Essential Abstractions in GCC

Partition: GCC Code Organization

Logical parts are:
- Build configuration files
- Front end + generic + generator sources
- Back end specifications
- Emulation libraries
  (eg. libgcc to emulate operations not supported on the target)
- Language Libraries (except C)
- Support software (e.g. garbage collector)
GCC Code Organization

Front End Code

- Source language dir: $(SOURCE_DIR)/gcc/<lang dir>
- Source language dir contains
  - Parsing code (Hand written)
  - Additional AST/Generic nodes, if any
  - Interface to Generic creation

Except for C – which is the “native” language of the compiler
C front end code in: $(SOURCE_DIR)/gcc

Optimizer Code and Back End Generator Code

- Source language dir: $(SOURCE_DIR)/gcc

Back End Specification

- $(SOURCE_DIR)/gcc/config/<target dir>/
  Directory containing back end code
- Two main files: <target>.h and <target>.md, e.g. for an i386 target, we have
  $(SOURCE_DIR)/gcc/config/i386/i386.md and
  $(SOURCE_DIR)/gcc/config/i386/i386.h
- Usually, also <target>.c for additional processing code
  (e.g. $(SOURCE_DIR)/gcc/config/i386/i386.c)
- Some additional files

Configuration

Preparing the GCC source for local adaptation:

- The platform on which it will be compiled
- The platform on which the generated compiler will execute
- The platform for which the generated compiler will generate code
- The directory in which the source exists
- The directory in which the compiler will be generated
- The directory in which the generated compiler will be installed
- The input languages which will be supported
- The libraries that are required
- etc.
Pre-requisites for Configuring and Building GCC 4.6.0

- ISO C90 Compiler / GCC 2.95 or later
- GNU bash: for running configure etc
- Awk: creating some of the generated source file for GCC
- bzip/gzip/untar etc. For unzipping the downloaded source file
- GNU make version 3.8 (or later)
- GNU Multiple Precision Library (GMP) version 4.3.2 (or later)
- mpfr Library version 3.0.0 (or later)
  (multiple precision floating point with correct rounding)
- mpc Library version 0.8.2 (or later)
- Parma Polyhedra Library (ppl) version 0.11
- CLooG-PPL (Chunky Loop Generator) version 0.15.11
- jar, or InfoZIP (zip and unzip)
- libelf version 0.8.12 (or later)  \(\text{for LTO}\)

Our Conventions for Directory Names

- GCC source directory : \$(SOURCE_DIR)
- GCC build directory : \$(BUILD)
- GCC install directory : \$(INSTALL)
- Important
  - \$(SOURCE_DIR) \ne \$(BUILD) \ne \$(INSTALL)
  - None of the above directories should be contained in any of the above directories

Commands for Configuring and Building GCC

This is what we specify

- cd \$(BUILD)
- \$(SOURCE_DIR)/configure <options>
  configure output: customized Makefile
- make 2> make.err > make.log
- make install 2> install.err > install.log

Order of Steps in Installing GCC 4.6.0

- Building pre-requisites
  Build and install in the following order with \(--prefix=/usr/local\)
  Run \$\text{ldconfig}\ after each installation
  - GMP 4.3.2
  - CPPFLAGS=--fexceptions ./configure --enable-cxx ...
  - mpfr 3.0.0
  - mpc 0.8.2
  - ppl 0.11
  - cloog-ppl 0.15.11
  - libelf 0.8.12
- Building gcc
  Follow the usual steps.
**Configuring GCC**

```
configure
  + config.in
  + config/*
  - config.cache
  + config.log
  + config.status
  + config.sub

configure
  + config.h.in
  + Makefile.in
  + config.h
  - Makefile

config.guess
config.sub
config.log
config.cache
config.status
```

**Steps in Configuration and Building**

<table>
<thead>
<tr>
<th>Usual steps for a other than GCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Download and untar the source</td>
</tr>
<tr>
<td>cd $(SOURCE_D)</td>
</tr>
<tr>
<td>./configure</td>
</tr>
<tr>
<td>make</td>
</tr>
<tr>
<td>make install</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Steps for GCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Download and untar the source</td>
</tr>
<tr>
<td>cd $(BUILD)</td>
</tr>
<tr>
<td>$(SOURCE_D)/configure</td>
</tr>
<tr>
<td>make</td>
</tr>
<tr>
<td>make install</td>
</tr>
</tbody>
</table>

*GCC generates a large part of source code during a build!*

**Building a Compiler: Terminology**

- The sources of a compiler are compiled (i.e. built) on *Build system*, denoted BS.
- The built compiler runs on the *Host system*, denoted HS.
- The built compiler compiles code for the *Target system*, denoted TS.

The built compiler itself runs on HS and generates executables that run on TS.

**Variants of Compiler Builds**

- **Native Build**: BS = HS = TS
- **Cross Build**: BS = HS ≠ TS
- **Canadian Cross**: BS ≠ HS ≠ TS

**Example**

- **Native i386**: built on i386, hosted on i386, produces i386 code.
- **Sparc cross on i386**: built on i386, hosted on i386, produces Sparc code.
Notation for a Compiler

Bootstrapping: The Conventional View
Bootstrapping: The Conventional View

Language need not change, but the compiler may change
Compiler is improved, bugs are fixed and newer versions are released

To build a new version of a compiler given a built old version:

- Stage 1: Build the new compiler using the old compiler
- Stage 2: Build another new compiler using compiler from stage 1
- Stage 3: Build another new compiler using compiler from stage 2

Stage 2 and stage 3 builds must result in identical compilers

⇒ Building cross compilers stops after Stage 1!

A Native Build on i386

Requirement: \( BS = HS = TS = i386 \)
- Stage 1 build compiled using \( cc \)
- Stage 2 build compiled using \( gcc \)
- Stage 3 build compiled using \( gcc \)
- Stage 2 and Stage 3 Builds must be identical for a successful native build
A Native Build on i386

Requirement: $BS = HS = TS = i386$

- Stage 1 build compiled using cc
- Stage 2 build compiled using gcc
- Stage 3 build compiled using gcc
- Stage 2 and Stage 3 Builds must be identical for a successful native build

Essential Abstractions in GCC
GCC Resource Center, IIT Bombay

Commands for Configuring and Building GCC Revisited

This is what we specify

- cd $(BUILD)
- $(SOURCE_D)/configure <options>
  configure output: customized Makefile
- make 2> make.err > make.log
- make install 2> install.err > install.log

Essential Abstractions in GCC
GCC Resource Center, IIT Bombay

Build for a Given Target

This is what actually happens!

- Generation
  - Generator sources in $(SOURCE_D)/gcc/gen*.c are read and generator executables are created in $(BUILD)/gcc/build
  - MD files are read by the generator executables and back end source code is generated in $(BUILD)/gcc

- Compilation
  - Other source files are read from $(SOURCE_D) and executables created in corresponding subdirectories of $(BUILD)

- Installation
  - Created executables and libraries are copied in $(INSTALL)
Examine the Build Process

Use the Build Browser `bb.py`
- Currently, it can only handle `make cc1`
- Reads the log post-facto and collects dependency information
- One can give queries interactively
- We will use it in the lab session

Building a MIPS Cross Compiler on i386

Requirement: BS = HS = i386, TS = mips
- Stage 1 build compiled using `cc`
- Stage 2 build compiled using `gcc`
  Its HS = mips and not i386!
Building a MIPS Cross Compiler on i386

Stage 1 Build

Stage 2 Build

Stage 2 build is inappropriate for cross build

Requirement: BS = HS = i386, TS = mips
- Stage 1 build compiled using cc
- Stage 2 build compiled using gcc
  Its HS = mips and not i386!

A More Detailed Look at Building

Partially generated and downloaded source is compiled into executables

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
Building a MIPS Cross Compiler on i386: A Closer Look

Stage 2 build is infeasible for cross build

- Stage 1 cannot build gcc but can build only cc1
- Stage 1 build cannot create executables
- Library sources cannot be compiled for mips using stage 1 build
- Stage 2 build is not possible

Stage 1 Build

mips assembly

Stage 2 Build

native cc, binutils, libraries

libraries

GCC Resource Center, IIT Bombay

30 June 2011 Config and Build: Configuration and Building: Basic Concepts 23/53

A Closer Look at an Actual Stage 1 Build for C

native cc, binutils, libraries

fixincl

gen*

c1

cpp

GCC sources

libraries

libiberty

GCC Resource Center, IIT Bombay

30 June 2011 Config and Build: Configuration and Building: Basic Concepts 24/53

A Closer Look at an Actual Stage 1 Build for C

native cc, binutils, libraries

fixincl

gen*

c1

cpp

GCC sources

GCC Resource Center, IIT Bombay

30 June 2011 Config and Build: Configuration and Building: Basic Concepts 24/53
A Closer Look at an Actual Stage 1 Build for C

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay

Generated Compiler Executable for All Languages

- Main driver
  - C compiler
  - C++ compiler
  - Fortran compiler
  - Ada compiler
  - Java compiler
  - Java compiler for generating main class
  - LTO driver
  - Objective C
  - Objective C++

Generated Compiler Executable for All Languages

- gcc for target
- target libraries
- libgcc

Difficulty in Building a Cross Compiler

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
Building a MIPS Cross Compiler on i386

GCC Source → Native cc

C mips

- Without headers
- Without libgcc

crossgcc1

Installed kernel headers + eglibc

crossgcc1

Initial libraries

C mips

GCC Source → Native cc

- Initial libraries
- C library source

crossgcc2

Final libraries
### Building a MIPS Cross Compiler on i386

![Diagram of crossgcc workflow]

- **Crossgcc1**: Initial libraries
- **Crossgcc2**: Final libraries
- **Native cc**: Native compiler
- **Crossgcc**: Cross compiler

### Problem with Native Build in Ubuntu 11.04

- GCC expects `asm` directory in `/usr/include`
- In Ubuntu 11.04, it is present in `/usr/include/i386-linux-gnu` and not in `/usr/include`
- Installing `gcc-multilib` using synaptic package manager creates the required symbolic links

### Common Configuration Options

- **--target**
  - Necessary for cross build
  - Possible host-cpu-vendor strings: Listed in `$(SOURCE_D)/config.sub`
- **--enable-languages**
  - Comma separated list of language names
  - Default names: `c, c++, fortran, java, objc`
  - Additional names possible: `ada, obj-c++, treelang`
- **--prefix=$(INSTALL)**
- **--program-prefix**
  - Prefix string for executable names
- **--disable-bootstrap**
  - Build stage 1 only
Building cc1 Only

- Add a new target in the Makefile.in
  
  `.PHONY cc1:
  cc1:
    make all-gcc TARGET-gcc=cc1$(exeext)
  `make all-gcc TARGET-gcc=cc1$(exeext)
- Configure and build with the command `make cc1`.

Configuring and Building GCC – Summary

- Choose the source language: C (``--enable-languages=c``)
- Choose installation directory: (``--prefix=<absolute path>``)
- Choose the target for non native builds: (``--target=sparc-sunos-sun``)
- Run: configure with above choices
- Run: `make`
  - generate target specific part of the compiler
  - build the entire compiler
- Run: `make install` to install the compiler

Tip
Redirect all the outputs:
```
$ make > make.log 2> make.err
```

Build failures due to Machine Descriptions

- Incomplete MD specifications ⇒ Unsuccessful build
- Incorrect MD specification ⇒ Successful build but run time failures/crashes
  (either ICE or SIGSEGV)

Part 3

Registering New Machine Descriptions
Registering New Machine Descriptions

- Define a new system name, typically a triple.
  e.g. spim-gnu-linux
- Edit \$(SOURCE_D)/config.sub to recognize the triple
- Edit \$(SOURCE_D)/gcc/config.gcc to define
  - any back end specific variables
  - any back end specific files
  - \$(SOURCE_D)/gcc/config/<cpu> is used as the back end directory
    for recognized system names.

Tip
Read comments in \$(SOURCE_D)/config.sub & \$(SOURCE_D)/gcc/config/<cpu>.

Registering Spim with GCC Build Process

We want to add multiple descriptions:

- Step 1. In the file \$(SOURCE_D)/config.sub
  Add to the case \$basic_machine
    - spim* in the part following
      # Recognize the basic CPU types without company name.
    - spim+-* in the part following
      # Recognize the basic CPU types with company name.

- Step 2a. In the file \$(SOURCE_D)/gcc/config.gcc
  In case \${target} used for defining cpu_type, i.e. after the line
  # Set default cpu_type, tm_file, tm_fp_file and xm_file ...
  add the following case

  spim+-* *
  cpu_type=spim
  ;;

  This says that the machine description files are available in the directory
  \$(SOURCE_D)/gcc/config/spim.

- Step 2b. In the file \$(SOURCE_D)/gcc/config.gcc
  Add the following in the case \${target} for
  # Support site-specific machine types.

  spim+-* *
  gas=no
gnu_ld=no
file_base="\`echo \${target}\ | sed \`s/-.*$//\`"
tm_file="\$(cpu_type)/\${file_base}.h"
tm_fp_file="\$(cpu_type)/\${file_base}.fp"
md_file="\$(cpu_type)/\${file_base}.md"
out_file="\$(cpu_type)/\${file_base}.c"
tmp_file="\$(cpu_type)/\${file_base}- protos.h"
md_out_file="\$(cpu_type)/\${file_base}.md";
echo \${target};
;;
Building a Cross-Compiler for Spim

- Normal cross compiler build process attempts to use the generated `cc1` to compile the emulation libraries (LIBGCC) into executables using the assembler, linker, and archiver.
- We are interested in only the `cc1` compiler.
- Use `make cc1`

Overview of Building a Cross Compiler

1. crossgcc1. Build a cross compiler with certain facilities disabled
2. Initial Library. Configure the C library using crossgcc1. Build some specified C run-time object files, but not rest of the library. Install the library’s header files and run-time object file, and create dummy libc.so
3. crossgcc2. Build a second cross-compiler, using the header files and object files installed in step 2
4. Final Library. Configure, build and install fresh C library, using crossgcc2
5. crossgcc. Build a third cross compiler, based on the C library built in step 4

Downloading Source Tarballs

Download the latest version of source tarballs

<table>
<thead>
<tr>
<th>Tar File Name</th>
<th>Download URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc-4.6.0.tar.gz</td>
<td>gcc.cybermirror.org/releases/gcc-4.6.0/</td>
</tr>
<tr>
<td>binutils-2.20.tar.gz</td>
<td>ftp.gnu.org/gnu/binutils/</td>
</tr>
<tr>
<td>Latest revision of EGLIBC</td>
<td>svn co svn://svn.eglibc.org/trunk eglibc</td>
</tr>
<tr>
<td>linux-2.6.33.3.tar.gz</td>
<td><a href="http://www.kernel.org/pub/linux/kernel/v2.6/">www.kernel.org/pub/linux/kernel/v2.6/</a></td>
</tr>
</tbody>
</table>
Setting Up the Environment for Cross Compilation

- Create a folder `crossbuild` that will contain the crossbuilt compiler sources and binaries.
  
  ```bash
  mkdir crossbuild
  cd crossbuild
  ```

- Create independent folders that will contain the source code of gcc-4.6.0, binutil, and eglibc.
  
  ```bash
  mkdir gcc
  mkdir eglibc
  mkdir binutils
  ```

- Create a folder that will contain the cross toolchain.
  
  ```bash
  mkdir install
  ```

- Create a folder that will have a complete EGLIBC installation, as well as all the header files, library files, and the startup C files for the target system.
  
  ```bash
  mkdir sysroot
  ```

  `sysroot` ≡ standard linux directory layout

Building Binutils

- Change the working directory to binutils.
  
  ```bash
  cd binutils
  ```

- Untar the binutil source tarball here.
  
  ```bash
  tar -xvf binutils-2.20.tar.gz
  ```

- Make a build directory to configure and build the binutils, and go to that directory.
  
  ```bash
  mkdir build
  cd build
  ```

  ```bash
  export prefix=<path_to_crossbuild/install>
  export sysroot=<path_to_crossbuild/sysroot>
  export host=i686-pc-linux-gnu
  export build=i686-pc-linux-gnu
  export target=mips-linux
  export target=powerpc-linux
  export linuxarch=mips
  export linuxarch=powerpc
  ```
Building Binutils

- Configure the binutils:
  ```
  crossbuild/binutils/build$. ../binutils-2.20/configure
  --target=$target --prefix=$prefix --with-sysroot=$sysroot
  ```

- Install the binutils:
  ```
  crossbuild/binutils/build$. make
  crossbuild/binutils/build$. make install
  ```

- Change the working directory back to crossbuild.
  ```
  crossbuild/binutils/build$. cd ~/crossbuild
  ```

Building First GCC

- Change the working directory to gcc.
  ```
  crossbuild$. cd gcc
  ```

- Untar the gcc-4.6.0 source tarball here.
  ```
  crossbuild/gcc$. tar -xvf gcc-4.6.0.tar.gz
  ```

- Make a build directory to configure and build gcc, and go to that directory.
  ```
  crossbuild/gcc$. mkdir build
  crossbuild/gcc$. cd build
  ```

  libgcc and other libraries are built using libc headers. Shared libraries like 'libgcc.s.so' are to be compiled against EGLIBC headers (not installed yet), and linked against 'libc.so' (not built yet). We need configure time options to tell GCC not to build 'libgcc.s.so'.

- Configure gcc:
  ```
  crossbuild/gcc/build$. ../gcc-4.6.0/configure
  --target=$target --prefix=$prefix --without-headers
  --with-newlib --disable-shared --disable-threads
  --disable-libssp --disable-libgomp --disable-libmudflap
  --enable-languages=c
  ```

  `--without-headers` ⇒ build libgcc without any headers at all. `--with-newlib` ⇒ use newlib header while building other libraries than libgcc.

  Using both the options together results in libgcc being built without requiring the presence of any header, and other libraries being built with newlib headers.

- Install gcc in the install folder:
  ```
  crossbuild/gcc/build$. PATH=$prefix/bin:$PATH make all-gcc
  crossbuild/gcc/build$. PATH=$prefix/bin:$PATH make install-gcc
  ```

  `--disable-threading` ⇒ enable the use of newlib header while building other libraries than libgcc.

- Change the working directory back to crossbuild.
  ```
  crossbuild/gcc/build$. cd ~/crossbuild
  ```
Installing Linux Kernel Headers

Linux makefiles are target-specific

• Untar the linux kernel source tarball.
  
  crossbuild$ .tar -xvf linux-2.6.33.3.tar.gz

• Change the working directory to linux-2.6.33.3
  
  crossbuild$ . cd linux-2.6.33.3

• Install the kernel headers in the sysroot directory:
  
  crossbuild/linux-2.6.33.3$ . PATH=$prefix/bin:$PATH make headers_install CROSS_COMPILE=$target- INSTALL_HDR_PATH=$sysroot/usr ARCH=$linuxarch

• change the working directory back to crossbuild.
  
  crossbuild/linux-2.6.33.3$ . cd ~/crossbuild

Installing EGLIBC Headers and Preliminary Objects

Using the cross compiler that we have just built, configure EGLIBC to install the headers and build the object files that the full cross compiler will need.

• Change the working directory to eglibc.
  
  crossbuild$ . cd eglibc

• Check the latest eglibc source revision here.
  
  crossbuild/eglibc$ . svn co svn://svn.eglibc.org/trunk eglibc

• Some of the targets are not supported by glibc (e.g. mips). The support for such targets is provided in the ‘ports’ folder in eglibc. We need to copy this folder inside the libc folder to create libraries for the new target.
  
  crossbuild/eglibc$ . cp -r eglibc/ports eglibc/libc

• Make a build directory to configure and build eglibc headers, and go to that directory.
  
  crossbuild/eglibc$ . mkdir build
  crossbuild/eglibc$ . cd build

• Configure eglibc:
  
  crossbuild/eglibc/build$ . BUILD_CC=gcc
  CC=$prefix/bin/$target-gcc AR=$prefix/bin/$target-ar RANLIB=$prefix/bin/$target-ranlib ../eglibc/libc/configure

  --prefix=/usr --with-headers=$sysroot/usr/include
  --build=$build --host=$target --disable-profile
  --without-gd --without-cvs --enable-add-ons

  EGLIBC must be configured with option ‘--prefix=/usr’, because the EGLIBC build system checks whether the prefix is ‘/usr’, and does special handling only if that is the case.

• We can now use the ‘install-headers’ makefile target to install the headers:
  
  crossbuild/eglibc/build$ . make install-headers
  install_root=$sysroot install-bootstrap-headers=yes

  ‘install-bootstrap-headers’ variable requests special handling for certain tricky header files.
  (autoconf 2.13 causes some problems. Get version 2.50 or later)

• There are a few object files that are needed to link shared libraries. We will build and install them by hand:
  
  crossbuild/eglibc/build$ . mkdir -p $sysroot/usr/lib
  crossbuild/eglibc/build$ . make csu/subdir_lib
  crossbuild/eglibc/build$ . cd csu
  crossbuild/eglibc/build$ . cp crt1.o crti.o crtn.o
  $sysroot/usr/lib
Installing EGLIBC Headers and Preliminary Objects

- Finally, ‘libgcc_s.so’ requires a ‘libc.so’ to link against. However, since we will never actually execute its code, it doesn’t matter what it contains. So, treating ‘/dev/null’ as a C source code, we produce a dummy ‘libc.so’ in one step:
  
  crossbuild/eglibc/build/csu$.
  $prefix/bin/$target-gcc
  -nostdlib -nostartfiles -shared -x c /dev/null -o
  $sysroot/usr/lib/libc.so

- change the working directory back to crossbuild.
  
  crossbuild/gcc/build$.
  cd ~/crossbuild

Building the Second GCC

With the EGLIBC headers and the selected object files installed, build a GCC that is capable of compiling EGLIBC.

- Change the working directory to build directory inside gcc folder.
  
  crossbuild$.
  cd gcc/build

- Clean the build folder.
  
  crossbuild/gcc/build$.
  rm -rf *

- Configure the second gcc:
  
  crossbuild/gcc/build$.
  ../gcc-4.6.0/configure
  --target=$target --prefix=$prefix --with-sysroot=$sysroot
  --disable-libssp --disable-libgomp --disable-libmudflap
  --enable-languages=c

Building Complete EGLIBC

With the second compiler built and installed, build EGLIBC completely.

- Change the working directory to the build directory inside eglibc folder.
  
  crossbuild$.
  cd eglibc/build

- Clean the build folder.
  
  crossbuild/eglibc/build$.
  rm -rf *

- Configure eglibc:
  
  crossbuild/eglibc/build$.
  BUILD_CC=gcc
  CC=$prefix/bin/$target-gcc
  AR=$prefix/bin/$target-ar
  RANLIB=$prefix/bin/$target-ranlib
  ../eglibc/libc/configure
  --prefix=/usr --with-headers=$sysroot/usr/include
  --build=$build --host=$target --disable-profile
  --without-gd --without-cvs --enable-add-ons
Building Complete EGLIBC

- install the required libraries in $sysroot:
  
crossbuild/eglibc/build$. PATH=$prefix/bin:$PATH make
crossbuild/eglibc/build$. PATH=$prefix/bin:$PATH make
install install_root=$sysroot

- change the working directory back to crossbuild.
  
crossbuild/gcc/build$. cd ~/crossbuild

At this point, we have a complete EGLIBC installation in '$sysroot', with header files, library files, and most of the C runtime startup files in place.

Building fully Cross-compiled GCC

Recompile GCC against this full installation, enabling whatever languages and libraries you would like to use.

- Change the working directory to build directory inside gcc folder.
  
crossbuild$. cd gcc/build

- Clean the build folder.
  
crossbuild/gcc/build$. rm -rf *

- Configure the third gcc.
  
crossbuild/gcc/build$. ../gcc-4.6.0/configure
  --target=$target --prefix=$prefix --with-sysroot=$sysroot
  --disable-libssp --disable-libgomp --disable-libmudflap
  --enable-languages=c

- Install the final gcc in the install folder:
  
crossbuild/gcc/build$. PATH=$prefix/bin:$PATH make
crossbuild/gcc/build$. PATH=$prefix/bin:$PATH make install

- change the working directory back to crossbuild.
  
crossbuild/gcc/build$. cd ~/crossbuild

At this point, we have a ready cross compile toolchain in $prefix, and EGLIBC installation in $sysroot.

Maintaining $sysroot Folder

Since GCC’s installation process is not designed to help construct sysroot trees, certain libraries must be manually copied into place in the sysroot.

- Copy the libgcc.s.so files to the lib folder in $sysroot.
  
crossbuild$. cp -d $prefix/target/lib/libgcc.s.so* $sysroot/lib

- If c++ language was enabled, copy the libstdc++.so files to the usr/lib folder in $sysroot.
  
crossbuild$. cp -d $prefix/target/lib/libstdc++.so* $sysroot/usr/lib

At this point, we have a ready cross compile toolchain in $prefix, and EGLIBC installation in $sysroot.
Part 5

Testing

Testing GCC

- Pre-requisites - Dejagnu, Expect tools
- Option 1: Build GCC and execute the command
  - make check
  - or
  - make check-gcc
- Option 2: Use the configure option --enable-checking
- Possible list of checks
  - Compile time consistency checks
    - assert, fold, gc, gcac, misc, rtl, rtlflag, runtime, tree, valgrind
  - Default combination names
    - yes: assert, gc, misc, rtlflag, runtime, tree
    - no
    - release: assert, runtime
    - all: all except valgrind

GCC Testing framework

- make will invoke runtest command
- Specifying runtest options using RUNTESTFLAGS to customize torture testing
  - make check RUNTESTFLAGS="compile.exp"
- Inspecting testsuite output: $(BUILD)/gcc/testsuite/gcc.log

Interpreting Test Results

- PASS: the test passed as expected
- XFAIL: the test unexpectedly failed
- UNSUPPORTED: the test is not supported on this platform
- ERROR: the testsuite detected an error
- WARNING: the testsuite detected a possible problem

GCC Internals document contains an exhaustive list of options for testing
Sample input file `test.c`:

```c
#include <stdio.h>
int main ()
{
    int a, b, c, *d;
    d = &a;
    a = b + c;
    printf ("%d", a);
    return 0;
}
```

For a powerpc architecture,

```bash
$ prefix/bin/powerpc-unknown-linux-gnu-gcc -o test test.c
```

Use `readelf` to verify whether the executable is indeed for powerpc

```bash
$ prefix/bin/powerpc-unknown-linux-gnu-readelf -lh test
```

ELF Header:
- Magic: 7f 45 4c 46 01 02 01 00 00 00 00 00 00 00 00 00
- Type: EXEC (Executable file)
- Machine: PowerPC

Program Headers:
- [Requesting program interpreter: /lib/ld.so.1]