Workshop on Essential Abstractions in GCC

Getting Started with GCC: Configuration and Building

GCC Resource Center

(www.cse.iitb.ac.in/grc)

Department of Computer Science and Engineering,
Indian Institute of Technology, Bombay

July 2009

Outline

- Code Organization of GCC
- Configuration and Building
- Registering New Machine Descriptions
- Testing GCC

Part 1

GCC Code Organization

Code Organization Overview

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)

Code Organization Overview

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)

Our conventions

GCC source directory : `$(SOURCE)`

Front End Code

- Source language dir: `$(SOURCE)/<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)/gcc`

Optimizer Code and Back End Generator Code

- Source language dir: `$(SOURCE)/gcc`

Back End Specification

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

Part 3

Configuration and Building

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

- 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 4.2 (or later)
- MPFR Library version 2.3.2 (or later)

Our Conventions for Directory Names

- GCC source directory : `$(SOURCE)`
- GCC build directory : `$(BUILD)`
- GCC install directory : `$(INSTALL)`
- Important
  - `$(SOURCE)` ≠ `$(BUILD)` ≠ `$(BUILD)`
  - None of the above directories should be contained in any of the above directories

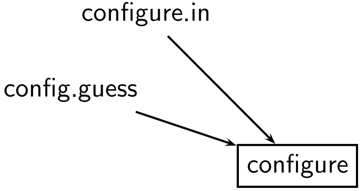
Configuring GCC

$\boxed{\text{configure}}$

Configuring GCC

$$\begin{tikzcd}
\text{config.guess} \arrow[dr] & \\
& \boxed{\text{configure}}
\end{tikzcd}$$

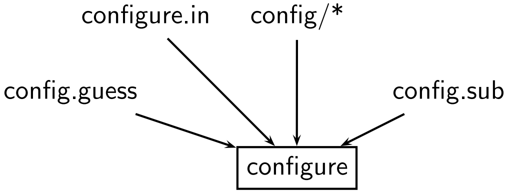
Configuring GCC



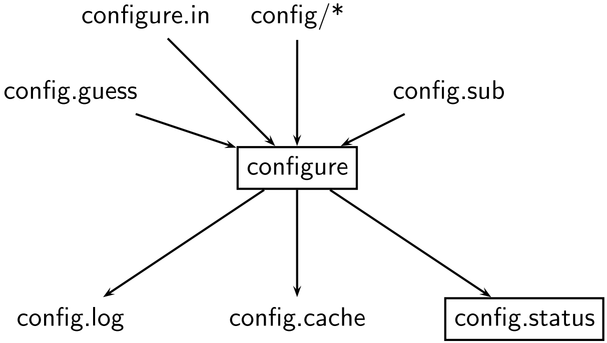
Configuring GCC

$$\begin{tikzcd}
 & \text{configure.in} \arrow[dr] & \text{config/*} \arrow[d] \\
\text{config.guess} \arrow[rr] & & \boxed{\text{configure}}
\end{tikzcd}$$

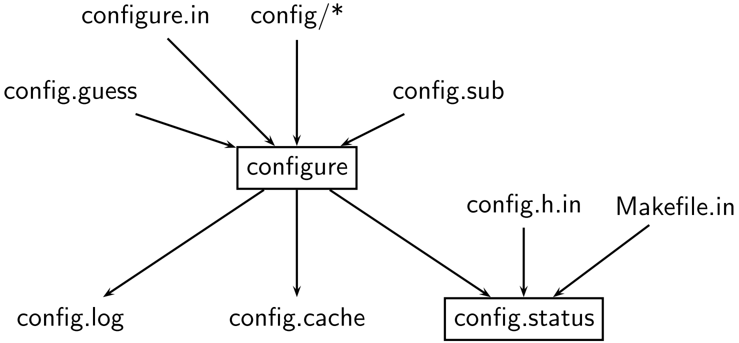
Configuring GCC


$$\begin{tikzcd}
 & \text{configure.in} \arrow[d] & \text{config/*} \arrow[dl] & \\
\text{config.guess} \arrow[r] & \boxed{\text{configure}} & & \text{config.sub} \arrow[ll]
\end{tikzcd}$$

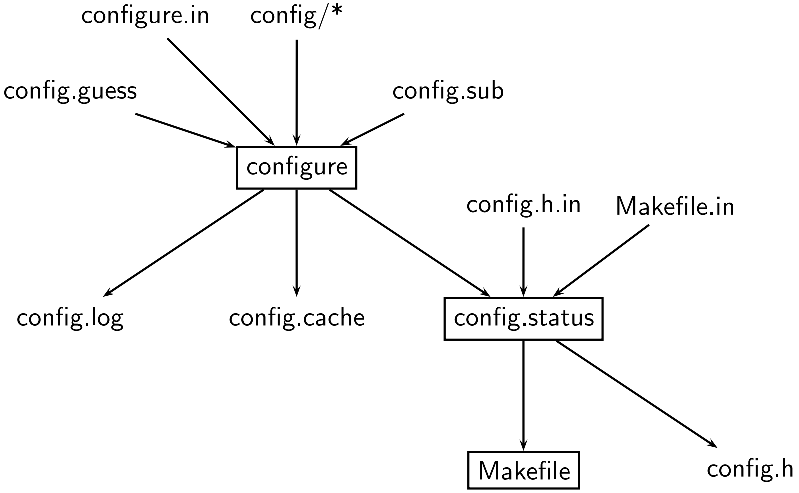
Configuring GCC



Configuring GCC

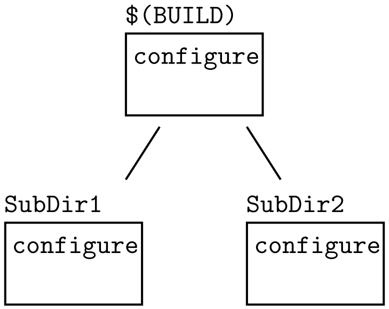


Configuring GCC



Alternatives in Configuration

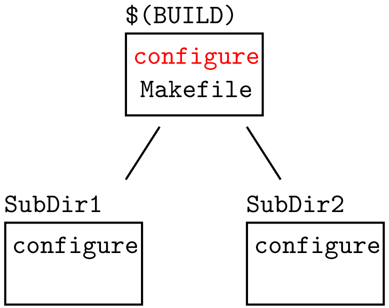
GCC 3.3



- Configure all directories recursively and create Makefiles
- Then run make in each directory recursively

Alternatives in Configuration

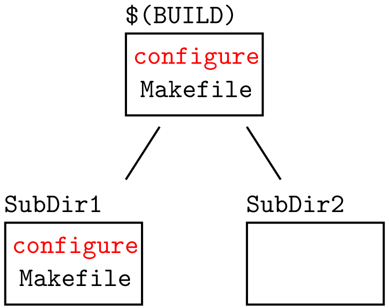
GCC 3.3



- Configure all directories recursively and create Makefiles
- Then run make in each directory recursively

Alternatives in Configuration

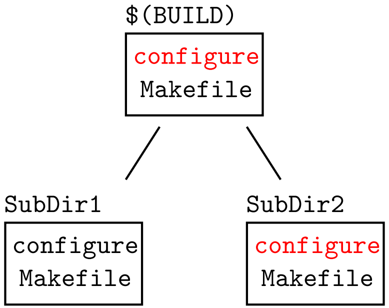
GCC 3.3



- Configure all directories recursively and create Makefiles
- Then run make in each directory recursively

Alternatives in Configuration

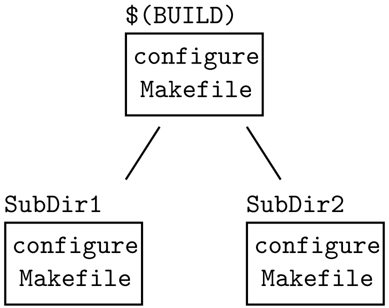
GCC 3.3



- Configure all directories recursively and create Makefiles
- Then run make in each directory recursively

Alternatives in Configuration

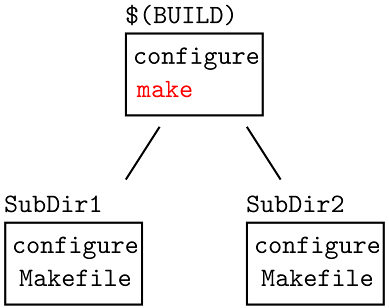
GCC 3.3



- Configure all directories recursively and create Makefiles
- Then run make in each directory recursively

Alternatives in Configuration

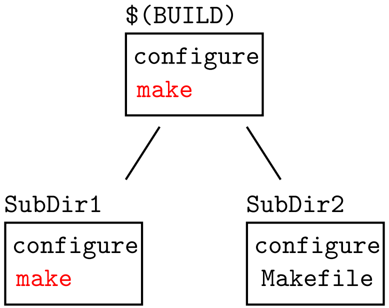
GCC 3.3



- Configure all directories recursively and create Makefiles
- Then run make in each directory recursively

Alternatives in Configuration

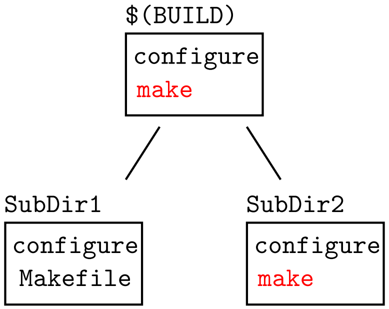
GCC 3.3



- Configure all directories recursively and create Makefiles
- Then run make in each directory recursively

Alternatives in Configuration

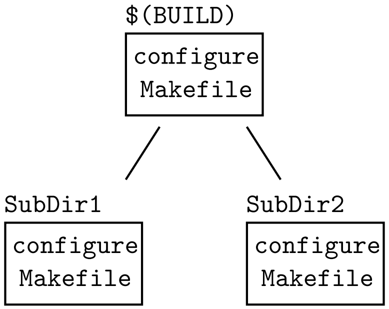
GCC 3.3



- Configure all directories recursively and create Makefiles
- Then run make in each directory recursively

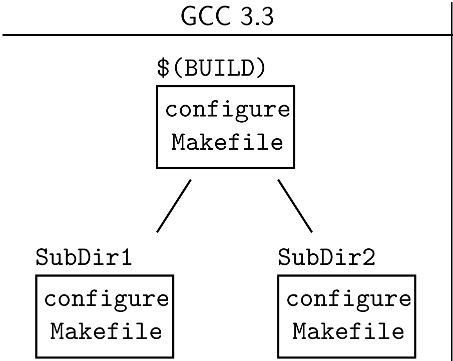
Alternatives in Configuration

GCC 3.3

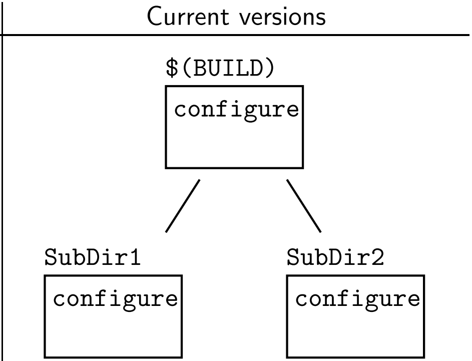


- Configure all directories recursively and create Makefiles
- Then run make in each directory recursively

Alternatives in Configuration



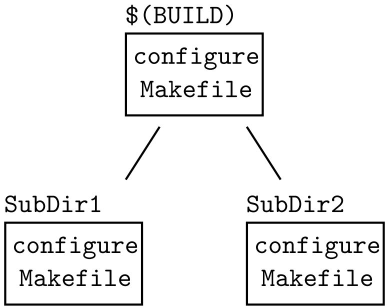
- Configure all directories recursively and create Makefiles
- Then run make in each directory recursively



- Configure a directory and create Makefile
- Run make and configure its subdirectories recursively

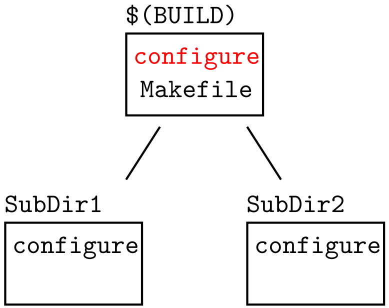
Alternatives in Configuration

GCC 3.3



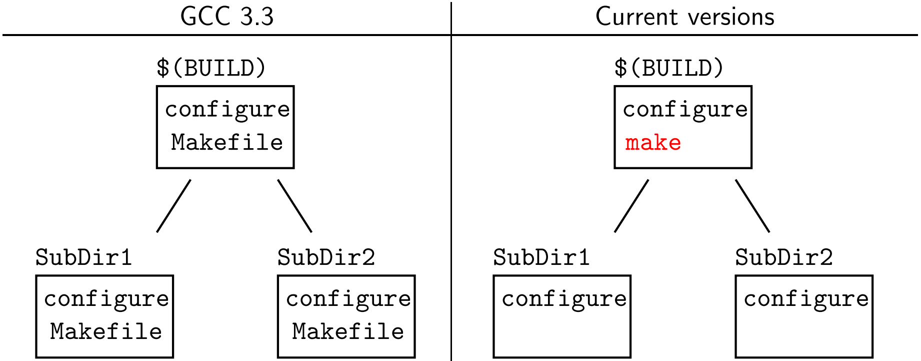
- Configure all directories recursively and create Makefiles
- Then run make in each directory recursively

Current versions



- Configure a directory and create Makefile
- Run make and configure its subdirectories recursively

Alternatives in Configuration



GCC 3.3

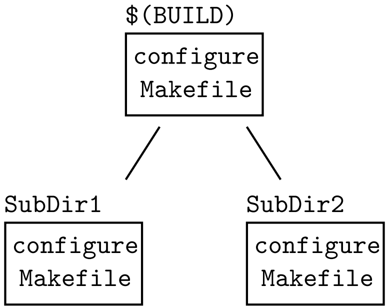
- Configure all directories recursively and create Makefiles
- Then run make in each directory recursively

Current versions

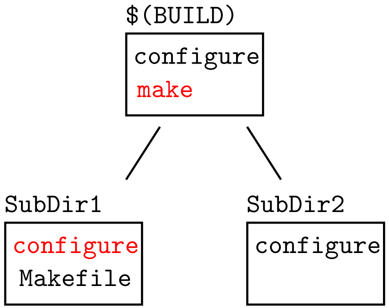
- Configure a directory and create Makefile
- Run make and configure its subdirectories recursively

Alternatives in Configuration

GCC 3.3	Current versions



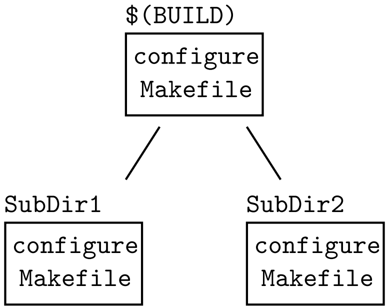
- Configure all directories recursively and create Makefiles
- Then run make in each directory recursively



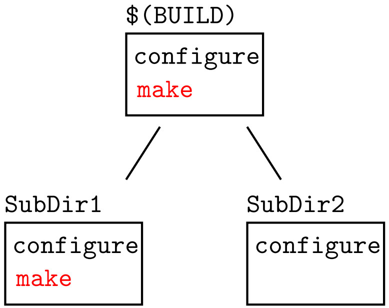
- Configure a directory and create Makefile
- Run make and configure its subdirectories recursively

Alternatives in Configuration

GCC 3.3	Current versions

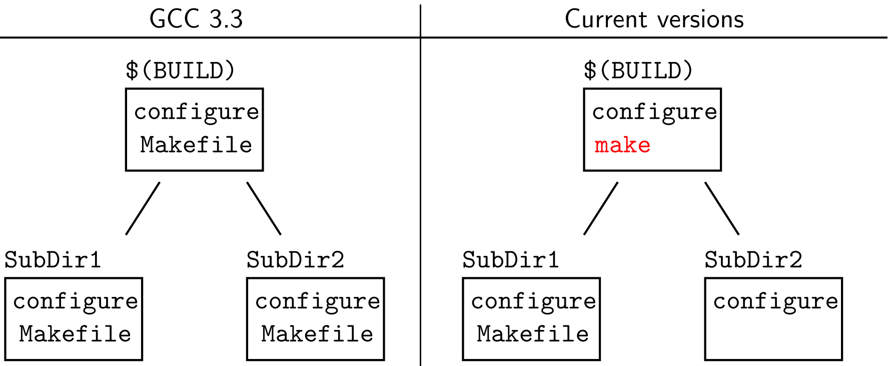


- Configure all directories recursively and create Makefiles
- Then run make in each directory recursively



- Configure a directory and create Makefile
- Run make and configure its subdirectories recursively

Alternatives in Configuration



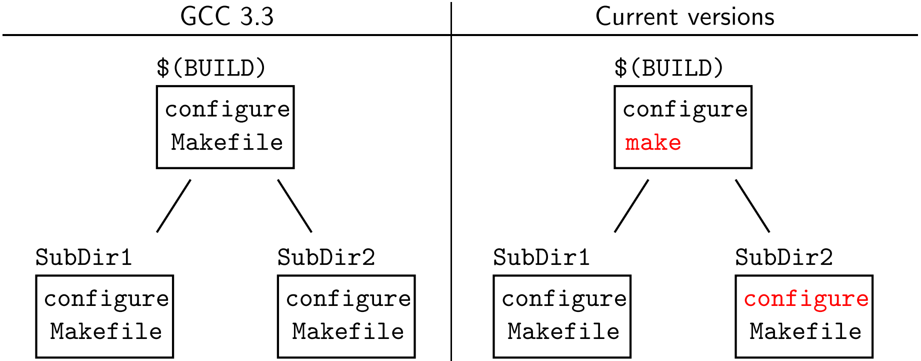
GCC 3.3:

- Configure all directories recursively and create Makefiles
- Then run make in each directory recursively

Current versions:

- Configure a directory and create Makefile
- Run make and configure its subdirectories recursively

Alternatives in Configuration



GCC 3.3

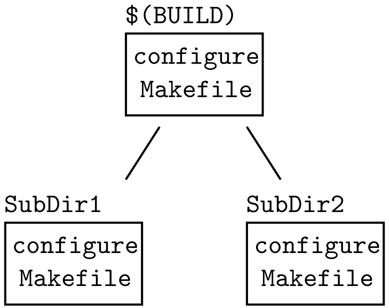
- Configure all directories recursively and create Makefiles
- Then run make in each directory recursively

Current versions

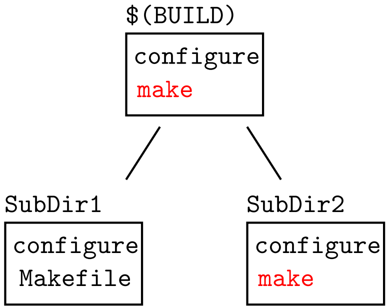
- Configure a directory and create Makefile
- Run make and configure its subdirectories recursively

Alternatives in Configuration

GCC 3.3	Current versions

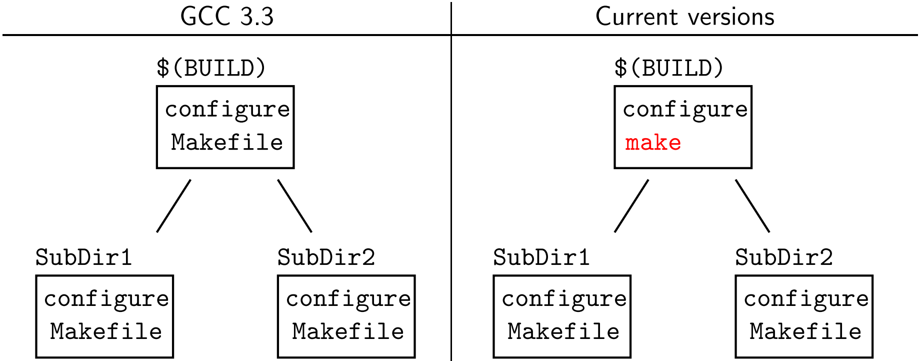


- Configure all directories recursively and create Makefiles
- Then run make in each directory recursively



- Configure a directory and create Makefile
- Run make and configure its subdirectories recursively

Alternatives in Configuration



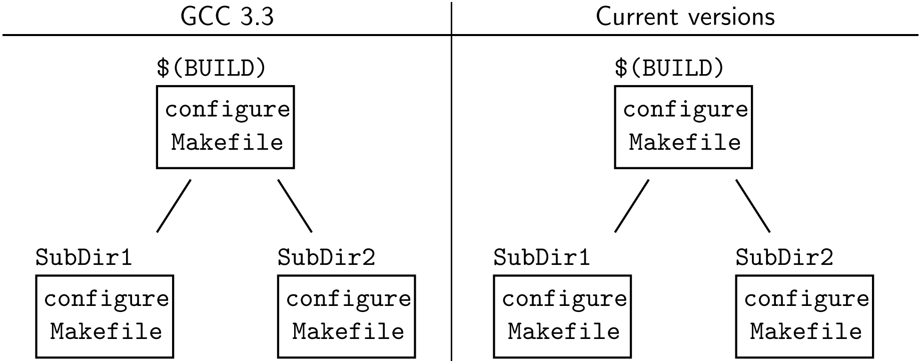
GCC 3.3:

- Configure all directories recursively and create Makefiles
- Then run make in each directory recursively

Current versions:

- Configure a directory and create Makefile
- Run make and configure its subdirectories recursively

Alternatives in Configuration



GCC 3.3:

- Configure all directories recursively and create Makefiles
- Then run make in each directory recursively

Current versions:

- Configure a directory and create Makefile
- Run make and configure its subdirectories recursively

Steps in Configuration and Building

Usual Steps	
• Download and untar the source	

Steps in Configuration and Building

Usual Steps	
• Download and untar the source • `cd $(SOURCE)`	

Steps in Configuration and Building

Usual Steps	
• Download and untar the source • `cd $(SOURCE)` • `./configure`	

Steps in Configuration and Building

Usual Steps	
• Download and untar the source • `cd $(SOURCE)` • `./configure` • `make`	

Steps in Configuration and Building

Usual Steps	
• Download and untar the source • `cd $(SOURCE)` • `./configure` • `make` • `make install`	

Steps in Configuration and Building

Usual Steps	Steps in GCC
• Download and untar the source • `cd $(SOURCE)` • `./configure` • `make` • `make install`	

Steps in Configuration and Building

Usual Steps	Steps in GCC
• Download and untar the source • `cd $(SOURCE)` • `./configure` • `make` • `make install`	• Download and untar the source

Steps in Configuration and Building

Usual Steps	Steps in GCC
• Download and untar the source	• Download and untar the source
• `cd $(SOURCE)`	• `cd $(BUILD)`
• `./configure`	
• `make`	
• `make install`	

Steps in Configuration and Building

Usual Steps	Steps in GCC
• Download and untar the source	• Download and untar the source
• `cd $(SOURCE)`	• `cd $(BUILD)`
• `./configure`	• `$(SOURCE)/configure`
• `make`	
• `make install`	

Steps in Configuration and Building

Usual Steps	Steps in GCC
• Download and untar the source	• Download and untar the source
• `cd $(SOURCE)`	• `cd $(BUILD)`
• `./configure`	• `$(SOURCE)/configure`
• `make`	• `make`
• `make install`	

Steps in Configuration and Building

Usual Steps	Steps in GCC
• Download and untar the source	• Download and untar the source
• `cd $(SOURCE)`	• `cd $(BUILD)`
• `./configure`	• `$(SOURCE)/configure`
• `make`	• `make`
• `make install`	• `make install`

Steps in Configuration and Building

Usual Steps	Steps in GCC
• Download and untar the source	• Download and untar the source
• `cd $(SOURCE)`	• `cd $(BUILD)`
• `./configure`	• `$(SOURCE)/configure`
• `make`	• `make`
• `make install`	• `make install`

Steps in Configuration and Building

Usual Steps	Steps in GCC
• Download and untar the source	• Download and untar the source
• `cd $(SOURCE)`	• `cd $(BUILD)`
• `./configure`	• `$(SOURCE)/configure`
• `make`	• `make`
• `make install`	• `make install`

GCC generates a large part of source code during configuration!

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 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

BS = HS = TS	Native Build
BS = HS ≠ TS	Cross Build
BS ≠ HS ≠ TS	Canadian Cross

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.

Bootstrapping

A compiler is just another program

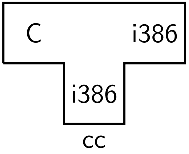
It is improved, bugs are fixed and newer versions are released

To build a new version given a built old version:

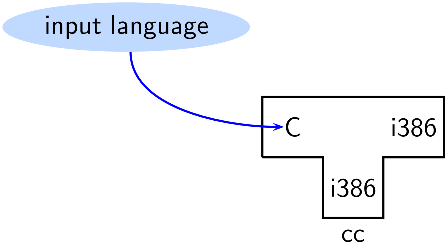
1. Stage 1: Build the new compiler using the old compiler
2. Stage 2: Build another new compiler using compiler from stage 1
3. Stage 3: Build another new compiler using compiler from stage 2
   Stage 2 and stage 3 builds must result in identical compilers

$\Rightarrow$ Building cross compilers stops after Stage 1!

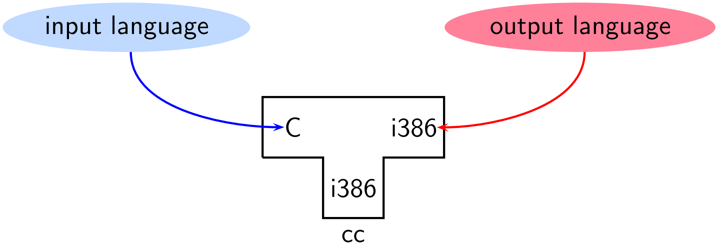
T Notation for a Compiler



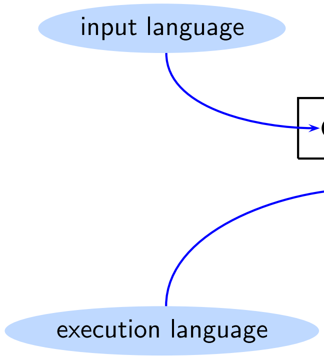
T Notation for a Compiler



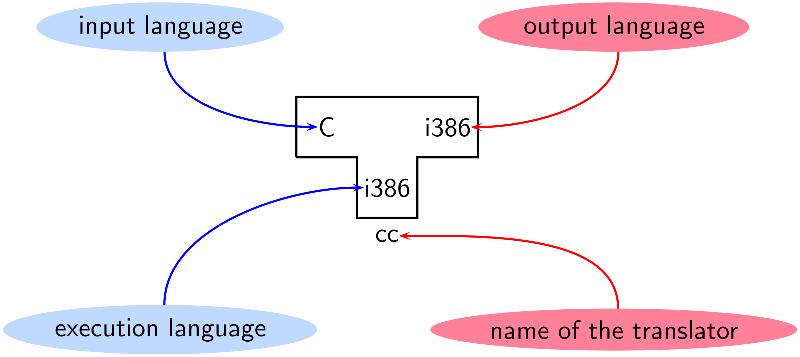
T Notation for a Compiler



T Notation for a Compiler



T Notation for a Compiler

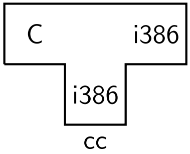


A Native Build on i386

GCC
Source

Requirement: BS = HS = TS = i386

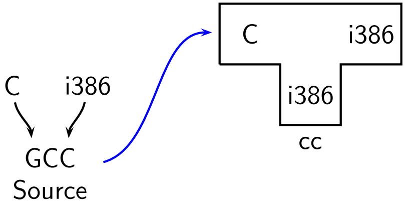
A Native Build on i386



GCC
Source

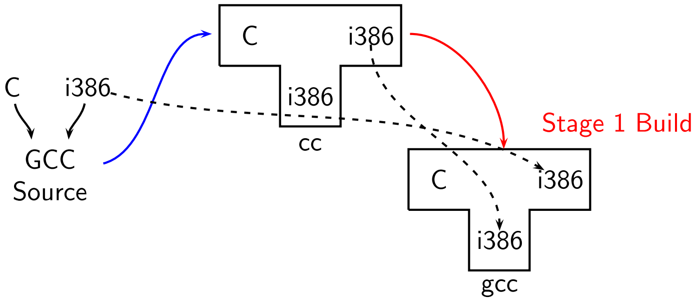
Requirement: BS = HS = TS = i386

A Native Build on i386



Requirement: BS = HS = TS = i386

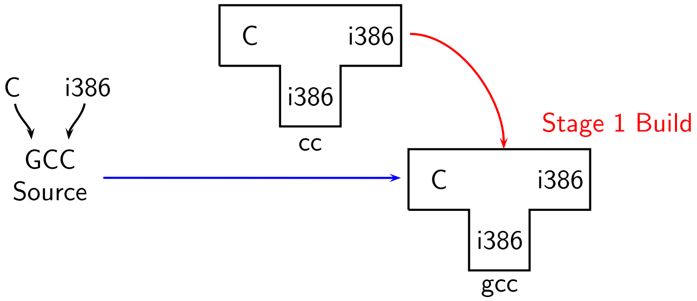
A Native Build on i386



Requirement: BS = HS = TS = i386

- Stage 1 build compiled using cc

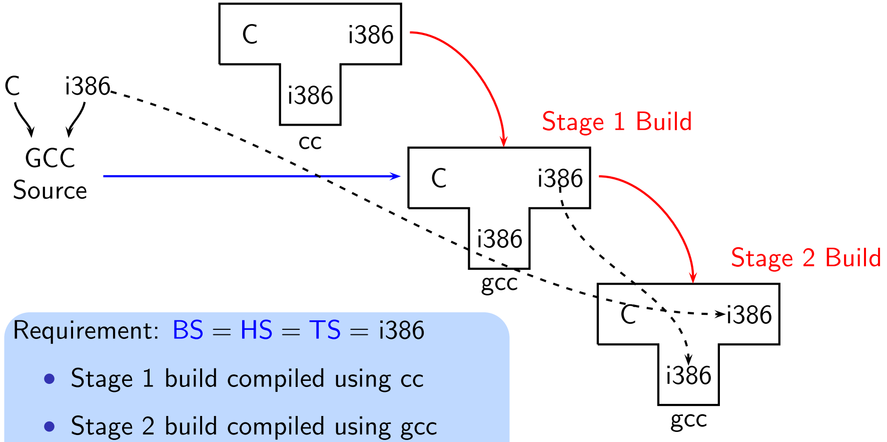
A Native Build on i386



Requirement: BS = HS = TS = i386

- Stage 1 build compiled using cc

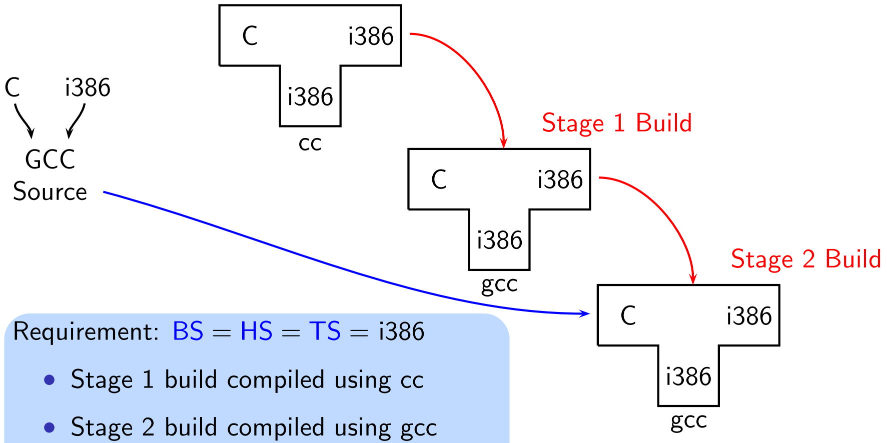
A Native Build on i386



Requirement: BS = HS = TS = i386

- Stage 1 build compiled using cc
- Stage 2 build compiled using gcc

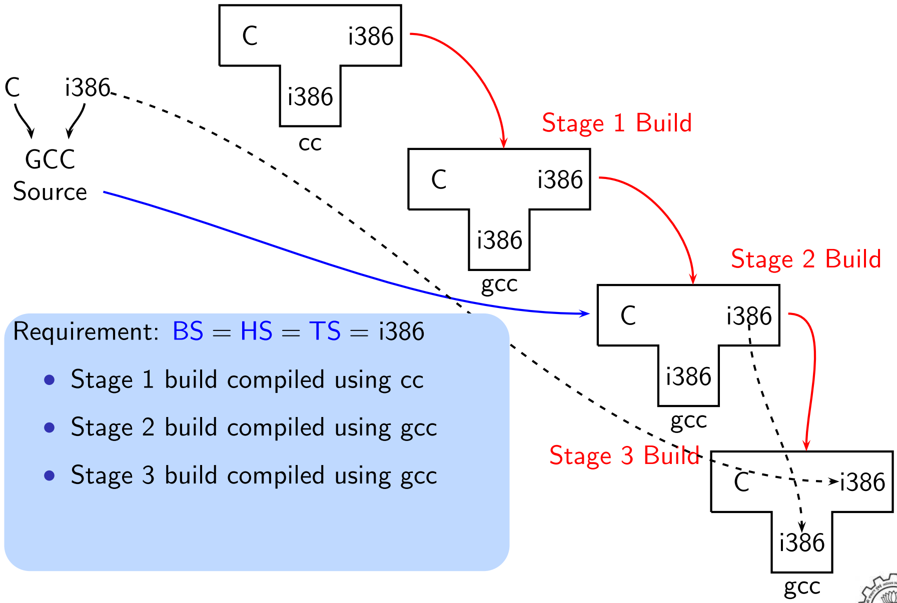
A Native Build on i386



Requirement: BS = HS = TS = i386

- Stage 1 build compiled using cc
- Stage 2 build compiled using gcc

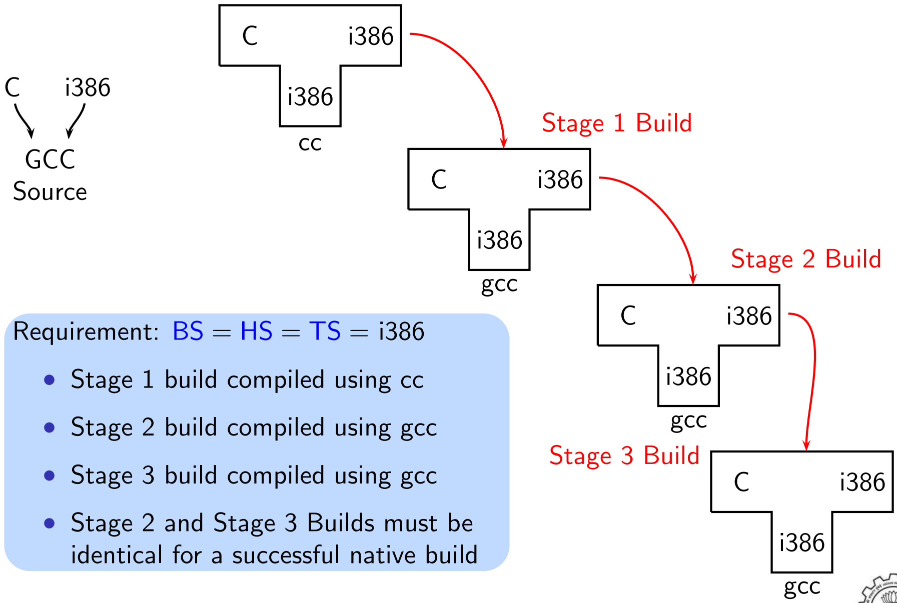
A Native Build on i386



Requirement: BS = HS = TS = i386

- Stage 1 build compiled using cc
- Stage 2 build compiled using gcc
- Stage 3 build compiled using gcc

A Native Build on i386



Requirement: BS = HS = TS = i386

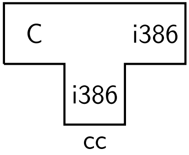
- Stage 1 build compiled using cc
- Stage 2 build compiled using gcc
- Stage 3 build compiled using gcc
- Stage 2 and Stage 3 Builds must be identical for a successful native build

A Cross Build on i386

GCC
Source

Requirement: BS = HS = i386, TS = mips

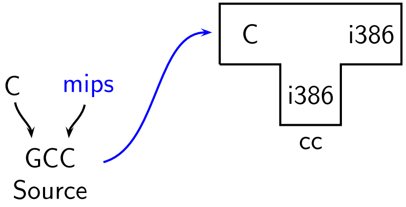
A Cross Build on i386



GCC
Source

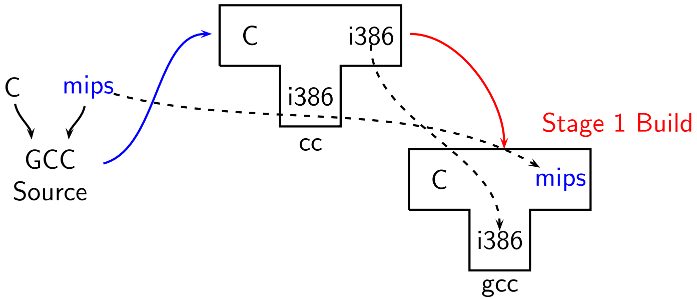
Requirement: BS = HS = i386, TS = mips

A Cross Build on i386



Requirement: BS = HS = i386, TS = mips

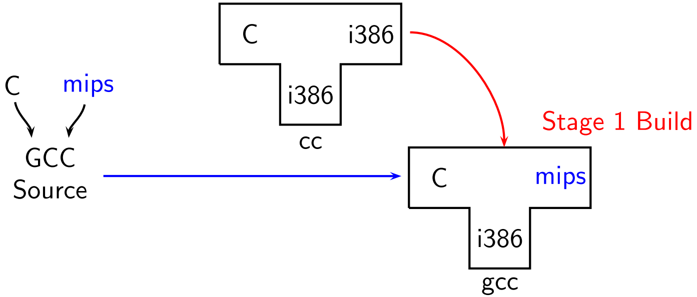
A Cross Build on i386



Requirement: BS = HS = i386, TS = mips

- Stage 1 build compiled using cc

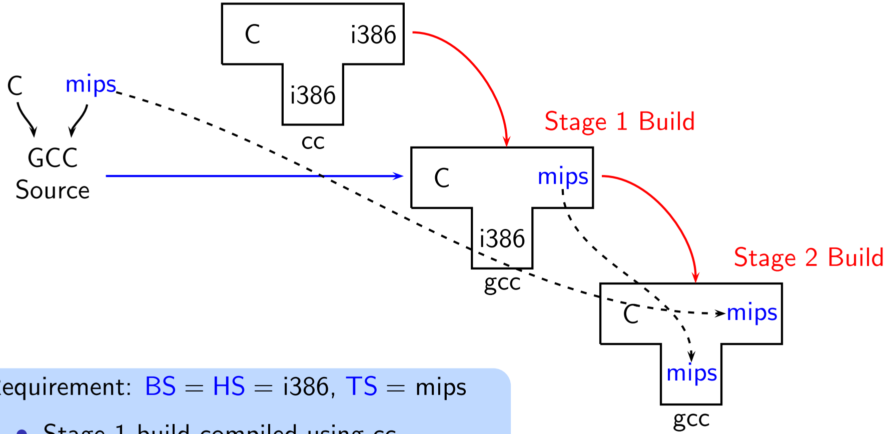
A Cross Build on i386



Requirement: BS = HS = i386, TS = mips

- Stage 1 build compiled using cc

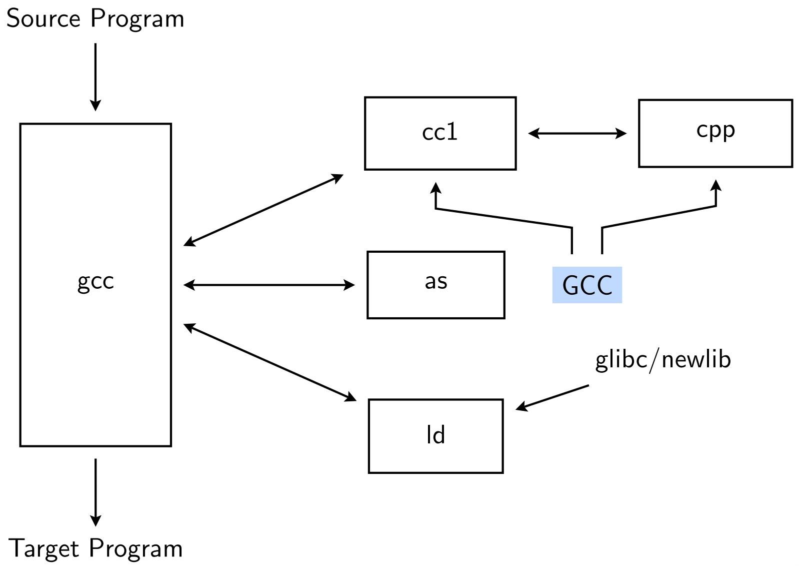
A Cross Build on i386



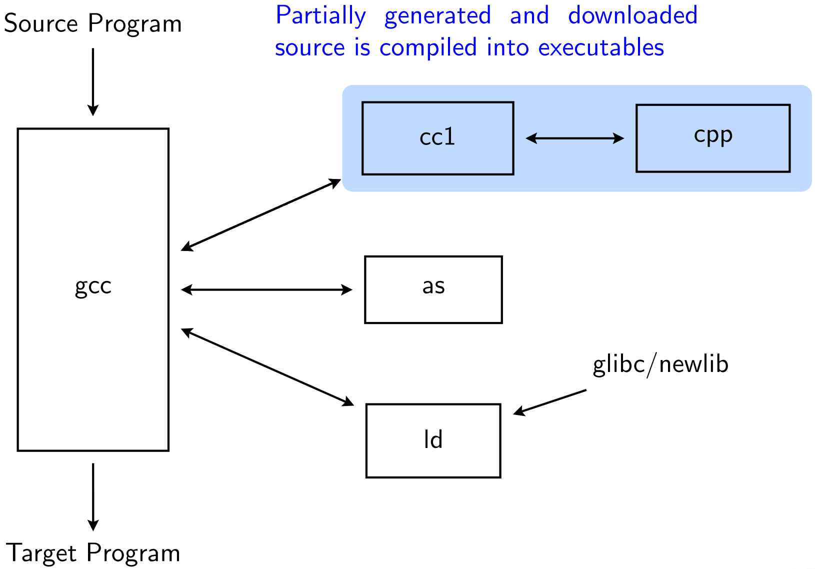
Requirement: BS = HS = i386, TS = mips

- Stage 1 build compiled using cc
- Stage 2 build compiled using gcc
  Its HS = mips and not i386!

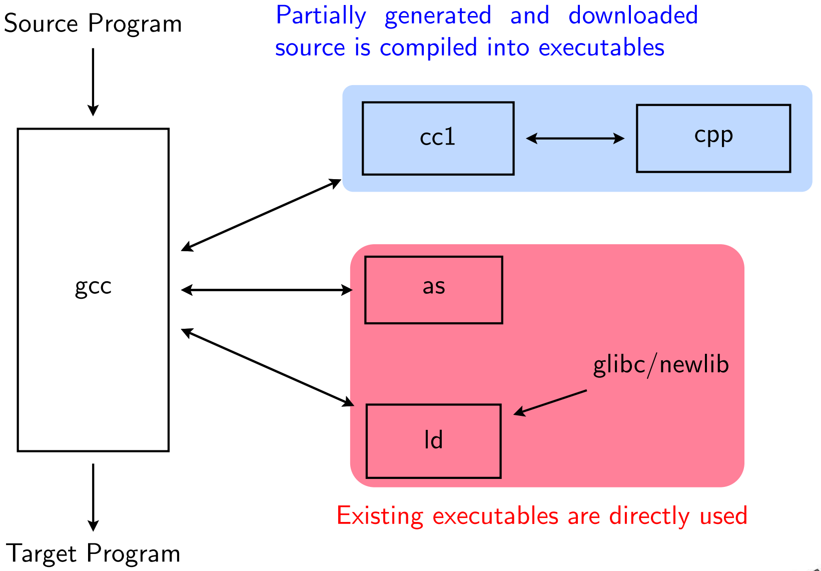
A More Detailed Look at Building



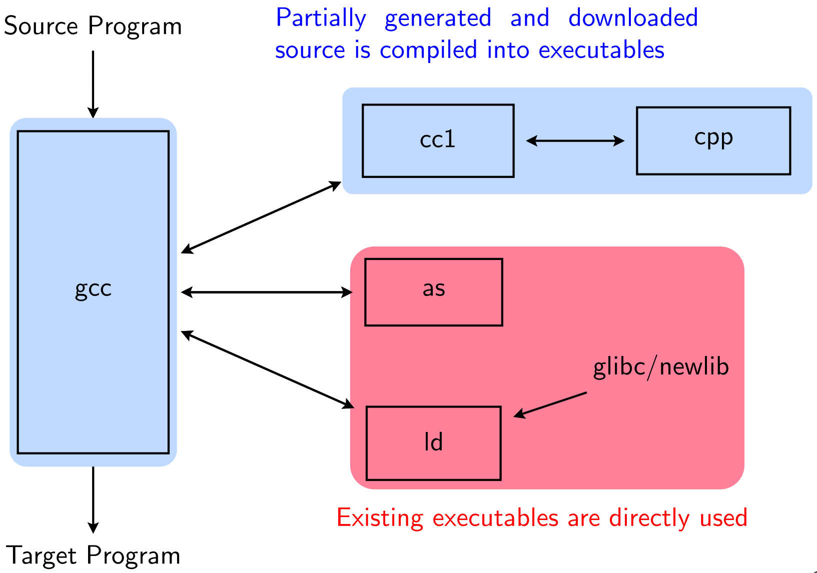
A More Detailed Look at Building


Source Program
Partially generated and downloaded source is compiled into executables
cc1
cpp
gcc
as
glibc/newlib
ld
Target Program

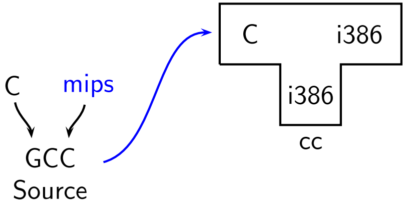
A More Detailed Look at Building


Source Program
Partially generated and downloaded source is compiled into executables
cc1
cpp
gcc
as
glibc/newlib
ld
Existing executables are directly used
Target Program

A More Detailed Look at Building

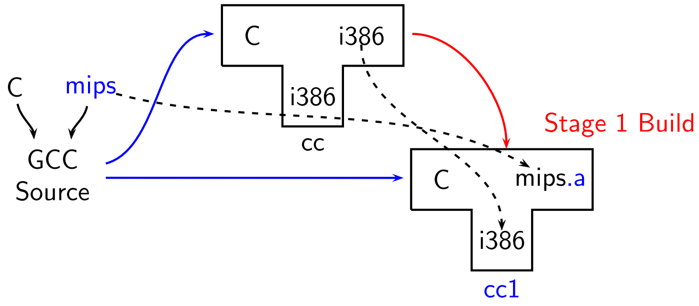

Source Program
Partially generated and downloaded source is compiled into executables
cc1
cpp
gcc
as
glibc/newlib
ld
Existing executables are directly used
Target Program

A More Detailed Look at Cross Build



Requirement: BS = HS = i386, TS = mips

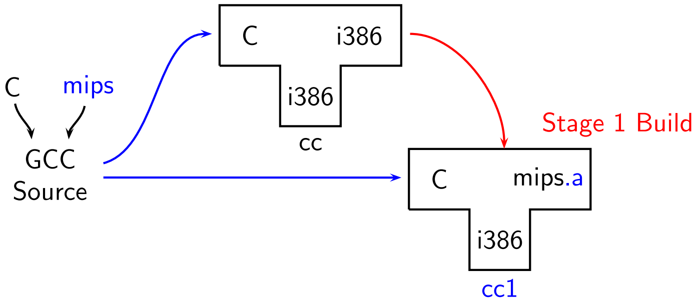
A More Detailed Look at Cross Build



Requirement: BS = HS = i386, TS = mips

- Stage 1 build consists of only cc1 and not gcc

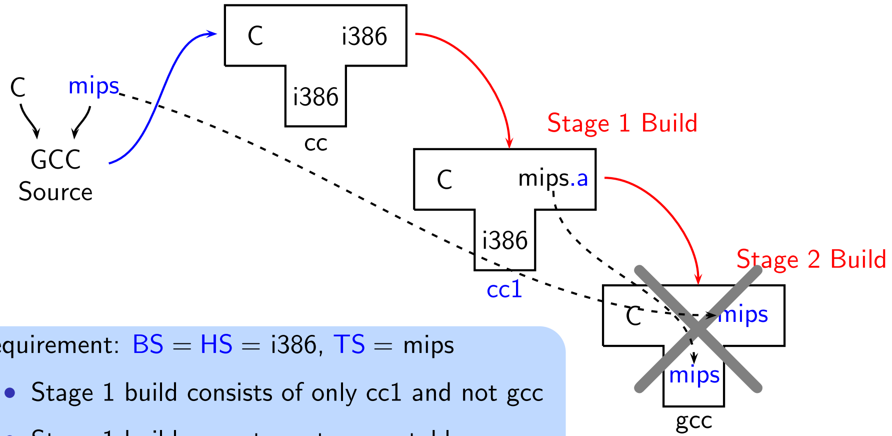
A More Detailed Look at Cross Build



Requirement: BS = HS = i386, TS = mips

- Stage 1 build consists of only cc1 and not gcc
- Stage 1 build cannot create executables
- Library sources cannot be compiled for mips using stage 1 build

A More Detailed Look at Cross Build



Requirement: BS = HS = i386, TS = mips

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

Cross Build Revisited

- Option 1: Build binutils in the same source tree as gcc
  Copy binutils source in `$(SOURCE)`, configure and build stage 1
- Option 2:
  - Compile cross-assembler (`as`), cross-linker (`ld`), cross-archiver (`ar`), and cross-program to build symbol table in archiver (`ranlib`),
  - Copy them in `$(INSTALL)/bin`
  - Build stage 1 of GCC

Information Required for Configuring GCC

- Build-Host-Target systems inferred for native builds
- Specify Target system for cross builds
  Build ≡ Host systems: inferred
- Build-Host-Target systems can be explicitly specified too
- For GCC: A "system" = three entities
  - "cpu"
  - "vendor"
  - "os"

  e.g. `sparc-sun-sunos`, `i386-unknown-linux`, `i386-gcc-linux`

Commands for Configuring and Building GCC

This is what *we* specify

- `cd $(BUILD)`

Commands for Configuring and Building GCC

This is what *we* specify

- `cd $(BUILD)`
- `$(SOURCE) configure <options>`

  `configure` output: customized `Makefile`

Commands for Configuring and Building GCC

This is what *we* specify

- `cd $(BUILD)`
- `$(SOURCE) configure <options>`

  `configure` output: customized `Makefile`
- `make 2> make.err > make.log`

Commands for Configuring and Building GCC

This is what *we* specify

- `cd $(BUILD)`
- `$(SOURCE) configure <options>`

  `configure` output: customized `Makefile`
- `make 2> make.err > make.log`
- `make install 2> install.err > install.log`

Build for a Given Machine

This is what actually happens!

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

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

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`)

Common Configuration Options

`--target`

- Necessary for cross build
- Possible `host-cpu-vendor` strings: Listed in `$(SOURCE)/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

Part 4

Registering New Machine Descriptions

Adding a New MD

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

  for recognized system names.

Tip

Read comments in `$(SOURCE)/config.sub` &
`$(SOURCE)/gcc/config/<cpu>`.

Registering Spim with GCC Build Process

Eventually, we want to add multiple descriptions:

- Step 1. In the file `$(SOURCE)/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 2. In the file `$(SOURCE)/gcc/config.gcc`
  - In case `${target}` used for defining cpu_type, add

    ```
    spim*-*-*)
         cpu_type=spim
         ;;
    ```

    This specifies the directory `$(SOURCE)/gcc/config/spim` in which the machine descriptions files are supposed to be made available.
  - In case `${target}` for

    ```
    # Support site-specific machine types.
    ```

    add

    ```
    spim*-*-*)
         gas=no
         gnu_ld=no
         tm_file=spim/${target_noncanonical}.h
         md_file=spim/${target_noncanonical}.md
         out_file=spim/${target_noncanonical}.c
         tm_p_file=spim/${target_noncanonical}-protos.h
         ;;
    ```

Part 5

Testing GCC

GCC testing framework

- Pre-requisites - `Dejagnu`, `Expect` tools
- Option 1: Build GCC and execute the command `$(BUILD)/gcc` directory

  `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`

GCC Internals document contains an exhaustive list of options for testing

Part 6

Summary

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
```

Lab Assignments

- Untar the GCC source provided and register the spim machine descriptions in the source.
- Configure GCC for spim target and build the compiler. Observe where the build process failed fails and try to find out why it fails.
- Configure with the option `--disable-bootstrap`. Does the build process fail now? Why?
- Add a new target in the `Makefile.in`

  ```
  cc1:
      make all-gcc TARGET-gcc=cc1$(exeext)
  ```

- Build with the command `make cc1`. Does the build process fail now? Why?