm-unknown-eabi/armv6m-unknown-eabi/

```

Outputs:

```text

include lib

```

```console

ls /home/durian/arm-unknown-eabi/armv6m-unknown-eabi/include

```

Outputs:

```text

assert.h math.h malloc.h stdio.h stdlib.h time.h ...

```

```console

ls /home/durian/arm-unknown-eabi/armv6m-unknown-eabi/lib

```

Outputs:

```text

libg.a librdimon.a crt0.o libm.a libc.a libnosys.a ...

```

Now we can compile a C/C++ program for ARM Cortex-M0 !

### Custom Linker File

Based on our hardware (FPGA board) and address location of defined memories we can create a linker file and pass it to the clang using -T argument.

[cortexm0.ld](./src/cortexm0.ld) file:

cortexm0.ld

```text

ENTRY(main)

MEMORY {

ROM(rx) : ORIGIN = 0x0, LENGTH = 0x11FFFF /\* 1.125 MB \*/

ROM2(rx) : ORIGIN = 0x120000, LENGTH = 0x20000 /\* 128 KB \*/

RAM(rwx) : ORIGIN = 0x800000, LENGTH = 0x400000 /\* 4 MB \*/

}

SECTIONS {

/\* The code should be loaded at address 0x0 \*/

/\* the dot (.) symbol is location counter \*/

. = ORIGIN(ROM);

.text : {

/\* When link-time garbage collection is in use (‘--gc-sections’), it is often useful

to mark sections that should not be eliminated. This is accomplished by

surrounding an input section’s wildcard entry with KEEP() \*/

KEEP(\*(.vector\_table));

. = ALIGN(4);

\_\_vec\_end\_\_ = .;

/\* Input sections: .text section in all files. \*/

\* (.text)

. = ALIGN(4);

\_\_end\_text\_\_ = .;

} > ROM /\* assign .text section to a previously defined region of memory 'ROM' \*/

. = ORIGIN(ROM2);

.rodata : {

\_\_dataro\_start\_\_ = .;

\*(.rodata)

. = ALIGN(4);

\_\_dataro\_end\_\_ = .;

} > ROM2

. = ORIGIN(RAM);

.data : {

\_\_data\_start\_\_ = .;

\* (.data);

. = ALIGN(4);

\_\_heap\_low = .; /\* for \_sbrk \*/

. = . + 0x10000; /\* 64kB of heap memory \*/

\_\_heap\_top = .; /\* for \_sbrk \*/

\_\_data\_end\_\_ = .;

} > RAM

```

1. Create a sample C file:

```C

#include <stdio.h>

int main() {

int a, b, i;

i = 0;

a = 0;

b = 10;

for (i = 0; i < 10; i++) {

a = a + b + i;

b = b + 2;

printf ("i = %d \n", i);

}

return a;

}

```

2. Compile the main.c file using ARM clang:

```console

/home/durian/arm-unknown-eabi/bin/clang main.c -o main -T cortexm0.ld --sysroot=/home/durian/arm-unknown-eabi/armv6m-unknown-eabi --target=armv6m-unknown-unknown-eabi -march=armv6m -mcpu=cortex-m0 -mfpu=none -mfloat-abi=soft

```

Output:

```text

ld.lld: error: undefined symbol: \_exit

ld.lld: error: undefined symbol: \_close

ld.lld: error: undefined symbol: \_fstat

ld.lld: error: undefined symbol: \_read

ld.lld: error: undefined symbol: \_write

...

```

These undefined function are stubs that are used by the newlib and must be defined by the user of the newlib according to the hardware. The user must define how the hardware performs the input and output operation by placing appropriate code (either in C or assembly machine code) in the function such as \\_read, \\_write, \\_exit, etc. Because our goal here is to use the LLVM infrastructure in profiling the program we do not need to implement these functions and a blank implementation will suffice.

3. Edit the sample C file:

```C

include <stdio.h>

void \_exit(int status) {

while(1) {}

}

void \_close(int status) {}

void \_isatty(int status) {}

void \_read(int status) {}

void \_kill(int status) {}

void \_getpid(int status) {}

void \_sbrk(int status) {}

void \_write(int status) {}

void \_fstat(int status) {}

void \_lseek(int status) {}

int main() {

int a, b, i;

i = 0;

a = 0;

b = 10;

for (i = 0; i < 10; i++) {

a = a + b + i;

b = b + 2;

printf ("i = %d \n", i);

}

return a;

}

```

4. Compile the main.c file using ARM clang:

```console

/home/durian/arm-unknown-eabi/bin/clang main.c -o main -T cortexm0.ld --sysroot=/home/durian/arm-unknown-eabi/armv6m-unknown-eabi --target=armv6m-unknown-unknown-eabi -march=armv6m -mcpu=cortex-m0 -mfpu=none -mfloat-abi=soft

```

Output:

```text

ld.lld: error: unable to find library -lclang\_rt.builtins-armv6m

clang: error: ld.lld command failed with exit code 1 (use -v to see invocation)

```

It seems that we still cannot compile our C program due to the clang compiler is unable to find the compiler-rt library (libclang\_rt.builtins-armv6m.a). The library is already installed but the compiler cannot find it. We solve this by creating a symbolic link to the library:

```console

ln -s /home/durian/arm-unknown-eabi/lib/baremetal/libclang\_rt.builtins-arm.a /home/durian/arm-unknown-eabi/armv6m-unknown-eabi/lib/libclang\_rt.builtins-armv6m.a

```

Now issuing the compile command works flawlessly:

```console

/home/durian/arm-unknown-eabi/bin/clang main.c -o main -T cortexm0.ld --sysroot=/home/durian/arm-unknown-eabi/armv6m-unknown-eabi --target=armv6m-unknown-unknown-eabi -march=armv6m -mcpu=cortex-m0 -mfpu=none -mfloat-abi=soft

```