Outline

• Code Organization of GCC
• Configuration and Building
• Registering New Machine Descriptions
• Building a Cross Compiler
• Testing GCC
Logical parts are:

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

- Source language dir: $(SOURCE_D)/<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_D)/gcc

Optimizer Code and Back End Generator Code

- Source language dir: $(SOURCE_D)/gcc

Back End Specification

- $(SOURCE_D)/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_D)/gcc/config/i386/i386.md and
  $(SOURCE_D)/gcc/config/i386/i386.h
- Usually, also <target>.c for additional processing code
  (e.g. $(SOURCE_D)/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)
  (for LTO)

Our Conventions for Directory Names

- GCC source directory: $(SOURCE_D)
- GCC build directory: $(BUILD)
- GCC install directory: $(INSTALL)
- Important
  - $(SOURCE_D) ≠ $(BUILD) ≠ $(INSTALL)
  - None of the above directories should be contained in any of the above directories
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 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.in
- config/*
- config.guess
- config.sub
- config.h.in
- Makefile.in
- config.log
- config.cache
- config.status
- config.h
- Makefile
- config.cache

### Steps in Configuration and Building

**Usual Steps**
- Download and untar the source
- cd $(SOURCE_DIR)
- ./configure
- make
- make install

**Steps in GCC**
- Download and untar the source
- cd $(BUILD)
- $(SOURCE_DIR)/configure
- $(SOURCE_DIR)/Makefile
- make
- make install

*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 \textit{Build system}, denoted BS.
- The built compiler runs on the \textit{Host system}, denoted HS.
- The compiler compiles code for the \textit{Target system}, denoted TS.

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

Variants of Compiler Builds

| BS = HS = TS | Native Build |
| BS = HS \neq TS | Cross Build |
| BS \neq HS \neq TS | Canadian Cross |

Example

- Native \textit{i386}: built on i386, hosted on i386, produces i386 code.
- Sparc cross on \textit{i386}: built on i386, hosted on i386, produces Sparc code.
T Notation for a Compiler

input language

output language

C

i386

cc

implementation or execution language

name of the translator

Bootstrapping: The Conventional View

input language

output language

Assembly language

Machine language

ass

m/c

m/c

Bootstrapping: The Conventional View

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
Bootstrapping: The Conventional View

Essential Abstractions in GCC
GCC Resource Center, IIT Bombay

Bootstrapping: GCC View

Essential Abstractions in GCC
GCC Resource Center, IIT Bombay

- 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

Notes
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
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`

This is what actually happens!

- **Generation**
  - Generator sources
    - `$(SOURCE_DIR)/gcc/gen*.c` are read and generator executables are created in `$(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 `$(SOURCE_DIR)` and executables created in corresponding subdirectories of `$(BUILD)`

- **Installation**
  - Created executables and libraries are copied in `$(INSTALL)`
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

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

Stage 2 build is inappropriate for cross build
A More Detailed Look at Building

Partially generated and downloaded source is compiled into executables

Existing executables are directly used

Notes

Building a MIPS Cross Compiler on i386: A Closer Look

Requirement: BS = HS = i386, TS = mips
Building a MIPS Cross Compiler on i386: A Closer Look

Requirement: $BS = HS = i386$, $TS = mips$

- *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 1 build is infeasible for cross build

we have not built binutils for mips
A Closer Look at an Actual Stage 1 Build for C

GCC sources

native cc + native binutils

libraries

libcpp: C preprocessor
 zlib: Data compression
 int1: Internationalization
 libdecnumber: Decimal floating point numbers
 libgomp: GNU Open MP

Essential Abstractions in GCC
GCC Resource Center, IIT Bombay
A Closer Look at an Actual Stage 1 Build for C

 GCC sources

 libraries

 native cc + native binutils

 libiberty

 fixincl

 gen*

 cc1

 xgcc

 cpp

 target binutils

 libgcc

 cc + binutils for stage 2
Difficulty in Building a Cross Compiler

Building gcc

Compiling libgcc

Building binutils

Notes

Building a MIPS Cross Compiler on i386

Native cc

Source

GCC

C

mips

without headers
without libgcc

crossgcc1
Building a MIPS Cross Compiler on i386

Installed kernel headers + eglibc

```
C  mips
  
GCC Source  Native cc
```

crossgcc1

Initial libraries

crossgcc2

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

```
crossgcc1

Initial libraries

GCC Source

Native cc

C library source

crossgcc2

Final libraries
```

Essential Abstractions in GCC

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

**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)
- 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 to
  - 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)
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.
Registering Spim with GCC Build Process

- 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_p_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"
    md_file="${cpu_type}/${file_base}.md"
    out_file="${cpu_type}/${file_base}.c"
    tm_p_file="${cpu_type}/${file_base}-protos.h"
    echo ${target}
    ;;
• 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.
  
  ```
  mkdir crossbuild
  cd crossbuild
  ```

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

• Create a folder that will contain the cross toolchain.
  
  ```
  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.
  
  ```
  mkdir sysroot
  ```

  `sysroot` ≡ standard linux directory layout
Setting the Environment Variables

Set the environment variables to generalize the later steps for cross build.

```
crossbuild$ export prefix=<path_to crossbuild/install>
crossbuild$ export sysroot=<path_to crossbuild/sysroot>
crossbuild$ export host=i686-pc-linux-gnu
crossbuild$ export build=i686-pc-linux-gnu
crossbuild$ export target=mips-linux OR
   export target=powerpc-linux
crossbuild$ export linuxarch=mips OR
   export linuxarch=powerpc
```

Building Binutils

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

- Untar the binutil source tarball here.
  ```
  crossbuild/binutils$ tar -xvf binutils-2.20.tar.gz
  ```

- Make a build directory to configure and build the binutils, and go to that directory.
  ```
  crossbuild/binutils$ mkdir build
  crossbuild/binutils$ cd build
  ```
Building Binutils

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

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

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

Building First GCC

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

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

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

  `libgcc` and other libraries are built using `libc` headers. Shared libraries like `libgcc.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.so`.
Building First GCC

- Configure gcc:
  ```bash
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:
  ```bash
crossbuild/gcc/build$ . PATH=$prefix/bin:$PATH make all-gcc
crossbuild/gcc/build$ . PATH=$prefix/bin:$PATH make install-gcc
```

- change the working directory back to crossbuild.
  ```bash
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:
  ```
  PATH=$prefix/bin:$PATH make headers_install CROSS_COMPILE=$target-
  INSTALL_HDRPATH=$sysroot/usr ARCH=$linuxarch
  ```

- Change the working directory back to crossbuild.
  ```
  crossbuild/eglibc$ 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
Installing EGLIBC Headers and Preliminary Objects

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

- Configure eglibc:
  
  ```sh
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:
  
  ```sh
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:
  
  ```sh
crossbuild/eglibc/build$ mkdir -p $sysroot/usr/lib
  crossbuild/eglibc/build$ make csu/subdir_lib
  crossbuild/eglibc/build$ cd csu
  crossbuild/eglibc/build/csu$ 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:

  ```bash
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.
  ```bash
crossbuild$.
  cd gcc/build
  
  Clean the build folder.
  ```bash
crossbuild/gcc/build$.
  rm -rf *
  
- Configure the second gcc:
  ```bash
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 the Second GCC

- install the second 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
  ```

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 $sysroot:
  ```bash
  crossbuild/eglibc/build$. PATH=$prefix/bin:$PATH make
  crossbuild/eglibc/build$. PATH=$prefix/bin:$PATH make
  install install_root=$sysroot
  ```

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

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

- Configure the third gcc:
  ```bash
  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 fully Cross-compiled GCC

- 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
  ```

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.so files to the lib folder in $sysroot.
  ```
crossbuild$: cp -d $prefix/$target/lib/libgcc*.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.
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
  ```bash
  make check RUNTESTFLAGS="compile.exp"
  ```
- Inspecting testsuite output: $(BUILD)/gcc/testsuite/gcc.log

Interpreting Test Results

- PASS: the test passed as expected
- XPASS: the test unexpectedly passed
- FAIL: the test unexpectedly failed
- XFAIL: the test failed as expected
- 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;
}
```

Use $prefix/bin$/target-gcc -o test test.c

For a powerpc architecture,

Use readelf to verify whether the executable is indeed for powerpc

Use readelf to verify whether the executable is indeed for powerpc