Workshop on Essential Abstractions in GCC

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 2010

Outline

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

Part 1

GCC Code Organization

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_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

Part 2

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 4.5.0

- 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 4.2 (or later)
- MPFR Library version 2.3.2 (or later)
  (multiple precision floating point with correct rounding)
- MPC Library version 0.8.0 (or later)
- Parma Polyhedra Library (PPL) version 0.10
- CLooG-PPL (Chunky Loop Generator) version 0.15
- jar, or InfoZIP (zip and unzip)
- libelf version 0.8.12 (or later) (for LTO)

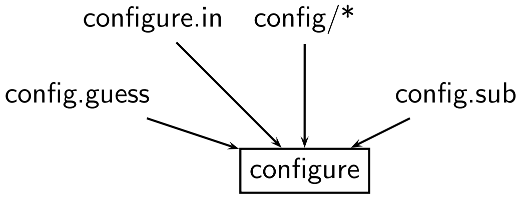
Our Conventions for Directory Names

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

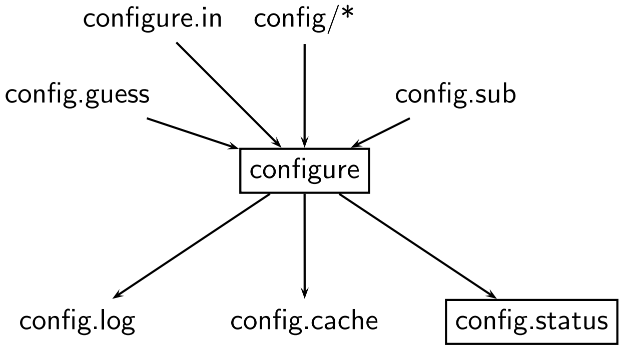
Configuring GCC

$\boxed{\text{configure}}$

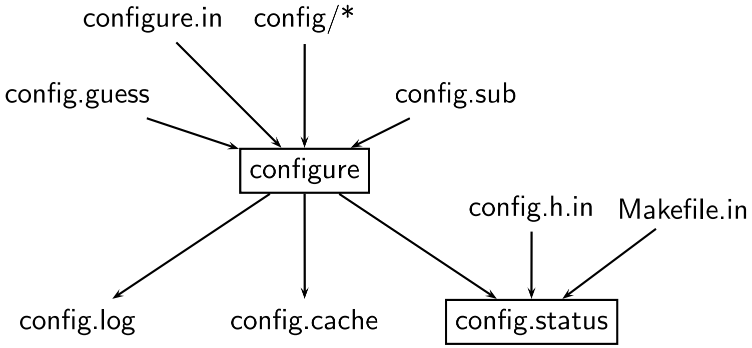
Configuring GCC


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

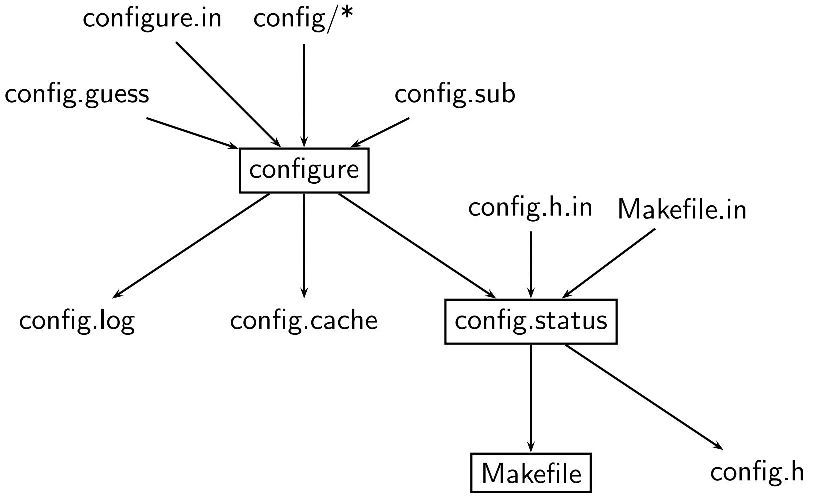
Configuring GCC



Configuring GCC



Configuring GCC



Steps in Configuration and Building

Usual Steps	
• Download and untar the source • `cd $(SOURCE_D)` • `./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_D)`	• `cd $(BUILD)`
• `./configure`	• `$(SOURCE_D)/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_D)`	• `cd $(BUILD)`
• `./configure`	• `$(SOURCE_D)/configure`
• `make`	• `make`
• `make install`	• `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 *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.

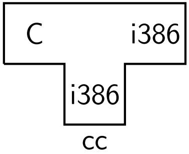
Variants of Compiler Builds

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

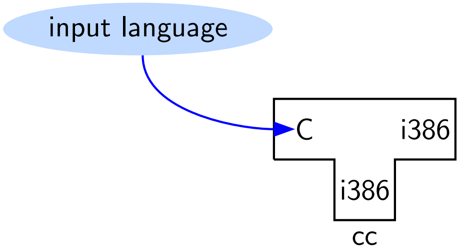
Example

Native i386: built on i386, hosted on i386, produces i386 code.
Sparc cross on i386: built on i386, hosted on i386, produces Sparc code.

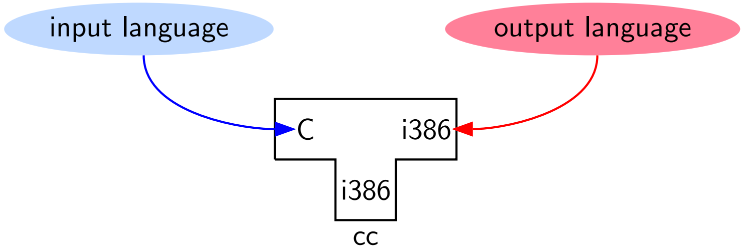
T Notation for a Compiler



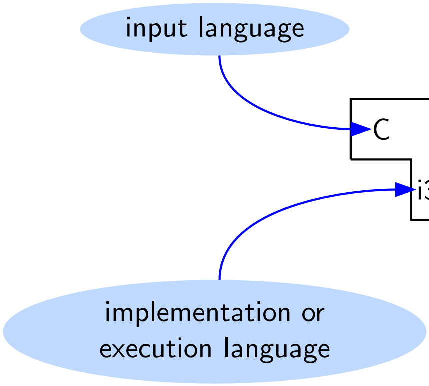
T Notation for a Compiler



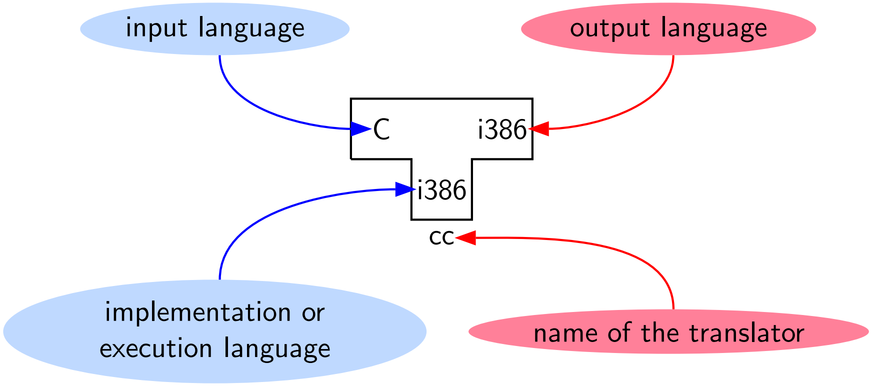
T Notation for a Compiler



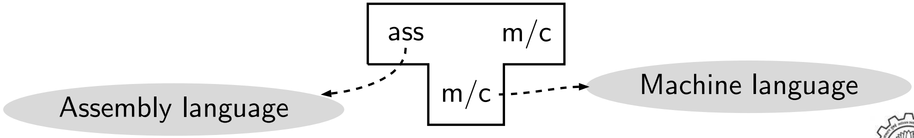
T Notation for a Compiler



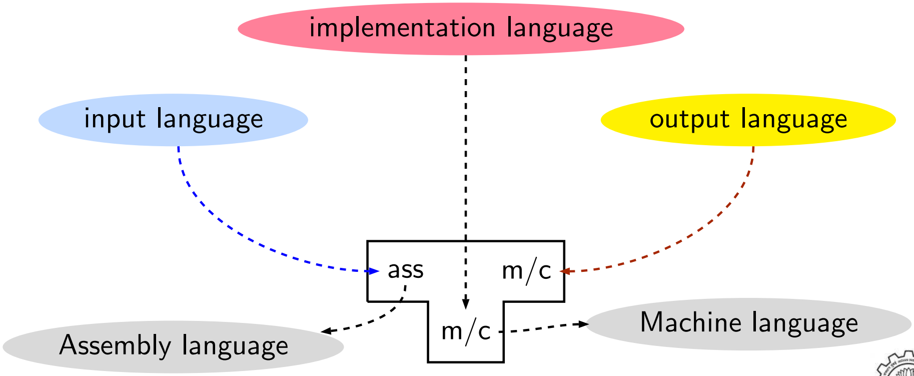
T Notation for a Compiler



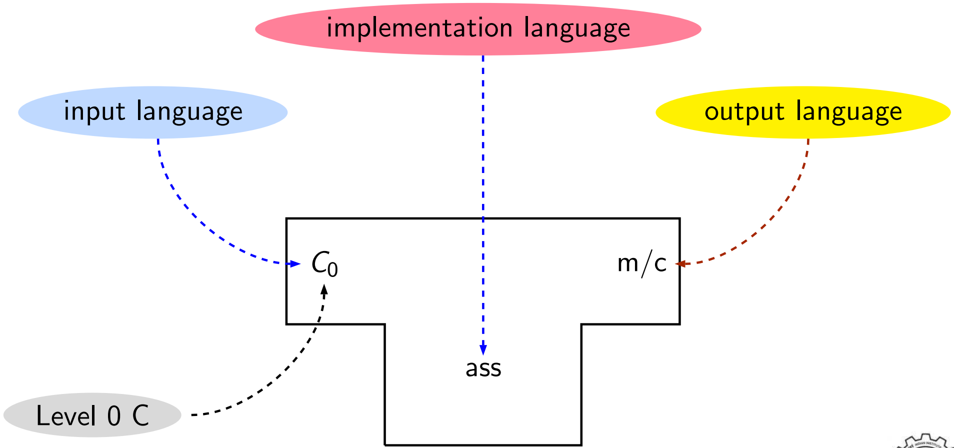
Bootstrapping: The Conventional View



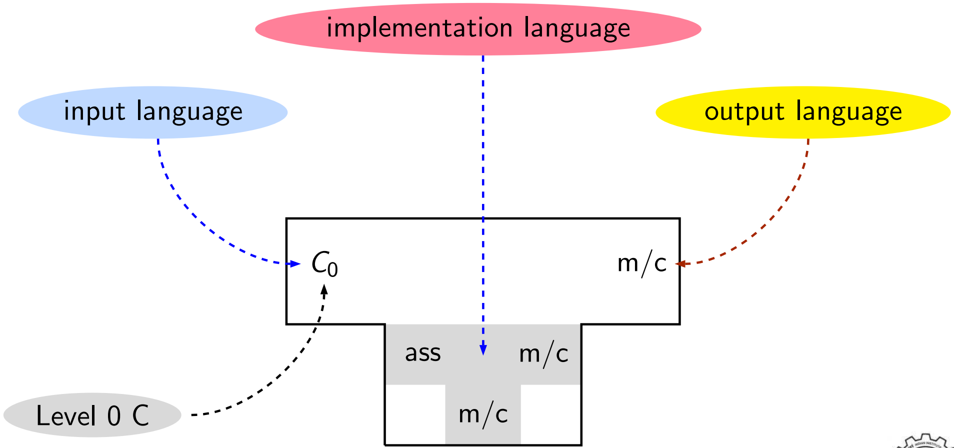
Bootstrapping: The Conventional View



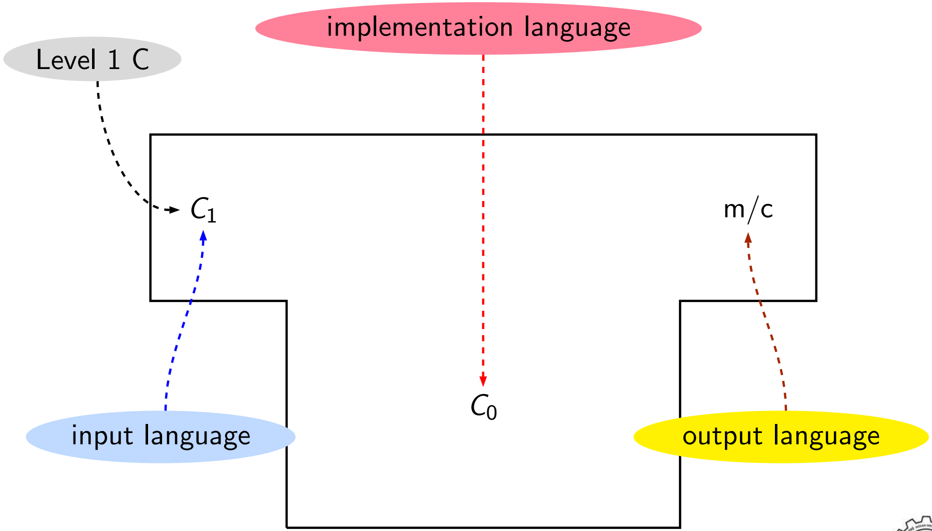
Bootstrapping: The Conventional View



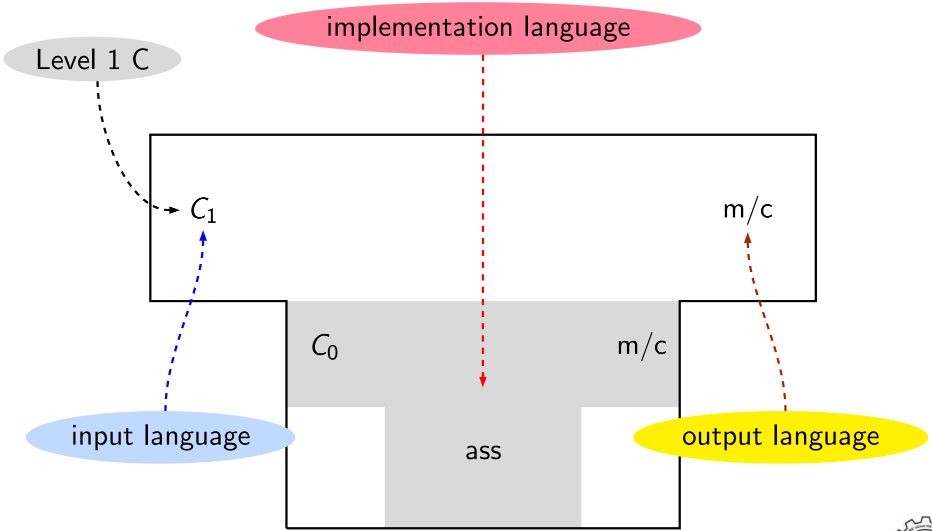
Bootstrapping: The Conventional View



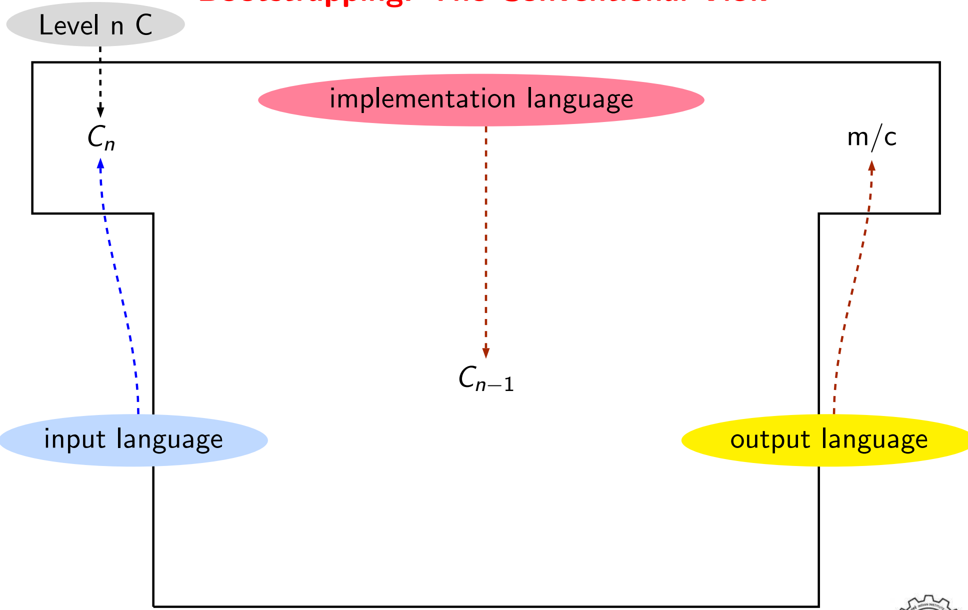
Bootstrapping: The Conventional View



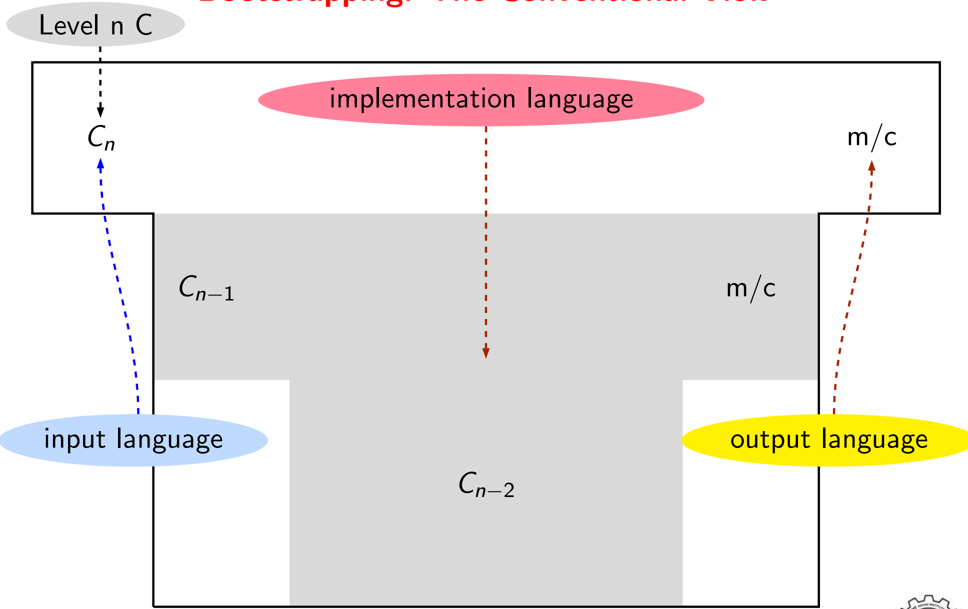
Bootstrapping: The Conventional View



Bootstrapping: The Conventional View



Bootstrapping: The Conventional View



Bootstrapping: GCC 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

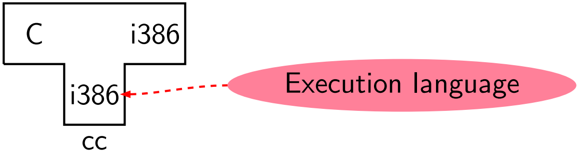
⇒ Building cross compilers stops after Stage 1!

A Native Build on i386

GCC
Source

Requirement: BS = HS = TS = i386

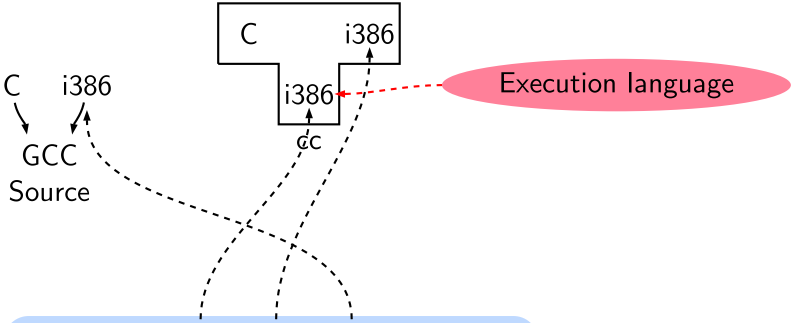
A Native Build on i386



GCC
Source

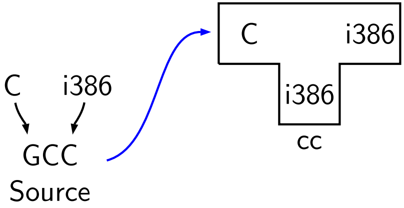
Requirement: BS = HS = TS = i386

A Native Build on i386



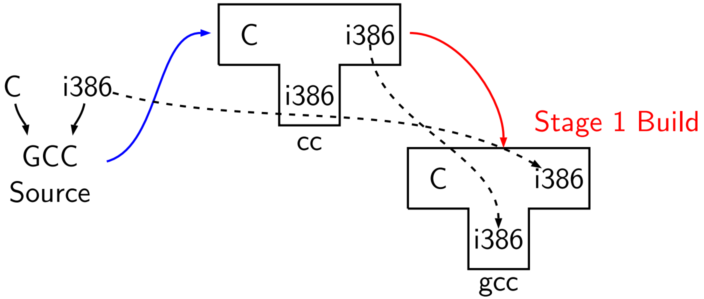
Requirement: BS = HS = TS = i386

A Native Build on i386



Requirement: BS = HS = TS = i386

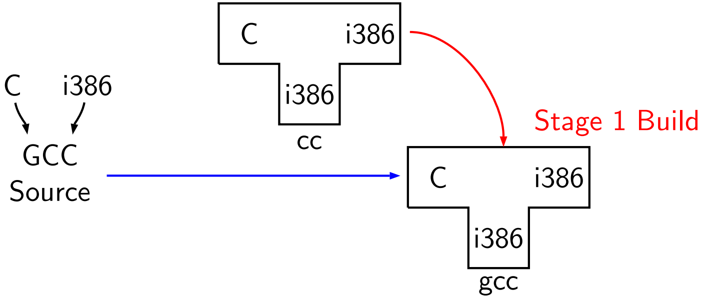
A Native Build on i386



Requirement: BS = HS = TS = i386

- Stage 1 build compiled using cc

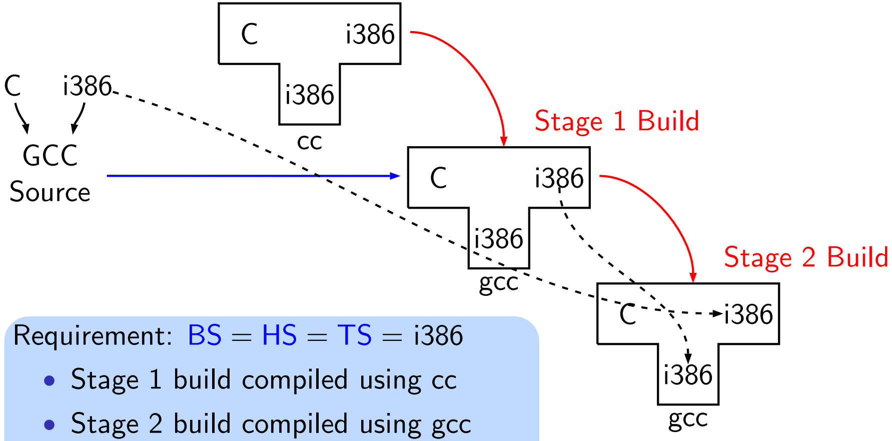
A Native Build on i386



Requirement: BS = HS = TS = i386

- Stage 1 build compiled using cc

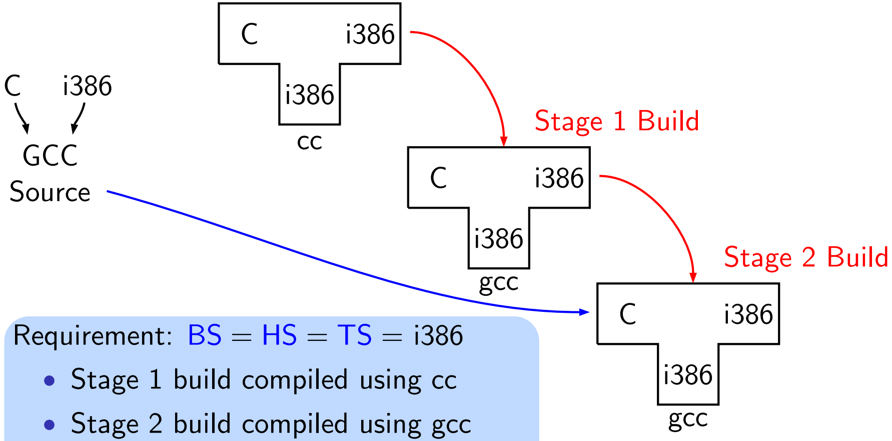
A Native Build on i386



Requirement: BS = HS = TS = i386

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

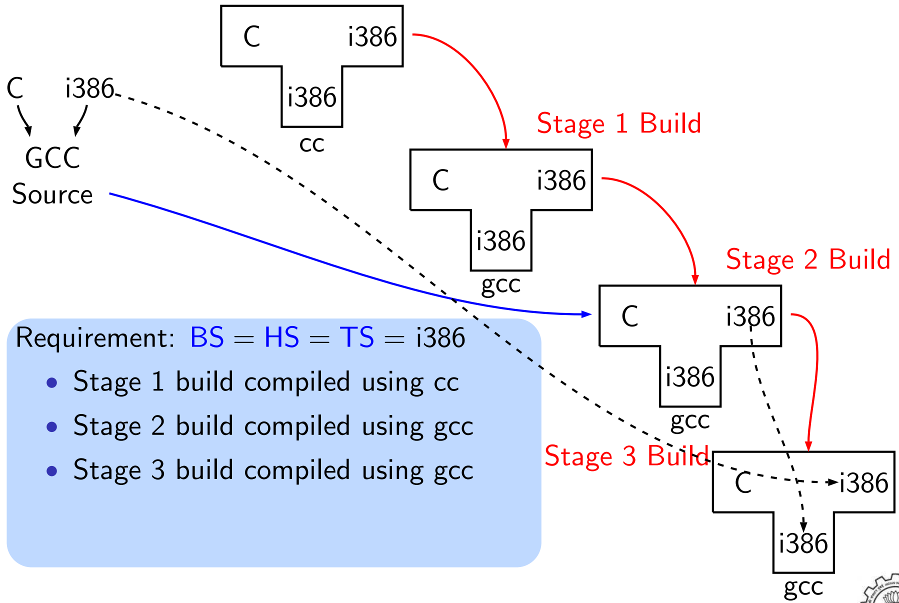
A Native Build on i386



Requirement: BS = HS = TS = i386

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

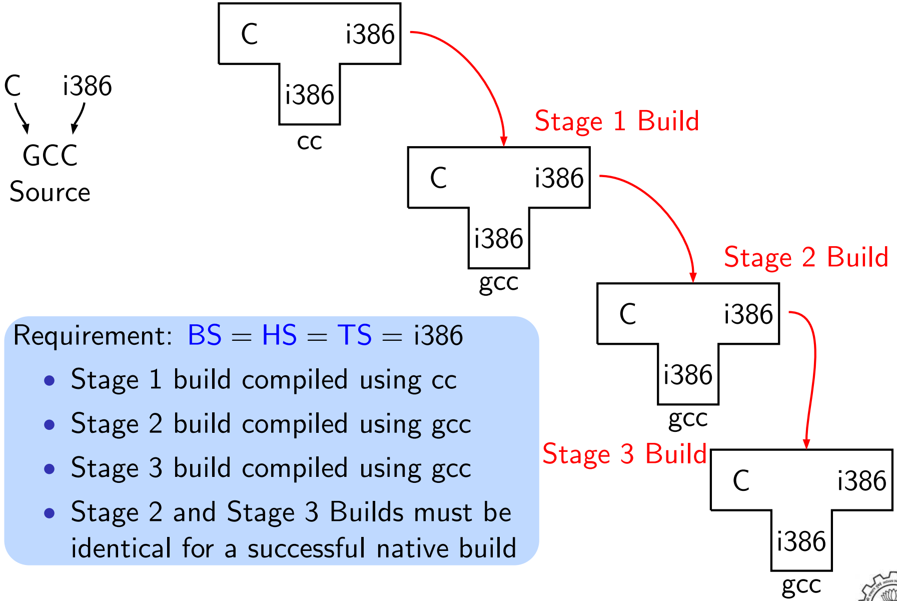
A Native Build on i386



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

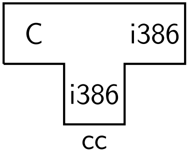
- Stage 1 build compiled using cc
- Stage 2 build compiled using gcc
- Stage 3 build compiled using gcc
- Stage 2 and Stage 3 Builds must be identical for a successful native build

A Cross Build on i386

GCC
Source

Requirement: BS = HS = i386, TS = mips

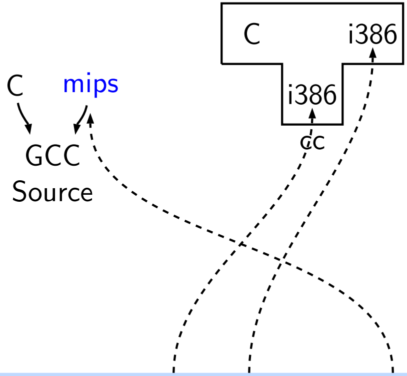
A Cross Build on i386



GCC
Source

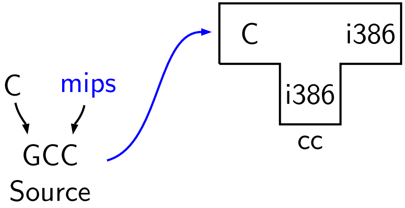
Requirement: BS = HS = i386, TS = mips

A Cross Build on i386



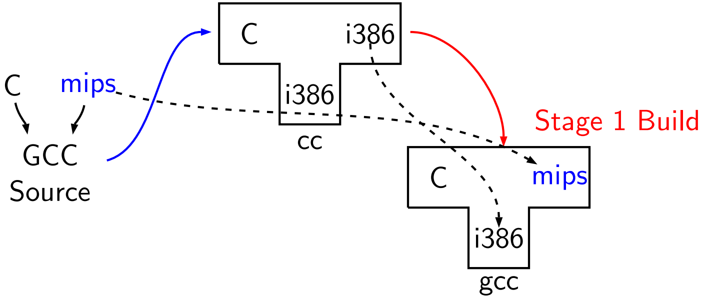
Requirement: BS = HS = i386, TS = mips

A Cross Build on i386



Requirement: BS = HS = i386, TS = mips

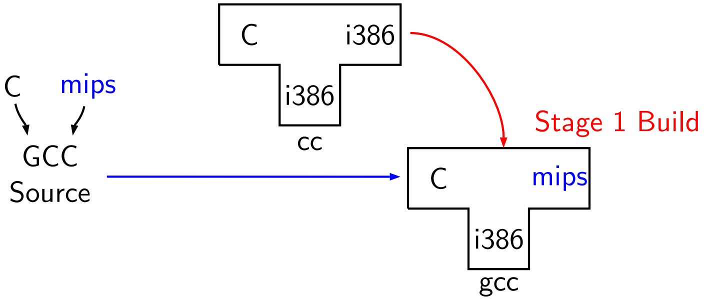
A Cross Build on i386



Requirement: BS = HS = i386, TS = mips

- Stage 1 build compiled using cc

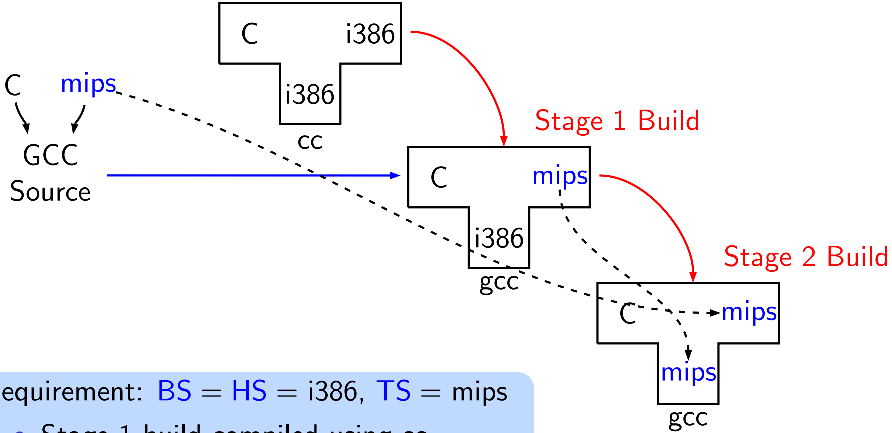
A Cross Build on i386



Requirement: BS = HS = i386, TS = mips

- Stage 1 build compiled using cc

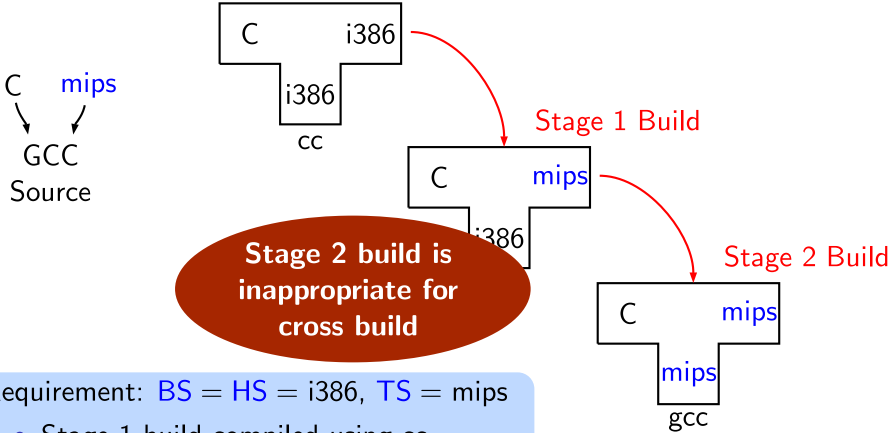
A Cross Build on i386



Requirement: BS = HS = i386, TS = mips

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

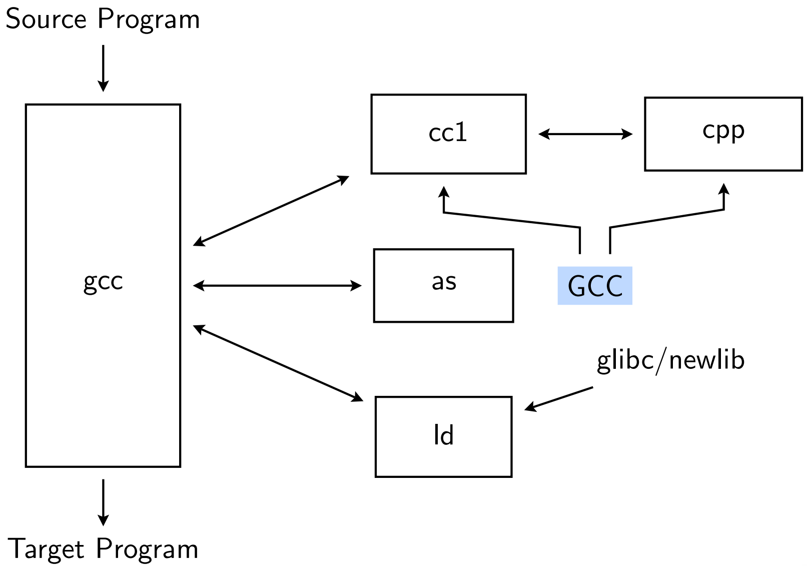
A Cross Build on i386



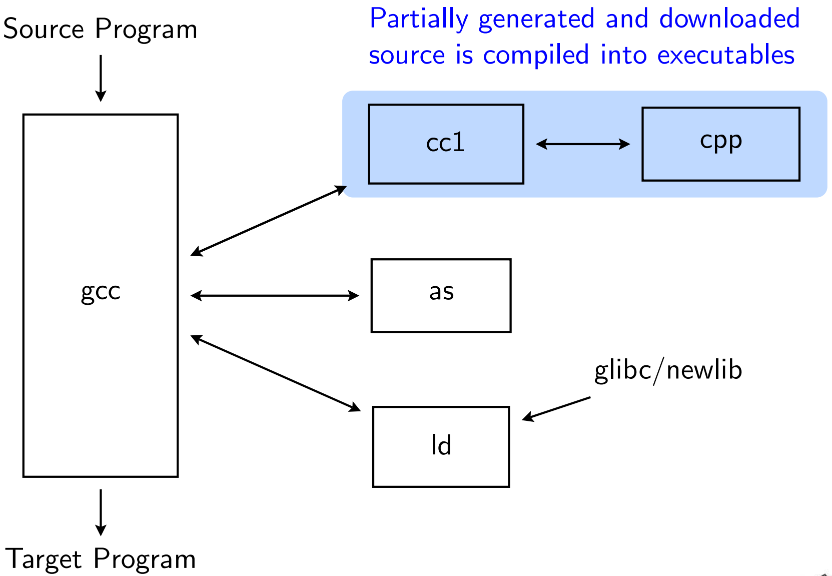
Requirement: BS = HS = i386, TS = mips

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

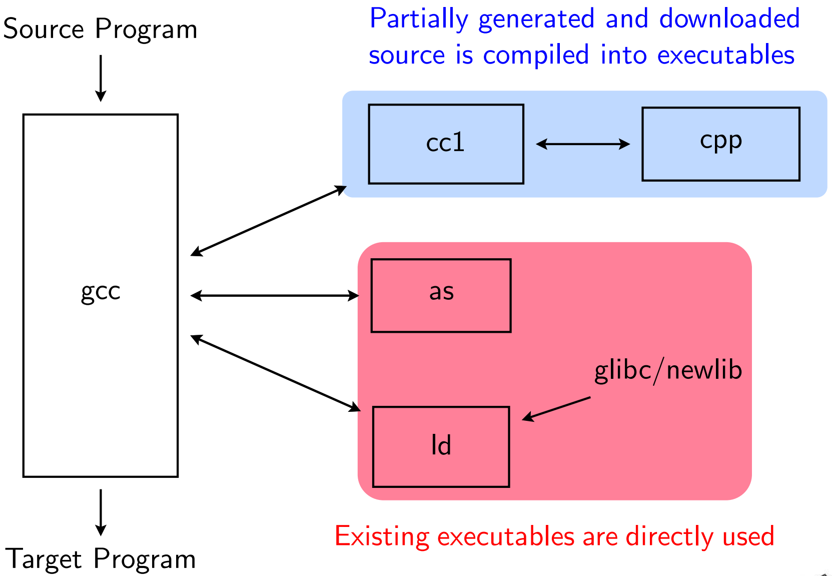
A More Detailed Look at Building



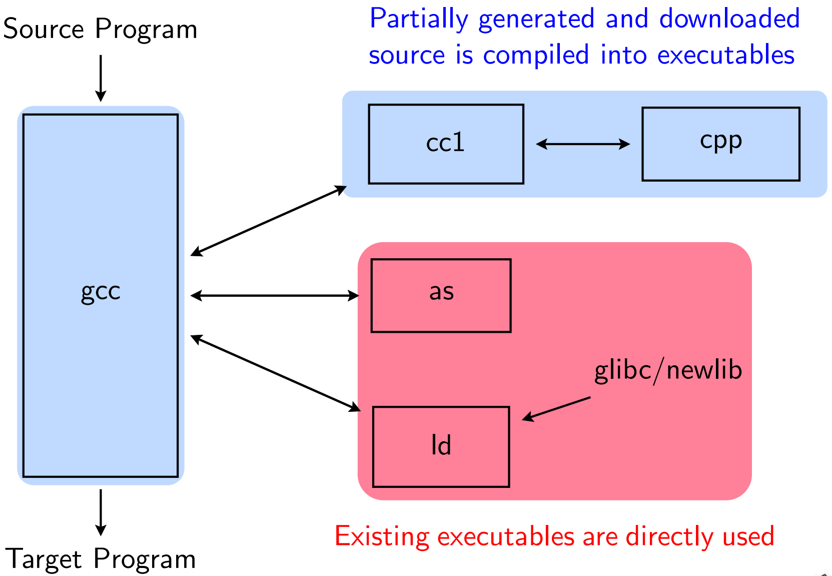
A More Detailed Look at Building


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

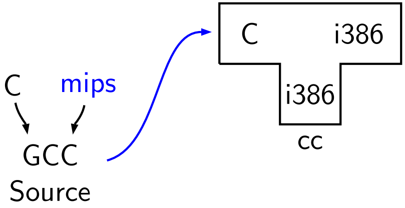
A More Detailed Look at Building


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

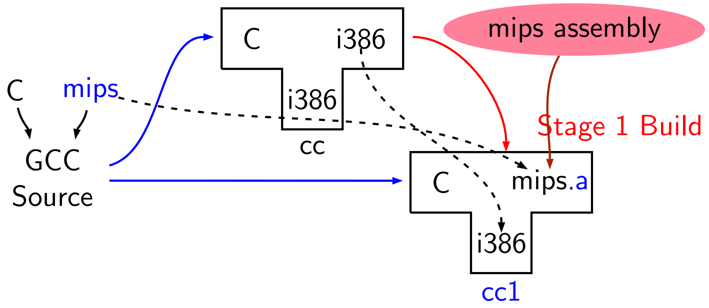
A More Detailed Look at Cross Build



Requirement: BS = HS = i386, TS = mips

we have not built binutils for mips

A More Detailed Look at Cross Build

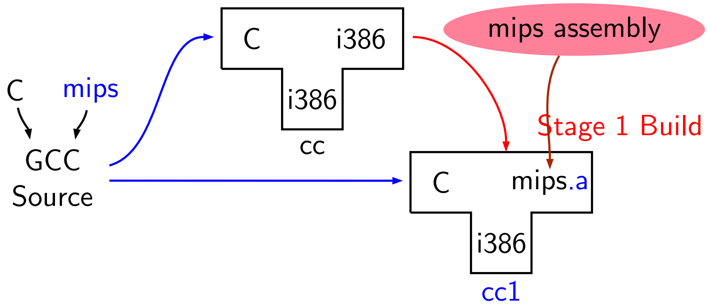


Requirement: BS = HS = i386, TS = mips

- *Stage 1 cannot build gcc but can build only cc1*

we have
not built binutils
for mips

A More Detailed Look at Cross Build

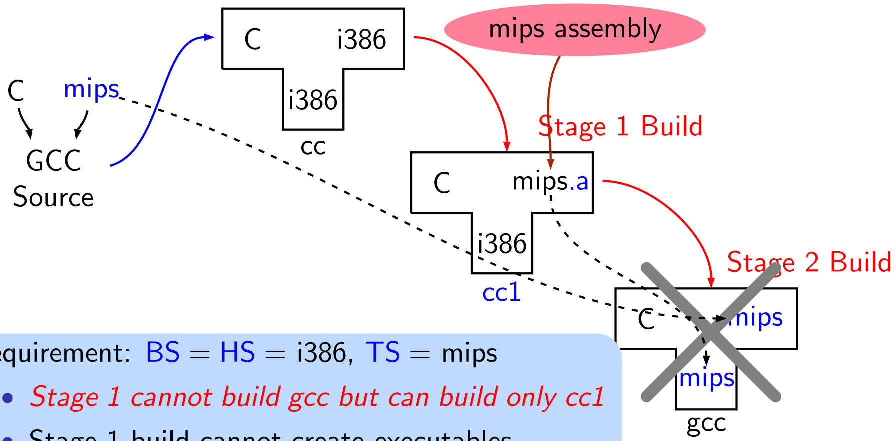


Requirement: BS = HS = i386, TS = mips

- *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

we have not built binutils for mips

A More Detailed Look at Cross Build

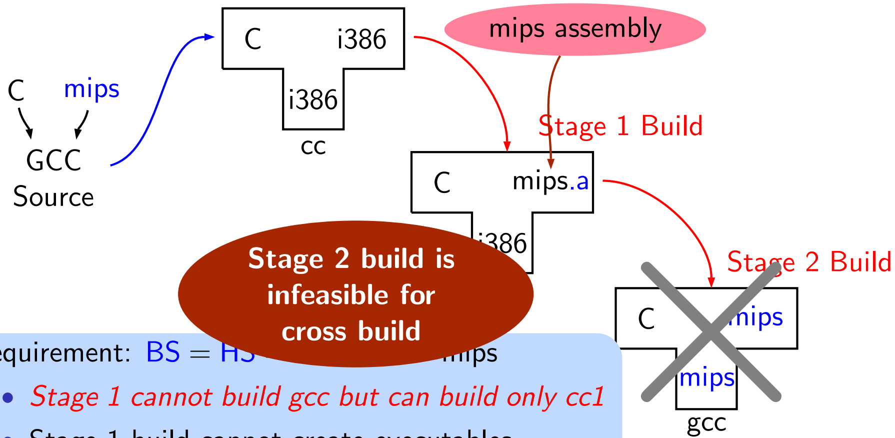


Requirement: BS = HS = i386, TS = mips

- *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 2 build is not possible

we have
not built binutils
for mips

A More Detailed Look at Cross Build



Stage 2 build is infeasible for cross build

Requirement: BS = HS 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 2 build is not possible

we have not built binutils for mips

Cross Build Revisited

- Option 1: Build binutils in the same source tree as gcc
  Copy binutils source in `$(SOURCE_D)`, configure and build stage 1
- Option 2:
  - Compile cross-assembler (`as`), cross-linker (`ld`), cross-archiver (`ar`), and cross-program to build symbol table in archiver (`ranlib`),
  - Copy them in `$(INSTALL)/bin`
  - Build stage GCC
  - Install newlib
  - Reconfigure and build GCC
    Some options differ in the two builds

Cross Build Revisited

- Option 1: Build binutils in the same source tree as `gcc`
  Copy binutils source in `$(SOURCE_D)`, configure and build stage 1
- Option 2:
  - Compile cross-assembler (`as`), cross-linker (`ld`), cross-archiver (`ar`), and cross-program to build symbol table in archiver (`ranlib`),
  - Copy them in `$(INSTALL)/bin`
  - Build stage GCC
  - Install newlib
  - Reconfigure and build GCC
    Some options differ in the two builds

Details to follow in the lecture on building a cross compiler

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_D)/configure <options>`
  `configure` output: customized `Makefile`

Commands for Configuring and Building GCC

This is what *we* specify

- `cd $(BUILD)`
- `$(SOURCE_D)/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_D)/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 sources (`$(SOURCE_D)/gcc/gen*.c`) are read and generator executables are created in `$(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)`

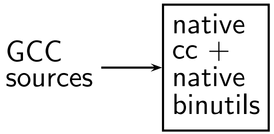
Build for a Given Machine

This is what actually happens!

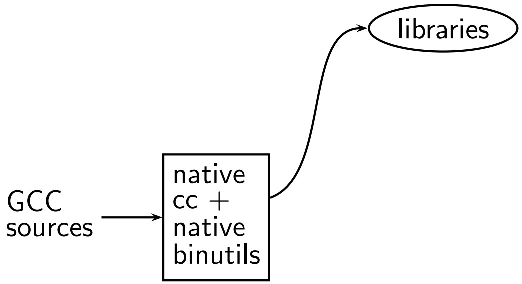
- Generation
  - Generator sources (`$(SOURCE_D)/gcc/gen*.c`) are read and generator executables are created in `$(BUILD)/gcc/build`
  - MD files are read by the generator executables and back end source code is generated in `$(BUILD)/gcc`
- Compilation
  Other source files are read from `$(SOURCE_D)` and executables created in corresponding subdirectories of `$(BUILD)`
- Installation
  Created executables and libraries are copied in `$(INSTALL)`

```
genattr
gencheck
genconditions
genconstants
genflags
genopinit
genpreds
genattrtab
genchecksum
gencondmd
genemit
gengenrtl
genmddeps
genoutput
genrecog
genautomata
gencodes
genconfig
genextract
gengtype
genmodes
genpeep
```

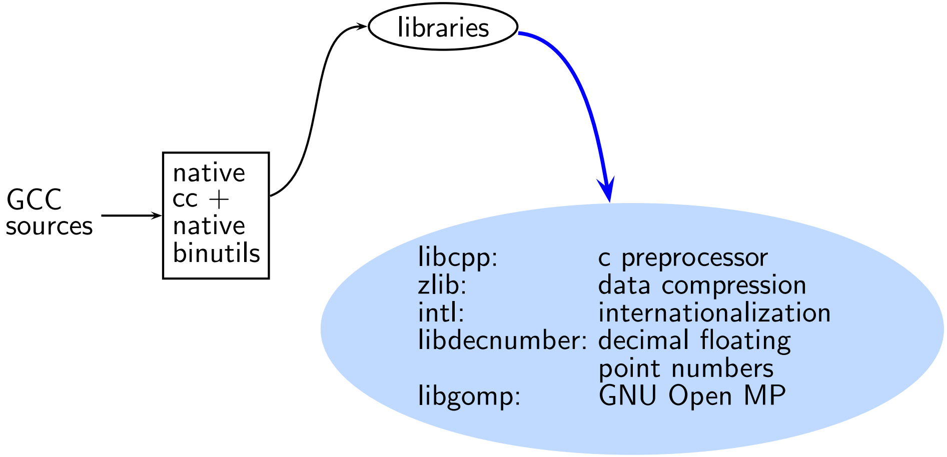
More Details of an Actual Stage 1 Build for C



More Details of an Actual Stage 1 Build for C

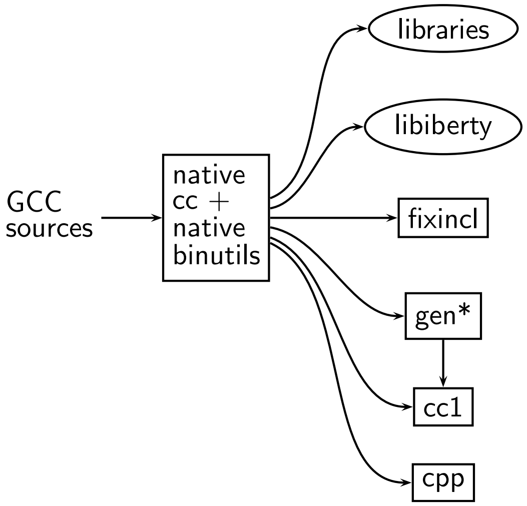


More Details of an Actual Stage 1 Build for C

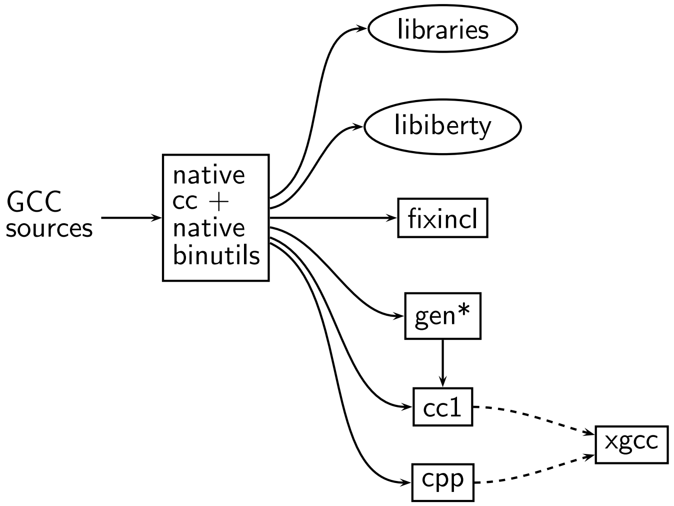


libcpp:	c preprocessor
zlib:	data compression
intl:	internationalization
libdecnumber:	decimal floating point numbers
libgomp:	GNU Open MP

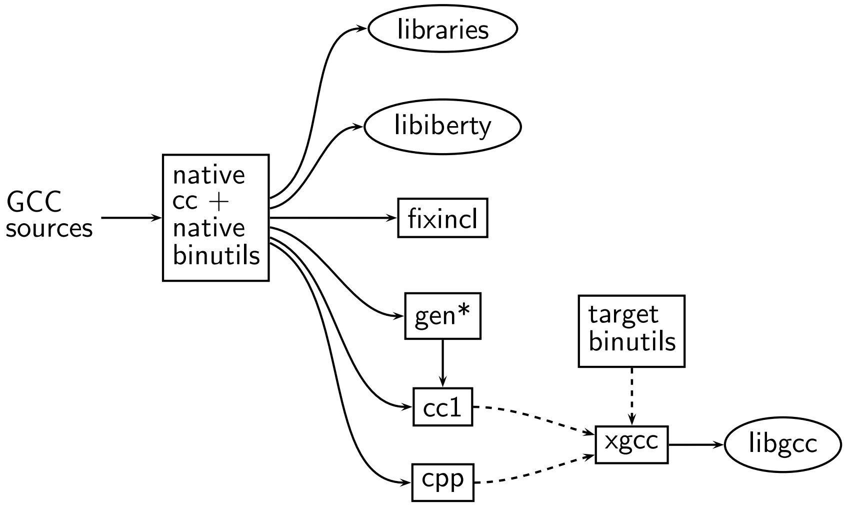
More Details of an Actual Stage 1 Build for C



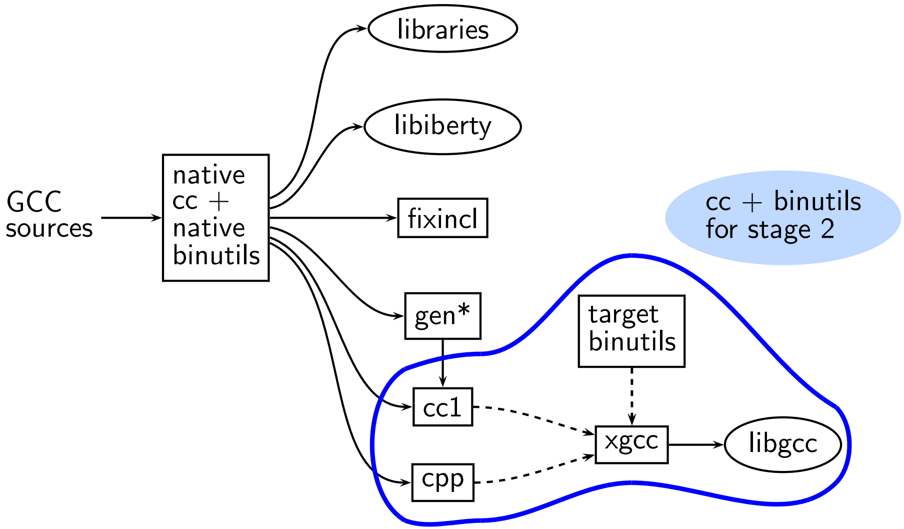
More Details of an Actual Stage 1 Build for C



More Details of an Actual Stage 1 Build for C



More Details of an Actual Stage 1 Build for C



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`)

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`.

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

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.`

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`.

Registering Spim with GCC Build Process

- 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}
    ;;
```

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`

Part 4

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
- 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

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
```